Linux for ARM

- **Автор(ы):** Pelmen Zharenny
- **Связаться с автором:** Telegram

- **Версия LFA:** 2.0-dev

Linux for ARM (далее — *LFA*) — подробное руководство по сборке Linux-системы из исходного кода для компьютеров на архитектуре ARM[1].

---

[1] На данный момент в руководстве описаны сведения о сборке Linux ОС для SoC Allwinner, Broadcom и Rockchip.

Предисловие

В декабре 2023 года я приобрёл себе одноплатник Orange Pi 3 LTS. Скачанная с офсайта Orange Pi система Debian оказалась настолько кривой, что я через какоето время удалил её, установив дистрибутив Armbian. У последнего был ряд преимуществ в виде первичной настройки системы (после первой загрузки запускалась консольная утилита, в которой человек создавал пользователя, от имени которого будет работать, настраивал время и локализацию и совершал ещё ряд каких-то действий). Но были и недостатки: у меня просто не сохранялся ряд настроек системы (да и браузер Firefox тоже вёл себя очень странно), да и сама система была собрана не очень корректно. Пробовал ставить Manjaro Linux, но и с ней были проблемы, а именно видеоартефакты.

Да и в тех дистрибутивах, что я использовал на моей Orange Pi, было огромное число абсолютно не нужного и лишнего ПО, которое только засоряло систему и занимало драгоценное место на SD-карте.

Поэтому я решился на сборку своей системы Linux из исходного кода. До этого я увлекался LFS, поэтому у меня был небольшой опыт сборки таких систем, но не было опыта *кросс-компиляции* системы для другой архитектуры.

LFA, на мой взгляд, является неплохим pet-проектом, которым я занимаюсь исключительно в свободное время. Будет что показать работодателям :).

Почему создан «очередной LFS», неужели нет готовых решений?

Основная причина создания LFA: необходимость зафиксировать для себя любимого те действия, которые я выполнял для сборки своей Linux-системы. Следовательно, LFA сама по себе предназначена только для меня

(автора руководства), однако я надеюсь, что информация отсюда окажется полезна и другим людям.

Создание руководства по сборке своего дистрибутива Linux для ARM-компьютеров в чём-то ново, поскольку достаточно известные руководства типа LFS или Linux для себя предназначены для x86/x86_64 компьютеров, а для ARM-девайсов особой информации не так уж и много, а руководство CLFS Embedded, на основе которого и создана LFA, давно заброшено (либо просто медленно развивается, ведь последняя версия датирована 2019 годом) и, как следствие, несколько устарело.

Что LFA предоставляет пользователю?

В данном руководстве вы не увидите информации о сборке пригодной к *использованию* системы, в которой будет браузер, офисный пакет, рабочее окружение и куча игр. Здесь не будет сведений о сборке системы, пригодной к использованию в IoT или умном доме. Предполагается, что вы дойдёте до этого сами.

LFA предназначена в первую очередь для того, чтобы показать вам, как потенциальному разработчику для Linux, отличия ARM-компьютеров от их x86_64 "собратьев". LFA, может быть, даст вам опыт в сборке программного обеспечения, а также, наверное, расскажет вам о строении Linux-систем. Но это не точно. Основная идея LFA заключается в том, что только вы решаете, собирать ли систему для ARM-ПК, а если и собирать, то *что* и *каким образом*. LFA предоставляет лишь шаблон, по которому можно это сделать.

Зачем мне нужна эта ваша LFA?

Я не знаю, зачем вы читаете это руководство. Я делал его для своих целей, но если оно вам почему-то пригодилось — ну что ж, я рад. Может быть, вы хотите собрать минималистичную встраиваемую операционную систему для управления вашими секс-игрушками? Системы Linux достаточно

гибкие и функциональные, поэтому у них множество сфер применения. Возможно, LFA можно приспособить к некоторым из них, но это уже задача не моя, а ваша.

Отличия от CLFS

Важным отличием от CLFS является то, что LFA полностью на русском языке. К тому же, LFA ориентирована на аудиторию, проживающую в странах СНГ, т.е. тех людей, которые хотя бы на базовом уровне владеют русским языком.

В LFA приводятся различные дополнительные сведения, касающиеся как в целом ОС, использующих ядро Linux, так и таковых систем, предназначенных для компьютеров с архитектурой ARM. В данном руководстве приведены рекомендуемые параметры сборки для некоторых конкретных моделей ПК.

Здесь используются относительно новые версии программного обеспечения, в то время как последний релиз CLFS Embedded для ARM был датирован 2019 годом. В новых версиях ПО исправляют ошибки и уязвимости, а также добавляют новый функционал.

В конце некоторых страниц может быть расположен блок «Смотрите также» со ссылками на дополнительные сведения, документацию или внешние ресурсы. Если вы хотите знать больше, можете ознакомиться с информацией по ссылкам из этого блока.

И в завершение, LFA предоставляет инструкции по сборке загрузчика U-Boot (на данный момент для плат на основе SoC Allwinner, Broadcom, Rockchip и для эмуляции в QEMU).

От авторов

Мы, разработчики LFA, ценим, что вы читаете это руководство. Надеемся, что оно принесёт вам как образовательную пользу, так и, возможно, практическую.

Если у вас возникли вопросы или проблемы, либо вы хотите внести свой вклад в развитие данного руководства, то, пожалуйста, оставьте запрос в нашем репозитории по адресу https://github.com/Linux-for-ARM/handbook.

С уважением, команда Linux for ARM.

Преимущества

Сборка своей системы с нуля для ARM-компьютеров поможет вам узнать, как всё работает вместе и как каждый компонент системы взаимодействует с другим. Возможно, собранная своими руками система будет чем-то лучше и предпочтительнее готовых вариантов. Кроме того, LFA даст опыт в сборке ПО из исходного кода, что может пригодиться при администрировании «обычных» ОС GNU/Linux, предназначенных для обычных x86-компьютеров.

В отличие от уже готовых систем Linux, здесь вам нужно настроить каждый аспект системы самостоятельно. Это заставляет вас читать множество руководств по работе с различными компонентами ОС. Кроме того, это даёт вам полный контроль над системой: вы точно знаете, какое ПО установлено, как оно настроено и где хранятся его файлы.

Другое ключевое преимущество — независимость от других сборщиков. Только вы решаете, что собирать, а что нет, какие применять патчи и как настраивать систему.

Прежде чем начать

Сборка своей системы — не самая простая задача. От вас потребуются знания в администрировании UNIX-систем для того, чтобы вы имели возможность устранять проблемы в процессе сборки, правильно выполнять ввод требуемых команд и при необходимости изменять ход сборки программного обеспечения в зависимости от ваших потребностей и желаний.

Во-первых, вы должны уметь пользоваться терминалом, в частности, владеть программами из состава `coreutils` [1], копировать и перемещать файлы и каталоги, просматривать содержимое директорий и файлов. Также ожидается, что у вас есть знания о процессе установки программного обеспечения в Linux.

Рекомендуем чаще обращаться к разделу «Вспомогательные материалы», в котором находятся ответы на часто задаваемые вопросы. В этот раздел часто добавляются новые сведения.

---

[1] Программы для работы с файлами (копирование, удаление, перемещение, создание), правами доступа, вычисления контрольных сумм и т.д.

Принятые обозначения

В руководстве используются следующие обозначения:

```
./configure --prefix=/usr --target=$LFA_TGT
```

Этот текст необходимо набрать в терминале в точности так, как указано, если иное не сказано в тексте.

Иногда строка разделяется до двух или более с использованием символа `\`:

```
ARCH=arm ./configure --prefix=/usr \
    --target=$LFA_TGT \
    --with-sysroot=$LFA_SYS \
    --disable-nls \
    --disable-threads
```

Обратите внимание на то, что после `\` должен быть переход на новую строку (`Enter`). Другие символы приведут к некорректному результату и ошибкам.

```
2024-02-27 18:58:13 [INFO] (mdbook::cmd::serve): Files changed:
["/home/admin/Work/lfa/src/typography.md"]
2024-02-27 18:58:13 [INFO] (mdbook::cmd::serve): Building book...
2024-02-27 18:58:13 [INFO] (mdbook::book): Book building has started
2024-02-27 18:58:13 [INFO] (mdbook::book): Running the html backend
```

Этот текст используется для отображения вывода в терминале.

Используется, чтобы подчеркнуть важную информацию, на которую следует обратить внимание.

Используется для ссылок на страницы руководства.

Используется для ссылок на внешние ресурсы.

Важно

Используется для указания на критически важную информацию. На неё следует обратить особое внимание.

Используется для указания на информацию рекомендательного характера. Не рекомендуется пропускать эти указания и внимательно с ними ознакомиться.

Опечатки и неточности

Если вы нашли в руководстве ошибку, опечатку или хотите предложить нам какое-либо изменение, которое, на ваш взгляд, важно для LFA, то, пожалуйста, оставьте запрос в нашем **репозитории** на GitHub. Мы открыты к диалогу, и вы, как читатель, всегда можете предложить свои замечания, улучшения и изменения.

Целевая архитектура

Предполагается, что LFA будет собираться для архитектуры ARMv8 (AArch64). Работа собранной по LFA системы проверялась на процессоре[1] Allwinner H6 (ARM Cortex-A53[2]). С другой стороны, в данном руководстве ещё остались инструкции, содержащий в том числе сведения для более старых архитектур семейства ARM в наследие от руководства CLFS Embedded. В ближайшее время мы не планируем удалять их или как-то актуализировать, по крайней мере это не будет сделано до релиза 2.0. Тем не менее, основной архитектурой для нас является ARMv8.

Для сборки LFA для того или иного AArch64-процессора мы будем использовать x86_64 хост. Компиляция ПО будет производиться посредством кросс-компилятора, который мы соберём в начале и будем использовать на протяжении всего руководства.

---

[1] В адрес подобных устройств куда справедливее использовать термин «SoC» (System on Chip, Система на Кристалле), но для простоты ограничимся понятием «процессор».

[2] Существуют процессоры Cortex-A, предназначенные для устройств, требующих относительно высокой производительности, Cortex-R для ПО, работающего в режиме реального времени и Cortex-M для микроконтроллеров и встраиваемых устройств. В данном руководстве идёт упор на процессоры Cortex-A.

Информация об используемом ПО

Как говорилось ранее, в LFA содержатся инструкции только по сборке базового программного обеспечения. Собранная система будет включать в себя базовое программное обеспечение для работы с файлами, процессами и сетью. Однако это не значит, что полученная система будет максимально компактной.

BusyBox-1.36.1

Объединяет крошечные версии многих распространённых утилит UNIX в один небольшой двоичный файл (1-2 Мбайт). Он заменяет большинство утилит, которые обычно находятся в GNU Coreutils, GNU Findutils и т.д.

- Домашняя страница: https://www.busybox.net
- Скачать: https://busybox.net/downloads/busybox-1.36.1.tar.bz2
- MD5 сумма: `0fc591bc9f4e365dfd9ade0014f32561`

GCC-13.2.0

Набор компиляторов GNU GCC.

- Домашняя страница: https://gcc.gnu.org
- Скачать: https://ftp.gnu.org/gnu/gcc/gcc-14.1.0/gcc-14.1.0.tar.xz
- MD5 сумма: `24195dca80ded5e0551b533f46a4481d`

GMP-6.3.0

Пакет с математическими библиотеками, которые предоставляют полезные функции для арифметики произвольной точности. Необходим

для сборки GCC.

- Домашняя страница: https://gmplib.org
- Скачать: https://ftp.gnu.org/gnu/gmp/gmp-6.3.0.tar.xz
- MD5 сумма: `956dc04e864001a9c22429f761f2c283`

LFA Bootscripts-1.0

Набор скриптов, используемых системой инициализации из BusyBox для запуска/остановки демонов и иных программ во время запуска/выключения LFA.

- Домашняя страница: https://github.com/Linux-for-ARM/lfa-bootscripts/
- Скачать: https://github.com/Linux-for-ARM/lfa-bootscripts/releases/download/v1.0/bootscripts-1.0.tar.xz
- MD5 сумма: `217d8f3d253f980691129e1c251379fc`

Linux-6.6.44

Ядро операционной системы.

- Домашняя страница: https://www.kernel.org
- Скачать: https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.6.44.tar.xz
- MD5 сумма: `613b7d75598dbf359b180c11acac39cc`

MPC-1.3.1

Математические функции для комплексных чисел. Необходим для сборки GCC.

- Домашняя страница: http://www.multiprecision.org/
- Скачать: https://ftp.gnu.org/gnu/mpc/mpc-1.3.1.tar.gz

- MD5 сумма: `5c9bc658c9fd0f940e8e3e0f09530c62`

MPFR-4.2.1

Функции для арифметики множественной точности. Необходим для сборки GCC.

- Домашняя страница: https://www.mpfr.org
- Скачать: https://ftp.gnu.org/gnu/mpfr/mpfr-4.2.1.tar.xz
- MD5 сумма: `523c50c6318dde6f9dc523bc0244690a`

TF-A-2.10.9

Проект Trusted Firmware-A предоставляет эталонную реализацию безопасного программного обеспечения для процессоров класса ARMv7-A и ARMv8-A.

- Домашняя страница: https://www.trustedfirmware.org/projects/tf-a/
- Скачать: https://raw.githubusercontent.com/Linux-for-ARM/packages/master/tf-a/trusted-firmware-a.git-refs_tags_lts-v2.10.9.tar.gz
- MD5 сумма: `42090a81f64db8d017354e86fdc2775f`

Wireless Tools-29

Набор инструментов для работы с Wireless Extensions (WE — API ядра Linux, позволяющий драйверу передавать в пользовательское пространство конфигурацию и статистику, характерные для беспроводных локальных сетей).

- Домашняя страница: https://hewlettpackard.github.io/wireless-tools/

- Скачать: https://hewlettpackard.github.io/wireless-tools/wireless_tools.29.tar.gz
- MD5 сумма: `e06c222e186f7cc013fd272d023710cb`

binutils-2.43

Этот пакет содержит компоновщик, ассемблер и другие утилиты для работы с объектными файлами.

- Домашняя страница: https://www.gnu.org/software/binutils
- Скачать: https://sourceware.org/pub/binutils/releases/binutils-2.43.tar.xz
- MD5 сумма: `a54bb52cd75555416f316fbbe885925a`

crust-0.6

Низкоуровневый компонент для плат на базе Allwinner, предназначенный для уравления питанием. Во время глубокого сна ядра процессора, контроллер DRAM и большинство встроенных периферийных устройств отключаются от питания, что позволяет снизить электропотребление на >80%. На платах без PMIC crust также отвечает за упорядоченное включение и выключение устройства.

- Домашняя страница: https://github.com/crust-firmware/crust
- Скачать: https://raw.githubusercontent.com/Linux-for-ARM/packages/master/crust/crust-0.6.tar.gz
- MD5 сумма: `e558d2365411c82d0393e47e57aa7bea`

iana-etc-20241122

Данные для сетевых служб и сервисов. Необходим для обеспечения надлежащих сетевых возможностей.

- Домашняя страница: https://www.iana.org/protocols
- Скачать: https://github.com/Mic92/iana-etc/releases/download/20241122/iana-etc-20241122.tar.gz
- MD5 сумма: `38064a8e7c2233e23911ef9d39360584`

musl-1.2.5

Минималистичная стандартная библиотека языка C.

- Домашняя страница: https://musl.libc.org
- Скачать: https://musl.libc.org/releases/musl-1.2.5.tar.gz
- MD5 сумма: `ac5cfde7718d0547e224247ccfe59f18`

rkbin-master

Прошивка BL31 для тех Rockchip SoC, для которых не обеспечена поддержка BL31 из состава TF-A.

- Домашняя страница: https://github.com/rockchip-linux/rkbin
- Скачать: https://raw.githubusercontent.com/Linux-for-ARM/packages/master/rkbin/rkbin-master.tar.xz
- MD5 сумма: `8e26a9aaeacd6f41839d204ca75cdfd7`

u-boot-2024.04

Загрузчик операционной системы, предназначенный для встраиваемых систем на MIPS, ARM, PowerPC и т.д.

- Домашняя страница: https://source.denx.de/u-boot/u-boot
- Скачать: https://source.denx.de/u-boot/u-boot/-/archive/v2024.04/u-boot-v2024.04.tar.bz2
- MD5 сумма: `7267d5902ea37ee56e71162a53b331df`

Подготовка к сборке

В данной главе приведены сведения, касаемые подготовки вашей хост-системы к сборке LFA. Вам требуется создать пользователя, настроить его окружение и создать базовую структуру каталогов собираемой системы.

Ход сборки LFA

1. **Подготовка к сборке** — на данном этапе мы создадим отдельного пользователя, от имени которого будем собирать систему, настроим его окружение и подготовим всё, что требуется для сборки LFA.
2. **Сборка кросс-компилятора** — поскольку сборка производится с компьютера x86_64 для компьютера AArch64, нам требуется собрать кросс-компилятор, позволяющий выполнить это. После сборки с его помощью базовой системы мы его удалим.
3. **Сборка базовой системы** — здесь мы собирём базовую систему, которая и будет являться той самой LFA.
4. **Настройка базовой системы** — на данном этапе требуется произвести настройку базовой системы: создать ряд конфигурационных файлов и при необходимости исправить существующие.
5. **Сборка ядра** — на данном этапе необходимо собрать ядро Linux с учётом всех ваших требований и пожеланий.
6. **Сборка загрузчика** — заключительный этап, на котором вы соберёте загрузчик U-Boot для конкретной модели компьютера, для которой вы собираете систему LFA.
7. **Конец** — сборка системы полностью завершилась. Теперь остаётся сделать img-образ, пригодный для записи на внешний носитель, который будет выступать в роли загрузочного в компьютере, для которого вы собирали LFA.

Требования к хосту

Оборудование

- Раздел на жёстком диске или просто свободное место, рекомендуемый объём которого - 10 Гб и более.
- Если оперативной памяти хост-компьютера мало (менее 4 Гб), рекомендуется создать раздел или файл подкачки. Кроме того, можно использовать `zram`.

Программное обеспечение

Первое и самое важное - на компьютере должна быть установлена ОС Linux. Сборка на других системах семейства UNIX не проверялась и не рекомендуется.

Если у вас нет на компьютере установленной системы Linux, допускается сборка LFA в операционной системе Windows в окружении WSL. Выбор дистрибутива для запуска в WSL не имеет значения, главное здесь только наличие в дистрибутиве указанного ниже программного обеспечения. Однако сборка LFA в WSL не тестировалась и может не работать вовсе.

На вашей хост-системе или в окружении WSL должно быть установлено ПО из списка ниже с указанными минимальными версиями. Для большинства современных дистрибутивов Linux это не должно быть особой проблемой. Самое главное, чтобы версия, которую предоставляет используемый вами дистрибутив, не была ниже, чем в списке далее.

- `bash-3.2` (`/bin/sh` должна быть ссылкой на `bash`)
- `bc-1.07` (для компиляции Linux)
- `binutils-2.13`
- `bison-2.7` (`/usr/bin/yacc` должен быть ссылкой на `bison`)

- `coreutils-8.1`
- `diffutils-2.8.1`
- `findutils-4.2.31`
- `flex-2.6.4`
- `gawk-4.0.1` (`/usr/bin/awk` должен быть ссылкой на `gawk`)
- `gcc-5.2` (влючающий компилятор языка C, C++)
- `grep-2.5.1a`
- `gzip-1.3.12`
- `linux-4.19`
- `m4-1.4.10`
- `make-4.0`
- `ncurses-6.3` (для сборки BusyBox, Linux и U-Boot)
- `patch-2.5.4`
- `perl-5.8.8`
- `python-3.4`
- `rsync-3.2.7` (для установки заголовков ядра на этапе сборки кросс-компилятора)
- `sed-4.1.5`
- `setuptools-66.1` (для компиляции U-Boot)[1]
- `swig-4.0` (для компиляции U-Boot)
- `tar-1.22`
- `texinfo-6.8` (для сборки binutils)
- `u-boot-tools-2023.01` (для сборки ядра Linux и работы с загрузчиком U-Boot)
- `xz-5.0`
- `wget-1.23` и `md5sum` (для скачивания исходного кода LFA)

> ⚠ **Внимание**
>
> Для некоторых моделей Allwinner SoC требуется сборка компонента `crust`, которая производится с помощью кросс-компилятора для архитектуры `or1k`. Здесь не приводится инструкций о его сборке, поскольку информация сразу о двух кросс-компиляторах (x86_64 -> ARM и x86_64 -> or1k) усложнит руководство и собьёт с толку тех читателей, кому or1k вовсе не нужен. Вы можете либо собрать нужный вам кросс-компилятор самостоятельно, либо использовать готовые пакеты: так, например, в репозиториях Arch Linux есть нужные пакеты с binutils и GCC для нужной архитектуры.

Некоторые дистрибутивы включают в свои репозитории метапакет, объединяющий большинство описанных выше утилит. В зависимости от дистрибутива Linux название этого пакета может меняться. Например, в Debian этот пакет называется `build-essential`. Рекомендуем вам установить сначала его, а потом доустановить все недостающие пакеты.

Проверка системных требований

Чтобы проверить наличие в вашей хост-системе всех необходимых версий необходимого ПО, выполните следующие команды:

```
cat > ver-check.sh << "EOF"
#!/bin/bash
# A script to list version numbers of critical development tools

# If you have tools installed in other directories, adjust PATH here

# For Linux for ARM (LFA) 2.0
# Forked from Linux from Scratch 12.2
(https://linuxfromscratch.org/lfs/view/stable-
systemd/chapter02/hostreqs.html)

LC_ALL=C
PATH=/usr/bin:/bin

bail() { echo -e "\e[1;31mFATAL:\e[0m $1"; exit 1; }
grep --version > /dev/null 2> /dev/null || bail "grep does not work"
sed '' /dev/null || bail "sed does not work"
sort   /dev/null || bail "sort does not work"

ver_check()
{
   if ! type -p $2 &>/dev/null
   then
     echo "ERROR: Cannot find $2 ($1)"; return 1;
   fi
   v=$($2 --version 2>&1 | grep -E -o '[0-9]+\.[0-9\.]+[a-z]*' |
head -n1)
   if printf '%s\n' $3 $v | sort --version-sort --check &>/dev/null
   then
     printf "OK:    %-9s %-6s >= $3\n" "$1" "$v"; return 0;
   else
     printf "ERROR: %-9s ver. $v is TOO OLD ($3 or later
required)\n" "$1";
     return 1;
   fi
}

ver_check2() {
   if ! type -p $2 &>/dev/null
   then
     echo "ERROR: Cannot find $2 ($1)"; return 1;
   fi
   v=$($2 -version 2>&1 | grep -E -o '[0-9]+\.[0-9\.]+[a-z]*' | head
-n1)
   if printf '%s\n' $3 $v | sort --version-sort --check &>/dev/null
   then
     printf "OK:    %-15s %-6s >= $3\n" "$1" "$v"; return 0;
```

```
   else
     printf "ERROR: %-15s ver. $v is TOO OLD ($3 or later
required)\n" "$1";
     return 1;
   fi
}

ver_kernel()
{
   kver=$(uname -r | grep -E -o '^[0-9\.]+')
   if printf '%s\n' $1 $kver | sort --version-sort --check
&>/dev/null
   then
     printf "OK:    Linux Kernel $kver >= $1\n"; return 0;
   else
     printf "ERROR: Linux Kernel ($kver) is TOO OLD ($1 or later
required)\n" "$kver";
     return 1;
   fi
}

# Coreutils first because --version-sort needs Coreutils >= 7.0
ver_check  Coreutils      sort            8.1 || bail "Coreutils too
old, stop"
ver_check  Bash           bash            3.2
ver_check  Bc             bc              1.07
ver_check  Binutils       ld              2.13
ver_check  Bison          bison           2.7
ver_check  Diffutils      diff            2.8.1
ver_check  Findutils      find            4.2.31
ver_check  Flex           flex            2.6.4
ver_check  Gawk           gawk            4.0.1
ver_check  GCC            gcc             5.2
ver_check  "GCC (C++)"    g++             5.2
ver_check  Grep           grep            2.5.1a
ver_check  Gzip           gzip            1.3.12
ver_check  M4             m4              1.4.10
ver_check  Make           make            4.0
ver_check  Ncurses        ncurses6-config 6.3
ver_check  Patch          patch           2.5.4
ver_check  Perl           perl            5.8.8
ver_check  Python         python3         3.4
ver_check  Rsync          rsync           3.2.7
ver_check  Sed            sed             4.1.5
ver_check2 Swig           swig            4.0
ver_check  Tar            tar             1.22
ver_check  Texinfo        texi2any        5.0
ver_check  "U-Boot Tools" mkimage         2023.01
```

```
ver_check  Xz             xz              5.0.0
ver_check2 Wget           wget            1.23
ver_kernel 4.19

if mount | grep -q 'devpts on /dev/pts' && [ -e /dev/ptmx ]
then echo "OK:    Linux Kernel supports UNIX 98 PTY";
else echo "ERROR: Linux Kernel does NOT support UNIX 98 PTY"; fi

alias_check() {
   if $1 --version 2>&1 | grep -qi $2
   then printf "OK:    %-4s is $2\n" "$1";
   else printf "ERROR: %-4s is NOT $2\n" "$1"; fi
}
echo "Aliases:"
alias_check awk GNU
alias_check yacc Bison
alias_check sh Bash

if [ "$(nproc)" = "" ]; then
   echo "ERROR: nproc is not available or it produces empty output"
else
   echo "OK: nproc reports $(nproc) logical cores are available"
fi
EOF

bash ver-check.sh
```

Примеры для разных дистрибутивов

В разделе ниже представлены команды для установки необходимого для сборки ПО в основные дистрибутивы Linux. Еси вы не нашли свой дистрибутив здесь, обратитесь к его репозиториям с ПО и документации.

Ubuntu/Debian

```
sudo apt install build-essential texinfo rsync u-boot-tools swig
```

---

[1] Это модуль языка Python, который может быть установлен с помощью пакетного менеджера `pip` (входит в состав Python и обычно устанавливается вместе с ним), либо с помощью пакетного менеджера вашего дистрибутива, если в его репозиториях поставляются пакеты для Python (в таком случае имя пакета, содержащего Python-модуль `setuptools`, может быть `python-setuptools` или `python3-setuptools`). Использование пакетного менеджера вашего дистрибутива вместо `pip` предпочтительнее, поскольку в таком случае `setuptools` будут установлены именно в систему, откуда интерпретатор Python будет иметь к нему доступ. С недавнего времени пакетный менеджер `pip` отключил «глобальную» установку Python-модулей в систему по умолчанию, став предпочитать установку модулей в виртуальное окружение Python.

О времени сборки пакетов

Время сборки пакетов во многом зависит от мощности компьютера. Но также на время влияют и иные факторы, такие как, например, версия компилятора и системы сборки, а также использование многопоточной сборки.

Поскольку от компьютера к компьютеру время сборки может меняться (на одном ПК пакет `some-pkg` собирается за 3 минуты, а на другом тот же `some-pkg` — за 3 недели), в руководстве введена специальная единица времени, которая называется «ОВС» (Относительное Время Сборки).

1 ОВС равна времени сборки первого пакета. К примеру, если первый пакет в этом руководстве собирается за 3 минуты, то 1 ОВС = 3 мин. Если время сборки какого-то пакета = 10 ОВС, то, переводя в минуты, это будет 30 минут.

> ОВС не даёт совсем точных значений, поскольку они зависят от многих факторов, включая версию компилятора GCC на хост-системе. ОВС нужна для *примерной* оценки времени сборки пакета.

Обратите своё внимание на то, что в данном руководстве некоторые пакеты предлагается конфигурировать перед сборкой вам самим (например, пакеты BusyBox, Linux, U-Boot). В зависимости от выставленных вами параметров сборки изменится и время сборки. ОВС, указанные в руководстве для всех пакетов, высчитывались с учётом *стандартных* настроек сборки пакетов.

Некоторые компьютеры могут контролировать тактовую частоту процессора. В Linux есть функционал, позволяющий менять профили производительности (энергосбережение, сбалансированный и производительность). Также в вашем хост-дистрибутиве может быть команда `powerprofilesctl`. Перед началом сборки вы можете использовать профиль, настроенный на максимальную

производительность (и, соответственно, максимальное энергопотребление). Очевидно, что это заставит хост-систему быстрее собирать LFA.

Самостоятельное вычисление ОВС

Для того, чтобы самостоятельно вычислить время сборки пакета, введите следующую команду для сборки:

```
tar -xf PROGRAM_NAME-PROGRAM_VERSION.tar.xz
cd PROGRAM_NAME-PROGRAM_VERSION

{ time \
  {
    # инструкции по сборке пакета, например:
    ./configure --prefix=/usr &&
    make CROSS_COMPILE=$LFA_TGT- &&
    make DESTDIR=$LFA_CROSS install
  }
} 2>&1 | tee ../build.log
```

Создание пользователя lfa

Рекомендуем выполнять сборку от имени отдельного пользователя, у которого будет доступ только к ограниченному набору файлов. Не рекомендуем вам собирать систему от имени текущего пользователя или пользователя `root`, так как, если вы ошиблись в наборе команд для сборки, есть вероятность порчи или потери пользовательских данных или поломки системы.

Вы можете использовать произвольного пользователя, но для упрощения настройки чистого рабочего окружения создайте нового пользователя с именем `lfa` как члена группы `lfa`:

> **Внимание**
>
> Если вы читаете PDF-версию руководства и хотите *скопировать* из него команды далее в терминал, рекомендуем вам не делать этого. Несмотря на то, что сами команды корректные (и правильно отображаются в PDF-книге), при копировании зачастую в буфер обмена попадают некорректные данные (такой проблемы нет, если вы читаете обычную HTML-версию LFA). Рекомендуем *перепечатывать* команды из PDF-версии LFA.

```
groupadd lfa
useradd -s /bin/bash -g lfa -m -k /dev/null lfa
```

Значения новых параметров:

`groupadd lfa` — создаёт новую группу `lfa`.

`-s /bin/bash` — указывает `/bin/bash` оболочкой по умолчанию для пользователя `lfa`.

`-g lfa` — добавляет пользователя `lfa` в группу `lfa`.

`-m` — создаёт домашнюю директорию пользователя `lfa` (по умолчанию в `/home/lfa`).

`-k /dev/null` — предотвращает копирование файлов из `/etc/skel` - каталога, в котором содержатся стандартные конфиги и иные файлы, которые обычно копируются в домашнюю директорию пользователя во время его создания.

Когда вы находитесь в терминале от имени пользователя `root` и переключаетесь на пользователя `lfa`, вам не требуется ввода пароля `lfa`. Однако когда вы переключаетесь на `lfa` с обычного пользователя, без пароля у вас не получится этого сделать.

Задайте для пользователя `lfa` новый пароль:

```
passwd lfa
```

> ⚠ **Внимание**
>
> Теперь вам необходимо войти от имени `lfa`. Для этого выполните:
>
> ```
> su - lfa
> ```
>
> Символ `-` указывает `su` запустить оболочку для входа в систему, а не оболочку, не предназначенную для входа. Разницу между двумя типами оболочек подробно описана в *bash(1)*.
>
> Если вы прервали сборку LFA досрочно и хотите продолжить её спустя какое-то время, то вам нужно будет выполнить вход в этого пользователя снова с помощью этой же команды.

Настройка окружения

После того, как вы создали нового пользователя, от имени которого будете собирать LFA, нужно настроить его окружение. Как минимум, требуется объявить ряд переменных окружения, которые мы будем использовать при сборке программ. В таких переменных содержатся сведения, неизменные от пакета к пакету. Например, путь, куда нужно устанавливать программы, целевая архитектураи т.п. Если после создания пользователя вы вошли в терминале от его имени, то приступайте к выполнению инструкций ниже. В противном случае от вас требуется сначала войти от имени `lfa` с помощью команды `su - lfa`.

