Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

«РАЗРАБОТКА СИСТЕМЫ СБОРКИ EMBEDDED LINUX»

Автор Яшин Александр Павлович _____
(Фамилия, Имя, Отчество) (подпись)

Направление подготовки (специальность) 09.03.01 –
(код, наименование)
Информатика и вычислительная техника

Квалификация бакалавр
(бакалавр, магистр)

Руководитель ВКР Ключев А. О., к.т.н. доцент _____
(Фамилия, И., О., ученое звание, степень) (подпись)

К защите допустить

Руководитель ОП Алиев Т.И., д.т.н, профессор _____
(Фамилия, И., О., ученое звание, степень) (подпись)

« 22 » мая 2019 г.

Санкт-Петербург, 2019 г.

Студент Яшин Александр Павлович Группа P3402 Факультет ПИиКТ

(Фамилия, Имя, Отчество)

Направленность (профиль), специализация 09.03.01 – Вычислительные машины, комплексы, системы и сети

ВКР принята «_____» _____ 20____ г.

Оригинальность ВКР _____ %

ВКР выполнена с оценкой _____

Дата защиты «_____» _____ 20____ г.

Секретарь ГЭК _____ _____

(Фамилия, И., О.) (подпись)

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

УТВЕРЖДАЮ

Руководитель ОП

Алиев Т.И. _____

(Фамилия, И.О.) (подпись)

« _____ » _____ 20 ____ г.

З А Д А Н И Е

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Студенту Яшину А.П. **Группа** P3402 **Факультет** ПИиКТ

Руководитель ВКР Ключев Аркадий Олегович, к.т.н., доцент, Университет ИТМО

(ФИО, ученое звание, степень, место работы, должность)

1 Наименование темы Разработка системы сборки Embedded Linux

Направление подготовки (специальность) 09.03.01 – Информатика и вычислительная техника

Направленность (профиль) Вычислительные машины, комплексы, системы и сети

Квалификация бакалавр

2 Срок сдачи студентом законченной работы « _____ » _____ 20 ____ г.

3 Техническое задание и исходные данные к работе

Изучить специфику работы встраиваемых устройств на основе Linux, провести обзор существующих решений, реализующих сборку Linux для встраиваемых устройств, провести сравнение.

Разработать систему сборки Embedded Linux, провести тестирование и апробацию.

Требования к системе сборки:

- Кроссплатформенность;
- Возможность вручную конфигурировать необходимые предустановленные в образ пакеты;
- Возможность одновременной сборки нескольких образов: минимального, серверного, расширенного;

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

4.1 Анализ предметной области;

4.2 Обзор существующих решений, реализующих сборку Linux для встраиваемых устройств;

4.3 Описание процесса проектирования, разработки и тестирования системы сборки;

4.4 Апробация результатов разработки.

5 Перечень графического материала (с указанием обязательного материала)

Презентация по проделанной работе (в формате PDF)

Слайды:

- ”Постановка задачи”

- ”Обзор существующих решений”

- ”Примеры результатов работы системы”

6 Исходные материалы и пособия

6.1 Аппаратные и программные средства встраиваемых систем [Электронный ресурс]: учебное пособие/ А.О. Ключев [и др.]. — Электрон. текстовые данные. — СПб.: Университет ИТМО, 2010. — 291 с. — Режим доступа: https://books.ifmo.ru/file/pdf/686.pdf

6.2 Yocto Project Mega-Manual [Электронный ресурс] // Linux Foundation. - 2010-2019. - Режим доступа: https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html

6.3 Armbian Documentation [Электронный ресурс] // Armbian. - 2019. - Режим доступа: https://docs.armbian.com

6.4 The Buildroot user manual [Электронный ресурс] // Buildroot Association. - 2019 - Режим доступа: https://buildroot.org/downloads/manual/manual.html

7 Дата выдачи задания « _____ » _____ 20 ____ г.