Первым делом требуется создать файл `~/.bash_profile`:

```
cat > ~/.bash_profile << "EOF"
exec env -i HOME=$HOME TERM=$TERM PS1='\u:\w\$ ' /bin/bash
EOF
```

При входе в систему от имени пользователя `lfa` начальной оболочкой обычно является оболочка входа в систему, которая читает файл `/etc/profile`, содержащий основные общесистемные настройки и переменные окружения, а затем `~/.bash_profile`. Команда `exec env -i ... /bin/bash` в последнем заменяет запущенную оболочку новой с абсолютно пустым окружением, за исключением переменных `$HOME`, `$TERM` и `$PS1`. Это гарантирует нам, что никакие нежелательные и потенциально опасные переменные окружения из хост-системы не «просочатся» в нашу среду сборки.

Новый экземпляр оболочки вместо `/etc/profile` и `~/.bash_profile` будет читать уже файл `~/.bashrc`. Создайте его:

```
cat > ~/.bashrc << "EOF"
set +h
umask 022
unset CFLAGS
LFA=$HOME/lfa
LC_ALL=C
PATH=$LFA/tools/bin:/bin:/usr/bin
export LFA LC_ALL PATH
EOF
```

Применение изменений

Для того, чтобы применить внесённые нами изменения, выполните:

```
source ~/.bash_profile
```

Значения параметров `~/.bashrc`:

Команда `set +h` отключает хеш-функцию BASH. Хеширование в общем случае является полезной вещью, поскольку BASH использует хеш-таблицу для запоминания полного пути к исполняемым файлам, чтобы не искать путь до программы в `$PATH` снова и снова. Однако новые программы для сборки, которые будут устанавливаться в `$LFA/tools/bin`, требуется использовать сразу же после их установки. После отключения хеширования оболочка BASH сразу найдёт только что установленные программы, не вспоминая о предыдущей версии ПО в другом месте.

Исполнение команды `umask 022` гарантирует, что вновь созданные файлы и каталоги могут быть записаны только их владельцем, но могут быть прочитаны и исполнены любым пользователем.

Далее во избежание сбоев во время сборки кросс-компилятора нам требуется «удалить» переменную окружения `CFLAGS`.

Переменная окружения `LFA` содержит путь до директории, в которой будем собирать систему.

`LC_ALL` управляет локализацией программ, заставляя сообщения, которые они выводят в терминал, следовать конвенциям указанной в этой переменной страны. Во избежание проблем сборки любые значения `LC_ALL`, отличные от `POSIX` или `C`, использовать не рекомендуется.

Переменная окружения `PATH` содержит пути до директорий, в которых содержатся исполняемые файлы. Благодаря этой переменной в терминале мы можем просто ввести `some_program` вместо указания полного пути `/usr/bin/some_program`. В эту переменную мы добавляем путь до двоичных исполняемых файлов собираемого нами кросс-компилятора (`$LFA/tools/bin`), а также «стандартные» для хост-системы директории `/bin` и `/usr/bin`. Указание пути до кросс-компилятора раньше, чем путей до других инструментов вкупе с отключением хеширования гарантирует, что для сборки системы у нас будут использованы только нужные программы из кросс-компилятора, а не из хост-системы.

⚠ Внимание

На некоторых популярных дистрибутивах присутствует недокументированный файл `/etc/bash.bashrc`, параметры из которого используются при инициализации BASH, что может привести к изменению окружения пользователя `lfa` таким образом, что это может повлиять на сборку критически важных пакетов LFA. Чтобы убедиться, что оболочка `lfa` чиста от лишних параметров и переменных окружения из этого недокументированного файла, проверьте его наличие и, если он присутствует, переместите его в другое место. Для этого от имени `root` выполните команду:

```
[ ! -e /etc/bash.bashrc ] || mv -v /etc/bash.bashrc
/etc/bash.bashrc.LFA
```

Команда выше проверяет наличие файла `/etc/bash.bashrc` и, если он есть, переименовывает его в `/etc/bash.bashrc.LFA`.

Когда пользователь `lfa` больше не нужен, вы можете спокойно восстановить этот файл, если он вам нужен. Обратите внимание на то, что в готовой системе LFA этот файл не нужен. Если вы хотите скопировать какие-либо конфиги из своей хост-системы, пропустите копирование `bash.bashrc`.

Установка переменных сборки

В данной части будет произведена установка ряда важных переменных, которые будут использоваться для сборки программ. Определите, для какой архитектуры семейства ARM вы будете собирать систему и в зависимости от этого выбирайте набор нужных вам переменных. Обращаем ваше внимание на то, что LFA предназначена в первую очередь для процессоров с архитектурой AArch64. Работа LFA на других архитектурах не проверялась. Поддержка других архитектур оставлена здесь "в наследие" от оригинальной CLFS.

> То, что это *дополнительные* переменные окружения, совсем не значит, что они являются *необязательными*. Мы вынесли объявление этих переменных окружения в отдельную страницу потому, что на предыдущей странице шла речь об общих переменных окружения. Переменные же на текущей странице предназначены исключительно для сборки ПО. Кроме того, если для общих переменных существует один единственный шаблон, который можно использовать для всех сборок LFA, то значения для переменных на данной странице пользователь выбирает самостоятельно в зависимости от оборудования, для которого он собирает эту систему.

Хост и цель

Напомним вам, что такое хост-компьютер и целевой компьютер. *Хост-компьютер* (host) — это ПК, на котором вы собираете кросс-компилятор и прочие вещи. А *целевой компьютер* (target) — тот ПК, для которого вы это собираете. В данном руководстве хост-компьютером является компьютер на архитектуре x86_64, а целевым — компьютер на архитектуре AArch64.

> ⚠ **Внимание**
>
> ---
>
> В настоящее время мы составляем список процессоров (бренд процессора, модель процессора, его архитектура и модели ПК, где он применяется), в котором будут указаны полные сведения о том, какие значения используются для переменных `LFA_FPU`, `LFA_ARCH` и `LFA_TGT`.
>
> ---
>
> В случае, если вы собрали систему, используя определённые значения, то, пожалуйста, оставьте запрос в нашем репозитории GitHub по этому поводу. Укажите бренд процессора, еего модель и его архитектуру, а также модель компьютера, для которого вы собирали систему. Кроме того, совсем не лишним будет, если вы укажете, как собралась ваша система: собралась ли она корректно или были всевозможные ошибки в процессе сборки или в процессе её функционирования.
>
> **Пример такого запроса (issue):**
>
> ```
> # Сборка системы для компьютера Orange Pi 3 LTS
>
> - **Бренд процессора:** Allwinner
> - **Модель процессора:** Allwnner H6
> - **Архитектура:** Cortex-A53 (AArch64)
> - **Статус сборки:** система собралась нормально
> - **Статус функционирования:** система работает корректно
>
> ## Значения переменных сборки
>
> `LFA_TGT`=`aarch64-linux-musleabihf`
>
> `LFA_HOST`=`...`
>
> `LFA_ARCH`=`armv8-a`
> ```

Для архитектуры AArch64

Для сборки кросс-компилятора вам нужно задать несколько переменных, которые будут зависеть от того, для какого оборудования вы хотите собрать LFA. Вам нужно выбрать триплет для целевой архитектуры, архитектуру процессора и т.д. Для выбора нужных значений пользуйтесь приведёнными на данной странице таблицами.

Установите триплеты для хоста и целевой машины:

```
export LFA_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
export LFA_TGT="aarch64-linux-musleabihf"
```

Немного про `LFA_TGT`.

Переменная окружения `LFA_TGT` хранит в себе значение *триплета* для целевой машины. Триплет принимает тип целевой машины, состоящий из следующих элементов: `<CPU>-<Vendor>-<KERNEL>-<OS>`. Поскольку поле `<Vendor>` часто не имеет значения, многие системы сборки (такие как, например, система сборки на основе `autoconf`, используемая всеми или практически всеми пакетами из LFA) позволяют его опустить.

Проницательный читатель LFA может задаться вопросом, почему мы применяем понятие «триплет» к четырёхкомпонентному имени. Компоненты `<Kernel>` и `<OS>` начинались как единое поле `<System>`. Такая форма применяется для многих ОС, например, `x86_64-unknown-freebsd`. Но две системы могут иметь одно и то же ядро, но быть слишком разными, чтобы использовать для них один и тот же триплет. Например, система Android, работающая на мобильном устройстве, полностью отличается от системы PostmarketOS, хотя обе они имеют одно и то же ядро Linux, запускаются на том же процессоре и, возможно, даже на одном и том же мобильном устройстве. Что и говорить — одна и та же система, имеющая одно и то же ядро, запускающаяся на одном и том же

компьютере, но имеющая разные стандартные библиотеки C (например, первая система использует `glibc`, а вторая — `musl`), не могут иметь одинаковый триплет.

Поэтому поле `<System>` было разделено на поля `<Kernel>` и `<OS>`, чтобы однозначно обозначить различные вариации систем. В нашем примере система Android имеет триплет `aarch64-unknown-linux-android`, а система PostmarketOS — `aarch64-linux-musleabihf`. Ну а слово «триплет» до сих пор осталось в профессиональном лексиконе.

Выберите архитектуру, для которой будете собирать систему:

```
export LFA_ARCH="архитектура"
```

ARCH	ARCH	ARCH	ARCH
armv8-a	armv8-m	armv8	armv8-r
armv8.1-a	armv8.1-m	armv8.1-r	

Например, для процессоров Cortex-A53 `$LFA_ARCH="armv8-a"`.

Запишите эти переменные в `~/.bashrc`, чтобы не вводить их значения каждый раз после входа от имени пользователя `lfa`:

```
cat >> ~/.bashrc << EOF
export LFA_HOST="$LFA_HOST"
export LFA_TGT="$LFA_TGT"
export LFA_ARCH="$LFA_ARCH"
EOF
```

> ⚠ **Внимание**
>
> Далее и на протяжении всего руководства, если вы собираете систему для AArch64, то **не используйте** переменные окружения `$LFA_FLOAT` и `$LFA_FPU`, а также пропускайте при вводе команд строки, содержащие эти переменные окружения. Например, если вы собираете систему для AArch64, то скрипту `configure` не следует передавать эти аргументы:
>
> ```
> --with-float=$LFA_FLOAT \
> --with-fpu=$LFA_FPU
> ```

Для других архитектур

Для сборки кросс-компилятора вам нужно задать несколько переменных, которые будут зависеть от того, для какого оборудования вы хотите собрать LFA. Вам нужно выбрать триплет для целевой архитектуры, архитектуру процессора и т.д. Для выбора нужных значений пользуйтесь приведёнными здесь таблицами.

Если ваш целевой процессор имеет аппаратную поддержку плавающей запятой, то установите переменную `LFA_FLOAT` в значение `hard` или `softfp`. Используйте `softfp`, если в будущем вы будете использовать в собранной системе ещё и программы, скомпилированные с помощью `soft`. В противном случае используйте `hard`. Если ваш целевой процессор не поддерживает плавающую запятую, используйте в качестве значения `LFA_FLOAT` вариант `soft`.

Если ваш процессор имеет одну из архитектур ARMv9, то хорошими для него будут следующие варианты: триплет `arm-linux-musleabi`, архитектура — `armv5t` и поддержка плавающей запятой — `soft`. ARMv9-процессоры обычно не имеют аппаратных возможностей работы с плавающей запятой.

```
export LFA_FLOAT="[hard, soft или softfp]"
```

Если вы выбрали `hard` или `softfp` для `LFA_FLOAT`, то теперь вам нужно установить, какое оборудование для работы с плавающей запятой используется в целевом процессоре (согласно таблице ниже):

```
export LFA_FPU="одно из значений из таблицы ниже"
```

FPU	FPU	FPU	FPU
`fpa`	`fpe2`	`fpe3`	`maverick`
`vfp`	`vfpv3`	`vfpv3-fp16`	`vfpv3-d16`
`vfpv3-d16-fp16`	`vfpv3xd`	`vfpv3xd-fp16`	`neon`

FPU	FPU	FPU	FPU
`neon-fp16`	`vfpv4`	`vfpv4-d16`	`fpv4-sp-d16`
`neon-vfpv4`			

Установите триплеты для хоста и целевой машины:

```
export LFA_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
export LFA_TGT="триплет для целевой машины"
```

Значение `$LFA_FLOAT`	Триплет
`soft` или `softfp`	`arm-linux-musleabi`
`hard`	`arm-linux-musleabihf`

Немного про `LFA_TGT`.

Переменная окружения `LFA_TGT` хранит в себе значение *триплета* для целевой машины. Триплет принимает тип целевой машины, состоящий из следующих элементов: `<CPU>-<Vendor>-<KERNEL>-<OS>`. Поскольку поле `<Vendor>` часто не имеет значения, многие системы сборки (такие как, например, система сборки на основе `autoconf`, используемая всеми или практически всеми пакетами из LFA) позволяют его опустить.

Проницательный читатель LFA может задаться вопросом, почему мы применяем понятие «триплет» к четырёхкомпонентному имени. Компоненты `<Kernel>` и `<OS>` начинались как единое поле `<System>`. Такая форма применяется для многих ОС, например, `x86_64-unknown-freebsd`. Но две системы могут иметь одно и то же ядро, но быть слишком разными, чтобы использовать для них один и тот же триплет. Например, система Android, работающая на мобильном устройстве, полностью отличается от системы PostmarketOS, хотя обе они имеют одно и то же ядро Linux, запускаются на том же процессоре и, возможно, даже на одном и том же мобильном устройстве. Что и говорить — одна и та же система, имеющая одно и то же ядро, запускающаяся на одном и том же

компьютере, но имеющая разные стандартные библиотеки C (например, первая система использует `glibc`, а вторая — `musl`), не могут иметь одинаковый триплет.

Поэтому поле `<System>` было разделено на поля `<Kernel>` и `<OS>`, чтобы однозначно обозначить различные вариации систем. В нашем примере система Android имеет триплет `aarch64-unknown-linux-android`, а система PostmarketOS — `aarch64-linux-musleabihf`. Ну а слово «триплет» до сих пор осталось в профессиональном лексиконе.

Выберите архитектуру, для которой будете собирать систему:

```
export LFA_ARCH="архитектура"
```

ARCH	ARCH	ARCH	ARCH
armv4t	armv5t	armv5te	armv6
armv6j	armv6k	armv6kz	armv6t2
armv6z	armv6-m	armv7	armv7-a
armv7-r	armv7-m	armv9-a	armv9

Запишите эти переменные в `~/.bashrc`, чтобы не вводить их значения каждый раз после входа от имени пользователя `lfa`:

```
cat >> ~/.bashrc << EOF
export LFA_HOST="$LFA_HOST"
export LFA_TGT="$LFA_TGT"
export LFA_ARCH="$LFA_ARCH"
export LFA_FLOAT="$LFA_FLOAT"
export LFA_FPU="$LFA_FPU"
EOF
```

Создание основных каталогов

Создайте каталог, в котором будет содержаться файлы кросс-компилятора. Для того, чтобы постоянно не указывать путь до него при сборке пакетов, объявите новую переменную окружения `$LFA_CROSS`:

```
export LFA_CROSS=$LFA/tools/$LFA_TGT

mkdir -pv $LFA_CROSS
ln -svf . $LFA_CROSS/usr

echo "export LFA_CROSS=\$LFA/tools/\$LFA_TGT" >> ~/.bashrc
source ~/.bashrc
```

Кроме того, вам необходимо создать директорию, где будет храниться исходный код компонентов:

```
mkdir -v src
```

Проверьте себя

В итоге в домашней папке пользователя `lfa` будет примерно такая структура файлов:

```
/home/lfa
|-- lfa/
|   `-- tools/
|       `-- aarch64-linux-musleabihf/
|           `-- usr/ -> .
`-- src/
```

И содержимое файла `~/.bashrc` после всех записей в него (в зависимости от выбранной вами архитектуры его содержимое может незначительно меняться):

```
set +h
umask 022
unset CFLAGS
LFA=$HOME/lfa
LC_ALL=C
PATH=$LFA/tools/bin:/bin:/usr/bin
export LFA LC_ALL PATH
export LFA_HOST="x86_64-cross-linux-gnu"
export LFA_TGT="aarch64-linux-musleabihf"
export LFA_ARCH="armv8-a"
export LFA_CROSS=$LFA/tools/$LFA_TGT
```

Скачивание пакетов

Перейдите в директорию `src/`, которую вы создали ранее:

```
cd src/
```

Скачайте файлы `wget-list` и `md5sums`, которые будут использованы для скачивания исходного кода компонентов системы:

```
wget https://linux-for-arm.github.io/lfa/2.0/wget-list
wget https://linux-for-arm.github.io/lfa/2.0/md5sums
```

И скачайте системные компоненты:

```
wget --input-file=wget-list --continue
```

Для проверки корректности скачивания пакетов вам нужно воспользоваться файлом `md5sums`:

```
md5sum -c md5sums
```

Сборка кросс-компилятора

В данной главе вы соберёте кросс-компилятор, необходимый для дальнейшей сборки LFA. Подробные сведения о том, зачем это нужно, вы можете получить в дополнительных материалах.

> ⚠ **Внимание**
>
> Перед выполнением инструкций по сборке пакета необходимо распаковать его от имени пользователя `lfa` и перейти в распакованную директорию с исходным кодом пакета (обычно директория имеет то же имя, что и архив с исходниками, но без расширения `.tar.*`) с помощью команды `cd ИМЯ_ДИРЕКТОРИИ`. В инструкциях по сборке предполагается, что используется командная оболочка BASH или совместимая с ней.

Во время компиляции большинства пакетов на экран будут выводиться различные сообщения, в том числе и предупреждения. Это предупреждения как правило об устаревшем использовании синтаксиса языка программирования C. Это не является проблемой, но вызывает предупреждение.

> ⚠ **Внимание**
>
> После установки пакета как в этой, так и в следующих главах, перейдите обратно в директорию `src/` (командой `cd ..` или `cd ../..` если сборка производилась в отдельной директории), а затем удалите каталог, в котором вы собирали этот пакет.

linux-headers

Заголовочные файлы ядра Linux, необходимые для сборки кросс-компилятора

- **Версия:** 6.6.44
- **Домашняя страница:** https://www.kernel.org
- **Время сборки:** 0.5 OBC

> ⚠ **Внимание**
>
> Обратите внимание, что исходный код `linux-headers` содержится в архиве с ядром Linux-6.6.44.

Настройка

Убедитесь, что дерево исходного кода Linux чистое и не содержит лишних файлов:

```
make mrproper
```

Установка

```
make ARCH=arm64 INSTALL_HDR_PATH=$LFA_CROSS headers_install
```

> ⚠ **Внимание**
>
> Обратите внимание на аргумент `ARCH=arm64`. Для 32-битных процессоров нужно заменить этот аргумент на `ARCH=arm`.
>
> Например, если ваш процессор имеет 64-битную архитектуру (ARMv8 или ARMv8.1), то оставьте этот аргумент без изменений. Однако если у вас иная 32-битная архитектура, то замените `ARCH=arm64` на `ARCH=arm`.

Если во время установки заголовков ядра (в частности, при исполнении второй команды `headers_install`) у вас возникли ошибки, проверьте, установлена ли в системе программа `rsync`.

Значения новых параметров:

`ARCH=arm64` — указывает `make` устанавливать заголовки для архитектуры `arm64`.

`INSTALL_HDR_PATH=$LFA_CROSS` — указывает *префикс*, в который будут установлены заголовки.

Содержимое пакета

- **Установленные заголовки:** `$LFA_CROSS/include/{asm,asm-generic,drm,linux,misc,mtd,rdma,scsi,sound,video,xen}/*.h`

Описание компонентов

- `$LFA_CROSS/include/asm/*.h` — заголовки Linux API ASM.
- `$LFA_CROSS/include/asm-generic/*.h` — общие заголовки Linux API ASM.
- `$LFA_CROSS/include/drm/*.h` — заголовки Linux DRM.
- `$LFA_CROSS/include/linux/*.h` — заголовки Linux API.
- `$LFA_CROSS/include/misc/*.h` — различные заголовки Linux API.
- `$LFA_CROSS/include/mtd/*.h` — заголовки Linux API MTD.
- `$LFA_CROSS/include/rdma/*.h` — заголовки Linux API RDMA.
- `$LFA_CROSS/include/scsi/*.h` — заголовки Linux API SCSI.
- `$LFA_CROSS/include/sound/*.h` — заголовки Linux API для работы со звуком.
- `$LFA_CROSS/include/video/*.h` — заголовки Linux API для работы с видео.
- `$LFA_CROSS/include/xen/*.h` — заголовки Linux API XEN.

Смотрите также:

- Сборка ПО из исходного кода;
- Кросс-компилятор.

---

binutils

Этот пакет содержит компоновщик, ассемблер и другие утилиты для работы с объектными файлами.

- **Версия:** 2.43
- **Домашняя страница:** https://www.gnu.org/software/binutils
- **Время сборки:** 1 ОВС

Настройка

Сборка пакета `binutils` должна происходить в отдельном каталоге. Создайте его:

```
mkdir -v build
cd build
```

Запустите скрипт `configure` для генерации предназначенных для сборки файлов `Makefile`:

```
../configure --prefix=$LFA/tools \
  --target=$LFA_TGT \
  --with-sysroot=$LFA_CROSS \
  --disable-nls \
  --enable-gprofng=no \
  --disable-werror \
  --disable-multilib
```

Значения новых параметров:

`--prefix=$LFA/tools` — указывает скрипту `configure` подготовиться к установке пакета в директорию `$LFA/tools`.

`--target=$LFA_TGT` — создаёт кросс-архитектурный исполняемый файл, который запускается на x86_64-системе, но создаёт файлы для `$LFA_TGT` -архитектуры.

`--with-sysroot=$LFA_CROSS` — сообщает `configure`, что `$LFA_CROSS` будет корнем кросс-компилятора.

`--disable-nls` — отключает сборку пакета с поддержкой интернационализации и локализации. В кросс-компиляторе это не нужно.

`--enable-gprofng=no` — отключает сборку `gprofng`, который не нужен в кросс-компиляторе.

`--disable-werror` — отключает остановку сборку при возникновении предупреждений.

`--disable-multilib` — отключает сборку multilib.

Сборка

```
make configure-host
make
```

Значения новых параметров:

`make configure-host` — проверяет окружение хоста и убеждается, что все необходимые инструменты доступны для компиляции `binutils`.

Установка

```
make install
```

Содержимое пакета

- **Установленные программы:** `addr2line`, `ar`, `as`, `c+filt`, `elfedit`, `gprof`, `ld`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings`, `strip`.
- **Установленные библиотеки:** `libibery.a`, `libbbfd.{a,so}`, `libopcodes.{a,so}`

Описание компонентов

- **Программы:**
  - `addr2line` — транслирует адреса программ в имена файлов и номера строк. Если задан адрес и имя исполняемого файла, он использует отладочную информацию в нём, чтобы определить, какой исходный файл или номер строки связаны с этим адресом.
  - `ar` — создаёт, изменяет и распаковывает ar-архивы.
  - `as` — GNU-ассемблер, который используется, в частности, в `gcc`.
  - `c++filt` — используется компоновщиком для "распутывания" символов C++ и Java и предотвращения столкновения перегруженных функций.
  - `elfedit` — получает и изменяет метаданные ELF-файлов.
  - `gprof` — отображение данных профиля графика вызовов.
  - `ld` — компоновщик, который объединяет несколько объектных и архивных файлов в один файл, перемещая их данные и связывая символьные ссылки.
  - `nm` — перечисляет символы, встречающиеся в данном объектном файле.
  - `objcopy` — копирует содержимое одного объектного файла в другой.
  - `objdump` — отображает информацию о данном объектном файле.
  - `ranlib` — генерирует индекс содержимого архива и сохраняет его в архиве.

    - `readelf` — отображает информацию об ELF-файле.
    - `size` - перечисляет размеры секций ELF-файла и размер для заданных файлов объектов.
    - `strings` — выводит для каждого заданного файла последовательности печатаемых символов, длина которых не меньше указанной (по умолчанию - четыре); для объектных файлов по умолчанию выводятся только строки из секций инициализации и загрузки, а для других типов файлов сканируется весь файл.
    - `strip` — удаляет символы из объектных файлов.
- **Библиотеки:**
    - `libiberty` — содержит функции, используемые различными программами GNU, включая `getopt`, `obstack`, `strerror`, `strtoul`.
    - `libbfd` — библиотека дескрипторов двоичных файлов.
    - `libopcodes` — библиотека для работы с опкодами - "читабельными текстовыми" версиями инструкций для процессора. Используется, например, в `objdump`.

gcc (проход 1)

Набор компиляторов GNU GCC.

- **Версия:** 13.2.0
- **Домашняя страница:** https://gcc.gnu.org
- **Время сборки:** 8 ОВС

> ⚠ **Внимание**
>
> Сейчас нам нужно собрать GCC со статической библиотекой `libgcc` и без поддержки многопоточности. Этот первый проход сборки делается главным образом для того, чтобы мы могли собрать с помощью этого компилятора стандартную библиотеку C (`musl`).

Подготовка

GCC требует, чтобы пакеты GMP, MPFR и MPC либо присутствовали на хосте, либо представлены в виде исходных текстов в дереве исходного кода GCC. Распакуйте их:

```
tar -xf ../gmp-6.3.0.tar.xz
tar -xf ../mpc-1.3.1.tar.gz
tar -xf ../mpfr-4.2.1.tar.xz

mv -v gmp-6.3.0 gmp
mv -v mpc-1.3.1 mpc
mv -v mpfr-4.2.1 mpfr
```

Настройка

Сборка пакета `gcc` должна происходить в отдельном каталоге. Создайте его:

```
mkdir -v build
cd build
```

Запустите скрипт `configure`:

> ⚠ **Внимание**
>
> Далее и на протяжении всего руководства, если вы собираете систему для AArch64, то **не используйте** переменные окружения `$LFA_FLOAT` и `$LFA_FPU`, а также пропускайте при вводе команд строки, содержащие эти переменные окружения. Например, если вы собираете систему для AArch64, то скрипту `configure` не следует передавать эти аргументы:
>
> ```
> --with-float=$LFA_FLOAT \
> --with-fpu=$LFA_FPU
> ```

```
../configure --prefix=$LFA/tools \
  --build=$LFA_HOST \
  --host=$LFA_HOST \
  --target=$LFA_TGT \
  --with-sysroot=$LFA_CROSS \
  --disable-nls \
  --disable-shared \
  --without-headers \
  --with-newlib \
  --enable-default-pie \
  --enable-default-ssp \
  --disable-decimal-float \
  --disable-libgomp \
  --disable-libmudflap \
  --disable-libssp \
  --disable-libvtv \
  --disable-libstdcxx \
  --disable-libatomic \
  --disable-libquadmath \
  --disable-threads \
  --enable-languages=c \
  --disable-multilib \
  --with-arch=$LFA_ARCH \
  --with-float=$LFA_FLOAT \
  --with-fpu=$LFA_FPU
```

Значения новых параметров:

`--host=$LFA_HOST` — указывает `configure` триплет машины, на которой будет выполняться GCC при кросс-компиляции. `$LFA_HOST` содержит название архитектуры хоста, на которой будем производить кросс-компиляцию для архитектуры `$LFA_TGT` .

`--disable-shared` — этот переключатель заставляет GCC связывать свои внутренние библиотеки статически.

`--without-headers` — указывает `configure` не использовать никаких заголовков из библиотек C. Это необходимо, поскольку мы ещё не собрали библиотеку C и чтобы предотвратить влияние окружения хоста.

`--with-newlib` — собрать `libgcc` без использования библиотек C.

`--enable-default-pie`, `--enable-default-ssp` — позволяют GCC по умолчанию компилировать программы с некоторыми средствами усиления безопасности.

`--disable-decimal-float` — отключить поддеркжу десятичной плавающей запятой (IEEE 754-2008). Нам это пока не нужно.

`--disable-libgomp` — не собирать библиотеки времени выполнения GOMP.

`--disable-libmudflap` — не собирать библиотеку `libmudflap` (библиотека, которая может быть использована для проверки правильности использования указателей).

`--disable-libssp` — не собирать библиотеки времени выполнения для обнаружения разбиения стека.

`--disable-libvtv` — не собирать `libvtv`.

`--disable-libstdcxx` — не собирать стандартную библиотеку C++.

`--disable-libatomic` — не собирать атомарные операции.

`--disable-libquadmath` — не собирать `libquadmath`.

`--disable-threads` — не искать многопоточные заголовочные файлы, поскольку для этой архитектуры (`$LFA_TGT`) их ещё нет. GCC сможет найти их после сборки стандартной библиотеки C.

`--enable-languages=c` — указывает `configure` собирать компилятор языка C.

`--disable-multilib` — поддержка multilib нам не нужна.

`--with-arch=$LFA_ARCH` — устанавливает выбранную ранее архитектуру ARM.

`--with-float=$LFA_FLOAT` — устанавливает ранее выбранный режим работы с плавающей запятой. **Не требуется, если вы собираете LFA для AArch64!**

`--with-fpu=$LFA_FPU` — устанавливает тип аппаратной плавающей запятой. Если `$LFA_FPU="soft"`, это значение игнорируется. **Не требуется, если вы собираете LFA для AArch64!**

Сборка

```
make all-gcc all-target-libgcc
```

Установка

```
make install-gcc install-target-libgcc
```

Содержимое пакета

На данный момент знать содержимое пакета GCC вам не требуется, поскольку сейчас мы собрали лишь небольшую его часть, предназначенную только для компиляции стандартной библиотеки C (`musl`). Информация о содержимом пакета GCC содержится на втором проходе сборки GCC.

musl

Минималистичная стандартная библиотека языка C.

- **Версия:** 1.2.5
- **Домашняя страница:** https://musl.libc.org
- **Время сборки:** 0.2 ОВС

Настройка

```
./configure CROSS_COMPILE=$LFA_TGT- \
  --prefix=/ \
  --target=$LFA_TGT
```

Сборка

```
make
```

Установка

```
make DESTDIR=$LFA_CROSS install
```

Содержимое пакета

- **Установленные программы:** `ld-musl`
- **Установленные библиотеки:** `libc.so.0`, `libcrypt.so.0`, `libdl.so.0`, `libm.so.0`, `libpthread.so.0`, `librt.so.0`

Описание компонентов

- **Программы:**
  - `ld-musl` — динамический компоновщик/загрузчик musl.
- **Библиотеки:**
  - `libc` — библиотека языка C.
  - `libcrypt` — криптографическая библиотека.
  - `libdl` — библиотека для динамического компоновщика/зарузчика.
  - `libm` — математическая библиотека.
  - `libpthread` — библиотека потоков POSIX.
  - `librt` — библиотека часов и таймера.

gcc (проход 2)

Набор компиляторов GNU GCC.

- **Версия:** 13.2.0
- **Домашняя страница:** https://gcc.gnu.org
- **Время сборки:** 8 ОВС

> ⚠ **Внимание**
>
> Сейчас мы собираем полноценную версию компилятора GCC для сборки остальной системы, используя уже готовую стандартную библиотеку C.

Подготовка

```
tar -xf ../gmp-6.3.0.tar.xz
tar -xf ../mpc-1.3.1.tar.gz
tar -xf ../mpfr-4.2.1.tar.xz

mv -v gmp-6.3.0 gmp
mv -v mpc-1.3.1 mpc
mv -v mpfr-4.2.1 mpfr
```

Настройка

> ⚠ **Внимание**
>
> Если вы собираете систему для AArch64, то **не используйте** переменные окружения `$LFA_FLOAT` и `$LFA_FPU`, а также пропускайте при вводе команд строки, содержащие эти переменные окружения. Например, если вы собираете систему для AArch64, то скрипту `configure` не следует передавать эти аргументы:
>
> ```
>    --with-float=$LFA_FLOAT \
>    --with-fpu=$LFA_FPU
> ```

```
mkdir -v build
cd build

../configure --prefix=$LFA/tools \
  --build=$LFA_HOST \
  --host=$LFA_HOST \
  --target=$LFA_TGT \
  --with-sysroot=$LFA_CROSS \
  --disable-nls \
  --enable-languages=c \
  --enable-c99 \
  --enable-long-long \
  --disable-libmudflap \
  --disable-multilib \
  --with-arch=$LFA_ARCH \
  --with-float=$LFA_FLOAT \
  --with-fpu=$LFA_FPU
```

Сборка

```
make
```

Установка

```
make install
```

Содержимое пакета

- **Установленные программы:** `gcc`, `gcov`
- **Установленные библиотеки:** `libgcc.a`, `libgcc_eh.a`, `libgcc_s.so`

Описание компонентов

- **Программы:**
  - `gcc` — компилятор языка C.
  - `gcov` — инструмент для тестирования покрытия, используется для анализа программ, чтобы определить, где оптимизация даст наибольший эффект.

Проверка кросс-компилятора

На данном этапе необходимо убедиться, что установленные ранее пакеты работают правильно. Внимательно изучите результаты вывода команд, и проверьте, что они строго соответствуют результатам вывода, приведенным ниже. Если есть несоответствия, значит инструкции на предыдущих этапах были выполнены некорректно.

Проверьте, используется ли правильный загрузчик программ:

```
echo "int main() {}" > main.c

$LFA_TGT-gcc -xc main.c
readelf -l a.out | grep "program interpreter"
```

Вывод должен быть таким:

```
      [Requesting program interpreter: /lib/ld-musl-aarch64.so.1]
```

> Если вы собирали систему для другой архитектуры семейства ARM, то различие будет в подстроке `ld-musl-aarch64.so.1`: вместо `aarch64` должно быть имя той архитектуры, для которой предназначен кросс-компилятор.

Если вывод не такой, как показано выше, или его вообще нет, значит, что что-то пошло не так. Исследуйте и проследите все шаги сборки всех пакетов до этого этапа, чтобы найти причину проблемы и устранить её. Прежде чем продолжать сборку LFA, необходимо решить эту проблему.

Удалите тестовый файл:

```
rm -v a.out
```

Очистка и сохранение

Удаление лишних файлов

Сборка кросс-компилятора завершена. Теперь нужно очистить директорию с исходным кодом (`~/src`) от лишних подкаталогов, образовавшихся во время сборки. Выполните команду:

```
for f in *; do
  if [ -d $f ]; then
    rm -rf $f
  fi
done
```

Эта команда удалит все директории в `src/`, оставив только архивы с исходным кодом ПО.

Значения новых параметров:

`for f in *` — символ `*` в данном случае означает «все файлы в текущей директории». Мы проходимся по содержимому `src/` для удаления лишних файлов.

`if [ -d $f ]` — выполняем проверку того, что файл `$f` — это директория. Поскольку мы удаляем распакованные из архивов директории, то нам нужно удалить только их, оставив архивы с исходным кодом не тронутыми.

`rm -rf $f` — если `$f` — директория, то удалить её.

> ⚠ **Внимание**
>
> Не удаляйте саму директорию `$LFA/tools`. Кросс-компилятор будет удалён только после окончания сборки базовой системы. В случае, если после сборки базовой системы вы захотите собрать дополнительное ПО, которое не описано в этом руководстве, то сборка будет также производиться посредством этого кросс-компилятора, поэтому не удаляйте его до тех пор, пока не окончите сборку всех необходимых вам программ.

Сохранение

Если вы собираетесь использовать этот кросс-компилятор для последующих сборок системы LFA, то рекомендуем вам сделать его резервную копию:

```
cd $LFA
tar -cJpf $HOME/lfa-cross-compiler-2.0.tar.xz .
```

Этой командой вы создадите архив `/home/lfa/lfa-cross-compiler-2.0.tar.xz` с содержимым директории `/home/lfa/lfa`, которая содержит в подкаталоге `tools/` кросс-компилятор. *В архив не будет добавлен исходный код компонентов системы, поскольку он находится в другой директории (`/home/lfa/src/`).*

Объявление дополнительных переменных

Теперь вам нужно объявить переменную `$LFA_SYS`, которая будет содержать путь до директории, в которой будет находиться собираемая базовая система LFA:

```
export LFA_SYS=$LFA/baseOS
echo "export LFA_SYS=\$LFA/baseOS" >> ~/.bashrc
```

Объявите переменные, содержащие пути до собранных компилятора, компоновщика и иных инструментов:

```
cat >> ~/.bashrc << EOF
export CC="$LFA_TGT-gcc --sysroot=$LFA_SYS"
export CXX="$LFA_TGT-g++ --sysroot=$LFA_SYS"
export AR="$LFA_TGT-ar"
export AS="$LFA_TGT-as"
export LD="$LFA_TGT-ld --sysroot=$LFA_SYS"
export RANLIB="$LFA_TGT-ranlib"
export READELF="$LFA_TGT-readelf"
export STRIP="$LFA_TGT-strip"
EOF
```

И примените изменения:

```
source ~/.bashrc
```

Сборка базовой системы

В этой главе мы начинаем всерьёз собирать систему LFA, используя кросс-компилятор из предыдущей главы. Порядок установки пакетов в этой главе должен строго соблюдаться, чтобы ни одна программа случайно не приобрела путь, ссылающийся на кросс-компилятор. По этой же причине не собирайте пакеты параллельно друг с другом, так как сборка сразу нескольких пакетов за раз хоть и уменьшит общее время сборки LFA, но приведёт к неправильной компиляции и, как следствие, неработоспособности базовой ОС.

Если вы хотите ускорить сборку системы, то лучше использовать многопоточную сборку пакетов. Для этого добавьте к команде `make` ключ `-jN`, где `N` - число потоков вашего процессора. Например:

```
make -j4
```

Кроме того, чтобы каждый раз не указывать `-jN`, вы можете объявить переменную окружения `MAKEFLAGS`, содержащую эту опцию:

```
export MAKEFLAGS="-jN" # либо MAKEFLAGS="-j$(nproc)"
# nproc возвращает число логических процессоров ПК
```

> ⚠ **Внимание**
>
> Никогда не передавайте `make` опцию `-j` без числа и не задавайте такой параметр в переменной `MAKEFLAGS`. Это позволит `make` порождать бесконечные задания на сборку и вызовет проблемы со стабильностью системы, вплоть до её полного зависания.

Для применения внесённых в `~/.bashrc` изменений выполните команду:

```
source ~/.bash_profile
```

Создание файлов и каталогов

Директория базовой ОС

Создайте директорию, в которой будут находиться файлы собранной базовой ОС:

```
mkdir -pv $LFA_SYS
```

Стандартные системные каталоги базовой ОС

Теперь пришло время создать некоторую структуру в целевой файловой системе базовой ОС. Создайте стандартное дерево каталогов, выполнив следующие команды:

```
mkdir -pv $LFA_SYS/{bin,boot,dev,etc,home}
mkdir -pv $LFA_SYS/lib/{firmware,modules}
mkdir -pv $LFA_SYS/{mnt,opt,proc,sbin,srv,sys}
mkdir -pv $LFA_SYS/var/{cache,lib,local,lock,log,opt,run,spool}
mkdir -pv $LFA_SYS/usr/{,local/}{bin,include,lib,sbin,share,src}

install -dv -m 0750 $LFA_SYS/root
install -dv -m 1777 $LFA_SYS/{var/,}tmp

if [ $LFA_TGT == "aarch64-linux-musleabihf" ]; then
    ln -svf lib $LFA_SYS/lib64
    ln -svf lib $LFA_SYS/usr/lib64
fi
```

Проверьте себя

После исполнения данных команд в директории `$LFA_SYS` должна быть такая структура:

```
/home/lfa/lfa/baseOS
|-- bin
|-- boot
|-- dev
|-- etc
|-- home
|-- lib
|   |-- firmware
|   `-- modules
|-- mnt
|-- opt
|-- proc
|-- root
|-- sbin
|-- srv
|-- sys
|-- tmp
|-- usr
|   |-- bin
|   |-- include
|   |-- lib
|   |-- local
|   |   |-- bin
|   |   |-- include
|   |   |-- lib
|   |   |-- sbin
|   |   |-- share
|   |   `-- src
|   |-- sbin
|   |-- share
|   `-- src
`-- var
    |-- cache
    |-- lib
    |-- local
    |-- lock
    |-- log
    |-- opt
    |-- run
    |-- spool
    `-- tmp
```

Создание ряда системных файлов

Обычно системы Linux хранят список смонтированных файловых систем в `/etc/mtab`. С учётом того, как устроена наша система, в качестве `/etc/mtab` в ней будет выступать ссылка на `/proc/mounts`:

```
ln -svf ../proc/mounts $LFA_SYS/etc/mtab
```

Создание пользователей и групп

Для того, чтобы пользователь `root` мог войти в систему и чтобы имя `root` было распознано, создайте в файлах `/etc/passwd` и `/etc/group` соответствующие записи:

```
cat > $LFA_SYS/etc/passwd << "EOF"
root::0:0:root:/root:/bin/ash
EOF
```

Вы можете захотеть создать следующих пользователей:

`bin:x:1:1:bin:/bin:/bin/false` — может быть полезен для совместимости с устаревшими приложениями;

`daemon:x:2:6:daemon:/sbin:/bin/false` — часто рекомендуется использовать непривилегированного пользователя для запуска демонов, чтобы ограничить их доступ к системе;

`adm:x:3:16:adm:/var/adm:/bin/false` — используется программами, которые выполняют административные задачи;

`lp:x:10:9:lp:/var/spool/lp:/bin/false` — используется в программах для печати;

`mail:x:30:30:mail:/var/mail:/bin/false` — используется для почтовых программ;

`news:x:31:31:news:/var/spool/news:/bin/false` — используется в программах для получения новостей;

`uucp:x:32:32:uucp:/var/spool/uucp:/bin/false` — часто используется для копирования файлов Unix-to-Unix с одного сервера на другой;

`operator:x:50:0:operator:/root:/bin/ash` — может быть использоан для предоставления операторам доступа к системе;

`postmaster:x:51:30:postmaster:/var/spool/mail:/bin/false` — обычно используется как учетная запись, которая получает всю информацию о проблемах с почтовым сервером;

`nobody:x:65534:65534:nobody:/:/bin/false` — используется в NFS.

Создайте файл, в котором будут указаны группы пользователей:

```
cat > $LFA_SYS/etc/group << "EOF"
root:x:0:
bin:x:1:
sys:x:2:
kmem:x:3:
tty:x:4:
tape:x:5:
daemon:x:6:
floppy:x:7:
disk:x:8:
lp:x:9:
dialout:x:10:
audio:x:11:
video:x:12:
utmp:x:13:
usb:x:14:
cdrom:x:15:
EOF
```

Вы можете захотеть создать следующие группы:

- `adm:x:16:root,adm,daemon`

- `console:x:17` — пользователи этой группы имеют доступ к консоли
- `mail:x:30:mail`
- `news:x:31:news`
- `uucp:x:32:uucp`
- `users:x:100:` — используется в программе `shadow` по умолчанию для новых пользователей
- `nogroup:x:65533` — используется некоторыми программами, которым **специально** не требуется группа
- `nobody:x:65534`

Логи

Программы `login`, `agetty` и `init` используют файл `lastlog` для записи информации о том, кто и когда вошёл в систему. Однако они не будут ничего туда записывать, если этого файла нет. Создайте файл `lastlog` и дайте ему соответствующие разрешения:

```
touch $LFA_SYS/var/log/lastlog
chmod -v 664 $LFA_SYS/var/log/lastlog
```

libgcc

При компиляции динамических библиотек с помощью GCC требуется, чтобы `libgcc` могла быть загружена во время выполнения программы. Поэтому нам нужно скопировать библиотеку `libgcc`, которая ранее была собрана для кросс-компилятора.

- **Версия:** 13.2.0
- **Домашняя страница:** https://gcc.gnu.org
- **Время сборки:** 0.01 ОВС

Подготовка

Объявите переменную окружения `LGCC_LIB`, которая будет содержать имя директории `lib{,64}`, где содержится необходимая нам библиотека (в зависимости от архитектуры название этой директории различается):

```
if [ $LFA_TGT == "aarch64-linux-musleabihf" ]; then
  LGCC_LIB="lib64"
else
  LGCC_LIB="lib"
fi
```

Значения новых параметров:

`if [ $LFA_TGT == "aarch64-linux-musleabihf" ]; then ...` — если вы собираете систему для 64-битной архитектуры, то нужная библиотека содержится в каталоге `$LFA_CROSS/lib64`. А для 32-битных архитектур семейства ARM нужная библиотека содержится в `$LFA_CROSS/lib`. В зависимости от целевой архитектуры мы вибираем, откуда копировать `libgcc_s.so.1` и куда.

Установка

Скопируйте библиотеку в директорию собираемой ОС:

```
cp -v $LFA_CROSS/$LGCC_LIB/libgcc_s.so.1 $LFA_SYS/$LGCC_LIB
```

Удалите из установленной библиотеки лишние для вас отладочные символы:

```
$STRIP $LFA_SYS/$LGCC_LIB/libgcc_s.so.1
```

Переменная `LGCC_LIB` нам больше не нужна, удалите её:

```
unset LGCC_LIB
```

Содержимое пакета

- **Установленные библиотеки:** `libgcc_s.so.1`

Описание компонентов

- `libgcc_s.so.1` — при компиляции динамически линкуемых программ с помощью GCC требуется, чтобы во время выполнения такой программы была загружена библиотека `libgcc_s.so.1` из состава GCC.

musl

Минималистичная стандартная библиотека языка C.

- **Версия:** 1.2.5
- **Домашняя страница:** https://musl.libc.org
- **Время сборки:** 0.2 ОВС

Настройка

```
./configure CROSS_COMPILE=$LFA_TGT- \
  --prefix=/ \
  --disable-static \
  --target=$LFA_TGT
```

Сборка

```
make
```

Установка

```
make DESTDIR=$LFA_SYS install-libs
```

Содержимое пакета

- **Установленные программы:** `ld-musl`
- **Установленные библиотеки:** `libc.so.0`, `libcrypt.so.0`, `libdl.so.0`, `libm.so.0`, `libpthread.so.0`, `librt.so.0`

Описание компонентов

- **Программы:**
  - `ld-musl` — динамический компоновщик/загрузчик musl.
- **Библиотеки:**
  - `libc` — библиотека языка C.
  - `libcrypt` — криптографическая библиотека.
  - `libdl` — библиотека для динамического компоновщика/зарузчика.
  - `libm` — математическая библиотека.
  - `libpthread` — библиотека потоков POSIX.
  - `librt` — библиотека часов и таймера.

busybox

Объединяет крошечные версии многих распространённых утилит UNIX в один небольшой двоичный файл (1-2 Мбайт). Он заменяет большинство утилит, которые обычно находятся в GNU Coreutils, GNU Findutils и т.д.

- **Версия:** 1.36.1
- **Домашняя страница:** https://www.busybox.net
- **Время сборки:** 0.3 ОВС

Настройка

Процесс настройки пакета `busybox` схож с процессом настройки ядра Linux. Параметры сборки записываются в файл `.config`. Можно сконфигурировать сборку в псевдографическом режиме (`make menuconfig`), а можно использовать стандартный конфиг (`make defconfig`). Вы можете сохранить файл `.config` для того, чтобы в будущем (в случае пересборки этой версии BusyBox или в случае сборки новой версии этого пакета) не конфигурировать пакет вновь.

Убедитесь, что дерево исходного кода BusyBox чистое и не содержит лишних файлов:

```
make mrproper
```

Далее требуется настроить пакет BusyBox, выбрав те опции, которые вам нужны, и убрать то, что вам не требуется. В зависимости от числа выбранных опций зависит в том числе и размер вашей системы,

однако BusyBox — вещь довольно минималистичная, и на размер системы влияет больше ядро Linux, его модули и файлы Device Tree.

```
make ARCH=arm64 menuconfig
```

Процесс настройки

В разделах `Archival Utilities`, `Coreutils`, `Console Utilities`, `Debian Utilities`, `klibc-utils`, `Editors` и т.д. выберите сборку программ и утилит в зависимости от того, в каких сценариях должна использоваться ваша система LFA, а также в зависимости от её конечного размера. Конечно, на её размер куда больше будут влиять ядро и файлы, необходимые для загрузки системы, но инструкции по уменьшению размера, к примеру, ядра Linux, ищите сами на просторах интернета.

Система инициализации

Поскольку чуть позже мы установим в LFA загрузочные скрипты, нам требуется система инициализации, которая эти скрипты будет исполнять. Для этого компилируйте BusyBox с поддержкой `init`, `halt`, `poweroff`, `reboot`. Кроме того, вам нужны программы `getty` и `login`.

```
Init Utilities  --->
  <*> halt      [CONFIG_HALT]
  <*> poweroff  [CONFIG_POWEROFF]
  <*> reboot    [CONFIG_REBOOT]
  <*> init      [CONFIG_INIT]
Login/Password Management Utilities  --->
  <*> getty     [CONFIG_GETTY]
  <*> login     [CONFIG_LOGIN]
```

mdev

В данном руководстве рекомендуется использовать `mdev` для динамического управления устройствами в директории `/dev`. Включите следующие опции:

```
Linux System Utilities  --->
  <*> mdev                          [CONFIG_MDEV]
    <*> Support /etc/mdev.conf      [CONFIG_FEATURE_MDEV_CONF]
    <*> Support loading of firmware
[CONFIG_FEATURE_MDEV_LOAD_FIRMWARE]
    <*> Support daemon mode         [CONFIG_FEATURE_MDEV_DAEMON]
```

Отключение опций

После конфигурирования вам нужно отлючить ряд возможностей, с которыми мы не смогли бы корректно собрать этот пакет.

Во-первых, отключите сборку `ifplugd` и `inetd`, поскольку их сборка вместе с `musl` имеет проблемы:

```
sed -i 's/\(CONFIG_\)\(.*\)\(INETD\)\(.*\)=y/# \1\2\3\4 is not
set/g' .config
sed -i 's/\(CONFIG_IFPLUGD\)=y/# \1 is not set/' .config
```

Отключите использование `utmp` / `wtmp`, поскольку `musl` их не поддерживает:

```
sed -i 's/\(CONFIG_FEATURE_WTMP\)=y/# \1 is not set/' .config
sed -i 's/\(CONFIG_FEATURE_UTMP\)=y/# \1 is not set/' .config
```

Отключите использование `ipsvd` для TCP и UDP, поскольку у него есть проблемы сборки вместе с `musl` (аналогично `inetd`):

```
sed -i 's/\(CONFIG_UDPSVD\)=y/# \1 is not set/' .config
sed -i 's/\(CONFIG_TCPSVD\)=y/# \1 is not set/' .config
```

Сборка компонента `tc` на ядрах Linux >= 6.8 вызывает ошибку. Отключите сборку `tc` если он вам не нужен:

```
sed -i 's/\(CONFIG_TC\)=y/# \1 is not set/' .config
```

> **Если вам нужен `tc`**
>
> - Примените **патч**;
> - **Обсуждение на** *lists.busybox.net*;

Сборка BusyBox с поддержкой PAM (Pluggable Authentication Modules) не требуется, поскольку PAM'а в LFA нет. Убедитесь, что собираете BusyBox без PAM:

```
sed -i 's/\(CONFIG_PAM\)=y/# \1 is not set/' .config
```

Обычно в системах подобных LFA не требуются пакетные менеджеры типа того же `dpkg`. К тому же, в BusyBox предоставляется достаточно «обрезанная» версия dpkg с некоторыми ограничениями. Да и подобных LFS руководствах (в том числе и в LFA) не рекомендуется использовать подобные пакетные менеджеры во избежание проблем и поломок системы. Если вам не нужен `dpkg`, отключите его сборку, чем освободите около 73 Кб памяти:

```
sed -i 's/\(CONFIG_DPKG\)=y/# \1 is not set/' .config
sed -i 's/\(CONFIG_DPKG_DEB\)=y/# \1 is not set/' .config
```

Тоже самое сделайте и с версией пакетного менеджера `rpm`:

```
sed -i 's/\(CONFIG_RPM\)=y/# \1 is not set/' .config
sed -i 's/\(CONFIG_RPM2CPIO\)=y/# \1 is not set/' .config
```

Сборка

```
make ARCH=arm64 CROSS_COMPILE=$LFA_TGT-
```

Установка

Установите пакет:

```
make ARCH=arm64 CROSS_COMPILE=$LFA_TGT- \
  CONFIG_PREFIX=$LFA_SYS install
```

> Заметьте, что BusyBox содержит множество программ, но все они объединены в один файл. Однако для удобства (чтобы, например, вводить не `busybox mv file1 file2`, а просто `mv file1 file2` как в обычных системах) в каталогах `$LFA_SYS/bin` и `$LFA_SYS/sbin` создаются ссылки на `busybox` с именами программ, которые содержит этот пакет.

Если вы собираетесь собирать ядро с помощью модулей, вам нужно убедиться, что `depmod.pl` доступен для выполнения на вашем хосте:

```
cp -v examples/depmod.pl $LFA/tools/bin
chmod -v 755 $LFA/tools/bin/depmod.pl
```

Содержимое пакета

- **Установленные программы:** `[`, `[[`, `arch`, `ascii`, `ash`, `awk`, `base32`, `base64`, `basename`, `bc`, `bunzip2`, `busybox` и другие[1]

Описание компонентов

- **Программы:**
  - `busybox` — реализация стандартных UNIX утилит.
  - все остальные — ссылки на `busybox`.

---

[1] Набор установленного ПО зависит от того, какие настройки вы указывали при конфигурировании пакета.

iana-etc

Данные для сетевых служб и сервисов. Необходим для обеспечения надлежащих сетевых возможностей.

- **Версия:** 20241122
- **Домашняя страница:** https://www.iana.org/protocols
- **Время сборки:** 0.01 ОВС

Установка

Скопируйте файлы `services` и `protocols` в `$LFA_SYS/etc` :

```
cp -v services protocols $LFA_SYS/etc
```

Содержимое пакета

- **Установленные файлы:** `/etc/protocols` и `/etc/services`

Описание компонентов

- `/etc/protocols` — описывает различные интернет-протоколы DARPA, которые доступны из подсистемы TCP/IP.
- `/etc/services` — обеспечивает сопоставление между дружественными текстовыми именами интернет-сервисов и соответствующими им номерами портов и типами протоколов.

wireless-tools

Набор инструментов для работы с Wireless Extensions (WE — API ядра Linux, позволяющий драйверу передавать в пользовательское пространство конфигурацию и статистику, характерные для беспроводных локальных сетей).

- **Версия:** 29
- **Домашняя страница:** https://hewlettpackard.github.io/wireless-tools/
- **Время сборки:** 0.1 OBC
- **Необходимые патчи:**
  - https://www.linuxfromscratch.org/patches/blfs/svn/wireless_tools-29-fix_iwlist_scanning-1.patch

---

Настройка ядра

Чтобы использовать Wireless Tools, ядро должно иметь соответствующие драйверы и прочие компоненты. Когда вы будете собирать ядро, не забудьте включить следующие опции:

```
[*] Networking support --->                                  [NET]
  [*] Wireless --->                                          [WIRELESS]
    <*/M> cfg80211 - wireless configuration API              [CFG80211]
    [*]   cfg80211 wireless extensions compatibility
[CFG80211_WEXT]
```

Настройка

Исправьте `Makefile`, чтобы можно было собрать пакет, используя наш кросс-компилятор:

```
sed -i s/gcc/\$\{LFA\_TGT\}\-gcc/g Makefile
sed -i s/\ ar/\ \$\{LFA\_TGT\}\-ar/g Makefile
sed -i s/ranlib/\$\{LFA\_TGT\}\-ranlib/g Makefile
```

Сборка

Существуют опции, которые можно передать в `make` и `make install`, чтобы уменьшить размер и функциональность этого пакета. В файле `INSTALL` вы можете узнать дополнительную информацию об этом.

```
make PREFIX=$LFA_SYS
```

Установка

```
make PREFIX=$LFA_SYS install
```

Содержимое пакета

- **Установленные программы:** `ifrename`, `iwconfig`, `iwevent`, `iwgetid`, `iwlist`, `iwpriv` и `iwspy`
- **Установленные библиотеки:** `libiw.so`

Описание компонентов

- **Программы:**
  - `ifrename` — переименовывает сетевые интерфейсы на основе различных статистичесих критериев.
  - `iwconfig` — настраивает беспроводную сеть.
  - `iwevent` — отображает события беспроводной сети, генерируемые драйверами и изменениями настроек.
  - `iwgetid` — собщает ESSID, NWID или адрес точки доступа беспроводной сети.
  - `iwlist` — получает подробную информацию о беспроводной сети.
  - `iwpriv` — настраивает дополнительные (частные) параметры интерфейса беспроводной сети.
  - `iwspy` — получает статистику беспроводной связи с определённого узла.
- **Библитеки:**
  - `libiw.so` — функции, необходимые для работы программ из этого пакета и API для других программ.

Настройка базовой системы

В данной части пойдёт речь о настройке базовой системы. Мы только собрали ряд нужных программ, но на этом всё. Однако для корректной работы только что собранной системы (как минимум для обеспечения её загрузки) нужно создать ряд конфигурационных файлов. Среди этих файлов будут `/etc/fstab`, содержащий информацию о монтировании файловых систем при загрузке, `/etc/mdev.conf` с информацией для `mdev`, предназначенного для динамического управления устройствами в диретории `/dev`, `/etc/profile` с настройками командной оболочки ash и ряд других конфигов, необходимых для работы системы.

Конечно, в более продвинутых и функциональных системах конфигурационных файлов куда больше, но в данном руководстве перечислен минимально возможный список кнфигов, подходящий для корректной работы LFA.

Создание /etc/fstab

Файл `/etc/fstab` используется рядом программ для определения того, какие файловые системы будут монтироваться по умолчанию, в каком порядке, а какие должны быть проверены (на наличие повреждений и ошибок) перед монтированием. Для нашей встраиваемой системы предполагается, что загрузчик укажет ядру Linux, где найти корневую ФС, поэтому здесь мы не будем указывать никаких других файловых систем.

Создайте пустой файл `/etc/fstab`:

```
cat > $LFA_SYS/etc/fstab << "EOF"
# Begin /etc/fstab

# file-system    mount-point    type    options             dump
fsck
EOF
```

При желании вы можете добавить в этот файл другие файловые системы, которые вы хотите монтировать автоматически при загрузке системы, например, разделы или файлы подкачки, разделы на различных накопителях типа NVMe, eMMC или SD или сетевые файловые системы.

Настройка mdev

`mdev` (является частью проекта BusyBox) — это замена `udev` с другой базой правил.

Создайте файл `/etc/mdev.conf`:

```
cat > $LFA_SYS/etc/mdev.conf << "EOF"
# /etc/mdev/conf

# Devices:
# Syntax: %s %d:%d %s
# devices user:group mode

# null does already exist; therefore ownership has to be changed
with command
null    root:root 0666  @chmod 666 $MDEV
zero    root:root 0666
grsec   root:root 0660
full    root:root 0666

random  root:root 0666
urandom root:root 0444
hwrandom root:root 0660

# console does already exist; therefore ownership has to be changed
with command
#console        root:tty 0600   @chmod 600 $MDEV && mkdir -p vc &&
ln -sf ../$MDEV vc/0
console root:tty 0600 @mkdir -pm 755 fd && cd fd && for x in 0 1 2 3
; do ln -sf /proc/self/fd/$x $x; done

fd0     root:floppy 0660
kmem    root:root 0640
mem     root:root 0640
port    root:root 0640
ptmx    root:tty 0666

# ram.*
ram([0-9]*)     root:disk 0660 >rd/%1
loop([0-9]+)    root:disk 0660 >loop/%1
sd[a-z].*       root:disk 0660 */lib/mdev/usbdisk_link
hd[a-z][0-9]*   root:disk 0660 */lib/mdev/ide_links
md[0-9]         root:disk 0660

tty             root:tty 0666
tty[0-9]        root:root 0600
tty[0-9][0-9]   root:tty 0660
ttyS[0-9]*      root:tty 0660
pty.*           root:tty 0660
vcs[0-9]*       root:tty 0660
vcsa[0-9]*      root:tty 0660

ttyLTM[0-9]     root:dialout 0660 @ln -sf $MDEV modem
```

```
ttySHSF[0-9]    root:dialout 0660 @ln -sf $MDEV modem
slamr           root:dialout 0660 @ln -sf $MDEV slamr0
slusb           root:dialout 0660 @ln -sf $MDEV slusb0
fuse            root:root  0666

# dri device
card[0-9]       root:video 0660 =dri/

# alsa sound devices and audio stuff
pcm.*           root:audio 0660 =snd/
control.*       root:audio 0660 =snd/
midi.*          root:audio 0660 =snd/
seq             root:audio 0660 =snd/
timer           root:audio 0660 =snd/

adsp            root:audio 0660 >sound/
audio           root:audio 0660 >sound/
dsp             root:audio 0660 >sound/
mixer           root:audio 0660 >sound/
sequencer.*     root:audio 0660 >sound/

# misc stuff
agpgart         root:root 0660  >misc/
psaux           root:root 0660  >misc/
rtc             root:root 0664  >misc/

# input stuff
event[0-9]+     root:root 0640 =input/
mice            root:root 0640 =input/
mouse[0-9]      root:root 0640 =input/
ts[0-9]         root:root 0600 =input/

# v4l stuff
vbi[0-9]        root:video 0660 >v4l/
video[0-9]      root:video 0660 >v4l/

# dvb stuff
dvb.*           root:video 0660 */lib/mdev/dvbdev

# load drivers for usb devices
usbdev[0-9].[0-9]       root:root 0660 */lib/mdev/usbdev
usbdev[0-9].[0-9]_.*    root:root 0660

# net devices
tun[0-9]*       root:root 0600 =net/
tap[0-9]*       root:root 0600 =net/

# zaptel devices
```

```
zap(.*)         root:dialout 0660 =zap/%1
dahdi!(.*)      root:dialout 0660 =dahdi/%1

# raid controllers
cciss!(.*)      root:disk 0660 =cciss/%1
ida!(.*)        root:disk 0660 =ida/%1
rd!(.*)         root:disk 0660 =rd/%1

sr[0-9]         root:cdrom 0660 @ln -sf $MDEV cdrom

# hpilo
hpilo!(.*)      root:root 0660 =hpilo/%1

# xen stuff
xvd[a-z]        root:root 0660 */lib/mdev/xvd_links
EOF
```

Создание /etc/profile

Файл `/etc/profile` содержит в себе общесистемные настройки командной оболочки. Создайте этот файл:

```
cat > $LFA_SYS/etc/profile << "EOF"
# /etc/profile

# Set the initial path
export PATH=/bin:/usr/bin

if [ `id -u` -eq 0 ] ; then
        PATH=/bin:/sbin:/usr/bin:/usr/sbin
        unset HISTFILE
fi

# Setup some environment variables.
export USER=`id -un`
export LOGNAME=$USER
export HOSTNAME=`/bin/hostname`
export HISTSIZE=1000
export HISTFILESIZE=1000
export PAGER='/bin/more '
export EDITOR='/bin/ed'

# End /etc/profile
EOF
```

Установка имени хоста

Во время загрузки система устанавливает имя хоста, на котором она работает (hostname). Имя хоста содержится в файле `/etc/hostname`. Создайте его:

```
echo "[lfa]" > $LFA_SYS/etc/hostname
```

Замените `[lfa]` на имя, присвоенное компьютеру. Не вводите здесь полное доменное имя (FQDN). Эта информация будет помещена в файл `/etc/hosts` в следующем разделе.

Настройка сети

Данные любого сайта находятся на физическом сервер. Чтобы браузер нашёл нужный сервер, он должен знать его IP-адрес. Узнать, какому IP соответствует домен, можно с помощью DNS-системы, до появления которой использовался текстовый файл `/etc/hosts` .

С помощью этого файла можно, например:

- Указать псевдоним для локальной сети;
- Запретить посещение какого-либо сайта;
- Просмотреть сайт до внесения его в DNS-систему (для этого нужно вписать нужный домен и соответствующий ему IP если данные DNS-системы ещё не обновлены).

> В современных системах даже заданная в `/etc/hosts` информация может быт перекрыта информацией из DNS.