Руководитель ВКР _____
(подпись)

Задание принял к исполнению _____ « _____ » _____ 20 ____ г.
(подпись)

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

АННОТАЦИЯ

ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Студент Яшин Александр Павлович
(Фамилия, Имя, Отчество)

Наименование темы ВКР Разработка системы сборки Embedded Linux

Наименование организации, где выполнена ВКР Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования Проектирование и реализация системы сборки Linux дистрибутивов для встроенных систем.

2 Задачи, решаемые в ВКР

2.1 Провести анализ существующих решений, реализующих сборку Linux для встраиваемых устройств;

2.2 Определить требования к системе сборки;

2.3 Спроектировать архитектуру системы сборки;

2.4 Реализовать программное обеспечение системы сборки в соответствии с разработанной архитектурой;

2.5 Произвести тестирование системы.

3 Число источников, использованных при составлении обзора 5

4 Полное число источников, использованных в работе 11

5 В том числе источников по годам

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
0	1	0	10	0	0

6 Использование информационных ресурсов Internet да, 6
(да, нет, число ссылок в списке литературы)

7 Использование современных пакетов компьютерных программ и технологий

Пакеты компьютерных программ и технологий	Раздел работы
Командный процессор bash	3, 4
Утилита debootstrap	3
Менеджер пакетов APT	3
ПО для развёртывания и управления приложениями в средах с поддержкой контейнеризации Docker	3

8 Краткая характеристика полученных результатов В ходе работы над дипломным проектом была спроектирована и реализована система сборки Linux дистрибутивов для встроенных систем. Система сборки была протестирована сотрудниками компании Emlid. Проведена апробация результатов работы системы. Созданная система сборки используется в коммерческом проекте Emlid Neutis-n5.

9 Полученные гранты, при выполнении работы нет
(название гранта)

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы нет
(да, нет)

а) 1 _____
(Библиографическое описание публикаций)

б) 1 _____
(Библиографическое описание выступлений на конференциях)

Студент _____ _____
(Фамилия, И., О.) (подпись)

Руководитель _____ _____
(Фамилия, И., О.) (подпись)

« _____ » _____ 20 ____ г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы. В настоящее время встраиваемые системы используются во сногих сферах человеческой деятельности. С помощью встраиваемых систем множество сложных процессов способны осуществляться легко, безопасно, вовремя, точно и непрерывно. Такие системы вносят серьезные изменения в повседневную жизнь людей.

Ядро Linux может работать на разных компьютерных архитектурах, большинство из которых довольно популярны во встроенном мире. Все базовые пакеты, позволяющие ОС выполнять основные задачи, подходят для кросс-компиляции, поэтому Linux может быть таким же распространенным, как микроконтроллеры и системы на кристалле (SoC).

Цель работы. Целью данного дипломного проекта является проектирование и реализация системы сборки Linux дистрибутивов для встроенных систем.

Задачи работы. Для достижения поставленной цели необходимо решить следующие задачи:

- провести анализ существующих систем сборки
- определить требования к системе сборки
- спроектировать архитектуру системы сборки
- реализовать программное обеспечение системы сборки в соответствии с разработанной архитектурой
- произвести тестирование и апробацию

Апробация результатов работы. Наличие документации позволило осуществить открытое тестирование приложения пользователями и, как результат, получить отзывы, сообщения об ошибках и пожелания к функциональности.

Объем и структура работы. Работа содержит 43 страницы печатного текста, 11 рисунков, список литературы, включающий 11 источников. Работа состоит из введения, четырех частей и заключения. Во введении обоснована актуальность работы, определены цель и задачи исследования. В первой части произведено краткое описание встроенных систем, а также анализ существующих решений, реализующих сборку Linux дистрибутивов. Вторая часть содержит описание процесса проектирования архитектуры системы сборки. Третья глава описывает реализацию программного обеспечения. Четвертая глава описывает тестирование реализованного программного обеспечения. В заключении приведены основные результаты работы и возможные направления для дальнейшего развития.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Встроенные системы

Встроенные вычислительные системы (ВВС) – специализированные (заказные) вычислительные системы (ВС), непосредственно взаимодействующие с объектом контроля или управления и объединенные с ним единой конструкцией [1].