Создайте базовый `/etc/hosts` файл:

```
cat > $LFA_SYS/etc/hosts << "EOF"
# Begin /etc/hosts

127.0.0.1 localhost
EOF
```

Смотрите также:

- **Файл `hosts` в Linux** (https://losst.pro);
- man: *hosts(5)*;

Сборка ядра

Ядро Linux — основной компонент операционной системы, выступающий промежуточным звеном между оборудованием и программным обеспечением ОС.

Общие рекомендации по сборке ядра

Первым делом нужно создать файл `.config`, содержащий параметры ядра. Для его создания можно воспользоваться следующими опциями:

- `make defconfig` — создаёт стандартный конфиг с учётом архитектуры компьютера, для которого производится сборка.
- `make oldconfig` — задаёт пользователю ряд вопросов о конфигурации ядра в текстовом режиме. Не позволяет изменить уже заданные параметры (изменение возможно после путём редактирования файла `.config` вручную).
- `make menuconfig` — настройка ядра в псевдографическом меню. Доступно разделение всяческих функций, опций и драйверов по категориям, справка по этим вещам и прочее.

После того, как вы настроили ядро, создав `.config` одним из способов, указанных ниже, рекомендуем вам сохранить созданный `.config` где-нибудь, чтобы использовать его в дальнейшем при возможных новых сборках ядра Linux.

Рекомендуем вам все ключевые компоненты ядра *встраивать* в ядро, а не компилировать в виде подключаемых *модулей*. Да и вообще рекомендуем вам оставить в конфигурации ядра (отмечено как `<*>` или `<M>` [1]) только то, что вам действительно необходимо. Это поможет вам сэкономить место на диске (размер одних только установленных в систему модулей легко может превысить объём всей системы без них) и упростить процесс загрузки системы (поскольку не придётся заботиться о том, какие модули загружать, а какие — нет).

В случае возникновения ошибки сборки, если рядом с этой ошибкой нет подробного текста о причине её возникновения, прочитайте весь вывод `make` — иногда сообщение о причине ошибки может быть очень далеко от последнего выведенного `make` сообщения. Это особенно актуально, если вы собираете ядро в несколько потоков.

---

[1] `<*>` означает, что эта функция будет встроена в двоичный файл ядра, а `<M>` — что эта функция будет скомпилирована как модуль.

linux

Ядро операционной системы.

- **Версия:** 6.6.44
- **Домашняя страница:** https://www.kernel.org
- **Время сборки:** 23 ОВС

Процесс сборки ядра состоит из нескольких процессов: конфигурирование, компиляция и установка. Прочитайте файл `README` в дереве исходного кода Linux, чтобы узнать об альтернативных методах, отличных от того, как конфигурируется ядро в этом руководстве.

Патчи: если вы собираете систему для компьютера, оснащённого SoC Rockchip, можете применить **патчи**.

Подготовка

Убедитесь в том, что дерево исходного кода ядра не содержит лишних файлов:

```
make mrproper
```

Разработчики ядра рекомендуют выполнять эту команду каждый раз, когда вы собираете Linux.

Настройка

> Поскольку вы создаёте встраиваемую систему, убедитесь, что все *ключевые* компоненты **встроены** в ядро, а не являются модулями (в меню, которое откроется по команде далеее опция отмечена как `<*>`, а не как `<M>`). Ключевыми обычно являются опции для поддержки консоли, видео, дисков и файловых систем, а также сети. Без них система не будет функционировать должным образом. Рекомендуется конфигурировать ядро без модулей, чтобы сэкономить место на диске и упростить процесс загрузки системы.

Откройте псевдографическое меню, в котором вам нужно выбрать все опции, которые нужны вам для корректной работы ядра на компьютере, для которого вы собираете систему:

```
make ARCH=arm64 CROSS_COMPILE=$LFA_TGT- menuconfig
```

Выбор платформы

Обязательно зайдите в раздел `Platform selection  --->` и отметьте там поддержку необходимых для вас SoC. Например, если я собираю систему для компьютера Orange Pi 3 LTS с Allwinner SoC, то в этом списке мне нужно отметить только

```
Platform selection  --->
  [*] Allwinner sunxi 64-bit SoC Family
```

FUSE

FUSE (англ. filesystem in userspace — «файловая система в пользовательском пространстве») — свободный модуль для ядер Unix-подобных операционных систем, позволяющий разработчикам создавать новые типы файловых систем, доступные для монтирования

пользователями без привилегий (прежде всего — виртуальных файловых систем); это достигается за счёт запуска кода файловой системы в пользовательском пространстве, в то время как модуль FUSE предоставляет связующее звено для актуальных интерфейсов ядра. С использованием средств FUSE разработаны, в частности, SSHFS, NTFS-3G, GlusterFS, ZFS.

```
File systems  --->
  <*/M> FUSE (Filesystem in Userspace) support [CONFIG_FUSE_FS]
```

Сборка

Скомпилируйте ядро, используя только что созданный `.config`:

```
make ARCH=arm64 CROSS_COMPILE=$LFA_TGT-
```

Теперь вам необходимо скомпилировать файлы devicetree:

```
make ARCH=arm64 CROSS_COMPILE=$LFA_TGT- dtbs
```

Установка

Следующие действия нужно выполнить, если вы собирали ядро с модулями:

При использовании модулей ядра может потребоваться файл `/etc/modprobe.conf`. Информация, касающаяся модулей и конфигурации ядра, находится в документации в директории `Documentation/`. Также будет не лишним прочитать документацию (man) *modprobe.conf(5)*.

```
make ARCH=arm64 CROSS_COMPILE=$LFA_TGT- \
  INSTALL_MOD_PATH=$LFA_SYS modules_install
```

Установка ядра и ряда дополнительных файлов

Конфигурационный файл `.config` содержит все настройки конфигурации только что собранного ядра. Было бы неплохим сохранить этот файл для дальнейшего пользования:

```
cp -v .config $LFA_SYS/boot/config-6.6.44
```

Скопируйте файл `System.map` в `/boot`:

```
cp -v System.map $LFA_SYS/boot/System.map-6.6.44
```

Полученное ядро будет находиться в директории `arch/arm64/boot`. Возможно, что там будет находиться несколько вариантов одного и того же ядра, просто с разным сжатием или добавлением помощников загрузчика. Следуйте инструкциям вашего загрузчика по копированию ядра в конечную систему. Например:

```
cp -iv arch/arm64/boot/Image $LFA_SYS/boot/vmlinuz-6.6.44
```

Для того, чтобы не указывать в опциях ядра полное имя `vmlinuz-6.6.44` создайте символическую ссылку:

```
ln -svf vmlinuz-6.6.44 $LFA_SYS/boot/vmlinuz
```

Установка файлов Devicetree

Создайте в `$LFA_SYS/boot` директорию `dtb-6.6.44`:

```
mkdir -pv $LFA_SYS/boot/dtb-6.6.44
```

И установите файлы Devicetree:

```
make ARCH=arm64 CROSS_COMPILE=$LFA_TGT- \
  INSTALL_DTBS_PATH=$LFA_SYS/boot/dtb-6.6.44 dtbs_install
```

Для того, чтобы не указывать в `boot.cmd`, который мы скоро напишем, версию ядра, сделайте символическую ссылку:

```
ln -svf dtb-6.6.44 $LFA_SYS/boot/dtb
```

Создание ссылок на ряд периодически изменяющихся файлов сделает проще их обновление до новой версии: нужно просто установить новую версию файла ядра и директории с devicetree в `$LFA_SYS/boot/` и обновить символические ссылки. Этим мы можем оставить предыдущие версии ядра и devicetree, которые могут понадобиться, если окажется, что их новые версии работают некорректно или не работают вовсе.

> ⚠ **Внимание**
>
> В `$LFA_SYS/boot/dtb-6.6.44` будут установлены скомпилированные файлы devicetree только для тех плат, поддержку которых вы указали при конфигурировании ядра. В этой директории будут созданы поддиректории с именами используемых в этих платах SoC, например:
>
> - `$LFA_SYS/boot/dtb-6.6.44/allwinner/`
> - `$LFA_SYS/boot/dtb-6.6.44/broadcom/`
> - `$LFA_SYS/boot/dtb-6.6.44/rockchip/`
>
> В этих поддиректориях будут содержаться файлы `*.dtb` для поддерживаемых плат. Если вам что-либо отсюда не нужно, то ради экономии места и уменьшения размера собранного дистрибутива вы можете удалить лишние файлы. Однако будьте готовы, что на каких-то компьютерах ваша система может не заработать.

Содержимое пакета

- **Установленные файлы:** `.config-6.6.44`, `vmlinuz-6.6.44`, `vmlinuz`, `System.map-6.6.44`, `dtb-6.6.44/*`, `dtb/`

Описание компонентов

- `.config-6.6.44` — содержит параметры сборки ядра.
- `vmlinuz-6.6.44`, `vmlinuz` — скомпилированное ядро Linux.
- `System.map-6.6.44` — список адресов и символов; в нём указаны точки входа и адреса всех функций и структур данных в ядре. Иногда полезен при отладке.
- `dtb-6.6.44/`, `dtb/` — директория с файлами devicetree.

Сборка загрузчика

Загрузчик операционной системы, предназначенный для встраиваемых систем на MIPS, ARM, PowerPC и т.д.

- **Версия:** 2024.04
- **Домашняя страница:** https://source.denx.de/u-boot/u-boot
- **Время сборки:** 10 ОВС

Внимание

Этот раздел содержит **общие** инструкции по сборке загрузчика. Поскольку сборка U-Boot специфична как для каждого SoC, так и, возможно, для разных плат/компьютеров, использующих эти SoC, в LFA очень тяжело описать сценарии сборки для каждой модели популярных ARM-компьютеров. Для получения подробных сведений о загрузчике U-Boot, пожалуйста, обратитесь к **его документации**.

В Дополнительных материалах приведены некоторые переведённые из официальной документации сведения, которые могут пригодиться во время сборки загрузчика и написания скриптов для загрузки ядра операционной системы. Перед началом сборки U-Boot рекомендуем ознакомиться с ними.

Кроме того, работа над этим разделом ещё идёт. Вы можете оказать нам неоценимую помощь, если дополните его новыми инструкциями. В частности, нужны сведения о файлах `boot.scr`, `boot.cmd` и `uInitrd`, а также информация о сборке `img`-образа с системой. Если вы готовы помочь нам, то можете либо создать **issue** с предложениями изменений, либо pull request с изменениями.

Примерный порядок сборки

Для большинства SoC перед сборкой U-Boot необходимо иметь скомпилированные дополнительные компоненты. Например, ARM Trusted Firmware. Для того, чтобы узнать, что и как нужно собирать, обратитесь к **документации (раздел «Board-specific docs»)** загрузчика U-Boot. Далее предоставлены общие инструкции для сборки загрузчика для плат, оснащённых Allwinner, Broadcom и Rockchip SoC.

В итоге примерный порядок сборки будет следующим:

1. Сборка ряда компонентов загрузчика, таких как, например, TF-A, `crust` или `rkbin`;
2. Сборка загрузчика;
3. Создание файлов `boot.scr`, `boot.cmd`, `uInitrd`;
4. Сохранение файла загрузчика для его записи на флешку или в `img`-образ с собранной системой LFA;

Смотрите также:

- **Getting started with U-Boot** (https://krinkinmu.github.io)
- **How U-boot loads Linux kernel** (https://krinkinmu.github.io)
- **U-boot boot script** (https://krinkinmu.github.io)

Allwinner

> ⚠ **Внимание**
>
> В данной части предоставлены *общие* инструкции, которые приведут к сборке работающего загрузчика U-Boot, пригодного для дальнейшего использования. Конечно, «кастомная» сборка U-Boot и ряда дополнительных компонентов с другими параметрами допускается, но на данный момент я не предоставляю иных инструкций кроме этих. Возможно, что в будущем я добавлю ряд советов по изменению параметров сборки, но сейчас у меня на это нет ни времени, ни сил, ни желания.

Смотрите также:

- **Страница U-Boot на linux-sunxi WiKi** (https://linux-sunxi.org/)

Allwinner: Сборка TF-A

Проект Trusted Firmware-A предоставляет эталонную реализацию безопасного программного обеспечения для процессоров класса ARMv7-A и ARMv8-A.

- **Версия:** 2.10.9
- **Домашняя страница:** https://www.trustedfirmware.org/projects/tf-a/
- **Время сборки:** 1 ОВС

Настройка

Вам нужно объявить переменную окружения `PLAT`, которая будет содержать имя целевой платформы для сборки:

```
export PLAT="целевая платформа"
```

Целевые платформы для сборки:

Сборка TF-A специфична для каждого SoC, в частности, специфично значение переменной `PLAT`, которая передаётся системе сборки `make`. Вы можете воспользоваться значениями из таблицы ниже:

SoC	Платформа
Allwinner A64	`sun50i_a64`
Allwinner H5	`sun50i_a64`
Allwinner H6	`sun50i_h6`
Allwinner H616	`sun50i_h616`
Allwinner H313	`sun50i_h616`

SoC	Платформа
Allwinner T507	`sun50i_h616`
Allwinner R329	`sun50i_r329`

Для поиска всех целевых платформ введите:

```
find plat/allwinner -name platform.mk
```

В файле `docs/plat/allwinner.rst` содержится дополнительная информация и приведены некоторые опции сборки.

Например, если в моей плате используется SoC Allwinner H6, то значение переменной `PLAT` будет равно `sun50i_h6`:

```
export PLAT="sun50i_h6"
```

Сборка

```
make CROSS_COMPILE=$LFA_TGT- DEBUG=1
```

Настройка окружения

Теперь вам нужно объявить переменную окружения `BL31`, содержащую путь до скомпилированной микропрограммы:

```
export BL31=$PWD/build/$PLAT/debug/bl31.bin
```

Необходимости в переменной `PLAT` больше нет, поэтому можете её удалить:

```
unset PLAT
```

> **Внимание**
>
> Не удаляйте эту директорию с исходным кодом TF-A (в которой вы его собирали) до тех пор, пока не соберёте загрузчик U-Boot!

> **Содержимое пакета**
>
> - **Установленные файлы:** `$PWD/build/$PLAT/debug/bl31.bin`

Описание компонентов

- `$PWD/build/$PLAT/debug/bl31.bin` — требуемый для сборки U-Boot компонент TF-A.

Allwinner: Установка кросс-компилятора or1k

Для некоторых SoC необходима сборка компонента crust, отвечающего за управление питанием. Чтобы его собрать, вам требуется кросс-компилятор для архитектуры or1k. В данном руководстве не приводится инструкций по сборке ещё одного кросс-компилятора, поскольку он требуется лишь для некоторых SoC, а также очень сильно раздувает и усложняет это руководство, которое я хочу оставить простым и кратким. Поэтому, если для вашего SoC вдруг понадобится crust, а для его сборки — кросс-компилятор or1k, то устанавливайте или собирайте его самостоятельно. Такие дистрибутивы, как Archlinux, в своих репозиториях предоставляют нужные пакеты gcc и binutils, выполняющие сборку ПО под эту архитектуру. Возможно, такие пакеты предоставляют и другие дистрибутивы.

Название необходимого кросс-компилятора: `or1k-none-elf` .

За дополнительной информацией о кросс-компиляторе обращайтесь к разделу дополнительных материалов.

Allwinner: Сборка SCP (crust)

Низкоуровневый компонент для плат на базе Allwinner, предназначенный для уравления питанием. Во время глубокого сна ядра процессора, контроллер DRAM и большинство встроенных периферийных устройств отключаются от питания, что позволяет снизить электропотребление на >80%. На платах без PMIC crust также отвечает за упорядоченное включение и выключение устройства.

- **Версия:** 0.6
- **Домашняя страница:** https://github.com/crust-firmware/crust
- **Время сборки:** 0.3 ОВС

> ⚠ **Внимание**
>
> Сборка пакета осуществляется с помощью кросс-компилятора `or1k-none-elf` .

Патчи: для `crust` вы можете найти дополнительные опциональные патчи в **репозитории Armbian**.

Если вы собираете систему для OrangePi 3 LTS, можете применить патч `add-defconfig-for-orangepi-3-lts` . Если вы собираете систему для другого компьютера, оснащённого SoC H3, H5, H6, A64, можете применить **соответствующий патч** для добавления их поддержки.

Настройка

```
make <имя платы>_defconfig
```

Список поддерживаемых материнских плат смотрите в директории `configs/`. Например, для Orange Pi 3 замените `<имя платы>` на `orangepi_3`. Прочитайте также **README**-файл из репозитория crust для получения дополнительных сведений о процессе сборки этой микропрограммы.

Сборка

```
make CROSS_COMPILE=or1k-none-elf- scp
```

Установка

Вам не нужно никуда копировать собранные программы, просто объявите новую переменную окружения `SCP`:

```
export SCP=$PWD/build/scp/scp.bin
```

> ⚠ **Внимание**
>
> Не удаляйте эту директорию с исходным кодом crust (в которой вы его собирали) до тех пор, пока не соберёте загрузчик U-Boot!

Содержимое пакета

- **Установленные файлы:** `$PWD/build/scp/scp.bin`

Описание компонентов

- `$PWD/build/scp/scp.bin` — требуемый для сборки U-Boot компонент crust, предназначенный для управления питанием.

Allwinner: Сборка U-Boot

Загрузчик операционной системы, предназначенный для встраиваемых систем на MIPS, ARM, PowerPC и т.д.

- **Версия:** 2024.04
- **Домашняя страница:** https://source.denx.de/u-boot/u-boot
- **Время сборки:** 10 ОВС

> ⚠ **Внимание**
>
> Предполагается, что у вас уже установлены нужные переменные окружения, в частности `BL31` и, опционально, `SCP` .

Патчи: для загрузчика Allwinner вы можете найти дополнительные опциональные патчи в **репозитории Armbian**.

Настройка

Директория `configs/` содержит шаблоны конфигурационных файлов для поддерживаемых [проектом U-Boot, а не LFA] плат в соответствии со следующей схемой наименования:

```
<имя платы>_defconfig
```

Вы можете использовать имя одного из этих файлов в качестве цели `make` для генерации конфигурационного файла `.config` . Например, шаблон конфигурации для платы Orange Pi 3 называется `orangepi_3_defconfig` . Соответственно, файл `.config` генерируется командой:

```
make orangepi_3_defconfig
```

Сборка

```
make CROSS_COMPILE=$LFA_TGT-
```

Сохранение образа U-Boot

Скопируйте скомпилированный файл в директорию `$LFA` для удобного доступа к нему в будущем:

```
cp -v u-boot-sunxi-with-spl.bin $LFA/bootloader.bin
```

Команда выше верна: мы действительно копируем файл загрузчика в директорию `$LFA`, а не в `$LFA_SYS`, как это делали обычно. Дело в том, что в базовой системе этот файл не особо нужен, а в `$LFA` содержатся различные файлы: кросс-компилятор, директория с базовой ОС, а теперь там будет и файл загрузчика. В будущем в этой директории мы соберём `img.xz`-образ вашей системы. Каталог `$LFA` отлично подходит для хранения разнотиповых файлов, чего не скажешь о `$LFA_SYS`, единственное предназначение которого — хранение файлов базовой системы.

Содержимое пакета

- **Установленные файлы:** `$LFA/bootloader.bin`

Описание компонентов

- `$LFA/bootloader.bin` — скомпилированный файл загрузчика U-Boot.

Broadcom

Поддерживаемые платы

SoC BCM7445 и BCM7260

Здесь U-Boot является загрузчиком третьего этапа, который запускается встроенным загрузчиком BOLT компании Broadcom.

BOLT загружает U-Boot как бинарный ELF-файл. Некоторые функции U-Boot, такие как работа с сетью, не реализованы, но есть некоторые другие важные функции, включая:

- Поддержка файловой системы `ext4` ;
- Поддержка FIT-образов;
- Поддержка файлов devicetree из FIT-образа вместо таковых файлов из BOLT.

SoC	`defconfig`
BCM7445	`bcm7445`
BCM7260	`bcm7260`

Raspberry Pi

Плата	Архитектура	`defconfig`
Raspberry Pi	**32 бит**	`rpi`
Raspberry Pi 1/Raspberry Pi Zero		`rpi_0_w`
Raspberry Pi 2		`rpi_2`
Raspberry Pi 3b		`rpi_3_32b`
Raspberry Pi 4b		`rpi_4_32b`

Плата	Архитектура	`defconfig`
Raspberry Pi 3b	**64 бит**	`rpi_3`
Raspberry Pi 3b+		`rpi_3_b_plus`
Raspberry Pi 4b		`rpi_4`

Общая конфигурация:

Плата	`defconfig`
Raspberry Pi 3b	`rpi_arm64` [1]
Raspberry Pi 3b+	
Raspberry Pi 4b	
Raspberry Pi 400	
Raspberry Pi CM 3	
Raspberry Pi CM 3+	
Raspberry Pi CM 4	
Raspberry Pi zero 2w	

Сборка

```
make CROSS_COMPILE=$LFA_TGT-
```

Сохранение образа U-Boot

```
cp -v u-boot $LFA/bootloader.bin
```

[1] `rpi_arm64_defconfig` использует дерево устройств, предоставляемое прошивкой, вместо встроенного в U-Boot. Это позволяет использовать один и тот же бинарник U-Boot для загрузки с разных плат.

Rockchip

Поддерживаемые платы

Выберите из списка ниже нужную вам материнскую плату. Алгоритм действий следующий: подставьте в аргумент системы сборки make вместо `<имя платы>_defconfig` имя вашей платы, указанное в столбце « `defconfig` », например, для платы Orange Pi 5, аргумент будет `orangepi-5-rk3588s_defconfig` .

SoC	Плата	defconfig
px30	**Rockchip Evb-PX30**	**evb-px30**
	Engicam PX30.Core C.TOUCH 2.0	px30-core-ctouch2-px30
	Engicam PX30.Core C.TOUCH 2.0 10.1	px30-core-ctouch2-of10-px30
	Firefly Core-PX30-JD4	firefly-px30
	Theobroma Systems PX30-µQ7 SoM - Ringneck	ringneck-px30
rk3288	**Google Jerry**	**chromebook_jerry**
	Google Mickey	chromebook_mickey
	Google Minnie	chromebook_minnie
	Google Speedy	chromebook_speedy
rk3308	**Radxa ROCK Pi S**	**rock-pi-s-rk3308**
rk3326	**Odroid-go Advance**	**odroid-go2**
rk3328	**FriendlyElec NanoPi R2C**	**nanopi-r2c-rk3328**
	FriendlyElec NanoPi R2C Plus	nanopi-r2c-plus-rk3328
	FriendlyElec NanoPi R2S	nanopi-r2s-rk3328

SoC	Плата	defconfig
	Pine64 Rock64	`rock64-rk3328`
	Radxa ROCK Pi E	`rock-pi-e-rk3328`
	Xunlong Orange Pi R1 Plus	`orangepi-r1-plus-rk3328`
	Xunlong Orange Pi R1 Plus LTS	`orangepi-r1-plus-lts-rk3328`
rk3399	**FriendlyElec NanoPC-T4**	**`nanopc-t4-rk3399`**
	FriendlyElec NanoPi M4	`nanopi-m4-rk3399`
	FriendlyElec NanoPi M4B	`nanopi-m4b-rk3399`
	FriendlyARM NanoPi NEO4	`nanopi-neo4-rk3399`
	Google Bob	`chromebook_bob`
	Google Kevin	`chromebook_kevin`
	Khadas Edge	`khadas-edge-rk3399`
	Khadas Edge-Captain	`khadas-edge-captain-rk3399`
	Khadas Edge-V	`khadas-edge-v-rk3399`
	Orange Pi RK3399	`orangepi-rk3399`
	Pine64 RockPro64	`rockpro64-rk3399`
rk3566	**Pine64 PineTab2**	**`pinetab2-rk3566`**
	Pine64 Quartz64-A Board	`quartz64-a-rk3566`
	Pine64 Quartz64-B Board	`quartz64-b-rk3566`
rk3568	**Banana Pi BPI-R2 Pro**	**`bpi-r2-pro-rk3568`**
	FriendlyElec NanoPi R5C	`nanopi-r5c-rk3568`
	FriendlyElec NanoPi R5S	`nanopi-r5s-rk3568`
	Hardkernel ODROID-M1	`odroid-m1-rk3568`
	Radxa ROCK 3 Model A	`rock-3a-rk3568`
	Generic RK3566/RK3568	*`generic-rk3568`*
rk3588	**Edgeble Neural Compute Module 6A SoM - Neu6a**	**`neu6a-io-rk3588`**

SoC	Плата	defconfig
	Edgeble Neural Compute Module 6B SoM - Neu6b	`neu6b-io-rk3588`
	FriendlyElec NanoPC-T6	`nanopc-t6-rk3588`
	Pine64 QuartzPro64	`quartzpro64-rk3588`
	Radxa ROCK 5A	`rock5a-rk3588s`
	Radxa ROCK 5B	`rock5b-rk3588`
	Xunlong Orange Pi 5	`orangepi-5-rk3588s`
	Xunlong Orange Pi 5 Plus	`orangepi-5-plus-rk3588`
	Generic RK3588S/RK3588	*`generic-rk3588`*
rv1126	**Edgeble Neural Compute Module 2 SoM - Neu2/Neu2k**	**`neu2-io-r1126`**

Таблица поддерживаемого оборудования в сокращённом виде. Выберите отсюда нужную вам плату. Если вашу плату, использующую Rockchip SoC, вы не нашли здесь, обратитесь к таблице из **документации U-Boot**, где содержится полный список поддерживаемых плат.

Rockchip: Сборка TF-A/rkbin

> ⚠ **Внимание**
>
> Перед тем, как начинать сборку TF-A или `rkbin`, определитесь, *что* вам нужно собирать. Для ряда SoC подойдёт TF-A, а для некоторых — `rkbin`. На данной странице приведены инструкции для сборки как TF-A, так и `rkbin`.
>
> `rkbin` требуется для следующих SoC:
>
> - rk3308
> - rk3568 (для него можно использовать либо TF-A, либо rkbin)
> - rk3588
>
> Если для вашего SoC нужен `rkbin`, пропустите сборку TF-A. Если для вашего SoC нужен TF-A, пропустите сборку `rkbin`.

TF-A

Проект Trusted Firmware-A предоставляет эталонную реализацию безопасного программного обеспечения для процессоров класса ARMv7-A и ARMv8-A.

- **Версия:** 2.10.9
- **Домашняя страница:** https://www.trustedfirmware.org/projects/tf-a/
- **Время сборки:** 1 ОВС

Настройка

Вам нужно объявить переменную окружения `PLAT`, которая будет содержать имя целевой платформы для сборки:

```
export PLAT="rkXXXX"
```

Замените `rkXXXX` на имя вашего SoC, например, для Rockchip RK3399 значение переменной `PLAT` будет `rk3399`.

Сборка

```
make realclean
make CROSS_COMPILE=$LFA_TGT-
```

Установка

Экспортируйте переменную окружения `BL31`, которая будет содержать путь до скомпилированного файла BL31:

```
export BL31="$PWD/путь/до/bl31"
```

Для PX30:

```
export BL31=$PWD/build/px30/release/bl31/bl31.elf
```

---

rkbin

> Прошивка BL31 для тех Rockchip SoC, для которых не обеспечена поддержка BL31 из состава TF-A.

- **Версия:** master
- **Домашняя страница:** https://github.com/rockchip-linux/rkbin
- **Время сборки:** 0 ОВС

Установка

В директории `rkbin/bin/` содержатся поддиректории для разных семейств SoC Rockchip. Выберите нужную поддиректорию и найдите в ней файл BL31. Экспортируйте переменную окружения `BL31`, которая будет содержать путь до файла BL31:

```
export BL31="$PWD/путь/до/bl31"
```

Например:

```
export BL31="$PWD/rkbin/bin/rk33/rk3308_bl31_v2.26.elf"
```

> ⚠ **Внимание**
>
> Не удаляйте директорию с исходным кодом TF-A/rkbin до тех пор, пока не соберёте загрузчик U-Boot!

Rockchip: TPL

Для некоторых SoC в U-Boot отсутствует поддержка инициализации DRAM. В этих случаях для получения полнофункционального образа загрузчика выполните следующие действия:

1. **Упакуйте образ U-Boot TPL/SPL** (https://docs.u-boot.org), используйте бинарник DDR из репозитория `rkbin` как `ROCKCHIP_TPL` при сборке U-Boot, либо следуйте следующему пункту №2:
2. **Упакуйте образ при помощи Rockchip `miniloader`** (https://docs.u-boot.org);

Установка

Для RK3308:

```
export
ROCKCHIP_TPL="$PWD/rkbin/bin/rk33/rk3308_ddr_589MHz_uartX_mY_v2.07.b
in"
```

Для RK3568:

```
export
ROCKCHIP_TPL="$PWD/rkbin/bin/rk35/rk3568_ddr_1560MHz_v1.13.bin"
```

Для RK3588:

```
export
ROCKCHIP_TPL="$PWD/rkbin/bin/rk35/rk3588_ddr_lp4_2112MHz_lp5_2736MHz
_v1.09.bin"
```

Rockchip: Сборка U-Boot

Загрузчик операционной системы, предназначенный для встраиваемых систем на MIPS, ARM, PowerPC и т.д.

- **Версия:** 2024.04
- **Домашняя страница:** https://source.denx.de/u-boot/u-boot
- **Время сборки:** 10 ОВС

> ⚠ **Внимание**
>
> Предполагается, что у вас уже установлены нужные переменные окружения, в частности `BL31` и, опционально, `ROCKCHIP_TPL`.

Настройка

Объявите переменную окружения `TARGET`, которая будет содержать название SoC, для которого производится сборка загрузчика.

```
export TARGET="целевая платформа"
```

Целевые платформы загрузчика:

SoC	defconfig
px30	`evb-px30`
rk3066	`mk808`
rk3288	`evb-rk3288`
rk3308	`evb-rk3308`
rk3328	`evb-rk3328`

SoC	defconfig
rk3368	evb-px5
rk3399	evb-rk3399
rk3568	evb-rk3568
rk3588	evb-rk3588

Например, если в моей плате используется Rockchip RK3588, то значение переменной `TARGET` будет равно `evb-rk3588`:

```
export TARGET="evb-rk3588"
```

Создайте базовый конфиг для сборки U-Boot (`defconfig`):

```
make ${TARGET}_defconfig
```

Сборка

```
make CROSS_COMPILE=$LFA_TGT-
```

Сохранение образа U-Boot

Скопируйте скомпилированный файл в директорию `$LFA` для удобного доступа к нему в будущем:

```
cp -v u-boot-rockchip.bin $LFA/bootloader.bin
```

Команда выше верна: мы действительно копируем файл загрузчика в директорию `$LFA` , а не в `$LFA_SYS` , как это делали обычно. Дело в том, что в базовой системе этот файл не особо нужен, а в `$LFA` содержатся различные файлы: кросс-компилятор, директория с базовой ОС, а теперь там будет и файл загрузчика. В будущем в этой директории мы соберём

`img.xz`-образ вашей системы. Каталог `$LFA` отлично подходит для хранения разнотиповых файлов, чего не скажешь о `$LFA_SYS`, единственное предназначение которого — хранение файлов базовой системы.

> **Содержимое пакета**
>
> - **Установленные файлы:** `$LFA/bootloader.bin`
>
> ### Описание компонентов
>
> - `$LFA/bootloader.bin` — скомпилированный файл загрузчика U-Boot.

Эмуляция в QEMU (ARM)

Загрузчик операционной системы, предназначенный для встраиваемых систем на MIPS, ARM, PowerPC и т.д.

- **Версия:** 2024.04
- **Домашняя страница:** https://source.denx.de/u-boot/u-boot
- **Время сборки:** 10 ОВС

Настройка U-Boot

Установите имя аргумента `qemu_*_defconfig` в зависимости от архитектуры, для которой вы собираете:

```
if [ $LFA_TGT == "aarch64-linux-musleabihf" ]
then
  QEMU_ARCH="arm64"
else
  QEMU_ARCH="arm"
fi
```

Сконфигурируйте пакет U-Boot:

```
make CROSS_COMPILE=$LFA_TGT- \
  qemu_${QEMU_ARCH}_defconfig
```

ACPI в QEMU

QEMU (начиная с версии v8.0.0) может предоставлять таблицы ACPI для ARM[1]. Для их поддержки нужны следующие настройки U-Boot:

```
CONFIG_CMD_QFW=y
CONFIG_ACPI=y
CONFIG_GENERATE_ACPI_TABLE=y
```

Вместо того, чтобы обновлять файл `.config` вручную, вы можете добавить опцию `acpi.config` в команду `make`. Например:

```
make CROSS_COMPILE=$LFA_TGT-  \
  qemu_${QEMU_ARCH}_defconfig \
  acpi.config
```

Сборка

```
make CROSS_COMPILE=$LFA_TGT-
```

В итоге будет сгенерирован двоичный файл `u-boot.bin`.

> ⚠ **Важно**
>
> В разделах о сборке загрузчика U-Boot для SoC Allwinner, Broadcom и Rockchip предполагается, что собранный двоичный файл с загрузчиком будет «встроен» в `img`-образ, который будет сгенерирован в разделе №8 «Сборка образа». Здесь же таких действий совершать не требуется — всё, что вам нужно, так это скопировать собранный двоичный файл загрузчика в специально отведённое для этого место, **не встраивая** его в генерируемый образ системы.

Сохранение образа U-Boot

Скопируйте скомпилированный файл в директорию `$LFA` для удобного доступа к нему в будущем:

```
cp -v u-boot.bin $LFA/bootloader.bin
```

Также вместо `u-boot.bin` вы можете использовать образ `u-boot-nodtb.bin`, если не хотите использовать скомпилированные файлы Devicetree.

Смотрите также:

- **QEMU ARM — Das U-Boot** (https://docs.u-boot.org/en/latest/).

---

[1] На x86 эти настройки уже включены по умолчанию. Для ARM и RISC-V по умолчанию используются device-tree.

Сборка загрузчика (для остальных)

> ⚠ **Внимание**
>
> В этом разделе приведены *общие* инструкции по сборке загрузчика для компьютеров/эмуляторов, не описанных в разделах выше. Используйте сведения отсюда в качестве опоры и примерного алгоритма действий.

Настройка

Директория `configs/` содержит шаблоны конфигурационных файлов для поддерживаемых [проектом U-Boot, а не LFA] плат в соответствии со следующей схемой наименования:

```
<имя платы>_defconfig
```

Эти файлы лишены настроек по умолчанию. Поэтому вы не можете использовать их напрямую. Вместо этого их имя служит в качестве цели `make` для генерации фактического конфигурационного файла `.config`. Например, шаблон конфигурации для платы Odroid C2 называется `odroid-c2_defconfig`. Соответствующий файл `.config` генерируется командой:

```
make odroid-c2_defconfig
```

Для плат на базе SoC Allwinner:

> На вики **linux-sunxi** также можно найти имя `defconfig` файла на соответствующей странице платы.

Вы можете сконфигурировать пакет командой:

```
make menuconfig
```

Сборка

> Для сборки вам по прежгнему нужен наш кросс-компилятор. Кроме того, в системе должны быть установлены пакеты `swig` и `python-setuptools` .

```
CROSS_COMPILE=$LFA_TGT- make
```

Компилятор Devicetree

Платам, использующим `CONFIG_OF_CONTROL` (т.е. почти всем), нужен компилятор Devicetree (`dtc`). Платам с `CONFIG_PYLIBFDT` требуется `pylibfdt` (библиотека Python для доступа к данным Devicetree). Подходящие версии этих библиотек включены в дерево U-Boot в директории `scripts/dtc` и собираются автоматически по мере необходимости.

Если вы хотите использовать их системные версии, используйте переменную `DTC` , в которой будет указан путь до `dtc` :

```
CROSS_COMPILE=$LFA_TGT- DTC=/usr/bin/dtc make
```

В этом случае `dtc` и `pylibfdt` не будут собраны. Система сборки проверит, что версия `dtc` достаточно новая. Она также убедится, что

`pylibfdt` присутствует, если это необходимо.

Обратите внимание, что инструменты **Host Tools** всегда собираются с включенной версией `libfdt`, поэтому в настоящее время невозможно собрать U-Boot с системной `libfdt`.

LTO

U-Boot поддерживает link-time optimisation, которая может уменьшить размер скомпилированных двоичных файлов, особенно при использовании SPL.

В настоящее время эта функция может быть включена на платах ARM путём добавления `CONFIG_LTO=y` в файл `defconfig`.

Однако в таком случае загрузчик будет собираться несколько медленнее, чем без LTO.

Установка

Процесс установки U-Boot специфичен для каждого компьютера. Обратитесь к документации U-Boot за получением конкретных данных.

Создание прочих загрузочных файлов

Теперь нужно закончить создание ряда файлов, которые используются, в первую очередь, при загрузке дистрибутива.

Загрузочные скрипты

Набор скриптов, используемых системой инициализации из BusyBox для запуска/остановки демонов и иных программ во время запуска/выключения LFA.

- **Версия:** 1.0
- **Домашняя страница:** https://github.com/Linux-for-ARM/lfa-bootscripts/
- **Время сборки:** 0.1 ОВС

---

inittab

Для начала создайте файл `/etc/inittab`, отвечающий за такие вещи как зарузка системы, её выключение и обработки поведения при нажатии комбинации `Ctrl`+`Alt`+`Del`.

```
cat >> $LFA_SYS/etc/inittab << "EOF"
# Begin /etc/inittab

::sysinit:/etc/rc.d/startup

tty1::respawn:/sbin/getty 38400 tty1
tty2::respawn:/sbin/getty 38400 tty2
tty3::respawn:/sbin/getty 38400 tty3
tty4::respawn:/sbin/getty 38400 tty4
tty5::respawn:/sbin/getty 38400 tty5
tty6::respawn:/sbin/getty 38400 tty6

::shutdown:/etc/rc.d/shutdown
::ctrlaltdel:/sbin/reboot

# End /etc/inittab
EOF
```

Скрипты

```
make DESTDIR=$LFA_SYS
```

Смотрите также:

- Заметки об ОС Linux. Часть 3. Система инициализации;
- **SysV init must die** (https://busybox.net/).

Создание uInitrd

> ⚠ **Внимание**
>
> Это заготовка страницы. Здесь приведены общие инструкции, тестирование которых не производилось. Окончательная версия страницы войдёт во вторую версию руководства LFA.

Загрузчик U-Boot требует наличие образа `uImage`. Сначала создайте директорию, в которой будет ряд файлов из собранной системы (`$LFA_SYS`):

```
cd $LFA

mkdir LFA-uImage
cd LFA-uImage

cp -rv $LFA_SYS/{bin,dev,etc,lib,proc,root,sbin,srv,sys,tmp,var} .
find . | cpio -H newc -ov --owner root:root > ../initramfs.cpio

cd ..
gzip initramfs.cpio
```

Создайте образ `uImage` с помощью программы `mkimage`:

```
mkimage -A arm64 -T ramdisk -n uInitrd -d initramfs.cpio.gz uInitrd
```

И скопируйте его в `$LFA_SYS/boot`:

```
cp -v uInitrd $LFA_SYS/boot
```

Смотрите также:

- **Image vs zImage vs uImage** (https://stackoverflow.com/);

- **U-Boot new uImage source file format (bindings definition)**;

Создание boot.scr

> ⚠ **Внимание**
>
> Это заготовка страницы. Здесь приведены общие инструкции, тестирование которых не производилось. Окончательная версия страницы войдёт во вторую версию руководства LFA.

В данном пункте приведены сведения о создании файла `boot.scr` для эмулятора QEMU и для реального оборудования. `boot.scr` — это загрузочный скрипт системы, предназначенный для U-Boot. Впоследствии мы скомпилируем его в `boot.cmd`, который и будет читать и исполнять загрузчик.

Смотрите также:

- **How to boot Linux kernel from u-boot?** (https://stackoverflow.com)
- **How U-boot loads Linux kernel** (https://krinkinmu.github.io)
- **U-boot boot script** (https://krinkinmu.github.io)

Создание `boot.scr` для QEMU

Создайте файл `$LFA_SYS/boot/boot.cmd`:

```
cat > $LFA_SYS/boot/boot.cmd << "EOF"
# higher load address; the default causes the initrd to be
overwritten when the bzImage is unpacked....
#setenv ramdisk_addr_r 0x8000000

echo "KERNEL LOAD ADDRESS: kernel_addr_r : ${kernel_addr_r}"
echo "INITRD LOAD ADDRESS: ramdisk_addr_r: ${ramdisk_addr_r}"
echo "FDT LOAD ADDRESS   : fdt_addr      : ${fdt_addr}"

# /vmlinuz are standard LFA symlink to the "latest installed kernel"
load ${devtype} ${devnum}:${distro_bootpart} ${kernel_addr_r}
/vmlinuz

#booti ${kernel_addr_r} ${ramdisk_addr_r} ${fdt_addr}
booti ${kernel_addr_r}
EOF
```

Скомпилируйте этот файл:

```
mkimage -C none \
  -A arm \
  -T script \
  -d $LFA_SYS/boot/boot.cmd \
  $LFA_SYS/boot/boot.scr
```

Для реального оборудования

Сборка образа и запуск системы

Теперь всё готово для сборки `*.img`-образа с вашей системой LFA. Этот образ в будущем можно будет записать на SD-карту или иной носитель, с которого поддерживается загрузка на компьютере, для которого вы собирали систему.

Общий процесс

В общем случае вам нужно собрать три образа:

1. **Заголовок**, в который будет записан загрузчик U-Boot;
2. **Системный раздел**, куда будет скопирована собранная система;
3. **Общий образ**, объединяющий заголовочный и системный разделы вместе.

Общий образ повторяет структуру разделов, которая в итоге будет получена после записи этого образа на SD- или иной накопитель, с которого будет производиться загрузка. Однако важно отметить, что это не единственный верный путь для создания готового образа системы — как минимум из-за того, что загрузчик системы может располагаться не просто на другом разделе, он может быть расположен на другом накопителе вообще (некоторые материнские платы оснащены SPI NOR Flash небольшого объёма, которого достаточно для установки туда U-Boot или иной микропрограммы). Конкретный алгоритм действий по созданию готового образа системы, ровно как и его структура и принципы его создания, являются сугубо индивидуальными и подбираются в зависимости от оборудования и задач сборщика. Ниже представлены общие инструкции для большинства материнских плат по типу Raspberry Pi, Orange Pi и подобных.

Смотрите также:

Если во время сборки образа у вас возникли вопросы и недопонимание, вы можете обратиться к проекту **Armbian**, формирующему готовые `img`-образы Linux. Скачайте такой образ и изучите его с помощью `fdisk` или `cfdisk`. Возможно, это натолкнёт вас на дальнейшие действия.

Сборка образа для QEMU

Для того, чтобы запустить собранную систему в эмуляторе QEMU, нам потребуется два файла: файл с кодом загрузчика U-Boot и файл с образом системы. Это отличается от запуска системы на реальном оборудовании, поскольку для реального компьютера обычно загрузчик и собранная ОС содержатся в одном файле.

Создание образа

С помощью `dd` создайте образ нужного вам размера (не меньше объёма, занимаемого собранной системой LFA, расположенной в `$LFA_SYS`).

```
dd if=/dev/zero bs=1M count=512 of=rootfs.img
```

Здесь размер образа составляет 512 МБ. В зависимости от требований и объёма собранной системы установите вместо 512 своё значение.

Создайте файловую систему внутри образа:

```
mkfs.ext4 -L BOOT \
  -O ^metadata_csum -F \
  -b 4096 \
  -E stride=2,stripe-width=1024 \
  -L rootfs rootfs.img
```

Копирование файлов

Теперь вам необходимо смонтировать созданный выше образ и скопировать в него файлы собранной системы (саму систему, ядро, Devicetree и сценарий загрузки):

```
mkdir -pv /tmp/lfa_rootfs
sudo mount -v rootfs.img /tmp/lfa_rootfs

sudo cp -rfv $LFA_SYS/* /tmp/lfa_rootfs
sync
```

> Все действия здесь выполняются от имени пользователя `lfa`. Поскольку здесь используется программа `sudo`, вам нужно добавить пользователя `lfa` в группу `wheel`.

После копирования файлов размонтируйте образ:

```
sudo umount /tmp/lfa_rootfs
```

Сборка образа для Allwinner

Данный раздел содержит инструкции по сборке загрузочного образа для плат, оснащённых Allwinner SoC.

Создание базовых файлов

Здесь вам нужно создать ряд образов (образ с загрузчиком и образ с собранной системой), которые потом будут «объединены» в один большой образ, пригодный для записи на загрузочный носитель и дальнейшей эксплуатации.

Создание образа с загрузчиком U-Boot

> ⚠ **Внимание**
>
> Обратите внимание на то, что в данном разделе предполагается, что вы запишете собранный образ на SD-карту. Тем не менее, некоторые платы с Allwinner SoC оснащены встроенной eMMC- или SPI NOR памятью, куда также можно установить загрузчик. Тем не менее, эти два варианта в руководстве не рассматриваются. Если вам нужно установить загрузчик на eMMC или SPI NOR Flash, воспользуйтесь сведениями из **документации U-Boot**.

Сначала создайте заголовок размером 2 Мб. Этот заголовок будет содержать скомпилированный ранее загрузчик.

```
dd if=/dev/zero bs=1M count=2 of=bootloader.img
```

Поскольку все системы на кристалле Allwinner пытаются найти загрузочный код в секторе № 256 (128 Кб) SD-карты, подключенной к

первому MMC-контроллеру, вам нужно записать собранный файл загрузчика по этому смещению в образ `bootloader.img`:

```
dd if=$LFA/bootloader.bin \
  conv=notrunc seek=128 bs=1k \
  of=bootloader.img
```

> ⚠ **Внимание**
>
> Вне зависимости от SoC (Allwinner, Broadcom или Rockchip), для которого вы собирали U-Boot, он был сохранён в файл `$LFA/bootloader.bin`. Так что имя файла в аргументе `dd if=...` правильное.

> Если вы собирали систему для *старых* Allwinner SoC (выпущенных до 2013 года), то эти SoC ищут загрузчик в секторе 16 (8 Кб). Для использования загрузчика U-Boot на старых SoC просто замените `seek=128` на `seek=8`.

Создание образа с базовой ОС

С помощью `dd` создайте образ нужного вам размера (не меньше объёма, занимаемого собранной системой LFA, расположенной в `$LFA_SYS`).

```
dd if=/dev/zero bs=1M count=512 of=rootfs.img
```

Здесь размер образа составляет 512 МБ. В зависимости от требований и объёма собранной системы установите вместо 512 своё значение.

Создайте файловую систему внутри образа:

```
mkfs.ext4 -L BOOT \
  -O ^metadata_csum -F \
  -b 4096 \
  -E stride=2,stripe-width=1024 \
  -L rootfs rootfs.img
```

Копирование файлов

Смонтируйте полученный образ и скопируйте в неё файлы нашей системы (саму систему, ядро, Devicetree и сценарий загрузки):

```
mkdir -pv /tmp/lfa_rootfs
sudo mount -v rootfs.img /tmp/lfa_rootfs

sudo cp -rfv $LFA_SYS/* /tmp/lfa_rootfs
sync
```

> Все действия здесь выполняются от имени пользователя `lfa`. Поскольку здесь используется программа `sudo`, вам нужно добавить пользователя `lfa` в группу `wheel` или `sudo`.

После копирования файлов размонтируйте образ:

```
sudo umount /tmp/lfa_rootfs
```

Создание окончательного образа

Объедините два образа в один:

```
dd if=rootfs.img conv=notrunc oflag=append bs=1M seek=2
of=bootloader.img
```

Создание таблицы разделов

Созданный нами образ нерабочий, поскольку ещё не содержит таблицу разделов. Создайте её с помощью `fdisk`:

```
fdisk bootloader.img
o
n
p
1
4096
+512M

w
```

> **Значения новых команд:**
>
> После начала редактирования мы вводим команду `o`, чтобы создать пустую таблицу разделов MBR. Затем командой `n` создаём новый раздел. Выбираем тип, номер и первый сектор. Дело в том, что размер сектора равен 512 байт, т.е. 1 Кб равен двум секторам. Размер заголовка 2 Мб, т.е. 2048 Кб или 4096 секоторов. Размер раздела можно указать в Мб. Сохранение и выход командой `w`.

Переименуйте файл `bootloader.img`, дав ему более логичное и подходящее имя:

```
mv -v bootloader.img lfa-2.0.img
```

Сжатие образа

При необходимости вы можете сжать образ с помощью `xz` или любого другого архиватора, который вам больше нравится. Я часто видел `img`-образы, сжатые с помощью `xz`, поэтому использую его:

```
xz lfa-2.0.img
```

Смотрите также:

- **How to prepare a SD card?** (https://docs.armbian.com/).
- **Allwinner SoC based boards** (сборка загрузчика для плат на базе Allwinner SoC).

Сборка образа для Broadcom

Данный раздел содержит инструкции по использованию U-Boot в SoC Broadcom 7445 и 7260 в качестве загрузчика третьего порядка, запускаемого загрузчиком BOLT компании Broadcom. BOLT загружает U-Boot как двоичный ELF-файл.

Запуск

Чтобы указать U-Boot, какой последовательный порт использовать для консоли, установите параметр `stdout-path` в узле `/chosen` дерева устройств, сгенерированного BOLT. Например:

```
BOLT> dt add prop chosen stdout-path s serial0:115200n8
```

Запишите двоичный файл `$LFA/bootloader.bin` в память платы, затем вызовите его из BOLT. Например:

```
BOLT> boot -bsu -elf flash0.u-boot1
```

Предполагается, что I-кеш и D-кеш уже включены при входе в U-Boot.

Запись образа системы

Создание образа с базовой ОС

С помощью `dd` создайте образ нужного вам размера (не меньше объёма, занимаемого собранной системой LFA, расположенной в `$LFA_SYS`).

```
dd if=/dev/zero bs=1M count=512 of=rootfs.img
```

Здесь размер образа составляет 512 МБ. В зависимости от требований и объёма собранной системы установите вместо 512 своё значение.

Создайте файловую систему внутри образа:

```
mkfs.ext4 -L BOOT \
  -O ^metadata_csum -F \
  -b 4096 \
  -E stride=2,stripe-width=1024 \
  -L rootfs rootfs.img
```

Копирование файлов

Смонтируйте полученный образ и скопируйте в неё файлы нашей системы (саму систему, ядро, Devicetree и сценарий загрузки):

```
mkdir -pv /tmp/lfa_rootfs
sudo mount -v rootfs.img /tmp/lfa_rootfs

sudo cp -rfv $LFA_SYS/* /tmp/lfa_rootfs
sync
```

> Все действия здесь выполняются от имени пользователя `lfa`. Поскольку здесь используется программа `sudo`, вам нужно добавить пользователя `lfa` в группу `wheel` или `sudo`.

После копирования файлов размонтируйте образ:

```
sudo umount /tmp/lfa_rootfs
```

Теперь вам остаётся записать образ на SD- или иной носитель, с которого может загружаться компьютер.

> **Смотрите также:**

- **BCM7445 and BCM7260** (https://docs.u-boot.org/);
- **Broadcom**;

Сборка образа для Rockchip

Данный раздел содержит инструкции по сборке загрузочного образа для плат, оснащённых Rockchip SoC.

Создание базовых файлов

Здесь вам нужно создать ряд образов (образ с загрузчиком и образ с собранной системой), которые потом будут «объединены» в один большой образ, пригодный для записи на загрузочный носитель и дальнейшей эксплуатации.

Создание образа с загрузчиком U-Boot

> ⚠ **Внимание**
>
> Обратите внимание на то, что в данном разделе предполагается, что вы запишете собранный образ на SD-карту. Тем не менее, некоторые платы с Rockchip SoC оснащены встроенной eMMC- или SPI NOR памятью, куда также можно установить загрузчик. Тем не менее, эти два варианта в руководстве не рассматриваются. Если вам нужно установить загрузчик на eMMC или SPI NOR Flash, воспользуйтесь сведениями из **документации U-Boot**.

Сначала создайте заголовок размером 2 Мб. Этот заголовок будет содержать скомпилированный ранее загрузчик.

```
dd if=/dev/zero bs=1M count=2 of=bootloader.img
```

Запись загрузчика на SD-

Запишите скомпилированный файл загрузчика (применимо ко всем платформам Rockchip кроме rk3128, которая не использует SPL):

```
dd if=$LFA/bootloader.bin \
  seek=64 of=bootloader.img
```

> ⚠ **Внимание**
>
> Вне зависимости от SoC (Allwinner, Broadcom или Rockchip), для которого вы собирали U-Boot, он был сохранён в файл `$LFA/bootloader.bin`. Так что имя файла в аргументе `dd if=...` правильное.

Создание образа с базовой ОС

С помощью `dd` создайте образ нужного вам размера (не меньше объёма, занимаемого собранной системой LFA, расположенной в `$LFA_SYS`).

```
dd if=/dev/zero bs=1M count=512 of=rootfs.img
```

Здесь размер образа составляет 512 МБ. В зависимости от требований и объёма собранной системы установите вместо 512 своё значение.

Создайте файловую систему внутри образа:

```
mkfs.ext4 -L BOOT \
  -O ^metadata_csum -F \
  -b 4096 \
  -E stride=2,stripe-width=1024 \
  -L rootfs rootfs.img
```

Копирование файлов

Смонтируйте полученный образ и скопируйте в неё файлы нашей системы (саму систему, ядро, Devicetree и сценарий загрузки):

```
mkdir -pv /tmp/lfa_rootfs
sudo mount -v rootfs.img /tmp/lfa_rootfs

sudo cp -rfv $LFA_SYS/* /tmp/lfa_rootfs
sync
```

> Все действия здесь выполняются от имени пользователя `lfa`. Поскольку здесь используется программа `sudo`, вам нужно добавить пользователя `lfa` в группу `wheel` или `sudo`.

После копирования файлов размонтируйте образ:

```
sudo umount /tmp/lfa_rootfs
```

Создание окончательного образа

Объедините два образа в один:

```
dd if=rootfs.img conv=notrunc oflag=append bs=1M seek=2
of=bootloader.img
```

Создание таблицы разделов

Созданный нами образ нерабочий, поскольку ещё не содержит таблицу разделов. Создайте её с помощью `fdisk`:

```
fdisk bootloader.img
o
n
p
1
4096
+512M

w
```

Значения новых команд:

После начала редактирования мы вводим команду `o`, чтобы создать пустую таблицу разделов MBR. Затем командой `n` создаём новый раздел. Выбираем тип, номер и первый сектор. Дело в том, что размер сектора равен 512 байт, т.е. 1 Кб равен двум секторам. Размер заголовка 2 Мб, т.е. 2048 Кб или 4096 секоторов. Размер раздела можно указать в Мб. Сохранение и выход командой `w`.

Переименуйте файл `bootloader.img`, дав ему более логичное и подходящее имя:

```
mv -v bootloader.img lfa-2.0.img
```

Сжатие образа

При необходимости вы можете сжать образ с помощью `xz` или любого другого архиватора, который вам больше нравится. Я часто видел `img`-образы, сжатые с помощью `xz`, поэтому использую его:

```
xz lfa-2.0.img
```

Сборка образа (для остальных)

> ⚠ **Внимание**
>
> Это заготовка страницы. Здесь приведены общие инструкции, тестирование которых не производилось. Сведения здесь приведены только для формирования у читателя общего представления о процессе сборки загрузочного образа LFA в том случае, если он собирал LFA для тех SoC, которые не были описаны в этом руководстве. Для получения информации о структуре загрузочного носителя (информация о разделах, их форматах и смещениях, а также иные данные), смотрите **документацию U-Boot** для каждого поддерживаемого SoC.
>
> Инструкции отсюда готовились для второй версии LFA. В более новых версиях руководства работоспособность и актуальность комманд отсюда не гарантируется.

Создание базовых файлов

Здесь вам нужно создать ряд образов (образ с загрузчиком и образ с собранной системой), которые потом будут «объединены» в один большой образ, пригодный для записи на загрузочный носитель и дальнейшей эксплуатации.

Создание образа с загрузчиком U-Boot

> ⚠ **Важно**
>
> Далее вам предлагается создать файл `bootloader.img`. Создавайте его только в том случае, если собирали систему для Allwinner, Broadcom или Rockchip SoC. Если вы собирали загрузчик для запуска в эмуляторе U-Boot, вам нужно пропустить действия из этого подпункта «Создание образа с загрузчиком U-Boot».

Сначала создадим заголовок размером 2 Мб:

```
dd if=/dev/zero bs=1M count=2 of=bootloader.img
```

И копируем сохранённый образ U-Boot по смещению 128:

```
dd if=$LFA/bootloader.bin \
  conv=notrunc seek=128 \
  of=bootloader.img
```

> ⚠ **Внимание**
>
> Вне зависимости от SoC (Allwinner, Broadcom или Rockchip), для которого вы собирали U-Boot, он был сохранён в файл `$LFA/bootloader.bin`. Так что имя файла в аргументе `dd if=...` правильное.

Создание образа с базовой ОС

Теперь нужно создать образ, в котором будут содержаться файлы собранной системы:

```
dd if=/dev/zero bs=1M count=512 of=rootfs.img
```

Здесь размер образа составляет 512 Мб. В зависимости от требований и объёма собранной системы установите вместо `512` своё значение.

Создайте файловую систему на этом разделе:

```
mkfs.ext4 -L BOOT \
  -O ^metadata_csum -F \
  -b 4096 \
  -E stride=2,stripe-width=1024 \
  -L rootfs rootfs.img
```

Копирование файлов

Смонтируйте полученный образ и скопируйте в неё файлы нашей системы (саму систему, ядро, Devicetree и сценарий загрузки):

```
mkdir -pv /tmp/lfa_rootfs
sudo mount -v rootfs.img /tmp/lfa_rootfs

sudo cp -rfv $LFA_SYS/* /tmp/lfa_rootfs
sync
```

> ⚠ **Внимание**
>
> Все действия здесь выполняются от имени пользователя `lfa`. Поскольку здесь используется программа `sudo`, вам нужно добавить пользователя `lfa` в группу `wheel`.

После копирования файлов размонтируйте образ:

```
sudo umount /tmp/lfa_rootfs
```

Создание окончательного образа

Объедините два образа в один:

```
dd if=rootfs.img conv=notrunc oflag=append bs=1M seek=2
of=bootloader.img
```

Если вы не генерировали образ `bootloader.img` (т.е. собираете систему для эмуляции в QEMU), то этот пункт вам нужно пропустить. Вместо него переименуйте образ `rootfs.img` в `bootloader.img`:

```
mv -v rootfs.img bootloader.img
```

Создание таблицы разделов

Созданный нами образ нерабочий, поскольку ещё не содержит таблицу разделов. Создайте её с помощью `fdisk`:

```
fdisk bootloader.img
o
n
p
1
4096
+512M

w
```

Значения новых команд:

После начала редактирования мы вводим команду `o`, чтобы создать пустую таблицу разделов MBR. Затем командой `n` создаём новый раздел. Выбираем тип, номер и первый сектор. Дело в том, что размер сектора равен 512 байт, т.е. 1 Кб равен двум секторам. Размер

заголовка 2 Мб, т.е. 2048 Кб или 4096 секоторов. Размер раздела можно указать в Мб. Сохранение и выход командой `w`.

Переименуйте файл `bootloader.img`, дав ему более логичное и подходящее имя:

```
mv -v bootloader.img lfa-2.0.img
```

Сжатие образа

При необходимости вы можете сжать образ с помощью `xz` или любого другого архиватора, который вам больше нравится. Я часто видел `img`-образы, сжатые с помощью `xz`, поэтому использую его:

```
xz lfa-2.0.img
```

Запись образа на SD-карту

Теперь вы можете записать полученный образ на SD-карту. Для этого можете использовать `dd`:

```
sudo dd if=lfa-2.0.img of=/dev/sdX
```

где `X` - буква (`a`, `b`, `c`, etc.) устройства, на которое будет производиться запись образа, например, `/dev/sdc`. Обратите внимание на то, что в некоторых случаях вместо `sdX` имя устройства может быть и `mmcblkX`.

Также для записи можете использовать программу Balena Etcher.

Смотрите также:

- **How to prepare a SD card?** (https://docs.armbian.com/).

Запуск в QEMU

Ранее вы сохранили образ U-Boot в файле `$LFA/bootloader.bin`. Вам потребуется сохранить этот файл и образ системы отдельно друг от друга для возможности запуска собранной системы LFA.

Минимальная команда для запуска эмулятора QEMU с загрузчиком U-Boot выглядит так:

- для AArch64:

```
qemu-system-aarch64 -machine virt \
  -nographic \
  -cpu cortex-a57 \
  -bios $LFA/bootloader.bin
```

- для других архитектур семейства ARM:

```
qemu-system-arm -machine virt \
  -nographic \
  -bios $LFA/bootloader.bin
```

Объяснение новых значений:

`-nographic` — обеспечивает вывод данных в терминал;

`-cpu cortex-a57` — по какой-то странной причине программе `qemu-system-aarch64` необходимо явно указать использование 64-битного процессора, иначе эмулятор запустится в 32-битном режиме.

`-bios $LFA/bootloader.bin` — использовать скомпилированный загрузчик U-Boot.

Вы также можете создать образ, в котором будут храниться сохранённые переменные окружения U-Boot. Это можно сделать, выполнив следующие действия:

- Создать образ `envstore.img` с помощью `qemu-img`:

```
qemu-img create -f raw $LFA/envstore.img 64M
```

- Передать команде для запуска виртуальной машины параметр `pflash drive`:

```
-drive if=pflash,format=raw,index=1,file=$LFA/envstore.img
```

Эмуляция блочных устройств

Поскольку командой выше вы только запустили загрузчик. Однако помимо загрузчика нужно передать аргументы, указывающие на образ операционной системы (генерацию такого образа см. далее в руководстве).

QEMU может эмулировать обычные блочные устройства. Добавьте следующие параметры в команду `qemu-system-aarch64` (или `qemu-system-arm`):

- MMC:

```
-device sdhci-pci,sd-spec-version=3 \
-drive if=none,file=disk.img,format=raw,id=MMC1 \
-device sd-card,drive=MMC1
```

- NVMe:

```
-drive if=none,file=disk.img,format=raw,id=NVME1 \
-device nvme,drive=NVME1,serial=nvme-1
```

- SATA:

```
-device ahci,id=ahci0 \
-drive if=none,file=disk.img,format=raw,id=SATA1 \
-device ide-hd,bus=ahci0.0,drive=SATA1
```

- USB:

```
-device qemu-xhci \
-drive if=none,file=disk.img,format=raw,id=USB1 \
-device usb-storage,drive=USB1
```

Во всех вариантах аргументов командной строки из этого пункта замените имя `disk.img` на название сгенерированного образа с системой. Подробнее о сборке образа с системой см. в пункте №8.1 «Сборка образа для QEMU».

Дополнительные устройства

Возможно, для работы вашей системы понадобится эмуляция дополнительных устройств. Ниже представлены аргументы для команды запуска виртуальной машины QEMU, включающие эмуляцию некоторых из них.

- Чтобы добавить видеоконтроллер, удалите аргумент `-nographic`, заменив его, например, следующим:

```
-serial stdio -device VGA
```

- Для добавления генератора псевдослучайных чисел, добавьте аргумент:

```
-device virtio-rng-pci
```

Ещё немного про виртуализацию

QEMU для ARM поддерживает специальную виртуальную машину `virt`, предоставляющую следующие базовые функции:

- Свободно настраиваемое количество ядер процессора;
- U-Boot, который загружается и выполняется по адресу `0x0`;
- Сгенерированный блоб дерева устройств, помещённый в начало оперативной памяти;
- Свободно конфигурируемый объём ОЗУ, описываемый в `*.dtb`;
- Последовательный порт PL011, описываемый в `*.dtb`;
- Таймер (для архитектуры ARMv7/ARMv8);
- PSCI для перезагрузки системы;
- Общий хост-контроллер PCI на базе ECAM, описываемый в `*.dtb`;

Кроме того, к PCI шине можно подключить ряд дополнительных периферийных устройств.