В силу специфики использования, обычно встроенные системы имеют низкое энергопотребление, предназначены для выполнения определенного набора задач и не являются универсальной вычислительной платформой, такой как компьютеры. По этим причинам, многие встроенные системы не используют операционную систему. Тогда программист проводит собственную разработку всего программного обеспечения, которое управляет оборудованием, практически не задействовав многозадачность и взаимодействие с пользователем. Программное обеспечение в таком случае работает непосредственно с оборудованием встроенной системы.

Однако в современном мире существуют более сложные встроенные системы, с расширенным списком решаемых задач. В такой многозадачной системе, должны быть реализованы следующие механизмы:

- распределение памяти;
- предоставление времени ЦП разным потокам, процессам;

В таком случае программист может использвать ОС, это подход позволяет сосредоточиться на решении задачи, возложенной на ВВС, делегировав управление аппаратными средствами ОС.

Однако операционные системы очень трудно создавать, и это приводит к увеличению количества строк кода в проекте. При этом важно отметить, что влияние на количество ошибок в коде оказывает количество строк, то есть чем больше строк, тем больше ошибок [1]. По этой же причине устранение

всех ошибок в практически значимом ПО слишком трудоёмко, вместо этого при его создании обычно пытаются достичь максимально возможного при заданных затратах уровня качества, как можно больше снизить вероятность проявления ошибок и ущерба от них. Так как операционные системы развиваются и поддерживаются в течение долгого периода времени[2], то во встроенных решениях часто используют уже существующие ОС.

1.2 Linux во встроенных системах

Большое распространение во встроенных системах получили ОС на основе Linux.

Дистрибутив Linux - это операционная система, в основе которой лежит ядро Linux и, зачастую, система управления пакетами. Дистрибутив может быть представлен в виде предварительно скомпилированных двоичных файлов и пакетов, собранных сопровождающими дистрибутива, или в виде источников в сочетании с инструкциями о том, как их скомпилировать.

Среда разработки в программировании встроенных систем обычно сильно отличается от сред тестирования и производства, они могут использовать разные архитектуры чипов, программные стеки и даже операционные системы. Во встроенных системах, поскольку аппаратная платформа зачастую выполняется на заказ, разработчик ОС обычно предпочитает генерировать дистрибутив с нуля, из источников.

Как правило, дистрибутив состоит из полного образа программного обеспечения для целевого устройства, включая ядро, драйверы устройств, библиотеки, прикладное программное обеспечение и загрузчик.

1.3 Обзор существующих систем сборки

В настоящее время существует достаточно большое количество систем, реализующих сборку Linux дистрибутивов для встроенных систем.

Система сборки embedded linux - механизм для построения создания Linux дистрибутивов, обладающий следующими свойствами:

- позволяет указать архитектуру оборудования;
- позволяет интегрировать приложения пользовательского пространства в образ;
- разрешает параллельную сборку;
- включает набор инструментов для кросс компиляции;

Системы сборки нацелены на решение задачи создания образа Linux дистрибутива. В рамках проведённого обзора были рассмотрены популярные продукты, позволяющие осуществлять сборку дистрибутивов Linux:

- **Open Embedded/Yocto** – система для создания полных встроенных изображений с нуля. Используется проектом Yocto в качестве системы сборки.
- **Buildroot** – система сборки, направленная на простоту использования.
- **Системы адаптации дистрибутивов** – системы, удалющие ненужные компоненты из настольных дистрибутивов.