См. раздел **Devicetree in QEMU** в документации U-Boot для получения информации о том, как увидеть дерево устройств, фактически сгенерированное QEMU.

Запись на SD-карту или иной носитель

Для записи img-образов можете использовать любую программу побайтового копирования. Например, входящую в состав пакета `coreutils` (а значит, присутствующую во всех дистрибутивах Linux) программу `dd`. Пример её использования:

```
sudo dd if=lfa-2.0.img of=/dev/mmcblk0p1 status=progress
```

Значения новых параметров:

`if=lfa-2.0.img` — параметр `if` указывает исходный файл, который будет записан.

`of=/dev/mmcblk0p1` — параметр `of` указывает конечный файл, *куда* будет записан исходный. Замените `mmcblk0p1` на нужное вам устройство. Список устройств (носителей информации) можете получить с помощью команды `lsblk` (на некоторых системах требует установки).

`status=progress` — по умолчанию `dd` не выводит прогресс записи, из-за чего у новичков часто возникает ощущение, что программа зависла. Указание прогресса записи покажет, сколько байт было записано в каждый конкретный момент времени, и сколько байт осталось записать.

Кроме `dd` вы можете использовать и другие программы, например, **Impression**. Она имеет простой графический интерфейс, соответствующий правилам оформления GNOME HIG.

Для записи img-образов, сжатых с помощью `xz` (или иного алгоритма), хорошо себя зарекомендовала программа **Balena Etcher**. В принципе, вы можете использовать её и для записи несжатых img-образов.

Обратите внимание на то, что записывать полученный образ вы должны на поддерживаемый вашим компьютером носитель! Так, например, если компьютер, для которого собиралась LFA, поддерживает загрузку только с SD-карт, то запись должна производиться именно на SD-карту.

Что далее?

После того, как вы собрали систему, вы можете обратиться к руководству BLFS, в котором предоставлена информация о сборке дополнительного ПО. Несмотря на то, что оно не совсем совместимо с LFA (вам придётся оптимизировать команды для сборки оттуда для использования кросс-компилятора LFA), данные оттуда вам всё-таки смогут пригодиться.

Не забывайте посещать наш канал в Telegram, чтобы быть в курсе последних изменений в LFA, а также Telegram-чат если у вас возникли вопросы или ошибки, либо если вам требуется обратная связь с разработчиком LFA.

Можете посетить проект The Linux Documentation Project, содержащий большое число man-страниц, всевозможных HOWTO и прочую документацию.

Рекомендуем также посетить следующие сайты:

- Опыт создания сборок Linux под одноплатники с поддержкой обновлений;
- Немного о ARM Security Extensions (aka ARM TrustZone);
- [Сборка системы для Orange Pi 5] Orange Pi 5 (как настоящий...);
- LFS для SBC;
- Сборка прошивки из исходников для Orange PI i96(Orange PI 2g-iot);
- Документация загрузчика U-Boot;
- LFS Hints

▼ Ну и конечно же вы можете отблагодарить автора за проделанную работу...

... отправив донат на карту Сбербанка:

2202206252335406

Вспомогательные материалы

Это необязательная часть, служащая дополнительным источником знаний. В некоторых случаях здесь могут быть приведены решения часто возникающих проблем и важные заметки по процессу сборки LFA.

В задачи этого раздела входит аккумулирование сведений как о процессе сборки своих систем, использующих ядро Linux, так и об ARM-компьютеров, для которых это руководство и предназначено.

От версии к версии этот раздел дополняется и актуализируется.

Процессоры ARM

ARM - семейство описаний и готовых топологий 32- и 64-битных микропроцессоров. К значимым семействам процессоров относятся ARM7, ARM9, ARM11 и Cortex. ARM-процессоры имеют низкое электропотребление, поэтому часто используются во встраиваемых системах и мобильных устройствах.

ARM7 (60-72 МГц)

Процессоры на этой архитектуре предназначены для мобильных телефонов и встраиваемых систем. Сейчас активно вытесняется семейством Cortex.

ARM9, ARM11 (до 1 ГГц)

Предназначено для мобильных устройств, КПК и встраиваемых систем высокой производительности.

Cortex-A

Пришли на смену ARM9 и ARM11.

Cortex-M

Пришли на смену ARM7. Предназначены для встраиваемых систем низкой производительности.

Архитектура

Для ARM-процессоров, начиная с ARMv7, были определены 3 профиля:

- **A** (Application) — для устройств, требующих высокой производительности;
- **R** (Real Time) — для приложений, работающих в реальном времени;
- **M** (Microcontroller) — для микроконтроллеров и встраиваемых устройств;

Система на кристалле (SoC)

Нас больше интересует не сама архитектура ARM, а *системы на кристалле* (СнК, SoC), использующие процессоры на этой архитектуре. Такие системы размещены на одной интегральной схеме и выполняют роль сразу нескольких устройств. Из-за небольших размеров таких систем они приобрели популярность в мобильной технике, фотоаппаратах, планшетных компьютерах, электронных книгах, умных часах и т.д., а также в одноплатных компьютерах, например, в Orange Pi, Raspberry Pi, Repka Pi и др.

Как правило, СнК содержат один или несколько микропроцессоров, блок памяти (ПЗУ, ОЗУ и т.д.), регуляторы напряжения и стабилизаторы питания и пр.

СнК потребляют меньше энергии, стоят дешевле и работают надёжнее, чем наборы микросхем с той же функциональностью. Меньшее количество корпусов упрощает монтаж. Однако проектирование и отладка одной большой СнК сложнее и дороже, чем проектирование и отладка серии маленьких микросхем.

Заметки об ОС Linux

В данном цикле обзорных статей будут рассмотрен ряд вопросов об операционных системах, использующих ядро Linux. Например, будут описаны основные компоненты, из которых состоят эти ОС, их принцип работы и предназначение. В данных статьях по большей мепе интересует *что* происходит, а не *как*, т.е. здесь не будет большого числа технических подробностей и мельчайших деталей, а просто краткое обзорное описание основных моментов, касаемых работы описываемых здесь ОС.

Заметки об ОС Linux. Часть 1. Обзор

Операционные системы

Определение, что же такое ОС, весьма объёмно и всеобъемлюще. Но вкратце можно сказать о том, что операционная система — это программное обеспечение, управляющее аппаратным обеспечением компьютера и предоставляющее какие-либо абстракции для остального прикладного ПО, которое запускается на компьютере. Примером таких абстракций можно назвать файл на жёстком диске: компьютер сам по себе не знает, что такое файл, ровно как и на жёстком диске или ином носителе информации у нас содержится последовательность данных, которую мы не можем назвать «файлом» в привычном для нас смысле этого слова. Уже операционная система предоставляет пользователю (и программисту, и программам, запускающимся на этой ОС) такую абстракцию, которая «преобразует» эти данные на жёстком диске в привычные нам файлы и папки.

В широком смысле под операционной системой подразумевается совокупность ядра и работающих поверх него программ и утилит.

Встраиваемые (embedded) системы

LFA основана на руководстве CLFS Embedded, в котором приведены сведения по сборке встраиваемой системы для архитектуры ARM. Поэтому неудивительно, что прямо сейчас описываются именно встраиваемые операционные системы. Встраиваемая ОС (Embedded OS) — такая ОС, которая работает, будучи встроенной непосредственно в девайс, которым она управляет. Т.е. это часть специализированного устройства или платформы, такой как, например устройство бытовой электроники или часть из системы умного дома.

Встроенный компьютер отличается от персонального тем, что встроенная система как правило спроектирована для одной или нескольких конкретных целей, в то время как ПК предназначены для выполнения широкого круга задач. Встроенный компьютер может иметь минимально возможную для выполнения конкретной задачи производительностью, что обеспечивает лёгкую и эффективную компьютерную платформу.

Семейство Unix

Среди современных операционных систем наиболее распространены два семейства: семейство операционных систем Windows и семейство операционных систем Unix. ОС семейства UNIX нашли применение на суперкомпьютерах, серверах, встраиваемых системах, мобильных устройствах, а с недавнего времени проникают и в сегмент компьютеров для обычных пользователей или офисного планктона.

Unix — это семейство переносимых, многозадачных и многопользовательских систем. Главная особенность таких систем заключается в том, что это *изначально* многопользовательские и многозадачные ОС, а также такие ОС созданы таким образом, чтобы их было легко переносить с одной процессорной архитектуры на другую. Модульная структура ОС этого семейства наряду с поддержкой широкого спектра микропроцессоров и других типов оборудования сделала такие ОС достаточно популярными и во встраиваемых решениях.

К семейству Unix относятся многие ОС, например FreeBSD, OpenBSD, NetBSD, macOS X (версии 10.+), minix и другие. Кроме того, к этому семейству, пусть и косвенно (поскольку они не основаны на исходном коде оригинальной UNIX или какого-то из её ответвлений), относятся и ОС, использующие ядро Linux.

ОС, использующие ядро Linux

Linux — это монолитное ядро операционной системы, распространяемое как свободное ПО на условиях лицензии GNU GPL (кроме несвободных

компонентов, включая драйверы, использующие прошивки, распространяемые под другими лицензиями). Linux можно назвать «сердцем» любой ОС, описываемой в данном руководстве, в том числе это «сердце» и собранного по этому руководству дистрибутива LFA.

Полноценная ОС не может существовать только из одного ядра, тк ей нужны программы, предназначенные для обеспечения работы и взаимодействия с этой ОС: стандартная библиотека языка C (glibc, musl, bionic, etc.), загрузчик ОС (GRUB, systemd-boot, syslinux, U-Boot), система инициализации (SysVInit, OpenRC, Runit, upstart, systemd) и набор стандартных утилит для работы с компонентами этой ОС (работа с файлами, правами доступа, процессами, пользователями, группами пользователей и т.п., обычно такие утилиты называются «coreutils»: GNU coreutils, uutils coreutils, etc.).

Свободное ПО

Операционная система, использующая ядро Linux, а также все её системные компоненты и большинство её пользовательских приложений — свободные программы. Свободное ПО отличается от несвободного тем, что его можно:

- запускать на любом количестве компьютеров без всевозможных лицензионных ограничений;
- распространять бесплатно или за деньги без каких-либо ограничений;

Кроме того, пользователи могут получать исходный код свободного ПО, изучать и модифицировать его, а также распространять модифицированные копии такого ПО.

Разработка ОС с ядром Linux

В отличие от распространённых несвободных ОС вроде Windows или macOS, Linux не имеет географического центра разработки, ровно как нет конкретной фирмы или компании, владеющей Linux, нет и единого

координационного центра разработки. Эта ОС — результат работы тысячи программистов по всему миру[1].

Компоненты ОС с ядром Linux

Ядро Linux

Как упоминалось ранее, ядро Linux — это «сердце» операционной системы, которое, к тому же, соответствует стандарту POSIX и распространяется под лицензией GNU GPLv2. Несмотря на *свободную* лицензию, в составе ядра есть и несвободные компоненты: как правило, это драйверы, использующие несвободные прошивки и иные составляющие.

Linux поддерживает многозадачность, виртуальную память, динамические библиотеки и прочее, что свойственно современным операционным системам. Это монолитное ядро с поддержкой динамически загружаемых модулей. В монолитных ядрах все их компоненты (например, драйверы) как бы встроены в одно большое ядро. Это является противоположностью, например, микроядрам, где очень многие компоненты являются отдельными частями, которые не входят непосредственно в состав ядра (например, это справедливо по отношению к ядру операционной системы Minix).

Из плюсов монолитных ядер можно отметить более прямой доступ к аппаратным средствам, а из минусов: бо́льший размер ядра и бо́льшее потребление оперативной памяти. Другой минус заключается в том, что когда мы подключаем к компьютеру какое-то новое устройство, драйвер которого существует для монолитного ядра, но в данный момент он не «встроен» в это ядро (т.е. ядро не было скомпилировано с поддержкой этого драйвера), то нам придётся пересобирать ядро, включив там поддержку этого драйвера, что иногда бывает не очень удобным вариантом.

В итоге ядро Linux обзавелось механизмом *модулей*, которые устанавливаются в отдельный каталог (например, в `/lib/modules`). Модули мы можем подключать, когда они нам нужны, а когда необходимость их использования сойдёт на нет, их можно выгрузить.

Загрузчик

Задача загрузчика — инициализировать и запустить ядро. На компьютерах с архитектурой ARM он может выполнять несколько больше функций, но в основном всё сводится именно к инициализации и запуску ядра ОС.

Система инициализации

Ядро запускает систему инициализации (`init`). В задачи `init` входит подготовка ОС к дальнейшей работе: инициализация консолей, монтирование виртуальных файловых систем и запуск компонентов системы. Процесс `/sbin/init` работает как демон и имеет `PID` = `1`.

Из основных реализаций системы инициализации в Linux можно отметить systemd, SysVInit, runit и OpenRC. В проекте LFA используется система инициализации из состава BusyBox.

`init` загружается первым процессом (пользовательского режима) после инициализации ядра и отвечает за дальнейшую загрузку системы. Для этого он может либо запускать загрузочные скрипты (написанные, как правило, на BASH или каком-то другом скриптовом языке), либо читает специальные конфигурационные файлы с параметрами загрузки (такое поведение есть, например, у systemd).

Во многих системах Linux (особенно коммерческих) получил наибольшее распространение systemd. Это довольно удобная и функциональная система инициализации, однако некоторые пользователи критикуют её за раздутость, наличие многих ненужных пользователям функций (некоторые компоненты из systemd заменяют `chroot`, `cron` и средства для ведения логов системы), а также несколько более высокое

потребление ресурсов компьютера по сравнению с другими системами инициализации. Лично для меня существенным недостатком [скорее не самого systemd, а ряда программ, «прибитых» к нему] является то, что ряд ПО «прибит» к нему и работа такого ПО на системах с другими инитами может быть несколько затруднительной. Примерами такого ПО можно привести рабочие окружения GNOME и KDE. Тем не менее, systemd — довольно продвинутая система инициализации, которая не даром набрала популярность во многих дистрибутивах Linux.

Стандартная библиотека языка C

Поскольку системные компоненты ОС Linux написаны преимущественно на языке программирования C, в системе присутствует стандартная библиотека этого языка, с которой динамически линкуются программы системы. Эта стандартная библиотека (`libc`) содержит системные вызовы и основные функции, которые используются в программе, например функции для работы с файлами, выделения памяти, форматирования вывода в консоль и т.д.

Существует множество различных стандартных библиотек C, самая распространённая из которых — GNU C Library (`glibc`). Она используется почти во всех известных дистрибутивах GNU/Linux. Однако её часто критикуют за «раздутость» и низкую скорость работы по сравнению с другими библиотеками.

В LFA используется стандартная библиотека C `musl` . Она была написана с нуля, без переиспользования существующего кода. При её написании уделялось внимание эффективной статической линковке программ. Эта стандартная библиотека получила распространение в минималистичных дистрибутивах Linux, например в Alpine Linux, OpenWRT и других. Применение `musl` может быть оправдано в минималистичных и встраиваемых системах, поэтому использование её в LFA в целом является хорошей идеей.

Стандартные утилиты UNIX (coreutils)

coreutils — это набор базовых утилит операционной системы, которые включают в себя программы для управления файлами (создание, удаление, копирование и перемещение файлов и каталогов, создание ссылок и блочных файлов), правами доступа к файлам (изменение прав доступа к файлу и каталогу, смена владельца и группы файла, etc.), проверки контрольных сумм файлов, работы с текстовой информацией и т.д.

Во многих дистрибутивах Linux в качестве реализации стандартных утилит выступает пакет GNU Coreutils, однако кроме него есть и другие. Например, существует реализация coreutils на языке программирования Rust (uutils coreutils). Во встраиваемых системах применяются более простые (а значит и более минималистичные и нетребовательные к ресурсам ПК) реализации, такие как BusyBox и ToyBox, которые объединяют все вышеописанные утилиты в один бинарный файл, имеющий размер от нескольких сотен Кбайт до 2-3 МБайт.

Командная оболочка (shell)

Для того, чтобы пользователь мог взаимодействовать с операционной системой и программами, которые в неё установлены, требуется какая-либо оболочка. Это может быть либо командный интерпретатор типа `sh`, `ash`, `zsh`, `csh` или `bash`, либо графическое пользовательское окружение типа GNOME или KDE.

В командной оболочке пользователь может либо давать команды ОС, либо запускать файлы, в которых содержатся списки этих команд (такие файлы называются скриптами).

Командные оболочки позволяют не только запускать какие-либо программы и передавать этим программам нужные аргументы и значения, но также объявлять свои переменные, в которых будут содержаться определённые значения, использовать циклы, конструкции ветвления, а также использовать другие возможности, которые обычно присущи языкам программирования.

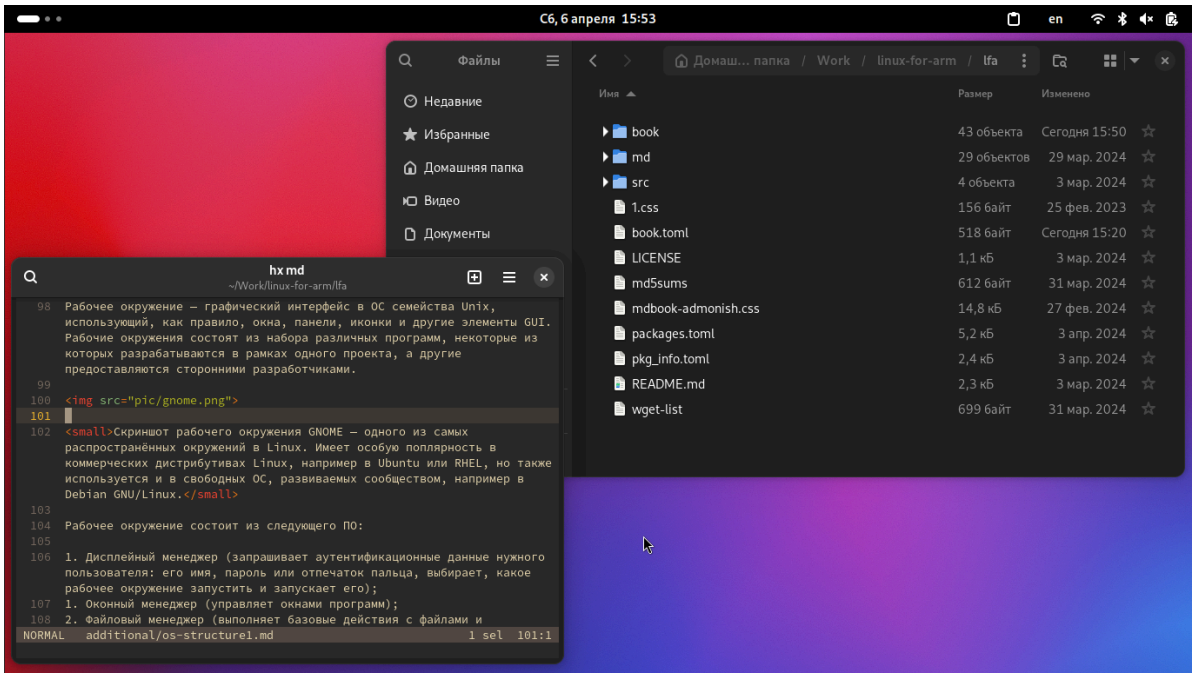
В операционных системах семейства Unix пользователь может использовать ветвления, циклы, функции, переменные, работать с арифметическими выражениями, сравнивать числа и строки, а также работать с текстовой информацией.

Эти возможности командных оболочек повсеместно используются во всех системах Unix, в том числе и дистрибутивах Linux. Начиная с процесса загрузки ОС, заканчивая какими-либо пользовательскими задачами. Преимущества Unix-оболочек используются и в этом руководстве при сборке системы.

Во многих дистрибутивах Linux в качестве командной оболочки используется BASH, однако во всяких минималистичных и/или встраиваемых системах можно встретить что-то вроде `ash` из состава BusyBox — микроверсию оболочки `sh`, обеспечивающую базовую функциональность shell'а. В других ОС семейства Unix можно встретить `csh` или `tcsh` (например, во FreeBSD), либо `zsh` (в macOS).

Рабочее окружение (Desktop Environment)

Рабочее окружение — графический интерфейс в ОС семейства Unix, использующий, как правило, окна, панели, иконки и другие элементы GUI. Рабочие окружения состоят из набора различных программ, некоторые из которых разрабатываются в рамках одного проекта, а другие предоставляются сторонними разработчиками.



Скриншот рабочего окружения GNOME — одного из самых распространённых окружений в Linux. Имеет особую поплярность в коммерческих дистрибутивах Linux, например в Ubuntu или RHEL, но также используется и в свободных ОС, развиваемых сообществом, например в Debian GNU/Linux.

Рабочее окружение состоит из следующего ПО:

1. Дисплейный менеджер (запрашивает аутентификационные данные нужного пользователя: его имя, пароль или отпечаток пальца, выбирает, какое рабочее окружение запустить и запускает его);
2. Оконный менеджер (управляет окнами программ);
3. Файловый менеджер (выполняет базовые действия с файлами и каталогами: просмотр содержимого директорий, переименование, перемещение, копирование, удаление файлов и директорий, создание ссылок и т.д.);
4. Панель, на которой расположены апплеты для взаимодействия с окружением (главное меню со списком приложений, часы, область уведомлений, панель управления, на которой можно изменять т.н. «быстрые параметры»: включение/выключение Wi-Fi, Bluetooth, регулировка звука и яркости экрана, кнопки для выключения/перезагрузки компьютера или выхода из текущей сессии пользователя, etc.);

В Unix традиционно за обеспечение работы графических пользовательских окружений отвечает X Window System. Это сервер, обеспечивающий базовые функции графической среды: отрисовку и работу окон (их перемещение на экране, изменение размера, etc.), взаимодействие с устройствами ввода (мышь и клавиатура) и т.д. За всё остальное отвечают, как правило, оконные менеджеры, работающие поверх X и определяющие интерфейс и взаимодействие с пользователем.

Оконные менеджеры могут использоваться как в составе рабочего окружения, так и отдельно от него.

В последнее время наблюдается тенденция на отказ от X Window System в пользу протокола Wayland. Он обладает некоторыми достоинствами по сравнению с X.org (по крайней мере это не монолитный и запутанный монстр типа X.org, в котором давным-давно не разбираются даже разработчики, а вполне себе элегантная и простая вещь, вести разработку которой в разы легче и куда проще добавлять новые функции). Кроме того, Wayland не имеет присущих X недостатков и уязвимостей.

Вместо оконных менеджеров для Wayland применяются композитные менеджеры. Среди таких композитных менеджеров для Wayland можно отметить:

1. Weston — эталонная реализация композитного менеджера Wayland;
2. Sway — аналог тайлинговых оконных менеджеров типа i3-wm для Wayland;
3. Mutter — из состава рабочего окружения GNOME;
4. KWin — из состава рабочего окружения KDE;

Рассмотрим строение рабочего окружения на примере GNOME Shell:

1. Дисплейный менеджер GDM;
2. Оконный/композитный менеджер Mutter;
3. Файловый менеджер Nautilus;
4. Оболочка GNOME Shell (включает в себя верхнюю панель с переключателем рабочих столов, часы и панель уведомлений, панель быстрых настроек, главное меню, dock со списком запущенных и закреплённых приложений, ряд диалогов и т.д.);

Все компоненты рабочих окружений, как правило, тесно интегрированы друг с другом и составляют целую «экосистему» связанного друг с другом ПО.

Во встраиваемых системах как правило нет графического пользовательского окружения, поскольку такие окружения достаточно тяжёлые для слабого встраиваемого оборудования.

[1] ровно поэтому нет и не может быть никаких «отечественных дистрибутивов Linux» или уж того хуже «отечественных операционных систем» (100% которых — просто дистрибутивы Linux, не более). Вклад российских разработчиков в развитие Linux ровно такой же, как и вклад любого другого разработчика из любого другого государства. Поэтому некорректно называть какой-то дистрибутив российским, французским, китайским и т.д.

Заметки об ОС Linux. Часть 2. Процесс загрузки

Большая часть сведений здесь взята из статьи Загрузка ОС на ARM на Хабре.

> ⚠ **Внимание**
>
> Здесь будут даны сведения о загрузке Linux на ARM. Особенности загрузки других ОС затронуты не будут, поскольку это не тема данного руководства.

Загрузчик ОС

Загрузчик обеспечивает запуск ОС и ряд сервисных функций, например проверку целостности образа ОС перед запуском, обновление ПО, самотестирование и т.д. Очень часто в качестве загрузчика можно встретить U-Boot или иные решения с открытым исходным кодом. Для x86_64 «собратьев» обычно применяются GRUB и systemd-boot.

Таким образом, если очень сильно упростить, то процесс загрузки ОС выглядит следующим образом:

Здесь знаком `//` отмечен момент подачи питания или сброса процессора. Такой простой способ запуска был у некоторых процессоров ARMv7. В более новых версиях процессоров процесс куда сложнее, чем на приведённой схеме.

Схема «Загрузчик» -> «ОС» очень удобна из практических соображений, ведь загрузчик берёт на себя всю низкоуровневую работу:

- Инициализирует память перед запуском ОС и загружает ядро ОС в память
- Инициализирует часть периферии
- Иногда реализует хранение двух образов ОС: текущего и резервного или образа для восстановления

Например, для запуска Linux на ARM загрузчик должен инициализировать память, хотя бы один терминал, загрузить образ ядра и Devicetree в память и передать управление ядру. Всё это описано в **документации** Linux. Код инициализации ядра Linux не будет делать сам то, что должен делать загрузчик.

Рассмотрим работу загрузчика на примере U-Boot:

1. После включения или сброса процессор загружает образ U-Boot в ОЗУ и передаёт управление на первую команду этого образа.
2. U-Boot инициализирует DDRAM.
3. U-Boot инициализирует драйверы загрузочного носителя (ЗН), например eMMC, NAND Flash и т.д.
4. U-Boot читает с ЗН область переменных конфигураций. В конфигурации задан скрипт загрузки, который U-Boot далее исполняет.
5. U-Boot выводит предложение прервать процесс загрузки и сконфигурировать устройство. Если за 2-3 секунды пользователь этого не сделает, запустится загрузочный скрипт.
6. Иногда скрипт начинается с поиска подходящего образа ОС для загрузки на всех доступных носителях. В других случаях ЗН задаётся в скрипте жёстко.
7. Скрипт загружает с ЗН в DDRAM образ ядра (`zImage`) и файл Devicetree с параметрами ядра.

8. Дополнительно скрипт может загрузить в память образ `initrd`.
9. `zImage` состоит из распаковщика и сжатого образа ядра. Распаковщик развёртывает ядро в памяти.
10. Начинается загрузка ОС.

Предзагрузчик

Однко в реальности почти никогда не бывает, чтобы команды загрузчика выполнялись первыми после включения или сброса процессора, тем более на современных ARM-процессорах.

Любое ядро ARM-процессора при сбросе начинает исполнение с адреса 0, где записан вектор `reset`. Старые процессоры буквально начинали загружаться с внешней памяти, отображённой по нулевому адресу, и тогда первая команда процессора была первой командой загрузчика. Однако для такой загрузки подходит только параллельная NOR Flash или ROM. Эти типы памяти работают очень просто: при подаче адреса они выдают данные. Характерный пример параллельной NOR Flash — микросхема BIOS в x86/x86_64 компьютерах.

В современных системах используются другие виды памяти, потому что они дешевле, а их объём больше. Это, например, NAND, eMMC, SPI/QSPI Flash. Эти типы памяти уже не работают по типу «подал адрес — читаешь данные», а значит, что для прямого исполнения команд из них не подходят. Даже для простого чтения нужно написать специальный драйвер, поэтому мы имеем проблему «курицы и яйца»: драйвер нужно откуда-то заранее загрузить.

По этой причине в современные ARM-процессоры интегрировано ПЗУ с предзагрузчиком. ПЗУ отображено в процессоре на адрес 0, и именно с этого адреса начинается исполнение команд процессором.

Задачи предзагрузчика:

1. Определение конфигурации подключенных устройств;
2. Определение загрузочного носителя (ЗН);

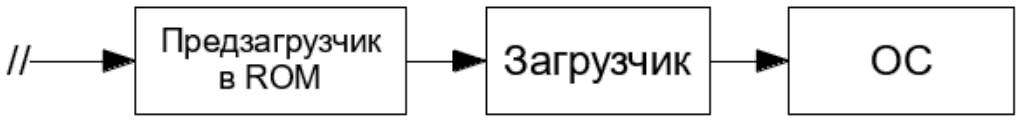
3. Инициализация устройств и ЗН;
4. Чтение загрузчика с ЗН;
5. Передача управления загрузчику.

Конфигурация предзагрузчика обычно устанавливается одним из двух способов:

- Схемотехнически — подключением определённых выводов процессора к земле или шине питания;
- Записывается в однократно программируемую память процессора на этапе производства.

В целом почти всегда есть возможность задать единственный вариант загрузки или основной и несколько альтернативных.

Подобный предзагрузчик устанавливается как в процессорах типа Cortex-A, так и в микроконтроллерах, даже таких маленьких, как Cortex-M0. Вместе с предзагрузчиком процедура запуска ОС выглядит так:



Код предзагрузчика пишется производителем конкретного SoC, а не компанией ARM, является частью SoC как продукта компании-производителя и защищён авторским правом.

Использования предзагрузчика в большинстве сценариев избежать нельзя.

Спойлер

У ARMv8-A есть специальная микропрограмма под названием ARM Trusted Firmware (TF-A). Это системное ПО, отвечающее, например, за управление питанием (PSCI). Этот код можно считать, в какой-то степени, BIOS для ARMv8. У более ранних процессоров (например, ARMv7) такого ПО нет.

TrustZone

В процессоры ARM Cortex-A и Cortex-R встраивается технология **TrustZone**. Эта технология позволяет на аппаратном уровне выделить два режима исполнения: Secure (Безопасный) и Non-Secure (Гостевой).

Эти процессоры в основном нацелены на рынок смартфонов и планшетных ПК, где TrustZone используется для создания в режиме Secure доверенной «песочницы» для исполнения кода, связанного с криптографией, DRM, хранением пользовательских данных.

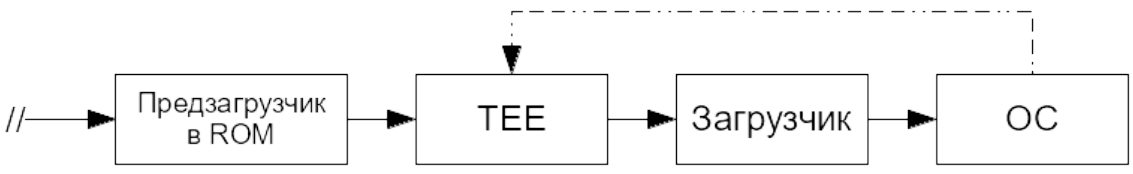
В режиме Secure при этом запускается специальная ОС, называемая в общем случае TEE (Trusted Execution Environment, Доверенная Среда Исполнения), а «нормальная» ОС, такая как Linux, iOS и т.д., запускается в режиме Non-Secure. При этом права доступа к некоторым устройствам ограничены для «нормальной» ОС, поэтому её ещё называют *гостевой ОС*.

Из-за наложенных ограничений гостевая ОС вынуждена время от времени вызывать функции TEE для исполнения некоторых операций. TEE продолжает существовать параллельно с гостевой ОС, и гостевая ОС не может с этим ничего поделать.

Например, гостевая ОС использует функции TEE для:

- Включения и выключения ядер процессора (в ARMv8-A это происходит через PSCI — часть TF-A);
- Хранения ключей, банковских карт и т.п.
- Хранения ключей полнодискового шифрования;
- Операций с криптографией;
- Отображения DRM-контента.

Процесс загрузки ОС на ARM (Cortex-A) примерно выглядит так:



На схеме пунктиром обозначен путь обращения из ядра ОС в TEE.

Смотрите также:

- **Boot process** (http://linux-sunxi.org/)
- **arm/BootProcess** (https://wiki.freebsd.org/arm/) — статья пусть и для FreeBSD, но для общего понимания сойдёт
- **How does ARM boot?** (https://fedevel.com/)

Заметки об ОС Linux. Часть 3. Система инициализации

Как было сказано ранее, `/sbin/init` — это программа, которая стартует первой в системе (в пользовательском режиме). Обычно имеет PID=1. Главная задача системы инициализации (далее — СИ) — довести систему до состояния, пригодного для использования. Для этого СИ может монтировать какие-либо разделы и файловые системы (так например на определённом этапе загрузки СИ выполняет монтирование ряда ФС), а также запускать установленные в системе демоны и иные программы. Основные параметры загрузки содержатся в файле `/etc/inittab`.

В некоторых СИ используются *уровни выполнения* — определённые стадии загрузки системы, на которых пользователю доступны какие-либо её возможности.

Помимо загрузки ОС, СИ заботится и о её выключении, завершая все сервисы и размонтируя файловые системы, после чего вызывая системный вызов для отключения ПК.

Реализации init'ов

Простейшие СИ

По умолчанию ядро после своей инициализации запускает программу `/sbin/init` и паникует, если не нашло её. Однако данное поведение можно и изменить, передав параметр `init=/путь/до/СИ`. Например, в качестве инита вы можете использовать вашу командную оболочку:

```
init=/bin/ash
```

Тогда первым процессом будет запущена оболочка `ash` из состава BusyBox. Однако помните, что в задачи системы инициализации входит подготовка ОС к дальнейшему функционированию: монтирование различных файловых систем (в т.ч. всяких `/dev/`, `/proc/`, `/sys/`), запуск демонов и иных программ. Если вместо обычной системы инициализации или программы, её заменяющей, вы запустите командную оболочку, то ничего из этого выполнено не будет — у вас просто запустится shell. Вы получите как бы рабочую систему с правами `root`, но не более того. Не будут работать логирование, графика, сеть и куча других вещей, которые запускает СИ.

Идя по пути усложнения вещей, мы можем написать shell-скрипт, который выполняет какие-то самые базовые вещи по настройке и запуску ОС. Например:

```
#!/bin/ash

export PATH=/bin:/sbin

# Монтируем всякие разные файловые системы
mount proc /proc -t proc
mkdir -p /dev/pts
mount -t devpts -o noexec,nosuid,gid=5,mode=0620 devpts /dev/pts
mount -t tmpfs none -o nodev,nosuid,noatime /tmp
mount -t tmpfs none -o nodev,nosuid,noatime /var/log
mount -t tmpfs none -o nodev,nosuid,noatime /var/run
mount -t tmpfs none -o nodev,nosuid,noatime /run
mount sysfs /sys -t sysfs

# Запускаем системный логгер
syslog
klogd -c 3

# Инициализация одного TTY
getty tty1 # Управление передаётся процессу `/sbin/getty`

# Когда мы завершим процесс `getty`, можно считать,
# что мы выключаем систему. Поэтому нам нужны
# действия по завершению её работы:

# Завершаем запущенные в скрипте сервисы
killall klogd
killall syslog

# Убиваем всё остальное
kill -TERM -1
kill -KILL -1

# Размонтируем все ФС
umount -a

exit
```

SysVInit

Ранее SysVInit был де-факто стандартом в дистрибутивах Linux, но в 2010-х были относительно успешные попытки заменить его на другие СИ.

Достоинства:

- Устоявшаяся и хорошо понятная классическая система;
- Простой процесс настройки;
- Нетребовательность к ресурсам компьютера;
- Следование философии UNIX;

Недостатки:

- Неудобная работа с сервисами;
- Каждый сервис — это обычный shell-скрипт;
- Отсутствие параллельного запуска загрузочных скриптов, из-за чего хотя бы один «зависший» скрипт застопорит загрузку/выключение системы;

Для SysVInit требуются загрузочные скрипты. С примером таких скриптов вы можете ознакомиться в **руководстве LFS**.

OpenRC

OpenRC — система инициализации с поддержкой зависимостей. Запускает необходимые системные сервисы в определённом порядке при загрузке, управляет ими во время работы системы и останавливает их при завершении работы. Имеет возможность параллельного запуска процессов для сокращения времени загрузки.

runit

runit — кроссплатформенная система инициализации для систем семейства UNIX, позиционирующая себя как замена SysVInit и других СИ. Как и другие системы инициализации, запускается первым процессом в системе. Работа runit состоит из трёх этапов:

1. runit запускает `/etc/runit/1` и ждёт, пока он завершится.
2. Запускает `/etc/runit/2`, который не должен завершаться до тех пор, пока система не остановится или не перезагрузится. В случае сбоя `/etc/runit/2` перезапускается. На этом этапе он также опционально

может обрабатывать сигнал `INT` (нажатие `Ctrl`+`Alt`+`Del` на клавиатуре).
3. Если `runit` было сказано остановить или перезагрузить систему, либо если этап № 2 завершился без ошибок, то запускается `/etc/runit/3`, который выполняет задачи по выключению, остановке или перезагрузке.

runit предоставляет следующие возможности:

- демонизация процессов;
- журналирование вывода процессов;
- запуск, остановка, перезапуск сервисов;
- выключение или запуск сервисов автоматически по мере их появления или удаления из списка;
- и т.д.

systemd

systemd — более крупная и сложная программа, чем все остальные, которые здесь описаны. Представляет собой менеджер системы и служб, который выполняется как самый первый процесс в системе и запускает остальные её компоненты. Параллельно запускает сервисы, а также имеет поддержку зависимостей у этих сервисов.

Поддерживает сценарии SysVInit, да собственно и является заменой последнего.

Среди прочих его функций можно отметить программу `journalctl` для журналирования системы, утилиты управления базовой конфигурацией системы (установка имени хоста, настройка даты и времени, etc.), ведение списка вошедших в систему пользователей, системных учётных записей, запущенных контейнеров, виртуальных машин, т.д.

systemd использует *юниты* — конфигурационные файлы, синтаксис которых напоминает формат `*.ini`, которые содержат в себе команды для монтирования файловых систем, запуска системных компонентов, включения подкачки и т.д.:

- сервисы (`*.service`);
- точки монтирования файловых систем (`*.mount`);
- устройства (`*.device`);
- сокеты (`*.socket`);
- etc.

Некоторым людям не особо нравится systemd за его монолитность (хотя в системах Linux куча других монолитных компонентов, то же ядро например) и то, что он пытается заменить собой кучу других компонентов системы, а не только `/sbin/init` .

Сравнение систем инициализации

Таблица взята из Gentoo WiKi.

Характеристика	SysVInit	OpenRC	runit
Поддерживаемые платформы	Linux/BSD	Linux/BSD	Linux/BSD/Da
Основной язык программирования	C	POSIX Shell + C	C
Основные зависимости	—	`init` (SysVInit или BSD)	—
Формат скрипта/ сервиса	Один скрипт (который может запускать несколько других)	Shell-скрипты	Shell-скрипты
Конфигурация для каждого сервиса	Нет	Есть	Нет

Характеристика	SysVInit	OpenRC	runit
Параллельная загрузка сервисов	Нет	Есть (опционально)	Есть (???)

Заметки об ОС Linux. Часть 4. Стандартные утилиты UNIX

Главное достоинство многих ОС семейства UNIX в том, что они модульные. Мы можем заменить один компонент системы, который нам не нравится, на другой, в случае. Например, мы можем удалить из системы рабочее окружение GNOME Shell и установить вместо него KDE Plasma. Либо мы можем создавать различные дистрибутивы одной ОС, единственное различие которых будет в разном наборе, казалось бы, однотипных программ. Всем же известно, что есть дистрибутивы Linux, использующие systemd, а есть дистрибутивы, использующие OpenRC или runit. А есть дистрибутивы, использующие glibc, но также есть и те, которые используют musl. Модульность является как плюсом для пользователей, ведь они могут найти систему, удовлетворяющую всем их требованиям, так и минусом, поскольку разные программы, делающие одно и тоже, могут различаться как по качеству, так и по набору функций, которые они могут выполнять. Но, тем не менее, наличие альтернатив каким-либо компонентам системы однозначно идёт на пользу таким системам.

Системные компоненты

В составе операционных систем семейства UNIX помимо ядра ОС, загрузчика и командной оболочки или иного интерфейса, с которым взаимодействует пользователь, входит ряд программ, которые предназначены, как правило, для администрирования и настройки системы. Среди этих программ есть утилиты для работы с файлами, правами доступа, процессами и прочим. Большинство этих утилит объединено в один пакет под названием `coreutils` (некоторые из них, предназначенные для администрирования ОС, использующих ядро Linux, находятся в отдельном пакете `util-linux` — как правило, это программы для работы с файловыми системами и подкачками, инициализации и

запуска TTY, отображения логов ядра и т.п. — т.е. специфичные для Linux программы).

Программы

Условно все программы в coreutils можно разделить на следующие категории:

Работа с файлами. Программы отсюда предназначены для создания, копирования, перемещения файлов и для изменения их метаданных.

Управление доступом. Эти утилиты применяются для изменения прав доступа к файлам и указания владельцев/группы владельцев файлов и каталогов.

Управление процессами. Программы нужны для отправки сигналов процессам, изменения их приоритета и отслеживанию запущенных в системе процессов.

Системные утилиты. Работа с TTY, получение информации о ядре Linux, работа с переменными окружения и т.д.

Реализации coreutils

Свои версии стандартных утилит UNIX имеют системы семейства *BSD и системы, использующие ядро Linux: в дистрибутивах GNU/Linux используется GNU Coreutils, а во встраиваемых системах как правило используется BusyBox, который содержит едаже больше утилит, чем в coreutils, хотя по размеру эти программы весьма небольшие и компактные.

- GNU Coreutils — используется в большинстве дистрибутивов GNU/Linux по умолчанию;
- uutils coreutils — реализация стандартных утилит UNIX на языке программирования Rust. Пытается обеспечить максимальную

совместимость с GNU-версией утилит;
- BusyBox — пакет, содержащий микроверсии программ из coreutils и ряда других проектов, объединяющий их в один небольшой исполняемый файл (размером до нескольких Мбайт). Обычно используется во встраиваемых системах;
- ToyBox — менее продвинутый и функциональный аналог BusyBox. Основная цель этого проекта — сделать Android самодостаточной системой, улучшив утилиты командной строки. В 2015 году Google объединила toybox с прошивкой AOSP и начала поставлять toybox с Android Marshmallow.

Состав

Работа с файлами

- `ls` — просматривает содержимое указанной директории;
- `cat` — выполняет конкатенацию содержимого файла в поток стандартного вывода;
- `mkdir` — создаёт директорию;
- `rmdir` — удаляет *пустую* директорию;
- `rm` — удаляет файлы или директории;
- `cp` — копирует файлы или директории;
- `mv` — перемещает файлы или директории;
- `ln` — создаёт жёсткие или символьные ссылки;
- `mktemp` — создаёт временный файл или директорию;
- `mknod` — создаёт именованный канал или устройство;
- `md5sum`, `sha256sum`, `*sum` — вычисляет контрольные суммы файлов;
- `[`, `test` — команда для проверки файла на его наличие, для проверки типа файла (файл, директория, ссылка), а также проверяет условия на истинность/ложность;
- `truncate` — уменьшает или увеличивает размер файла;

Работа с дисками

- `dd` — утилита побайтового копирования файлов;
- `df` — выводит информацию об использовании дисков;
- `du` — выводит информацию о размере файлов;

Работа с правами доступа

- `chown` — изменяет владельца и группу владельца файла;
- `chmod` — изменяет права доступа к файлу;

Управление процессами

- `nice` — изменяет приоритет процессов;
- `nproc` — выводит число ядер/потоков процессора, доступных процессу;

Работа с данными

- `cut` — печатает выбранные части строк;
- `expand` — конвертирует символы табуляции (`\t`) в пробелы;
- `od` — выгружает файлы в восьмеричном и других форматах;
- `sort` — сортирует строки;
- `uniq` — печатает или убирает повторяющиеся строки;
- `comm` — построчно сравнивает два файла;
- `head` — выводит первые n-строк из файла;
- `tail` — выводит последние n-строк из файла;
- `wc` — считает строки, символы и байты в файле;

Управление пользователями

- `groups` — сообщает о членстве пользователя в группах;

- `hostid` — выводит идентификатор хоста в шестнадцатеричном формате;
- `id` — выводит EUID, GID и членство в группах указанного пользователя;
- `logname` — сообщает имя пользователя, от имени которого произошёл вход в систему;
- `users` — выводит список пользователей, залогиненных в системе;
- `who` — сообщает, кто вошёл в систему;
- `whoami` — сообщает имя пользователя, связанное с текущим EUID;

util-linux

Из состава пакета `util-linux` можно также отметить следующие программы:

- `dmesg` — выводит буфер сообщений ядра в stdout;
- `lsblk` — выводит список блочных устройств в stdout;
- `mount` — монтирует файловую систему;
- `umount` — размонтирует файловую систему;
- `su` — выполняет вход от имени указанного пользователя;
- `kill` — посылает сигналы процессам;

Смотрите также:

- **Core utilities** (https://wiki.archlinux.org)

Управление пакетами

Использование пакетных менеджеров является привычным способом управления пакетами во многих системах, использующих ядро Linux, однако LFA не предполагает использование таких решений, предпочитая использованию готовых инструментов сборку нужного ПО из исходного кода.

Мы не рекомендуем вам использовать уже существующие пакетные менеджеры в собранной системе LFA по ряду причин:

- Рассмотрение вопросов управления пакетами отвлекает внимание от целей LFA - изучения строения Linux-систем и их процесса проектирования и сборки для ARM-компьютеров.
- Типичные пакетные менеджеры предназначены для конкретных дистрибутивов GNU/Linux. Использование таких ПМ в LFA может привести к поломке системы.
- Существует множество решений для управления программным обеспечением, каждое из которых имеет свои достоинства и недостатки. Найти идеальное для конкретной задачи решение - задача не авторов LFA, а пользователя.

Обновления

Пакетный менеджер облегчает обновление ПО до новых версий. Как правило, для обновления пакета до новой версии можно использовать инструкции из LFA. Однако есть некоторые моменты, о которых необходимо помнить при обновлении пакетов, особенно в рабочей системе:

- Если вы *уже* собрали систему по LFA и с момента сборки прошло некоторое время, то нет нужды устанавливать обновления для пакетов GCC, GMP, MPC, MPFR, Binutils. Их всё равно в системе нет, они используются только для сборки LFA.

- Если вам необходимо обновить стандартную библиотеку C (`musl`), лучше всего будет полностью пересобрать LFA. Несмотря на то, что вы можете просто пересобрать все пакеты в порядке их зависимостей, мы вам не рекомендуем этого делать.
- Если пакет, содержащий разделяемую библиотеку (shared library), обновляется и, если имя библиотеки меняется, то все пакеты, динамически связанные с этой библиотекой, должны быть пересобраны для связи с этой библиотекой (обратите внимание, что нет никакой корреляции между версией пакета и именем библиотеки). Например, есть пакет `foo-1.0.0`, который устанавливает библиотеку `libfoo.so.1`. Допустим, вы обновляете пакет до более новой версии `foo-1.1.7`, который устанавливает библиотеку `libfoo.so.2`. В этом случае все пакеты, динамически связанные с `libfoo.so.1`, должны быть перекомпилированы для связи с новой `libfoo.so.2`. *Заметьте, что вы не должны удалять предыдущие библиотеки, пока не будут перекомпилированы зависимые пакеты.*
- Если вы обновляете работающую систему, обратите внимание на пакеты, в которых для установки файлов в систему используется команда `cp`, а не `install`. Последняя команда обычно безопаснее, если исполняемый файл или библиотека уже загружена в память.

Методы управления пакетами

Ниже перечислены некоторые распространённые методы управления ПО. Прежде чем принять решение о выборе пакетного менеджера, рассмотрите указанные ниже методы и определите, лучше ли они тупых и раздутых ПМ.

Держать всё в своей голове

Эта техника применяется в данном руководстве. Мы считаем, что вы, как пользователь, вполне в состоянии запомнить пару десятком установленных в систему пакетов, к тому же, здесь приведены основные

сведения о ПО, используемом в LFA: начиная от описания и версии пакета, заканчивая их содержимым (например, в нашем руководстве приведены краткие описания устанавливаемых пакетами программ и библиотек). Данный метод хорош в крайне минималистичных и небольших системах, в которых либо не предполагается управление пакетами как таковое, либо, если и предполагается, то установка или удаление ПО планируется крайне редко.

Установка в отдельные каталоги

Эта техника допускает наличия в системе нескольких версий одного и того же пакета. К тому же, она значительно упрощает удаления пакетов, ведь для осуществления этого требуется всего лишь удалить директорию, в которую установлен пакет.

Каждый пакет устанавливается в отдельный каталог. Например, пакет `foo-1.0.0` будет установлен в `/usr/pkg/foo-1.0.0`. Из минусов такого подхода можно отметить разрастание переменных окружения `PATH`, `LD_LIBRARY_PATH` и др.

Использование самодостаточных пакетов

Вы можете использовать и самодостаточные пакеты. Мы не проверяли их работоспособность в LFA, поэтому не можем гарантировать то, что это возможный вариант.

Использование пакетных менеджеров

Несмотря на то, что LFA не использует пакетные менеджеры, поскольку основным её предназначением является именно ручная компиляция программ, вы можете установить какой-либо пакетный менеджер в вашу собранную систему. Обратите внимание, что использование пакетников из распространённых дистрибутивов может сломать вашу LFA, поскольку эти пакетные менеджеры могут не знать о том, какое ПО у вас уже

установлено в системе и пытаться переустановить его. Поэтому используйте пакетные менеджеры осторожно.

Сборка ПО из исходного кода

Концепция некоторых систем Linux, а иногда и некоторых других ОС семейства Unix неразрывно связана с компиляцией программного обеспечения из исходного кода. Да и многие руководства, связанные с Linux, например LFS, CLFS или LFA также построены вокруг компиляции ПО. Конечно, многие системы Linux снабжены удалёнными хранилищами уже скомпилированных пакетов, но бывают и случаи, когда в репозиториях нет нужного пакета и его приходится собирать из исходников самостоятельно.

Конечно, LFA может дать вам опыт в сборке ПО из исходного кода, однако у пользователей (особенно начинающих) периодически возникали вопросы по поводу компиляции программ; эта страница содержит ответы на основные из вопросов.

Системы сборки

Современное программное обеспечение достаточно крупное и сложное. Если какую-то простую программу мы сможем скомпилировать всего лишь одной командой, например:

```
# C
gcc main.c -o main

# C++
g++ main.cpp -o main

# Rust
rustc main.rs
```

то более сложные вещи компилируются в несколько команд. Число таких команд может превышать десятки, а то и сотни. Для того, чтобы не набирать их все вручную, можно пойти двумя путями:

1. Написать BASH-скрипт, последовательно выполняющий команды для сборки.
2. Использовать специальную программу, принимающую на вход определённый файл со сборочными инструкциями, и выполняющая его.

Для чего-то простого вполне сойдёт и BASH-скрипт: это просто и быстро, да и интерпретатор BASH (или совместимого с ним языка) присутствует практически во всех ОС семейства Unix. Но что делать, если скрипт выполнил, условно говоря, из 100 команд всего 10, а на 11 произошла ошибка и скрипт прервал свою работу? Мы можем как-то исправить ситуацию и начать сборку заново. Но тогда, когда мы вновь будем исполнять те команды, которые ранее были успешно выполнены, мы просто потеряем время. А если мы хотим производить сборку пакета в несколько потоков, чтобы сократить общее время сборки? BASH-скрипты, насколько я знаю, этого не позволяют. А если нам нужно как-то кастомизировать сборку пакета, скомпилировав только то, что нам необходимо в конкретной ситуации, а всё, что ненужно, пропустить?

Использование простых скриптов для сборки можно оправдать разве что для каких-то простых проектов, для чего-то посложнее и были придуманы системы сборки. Существует множество разных систем сборки, наиболее популярные из которых GNU Make и meson+ninja. В LFA вы встречались только с `make`, а, например, если заглянете в руководство **BLFS**, то можете увидеть у некоторых пакетов и другие системы сборки, например CMake или meson+ninja. Последняя особенно полюбилась разработчикам рабочего окружения GNOME и многие из проектов этого окружения уже используют meson+ninja для своей сборки.

Как определить, какая система сборки используется у пакета?

Если вы собираете какой-либо пакет из состава LFA, LFS или BLFS, то об этом можете не беспокоиться: просто используйте инструкции, которые приведены в этих руководствах. В случае, если по поводу сборки пакета не сказано и в документации (например, обычно *общие* инструкции по

сборке приводятся в файле `INSTALL`, который находится в директории с исходниками программы. Там даны лишь общие сведения по поводу использования основных команд и всё, конкретных сборочных инструкций там как правило нет), то вам придётся определить систему сборки самостоятельно. Например, посмотрите содержимое директории с исходным кодом программы:

Система сборки	Типичные для неё файлы
GNU Make	`Makefile`, `Makefile.in`, `configure`, `aclocal.m4`, `config.guess`, `config.in`, `config.sub`
CMake	`CMake.list`
`meson` + `ninja`	`meson.build`

Общий порядок сборки

№	Название пункта	Выполнение
1	Скачивание архива с исходным кодом	Выполняется всегда
2	Скачивание необходимых сторонних патчей	Выполняется при необходимости
3	Распаковка архива с исходниками; переход в директорию с исходниками	Выполняется всегда
4	Применение патчей	Выполняется при необходимости
5	Конфигурирование системы сборки; настройка пакета перед сборкой	Выполняется при необходимости и/или при наличии возможности настройки
6	Сборка пакета	Выполняется при наличии возможности сборки (например,

№	Название пункта	Выполнение
		если вы собираете программу, написанную на скриптовом ЯП, то сборка тут необходима не всегда, под сборкой подразумевается не только компиляция исходников, но и иные действия с ними)
7	Установка пакета в систему	Выполняется всегда

Примеры сборок пакетов

Рассмотрим процесс сборки пакетов из исходного кода на примере BusyBox и iana-etc.

BusyBox

Шаг 1. Скачивание исходного кода:

```
wget https://busybox.net/downloads/busybox-1.36.1.tar.bz2
```

Шаг 2. Скачивание необходимых сторонних патчей:

Скачивание патчей здесь не требуется.

Шаг 3. Распаковка архива с исходниками; переход в директорию с исходниками:

```
tar -xf busybox-1.36.1.tar.bz2
cd busybox-1.36.1
```

Шаг 4. Применение патчей:

Применение патчей здесь не требуется.

Шаг 5. Конфигурирование системы сборки; настройка пакета перед сборкой:

```
make mrproper
make defconfig
```

На этапе настройки системы сборки и конфигурирования пакета перед сборкой создаётся ряд файлов либо с параметрами сборки (примером этого может служить файл `.config`, содержащий параметры сборки ядра Linux или пакета BusyBox), либо файл со сборочными инструкциями (например, генерация файлов `Makefile` после исполнения скрипта `configure` с переданными ему ключами с параметрами сборки).

Шаг 6. Сборка пакета:

```
make -j4
```

Если программа написана на компилируемом языке программирования, то на этапе сборки вызывается компилятор нужного языка программирования, чтобы сгенерировать исполняемые двоичные файлы из исходников. Если программа написана на интерпретируемом ЯП, то, как правило, если пакет поддерживает «сборку», то на данном этапе исходный код просто подготавливается для его дальнейшей установки в систему.

Шаг 7. Установка пакета в систему:

```
sudo make install
```

На этом этапе собранные файлы копируются либо в системные директории, либо в другое место, указанное либо на этапе конфигурирования, либо непосредственно сейчас.

iana-etc

Шаг 1. Скачивание исходного кода:

```
wget https://github.com/Mic92/iana-
etc/releases/download/20240125/iana-etc-20240125.tar.gz
```

Шаг 2. Скачивание необходимых сторонних патчей:

Скачивание патчей здесь не требуется.

Шаг 3. Распаковка архива с исходниками; переход в директорию с исходниками:

```
tar -xf iana-etc-20240125.tar.gz
cd iana-etc-20240125
```

Шаг 4. Применение патчей:

Применение патчей здесь не требуется.

Шаг 5. Конфигурирование системы сборки; настройка пакета перед сборкой:

Конфигурирование и настройка здесь не требуется.

Шаг 6. Сборка пакета:

Сборка пакета здесь не требуется.

`iana-etc` — просто набор текстовых файлов. Тут нечего компилировать или подготавливать к установке.

Шаг 7. Установка пакета в систему:

```
cp -v services protocols $LFA_SYS/etc/
```

Просто вручную копируем нужные нам файлы в системную директорию LFA.

Решение проблем

- У пакета, который я собираю, используется система сборки GNU Make, однако нет файла `Makefile`, без которого я не могу собрать пакет. Что делать?

    - Сгенерируйте пакет с помощью скрипта `configure`. Например:

```
./configure --prefix=/usr
```

- У меня нет файла `configure`, хотя используется GNU Make!
    - Используйте программу `autoreconf` из пакета `autoconf` для генерации `configure`.

Кросс-компилятор

Кросс-компилятор — такой компилятор, который генерирует двоичные файлы для платформы, отличной от той, на которой этот компилятор запускается. Он может быть полезен, например, когда нужно скомпилировать программу для той платформы, экземпляров которой в данный момент в наличии нет, либо когда сборка непосредственно на этой (целевой) платформе невозможна или нецелесообразна, например, когда нужно собрать программу для маломощного встраиваемого ARM-компьютера, сборка программы непосредственно на котором будет происходить либо очень долго, либо не происходить вообще ввиду ограниченной производительности такого компьютера.

- **Хост** (host) — компьютер, **на** котором производится сборка;
- **Цель** (target) — компьютер, **для** которого производится сборка.

Строение кросс-компилятора

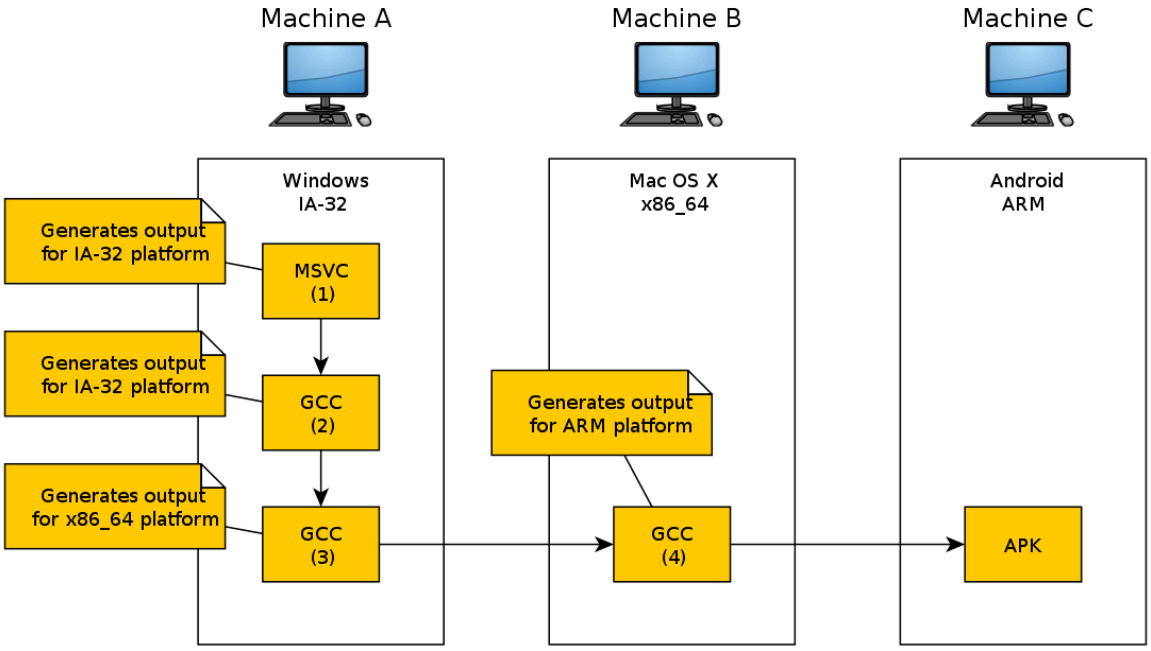
Помимо само*го* кросс-компилятора требуются скомпилированные для целевой платформы пакет `binutils` (особенно важно наличие GNU Assembler), содержащий программы для работы с объектными файлами, стандартная библиотека языка C (в LFA используется библиотека Musl) и заголовки ядра Linux.

Канадский крест

Канадский крест — это метод сборки кросс-компилятора для целевых архитектур. Например:

- Фирменный родной компилятор для компьютера (№1) используется для сборки «родного» компилятора для компьютера (№2).

- Родной компилятор для компьютера (№2) используется для сборки кросс-компилятора с компьютера A на компьютер B (№3).
- Кросс-компилятор с компьютера A для компьютера B используется для сборки кросс-компилятора для компьютера C (№4).



Смотрите также:

- **Cross compiler** (https://en.wikipedia.org/).

FPU в ARM-процессорах

FPU

FPU — это часть процессора для выполнения различных математических операций над вещественными числами. Он поддерживает работу с ними на уровне *примитивов* — загрузка, выгрузка вещественного числа (в/из специализированных регистров) или математическая операция над ними выполняется одной командой, за счёт чего достигается значительное ускорение таких операций. Типичными операциями является сложение, вычитание, умножение, деление и вычисление квадратного корня. Некоторые FPU также могут выполнять различные трансцендентные функции, такие как экспоненциальные или тригонометрические вычисления, но их точность может быть низкой, поэтому некоторые системы предпочитают вычислять эти функции программно.

VFP

Технология VFP (Vector Floating Point) — это сопроцессорное расширение блока **FPU (Floating-Point Unit)** для архитектуры ARM (в ARMv8 реализовано иначе — сопроцессоры там не определены). Он обеспечивает недорогие вычисления с плавающей запятой одинарной и двойной точности. Архитектура VFP предполагала поддержку выполнения коротких инструкций «векторного режима», но они последовательно обрабатывали каждый элемент вектора и, таким образом, не обеспечивали производительности истинного векторного параллелизма с одной инструкцией и множеством данных (**SIMD**). Поэтому векторный режим был удалён вскоре после появления, а на смену ему пришёл гораздо более мощный Advanced SIMD, также известный как **Neon**.

Некоторые устройства, такие как ARM Cortex-A8, имеют урезанный модуль VFPLite вместо полноценного VFP, и требуют примерно в 10 раз больше

тактов на операцию с плавающей запятой. Архитектуры до ARMv8 реализовывали плавающую запяту/SIMD с помощью интерфейса сопроцессора. Другие модули с плавающей запятой и/или SIMD, встречающиеся в ARM процессорах, использующих математический сопроцессор, включают FPA, FPE, iwMMXt, некоторые из которых были реализованы программно, но моги быть реализованы и аппаратно. Они обеспечивают ту же функциональность, что и VFP, но не совместимы с ним по коду. FPA10 также обеспечивает расширенную точность, но реализует корректное округление только в одинарной точности.

VFPv1

Устарел.

VFPv2

Опциональное расширение набора инструкций в архитектурах ARMv5TE, ARMv5TEJ и ARMv6. VFPv2 имеет 16 64-битных регистров FPU.

VFPv3 или VFPv3-D32

Реализовано в большинстве процессоров Cortex-A8 и A9 ARMv7. Обратно совместим с VFPv2, за исключением того, что не может отлавливать исключения с плавающей точкой. VFPv3 имеет 32 64-битных регистра FPU в качестве стандарта, добавляет инструкции VCVT для преобразования между скалярами, `float` и `double`, добавляет немедленный режим в VMOV, чтобы константы могли быть загружены в регистр FPU.

VFPv3-D16

Как и выше, но только с 16 64-битными регистрами FPU. Реализован в процессорах Cortex-R4 и R5, а также в Cortex-A9.

VFPv4 или VFPv4-D32

Реализован в процессорах Cortex-A12, Cortex-A15. В Cortex-A7 опционально устанавливается VFPv4-D32 в случае использования FPU с Neon.

VFPv4-D16

Реализован в Cortex-A5 и Cortex-A7 если они используют FPU без Neon.