1.3.1 Open Embedded/Yocto

Проект Yocto определяется как «проект с открытым исходным кодом, который предоставляет шаблоны, инструменты и методы, помогающие создавать пользовательские системы на основе Linux для встроенных продуктов независимо от аппаратной архитектуры»[3]. Это набор рецептов, конфигурационных файлов и зависимостей, используемых для создания пользовательского образа Linux, адаптированного к конкретным пользовательским потребностям. Yocto использует Openembedded в качестве своей системы сборки. Технически это два отдельных проекта, названия проектов часто используются взаимозаменяемо. Результат сборки в целом состоит из трех компонентов:

- **Целевые двоичные файлы** – к ним относятся загрузчик, ядро, модули ядра, образ корневой файловой системы и любые другие вспомогательные файлы, необходимые для развертывания Linux на целевой платформе.

- **Хранилище пакетов** – это набор пакетов программного обеспечения, доступных для установки на устройство. Есть возможность выбрать формат пакета (например, deb, rpm, ipk) в зависимости от потребностей. Некоторые из них могут быть предварительно установлены в целевые исполняемые файлы, однако можно создавать пакеты для установки в развернутую и функционирующую систему. Все части rootfs являются пакетами (рис. 1.1).

- **Целевая SDK** – это набор библиотек и заголовочных файлов, представляющих программное обеспечение, установленное на вашей целевом устройстве. Заголовочные файлы используются разработчиками приложений при создании своего кода, чтобы гарантировать, что они связаны с соответствующими библиотеками.

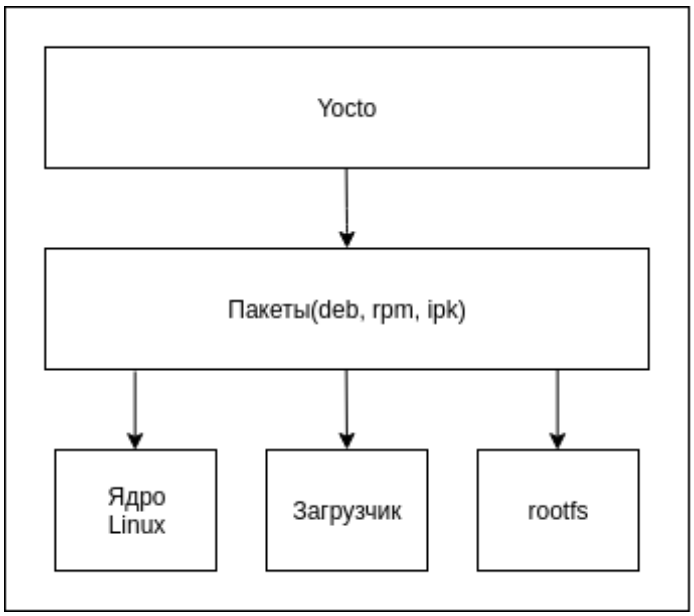

Рисунок 1.1 – Структура yocto образа

Проект Yocto широко используется в отрасли и имеет поддержку многих влиятельных компаний. Кроме того, у этого есть большое и энергичное сообщество разработчиков и экосистема, способствующая этому. Сочетание энтузиастов с открытым исходным кодом и корпоративных спонсоров помогает вести проект[3].

Преимущества использования Yocto:

- Проект легко расширяется через **слои**, которые можно публиковать независимо для добавления дополнительных функций или для хранения настроек, уникальных для целевой системы.

- Широкая поддержка со стороны многих производителей полупроводников и плат.

- Настройки для отдельного приложения могут храниться в слое для инкапсуляции и изоляции. Настройки, уникальные для слоя, как правило, хранятся как часть самого слоя, что позволяет применять одни и те же настройки одновременно к нескольким системным конфигурациям.

- Yocto также предоставляет четко определенный уровень приоритета за-

дач и возможность их переопределения. Это позволяет определить порядок применения слоев и поиска метаданных. Это также позволяет переопределить настройки в слоях с более высоким приоритетом.