VFPv5-D16-M

Реализовано в Cortex-M7 при наличии опции ядра с плавающей точкой одинарной и двойной точности.

В Debian Linux и производных, таких как Ubuntu и Linux Mint, armhf (ARM hard float) относится к архитектуре ARMv7, включая дополнительное аппаратное расширение VFP3-D16 с плавающей точкой (и Thumb-2). Программные пакеты и инструменты кросс-компиляторов используют суффиксы armhf и arm/armel для различия.

Смотрите также:

- **Почти все, что вы хотели знать про плавающую точку в ARM, но боялись спросить**;

Device Tree

ARM-компьютеры, в отличие от «классических» x86_64, используют специальный компонент под названием SoC (System on Chip), включающий в себя процессор и много других устройств, которые находятся на одной, как правило небольшой, микросхеме. К таким компьютерам можно отнести, к примеру, Raspberry Pi, Orange Pi, Banana Pi, Repka Pi, Pinebook и т.д.

SoC нуждается разве что во внешнем ОЗУ и некоторой периферии или разъёмов для подключения этой периферии, при этом SoC реализует в себе множество возможных, а иногда и взаимоисключающих конфигураций устройств. Таким образом, в системе должен быть механизм, позволяющийй передать ядру Linux информацию о том, какая из конфигураций соответствует тому, что распаяно на материнской плате. Данным механизмом является Devicetree.

> Важно понимать, что Devicetree — это не что-то очень крупное и сложное, а всего лишь *формат* описания конфигурации оборудования компьютера. Ядро читает эту конфигурацию для обнаружения нужного оборудования в компьютере и, таким образом, обеспечивает поддержку только тех устройств, которые описаны в файле Devicetree (при условии наличия драйверов для этих устройств).

Файлы

`*.dts`

Файл с расширением `*.dts` — представление дерева устройств в человекочитабельном текстовом формате, синтаксис которого похож на

JSON. Компилируется программой `dtc` в бинарный файл `*.dtb`.

Пример содержимого `*.dts`:

```
/dts-v1/;

/ {
  interrupt-parent = <0x01>;
  #address-cells = <0x01>;
  #size-cells = <0x01>;
  model = "OrangePi 3 LTS";
  compatible = "xunlong,orangepi-3-lts\0allwinner,sun50i-h6";

  aliases {
    mmc0 = "/soc/mmc@4020000";
    mmc1 = "/soc/mmc@4021000";
    mmc2 = "/soc/mmc@4022000";
    ethernet0 = "/soc/ethernet@5020000";
    serial0 = "/soc/serial@5000000";
  };

  cpus {
    #address-cells = <0x01>;
    #size-cells = <0x01>;

    cpu@0 {
      compatible = "arm,cortex-a53";
      device_type = "cpu";
      reg = <0x00>;
      enable-method = "psci";
      clocks = <0x06 0x15>;
      clock-latency-ns = <0x3b9b0>;
      #cooling-cells = <0x02>;
      operating-points-v2 = <0x07>;
      cpu-supply = <0x08>;
      phandle = <0x0a>;
    };
    ...
  };
  ...
}
```

`*.dtb`

`*.dtb` — уже двоичное (бинарное) представление информации из файла `*.dts`, которое читается ядром Linux.

`*.dtbo` (overlay)

В дистрибутивах Raspbian и Armbian есть механизм, позволяющий во время загрузки применять к `*.dtb`-файлам патчи. Такой патч, содержащий какую-либо аппаратную возможность платы, называется overlay. Формат такого файла идентичен формату `*.dtb`. Обычно расположены в директории `/boot/dtb/<имя SoC>/overlay`.

Пример содержимого `/boot/dtb/<имя SoC>/overlay/`:

```
$ ls /boot/dtb/allwinner/overlay
sun50i-a64-fixup.dtbo              sun50i-h6-fixup.dtbo
sun50i-a64-pine64-7inch-lcd.dtbo   sun50i-h6-i2c1.dtbo
sun50i-a64-pps-gpio.dtbo           sun50i-h6-i2c2.dtbo
sun50i-a64-uart1.dtbo
```

Откуда брать эти файлы

В новых версиях ядра существует целая библиотека с ванильными `*.dt{b,s}` файлами для многих известных компьютеров на архитектуре ARM. Для компиляции этих файлов (нужна программа `dtc`), которые находятся в дереве исходного кода Linux, введите:

```
make CROSS_COMPILE=$LFA_TGT- ARCH=arm64 dtbs
```

В зависимости от параметров, указанных при конфигурировании ядра (в частности, в зависимости от того, поддержку каких SoC вы включили/отключили), будут скомпилированы нужные файлы `*.dtb`. После их

компиляции вам нужно будет создать в директории `/boot` поддиректорию, в которую будут скопированы эти файлы.

Если в составе ядра нет `*.dts`-файла для вашего компьютера, то попробуйте взять его из дстрибутива (Armbian, Raspbian, etc.), где этот файл есть. Однако я не могу гарантировать в этом случае полную работоспособность системы с «чужим» файлом описания устройств.

Смотрите также:

- **Linux and the Devicetree** (https://docs.kernel.org/);
- **Device Tree** (https://www.altlinux.org/);
- **https://www.devicetree.org/**.

Загрузчик U-Boot

В данном разделе приведены общие сведения об использовании загрузчика U-Boot. Основная их часть была взята из **документации**.

Переменные окружения

Загрузчик U-Boot поддерживает пользовательскую конфигурацию с помощью переменных окружения, значения которых можно записать на постоянном носителе (например, во flash-памяти).

Переменные окружения устанавливаются с помощью команды `env set` (алиас `setenv`), выводятся в консоль с помощью команды `env print` (алиас `printenv`) и сохраняются в постоянном хранилище с помощью команды `env save` (`saveenv`). Использование `env set` без указания значения переменной может использоваться для удаления переменной из окружения. Пока вы не сохраняете окружение, вы работаете с его копией в оперативной памяти. Если область в постоянном хранилище, содержащая окружение, будет случайно стёрта, U-Boot создаст окружение по умолчанию.

Некоторые настройки контролируются переменными окружения, поэтому установка переменной может изменить поведение загрузчика (например, таймаут автозагрузки, автозагрузка с `tftp` и т.п.).

Окружение в текстовых файлах

Для каждой конкретной платы компьютера может быть создано своё уникальное окружение, переменные которого хранятся в файле `*.env`, имеющий простой текстовый формат. Базовое имя этого файла содержится в параметре `CONFIG_ENV_SOURCE_FILE` или, если он пуст, в `CONFIG_SYS_BOARD`.

Этот файл хранится в каталоге, отведённом для конкретной платы, и имеет расширение `.env`. Пример пути до этого файла:

```
board/<производитель>/<плата>/<CONFIG_ENV_SOURCE_FILE>.env
```

или:

```
board/<производитель>/<плата>/<CONFIG_SYS_BOARD>.env
```

Это обычный текстовый файл, в котором хранятся переменные в формате `var=value`. Поддерживаются пустые строки и многострочные переменные. Пробелы перед `=` не допускаются.

Для добавления дополнительного текста к значению переменной можно использовать конструкцию `var+=value`. Этот текст объединится со значением переменной во время сборки и станет доступным загрузчику как единое значение. Переменные могут содержать символ `+`, но если вам вдруг захочется иметь значение переменной, заканчивающееся на этот символ, то поставьте перед `+` обратную косую черту, чтобы скрипт загрузчика знал, что вы не добавляете значение к существующей переменной, а объявляете новую:

```
maximum\+=value
```

Этот файл может включать комментарии в стиле C. Пустые линии и многострочные переменные также поддерживаются, кроме того, вы можете использовать привычные директивы препроцессора C и определения `CONFIG` из конфига вашей платы.

Например, для платы `snapper9260` вы можете создать текстовый файл с именем `board/bluewater/snapper9260.env`, содержащий информацию об окружении:

```
stdout=serial
#ifdef CONFIG_VIDEO
stdout+=,vidconsole
#endif
bootcmd=
    /* U-Boot script for booting */

    if [ -z ${tftpserverip} ]; then
        echo "Use 'setenv tftpserverip a.b.c.d' to set IP address."
    fi

    usb start; setenv autoload n; bootp;
    tftpboot ${tftpserverip}:
    bootm
failed=
    /* Print a message when boot fails */
    echo CONFIG_SYS_BOARD boot failed - please check your image
    echo Load address is CONFIG_SYS_LOAD_ADDR
```

Настройки, которые являются общими для нескольких плат, можно подключать к `.env`-файлу конкретной платы с помощью директивы `#include`, которой будет передан файл, находящийся в директории `include/env`, содержащий настройки окружения. Например:

```
#include <env/ti/mmc.env>
```

Старый C-стиль окружения

Если параметр `CONFIG_ENV_SOURCE_FILE` пуст и имя `.env`-файла по умолчанию отсутствует, то вместо него используется окружение в старом C-стиле (см. ниже).

Традиционно, стандартное окружение создаётся в файле `include/env_default.h`. Оно может быть дополнено различными определениями `CONFIG`. В частности, вы можете определить `CFG_EXTRA_ENV_SETTINGS` в конфиге конкретной платы, чтобы добавить переменные окружения.

Список переменных окружения

Некоторые параметры конфигурации загрузчика можно задать с помощью переменных окружения. Во многих случаях значения в окружении по умолчанию берётся из опции `CONFIG` — для этого см. `include/env_default.h`.

Неполный список переменных (более полный список см. в **документации U-Boot**):

`autostart`

Если установлено значение `yes` (фактически любая строка, начинающаяся с `1`, `y`, `Y`, `t` или `T`), образ, загруженный с помощью одной из перечисленных ниже команд, будет запущен автоматически внутренним вызовом команды `bootm`.

- `bootelf` — загрузка из ELF-файла, загруженного в оперативную память;
- `bootp` — загрузка образа по сети используя протокол BOOTP/TFTP;
- `dhcp` — загрузка образа по сети используя протокол DHCP/TFTP;
- `diskboot` — загрузка с IDE-устройства;
- `nboot` — загрузка с NAND-устройства;
- `nfs` — загрузка образа по сети используя протокол NFS;
- `rarpboot` — загрузка образа по сети используя протокол RARP/TFTP;
- `scsiboot` — загрузка с SCSI-устройства;
- `tftpboot` — загрузка образа по сети используя TFTP-протокол;
- `usbboot` — загрузка с устройства USB;

Если переменная окружения `autostart` не имеет значения, начинающегося с `1`, `y`, `Y`, `t` или `T`, образ, переданный команде `bootm`, будет скопирован по адресу загрузки (и в конечном итоге распакован), но **не будет** запущен. Это можно использовать для загрузки и распаковки произвольных данных.

---

baudrate

Используется для установки скорости передачи данных по UART — значение по умолчанию указано в параметре `CONFIG_BAUDRATE` (по умолчанию равна 115200).

---

bootdelay

Задержка перед автоматическим запуском `bootcmd`. В течение этого времени пользователь может выбрать вход в оболочку или меню загрузки U-Boot (если `CONFIG_AUTOBOOT_MENU_SHOW=y`):

- 0 — автозагрузка без задержки, но её можно остановить вводом клавиши;
- 1 — отключение автозагрузки;
- 2 — автозагрузка без задержки и без проверки на прерывание.

Значение по умолчанию определяется параметром `CONFIG_BOOTDELAY`. Значение `bootdelay` переопределяется значением `/config/bootdelay` в Device Tree, если `CONFIG_OF_CONTROL=y`.

---

bootcmd

Команда, которая выполняется, если пользователь не вошёл в оболочку U-Boot во время задержки загрузки.

---

bootargs

Аргументы командной строки, которые будут переданы во время загрузки операционной системе или двоичному образу.

`fdt_high`

Если установлено, этот параметр ограничит максимальный адрес, в который будет скопирован образ FDT (Flattened Device Tree) при загрузке. Например, если у вас система с 1 Гб памяти по физическому адресу 0x10000000, а ядро Linux распознаёт только первые 704 МБ как low memory, вам может понадобиться установить `fdt_high=0x3C000000`, чтобы FDT был скопирован по максимальному адресу low memory 704 МБ, чтобы ядро Linux могло получить к нему доступ во время процесса загрузки.

Если этот параметр имеет специальное значение `0xffffffffff` (32-битные устройства) или `0xffffffffffffff` (64-битные устройства), то FDT вообще не будет копироваться при загрузке.

`initrd_high`

Ограничивает позиционирование образа `initrd`. Если эта переменная не установлена, образы `initrd` будут копироваться по максимально возможному адресу в оперативной памяти; обычно это то, что нужно, поскольку позволяет обеспечить максимальный размер образа `initrd`. Однако если по какой-то причине вы хотите быть уверены, что образ `initrd` будет загружен ниже предела `CFG_SYS_BOOTMAPSZ`, вы можете установить для этой переменной значение `no`, `off` или `0`. В качестве альтернативы можно задать максимальный верхний адрес для использования (U-Boot всё равно будет проверять, не перезаписывает ли этот образ стек и данные загрузчика).

Например, вы имеете систему с 16 МБ ОЗУ и хотите зарезервировать 4 МБ для использования Linux, вы можете это сделать, добавив `mem=12M` к значению переменной `bootargs`. Однако теперь вы должны будете убедиться, что образ `initrd` будет размещён в первых 12 МБ — это можно сделать с помощью:

```
setenv initrd_high 00c00000
```

Если этот параметр использует специальное значение `0xffffffffff` (32-битные устройства) или `0xffffffffffffff` (64-битные устройства), это будет означать для загрузчика, что все адреса являются легальными для ядра Linux, включая адреса во флеш-памяти. В этом случае U-Boot вообще не будет копировать ramdisk. Это может быть полезно для уменьшения времени загрузки системы, но требует, чтобы эта функция поддерживалась ядром Linux. Кроме того, нужно, чтобы пользователь убедился в отсутствии дублирования с другими частями образа, такими как BSS ядра Linux. Этот параметр не следует включать по умолчанию, а только в рамках оптимизации развёртывания ОС.

`loadaddr`

Устанавливает стандартный адрес загрузки для таких команд, как `bootp`, `rarpboot`, `tftpboot`, `loadb` или `diskboot`. Обратите внимание, что оптимальные значения по умолчанию в разных архитектурах могут различаться. Например, на 32-битных ARM используется некоторое смещение от начала памяти, так как в ядре Linux zImage имеет самораспаковывающийся декомпрессор, и лучше, если мы не будем вмешиваться в работу этого декомпрессора.

`silent_linux`

Если эта переменная установлена, то Linux будет загружаться в «тихом режиме».

Расположения образов

Следующие переменные содержат расположение образов, используемых при загрузке. Столбец «Образ» указывает роль образа и не является именем переменной окружения. Остальные столбцы — имена переменных. «Имя файла» — имя файла на TFTP-сервере. «Адрес ОЗУ» — место в ОЗУ, куда будет загружен образ, а «Расположение Flash» — адрес образа во flash-памяти NOR или смещение во flash-памяти NAND.

> ⚠ **Важно**
>
> Эти переменные не обязательно должны быть определены для всех плат, некоторые платы в настоящее время используют другие переменные для этих целей, а некоторые используют эти переменные для других целей.

Также обратите внимание, что большинство из этих переменных — это просто общепринятый набор имён переменных, используемых в некоторых других определениях переменных, но не закодированных в коде загрузчика.

Образ	Имя файла	Адрес ОЗУ	Расположение Flash
Ядро Linux	`bootfile`	`kernel_addr_r`	`kernel_addr`
Блоб Devicetree	`fdtfile`	`fdt_addr_r`	`fdt_addr`
ramdisk	`ramdiskfile`	`ramdisk_addr_r`	`ramdisk_addr`

При задании адресов ОЗУ для `kernel_addr_r`, `fdt_addr_r` и `ramdisk_addr_r` необходимо учитывать несколько типов ограничений. Одним из таких типов является требование к полезной нагрузке. Например, devicetree **ДОЛЖНО** загружаться по адресу, выровненному по 8 байтам, так как этого требует спецификация. Аналогичным образом операционная система может определять ограничения на то, где в памяти

может находиться полезная нагрузка. Это документировано, например, в Linux, в документах **Booting ARM Linux** и **Booting AArch64 Linux**. Наконец, существуют практические ограничения. Мы не знаем, какой размер конкретной полезной нагрузки будет задействовать пользователь, но каждая полезная нагрузка не должна перекрываться, иначе она будет повреждать другую полезную нагрузку. Аналогичная проблема может возникнуть, если полезная нагрузка окажется в области OS BSS. По этим причинам нам нужно убедиться, что значения по умолчанию здесь не приведут к сбоям в загрузке и достаточно объяснимы, чтобы их можно было оптимизировать по времени загрузки или скорректировать для конфигураций с меньшим объемом памяти.

На разных архитектурах у нас будут разные ограничения. Выжно следовать всем документированным требованиям, чтобы наилучшим образом обеспечить совместимость. В **документации U-Boot** приведены примеры, показывающие, как обеспечить разумные значения по умолчанию в различных случаях.

Загрузка DTBO

Оверлеи Devicetree — это почти те же Devicetree-файлы, но с несколько иным синтаксисом. Пожалуйста, обратитесь к файлу `dt-object-internal.txt` в исходном коде компилятора devicetree за информацией о внутреннем формате оверлеев: **https://git.kernel.org/pub/scm/utils/dtc/dtc.git/tree/Documentation/dt-object-internal.txt**

Способы использования оверлеев в U-Boot

Существует два способа применения оверлеев:

1. Включить и определить оверлеи в FIT-образ и автоматически их применять;
2. Вручную применять оверлеи;

В оставшейся части документа будет рассмотрено использование оверлеев с помощью ручного подхода. Информацию об использовании оверлеев как части образа FIT можно найти в документе `doc/uImage.FIT/overlay-fdt-boot.txt` загрузчика U-Boot.

Ручная загрузка и применение оверлеев

1. Определите, где разместить базовый devicetree и оверлей к нему. Убедитесь, что у вас достаточно места для наложения оверлея.

```
=> setenv fdtaddr 0x87f00000
=> setenv fdtovaddr 0x87fc0000
```

2. Загрузите базовый devicetree и оверлей к нему:

```
=> load ${devtype} ${bootpart} ${fdtaddr} ${bootdir}/base.dtb
=> load ${devtype} ${bootpart} ${fdtovaddr}
${bootdir}/overlay.dtbo
```

3. Установите базовый devicetree как рабочее дерево FDT:

```
=> fdt addr $fdtaddr
```

4. Увеличте дерево FDT настолько, чтобы было возможно применить к нему все возможные (нужные для вас?) оверлеи:

```
=> fdt resize 8192
```

5. Примените оверлей:

```
=> fdt apply $fdtovaddr
```

6. Загрузите систему, как это делается с традиционным `dtb`.

Для `bootm`:

```
=> bootm ${kerneladdr} - ${fdtaddr}
```

Для `bootz`:

```
=> bootz ${kerneladdr} - ${fdtaddr}
```

Обратите внимание, что в случае ошибки и базовый devicetree, и его оверлеи будут аннулированы, поэтому сохраните копии, чтобы избежать их перезагрузки.

Разделы загрузочного носителя

Использование

```
<command> <interface> [devnum][.hwpartnum][:partnum|#partname]
```

Описание

Многие команды U-Boot позволяют указывать разделы (или целые устройства), используя общий синтаксис (приведён выше).

`interface`

Используется для доступа к разделу(ам) устройства вроде `mmc` или `scsi`. Для получения полного списка поддерживаемых интерфейсов обратитесь к массиву `uclass_idname_str` в файле `drivers/block/blk-uclass.c`.

`devnum`

Номер устройства (по умолчанию `0`).

`hwpartnum`

Номер аппаратного раздела. Все устройства имеют как минимум один аппаратный раздел. На большинстве устройств аппаратный раздел `0` определяет всё устройство. На eMMC аппаратный раздел `0` — это пользовательский раздел, аппаратные разделы `1` и `2` —

загрузочные, аппаратный раздел `3` — раздел RPMB, а последующие — аппаратный пользовательские разделы общего назначения.

По умолчанию `0`.

`partnum`

Номер раздела, начинающийся с единицы. Номер раздела `0` означает, что всё устройство будет использоваться как один «раздел».

`partname`

Имя раздела. Это метка раздела для таблицы GPT. Для MBR разделов используется следующий синтаксис:

```
<devtype><devletter><partnum>
```

`devtype`

Поле `devtype` устанавливается в зависимости от класса устройства:

`devtype`	Класс
`hd`	IDE/SATA
`sd`	SCSI
`usbd`	USB
`mmcsd`	eMMC/SD
`xx`	Другое

См. функцию `part_set_generic_name()` в файле `disk/part.c` для получения полного списка.

---

devletter

Номер устройства как смещение от `a`. Например, устройство 2 будет иметь букву устройства `c`.

---

partnum

Номер раздела. Это то же самое, что и выше.

Если не указаны ни `partname`, ни `partnum` и имеется таблица разделов, то используется раздел 1. Если таблицы разделов на устройстве нет, то всё оно используется как один «раздел». Если не указано ни одно из `devnum`, `hwpartnum`, `partnum` или `partname`, или указано только `-`, то `devnum` по умолчанию принимает значение переменной окружения `bootdevice`.

Примеры

Просмотреть содержимое корневой директории на устройстве MMC 2, аппаратном разделе 1 и разделе №3:

```
ls mmc 2.1:3 /
```

Загрузить `/kernel.itb` в адрес `0x80000000` с устройства SCSI 0, аппаратного раздела 0 и раздела, имеющего название `boot`:

```
load scsi #boot 0x80000000 /kernel.itb
```

Вывести UUID раздела SATA-устройства `$bootdevice`, аппаратного раздела 0 и номера раздела 0:

```
part uuid sata -
```

Приложения

Данная глава содержит дополнительные сведения, такие как, например, приложения, а также иную информацию, которая напрямую не относится к руководству, но, тем не менее, может понадобиться при сборке системы.

Список ПК, для которых собиралась LFA

Orange Pi 3 LTS

- **Бренд процессора:** Allwinner
- **Модель процессора:** H6
- **Архитектура:** Cortex-A53 (AArch64)
- **Версия LFA:** 1.0
- **Дата сборки:** 20.05.2024

Переменные сборки

- `$LFA_HOST=x86_64`
- `$LFA_TGT="aarch64-linux-musleabihf"`
- `$LFA_ARCH="armv8-a"`

Загрузочные скрипты и конфигурационные файлы

Загрузочные скрипты и ряд конфигурационных файлов содержится в директории `$LFA_SYS/etc/*`. В данном приложении приведены листинги этих скриптов и их описания.

/etc/rc.d/shutdown

`/sbin/init` исполняет этот скрипт для выключения системы.

Что делает скрипт

1. Последовательно останавливает загрузочные скрипты, расположенные в `/etc/rc.d/scripts/` ;
2. Синхронизирует системные часы с часами компьютера;
3. Отключает все swap-разделы и/или swap-файлы;
4. Убивает все незавершённые процессы;
5. Размонтирует все подключенные файловые системы;

Листинг

```
#!/bin/ash
# System shutdown script
# (C) 2024 mskrasnov <https://github.com/mskrasnov>

. /etc/rc.d/init.d/functions

echo -e "\nSystem is going down for reboot or halt now.\n"
echo -e "\n\e[1mStarting stop scripts.\e[0m"

export PATH=/bin:/sbin

for i in /etc/rc.d/scripts/*
do
  if [ -x $i ]; then
    $i stop
  fi
done

if [ -x /sbin/hwclock ] && [ -e /dev/rtc0 ]; then
  print_msg "Syncing system clock to hardware clock..."
  hwclock --systohc --utc
  check_status
fi

if [ -x /sbin/swapoff ] && [ -s /etc/fstab ]; then
  print_msg "Disabling swap space..."
  swapoff -a
  check_status
fi

print_msg "Syncing all filesystems..."
sync
check_status

echo "Killing all processes..."
kill -TERM -1
kill -KILL -1

echo "Unmounting all filesystems..."
umount -a -r

exit 0
```

/etc/rc.d/startup

`/sbin/init` выполняет этот скрипт для запуска системных компонентов и служб.

Что делает скрипт

1. Монтирует файловые системы `/proc`, `/sys`, `/tmp`, `/dev`;
2. Монтирует в `tmpfs` все директории, файлы в которых очень часто изменяются (например, логи). Это нужно для того, чтобы продлить жизнь ненадёжным SD- и eMMC-накопителям, на которых обычно будет запускаться система LFA;
3. Запускает `mdev`;
4. Настраивает системные часы;
5. Включает подкачку (swap), если она есть;
6. Настраивает сетевой интерфейс `lo`;
7. Последовательно запускает загрузочные скрипты из директории `/etc/rc.d/scripts/`.

Листинг

```
#!/bin/ash
# System startup script
# (C) 2024 mskrasnov <https://github.com/mskrasnov>

. /etc/rc.d/init.d/functions

export PATH=/bin:/sbin

function mount_to_tmpfs() {
  mount -t tmpfs none -o nodev,nosuid,noatime /var/log &&
  mount -t tmpfs none -o nodev,nosuid,noatime /var/run &&
  mount -t tmpfs none -o nodev,nosuid,noatime /run
}

echo "Mounting kernel virtual file systems..."
mount -vt proc  none /proc
mount -vt sysfs none /sys
mount -vt tmpfs /tmp /tmp

mkdir -pv /dev/pts
mkdir -pv /dev/shm

echo "/sbin/mdev" > /proc/sys/kernel/hotplug

print_msg "Starting mdev..."
mdev -s
check_status

print_msg "Mounting /dev/pts..."
mount -t devpts none /dev/pts
check_status

print_msg "Mounting shared memory..."
mount -t tmpfs none /dev/shm
check_status

print_msg "Mounting temporary, log and variable dirs..."
mount_to_tmpfs
check_status

if [ -x /sbin/hwclock ] && [ -e /dev/rtc0 ]; then
  print_msg "Setting system clock..."
  hwclock --hctosys --utc
  check_status
```

```
fi

if [ -x /sbin/swapon ]; then
  print_msg "Enabling swap..."
  swapon -a
  check_status
fi

print_msg "Setting up interface [lo]..."
ifconfig lo up 127.0.0.1
check_status

echo -e "\n\e[1mRunning bootscripts\e[0m"

for i in /etc/rc.d/scripts/*
do
  if [ -x $i ]; then
    $i start
  fi
done

exit 0
```

/etc/rc.d/init.d/functions

Общие функции для скриптов системы инициализации (`startup`, `shutdown`, загрузочные скрипты). Сейчас там расположены только функции для печати сообщений в терминал.

Что делает скрипт

Объявляет функции `print_msg` и `check_status`.

Листинг

```
#!/bin/ash
# Shared functions and variables
# (C) 2024 mskrasnov <https://github.com/mskrasnov>

function print_msg() {
  echo -ne "[ \e[1;33m***\e[0m ] $1 "
}

function check_status() {
  local ERR=$?
  if [ $ERR -eq 0 ]; then
    echo -e "[ \e[32mOK\e[0m ]"
  else
    echo -e "[\e[31mERRR\e[0m] (code: $ERR)"
  fi
}
```

/etc/rc.d/init.d/scripts/

Скрипты в этой директории запускают конкретные системные компоненты (обычные программы или демоны). Содержат команды для запуска, остановки и перезапуска этих компонентов.

Использование

```
/etc/rc.d/init.d/scripts/SCRIPT_NAME {start|stop|restart}
```

- `start` — запустить скрипт;
- `stop` — остановить скрипт;
- `restart` (опционально) — перезапустить скрипт.

Листинг

Рассмотрим на примере `/etc/rc.d/init.d/scripts/syslog.sh`:

```
#!/bin/ash
# (C) 2024 mskrasnov <https://github.com/mskrasnov>

. /etc/rc.d/init.d/functions

SYSLOG_ROT_SIZE=65536

case $1 in
  start)
    print_msg "Starting syslog..."
    syslogd -m 0 -s $SYSLOG_ROT_SIZE -L
    check_status

    print_msg "Starting klogd..."
    klogd
    check_status
  ;;

  stop)
    print_msg "Stopping klogd..."
    killall klogd
    check_status

    print_msg "Stopping syslogd..."
    killall syslogd
    check_status
  ;;

  restart)
    $0 stop
    $0 start
  ;;

  *)
    echo "Usage: $0 {start|stop|restart}"
    exit 1
  ;;
esac
```

Литература

- Интернет-ресурсы
  - **https://clfs.org/view/clfs-embedded/arm/index.html**
  - https://docs.u-boot.org/en/latest/build/gcc.html#building
  - https://en.wikipedia.org/wiki/ARM_architecture_family
  - https://en.wikipedia.org/wiki/ARM_Cortex-A53
  - https://en.wikipedia.org/wiki/Floating-point_unit
  - https://habr.com/ru/articles/763996/
  - https://habr.com/ru/companies/aladdinrd/articles/338806/
  - https://help.reg.ru/support/dns-servery-i-nastroyka-zony/rabota-s-dns-serverami/fayl-hosts-na-linux#0
  - https://linux-sunxi.org/Manual_build_howto
  - https://linux-sunxi.org/U-Boot#Compile_U-Boot
  - https://smarden.org/runit/
  - https://wiki.archlinux.org/title/core_utilities
  - https://wiki.archlinux.org/title/Systemd_(%D0%A0%D1%83%D1%81%D1%81%D0%BA%D0%B8%D0%B9)
  - https://wiki.gentoo.org/wiki/Comparison_of_init_systems
  - https://www.altlinux.org/Device_Tree
  - https://linuxfromscratch.org/lfs/view/stable-systemd/chapter02/hostreqs.html

Roadmap

- ✅ Обновление основных пакетов (Linux, BusyBox, GCC, Binutils, etc.) до новых версий;
- ✅ Перевод на русский язык;
- ✅ Инструкции по сборке загрузчика U-Boot;
- ✅ Инструкции по созданию `img`-образа системы;

Что нового в этом релизе

LFA 2.0

Вторая крупная версия Linux for ARM (LFA). Основные изменения:

1. Расширение и конкретизация сведений о сборке загрузчика U-Boot для Allwinner, Broadcom, Rockchip и QEMU;
2. Переработка раздела №7: расширил сведения о файлах `boot.cmd` и `boot.scr` ;
3. Переработка раздела №8: добавил для каждого поддерживаемого SoC и эмулятора конкретные инструкции по сборке `img`-образа и загрузке с него;
4. Представлен Automated Linux for ARM (ALFA) — программа для автоматизации сборки пакетов ПО из исходного кода по инструкциям LFA;
5. Расширены сведения о переменных окружения и многопоточной сборке программ;

LFA 1.2

Мелкий промежуточный выпуск

LFA 1.1.1

- Обновление binutils 2.42 -> 2.43;
- Обновление linux 6.1.103 -> 6.6.44;
- Исправление сборочных инструкций ядра Linux;

LFA 1.1

- Понижение версии Linux с 6.9 до 6.1.103 (LTS)

LFA 1.0

Это первая версия руководства LFA, в которой изначально доступно:

- Сборка кросс-компилятора x86_64 -> ARM
- Сборка загрузчика U-Boot для Allwinner SoC [инструкции нестабильны]
- Инструкции по созданию `img`-образа системы [инструкции нестабильны]
- Более новые версии пакетов по сравнению с оригинальной CLFS
- Полный перевод на русский язык

Список изменений

Изменения

- Переработан раздел объявления переменных сборки для не-AArch64-архитектур
- Добавлены пояснения по поводу триплета целевой машины
- Добавлены замечания о профилях производительности
- Добавлена команда для применения изменений параметров BASH
- Объяснено применение команды `su`
- Расширены сведения о переменной `MAKEFLAGS`; добавлена команда для применения параметров BASH
- Добавлена информацию о `/etc/bash.bashrc`
- Добавлен скрипт для проверки наличия необходимых версий ПО на хосте
- Создан отдельный раздел с информацией о сборке загрузочного образа (#75)
- Добавлены сведения о `boot.cmd` и `boot.cmd` (fix #34)
- Добавлены сведения об uImage (fix #33)
- Обновлены ссылки на внешние ресурсы (#73)

Обновления

20.05.2025

- Выпущен релиз 2.0