Недостатки использования Yocto:

- Время сборки и ресурсы разработки достаточно высоки[3]. Так как количество пакетов, которые необходимо собрать, является значительным, а сборка происходит в несколько потоков. Рабочие станции для разработчиков Yocto, как правило, представляют собой большие системы. Однако, в Yocto имеется встроенный механизм кэширования, который позволяет повторно использовать ранее созданные компоненты, когда параметры для создания определенного пакета не изменились.

1.3.2 Buildroot

Проект Buildroot это «простой, эффективный и простой в использовании инструмент для генерации встроенных систем Linux посредством кросс-компиляции»[4]. Он разделяет многие из тех же целей, что и проект Yocto, однако он сфокусирован на простоте использования. По умолчанию Buildroot отключит все необязательные настройки времени компиляции для всех пакетов, что приведет к созданию образа с минимальным объемом. Разработчик системы должен будет включить настройки, необходимые для его встроенной системы.

Buildroot собирает все компоненты из исходного кода, но не поддерживает управление пакетами. Нет механизма для установки новых пакетов в работающую систему. Результат сборки в целом состоит из трех компонентов:

- **Основные модули** – ядро, загрузчик и модули ядра, соответствующие целевому оборудованию(рис. 1.2).;
- **rootfs** – образ корневой файловой системы и любые другие вспомогательные файлы, необходимые для развертывания Linux на целевой платформе;
- **Набор инструментов,** которые использовалась для сборки всех целевых двоичных файлов;

Преимущества использования Buildroot:

- Базовая система сборки написана на Make[4] и достаточно коротка, чтобы позволить разработчику понять всю систему, в то же время будучи достаточно расширяемой для удовлетворения потребностей разработчиков встраиваемых Linux-систем; Ядро Buildroot обычно обрабатывает только общие случаи использования, но его можно расширять с помощью сценариев;

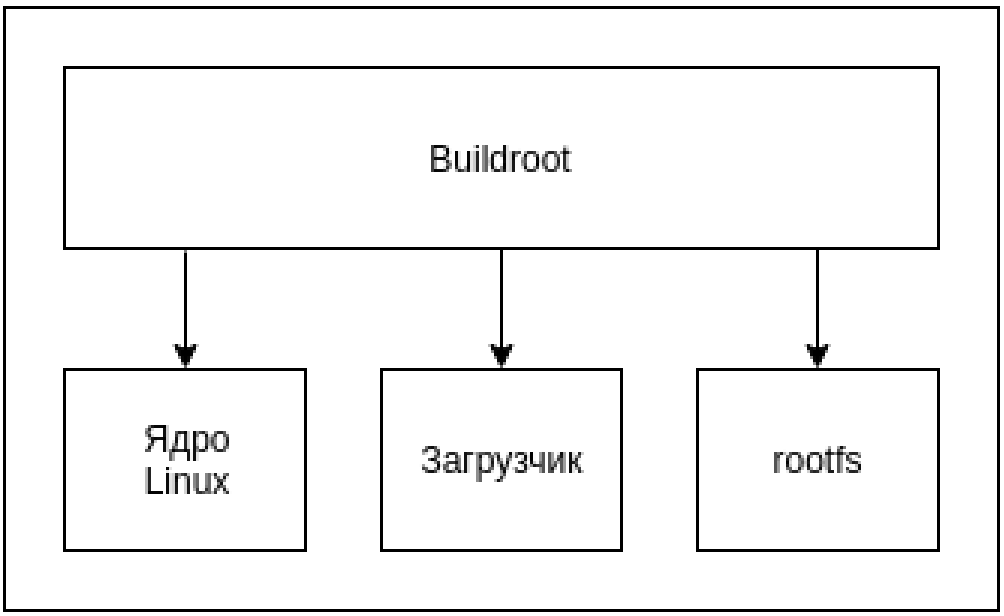

Рисунок 1.2 – Структура buildroot образа

- Система Buildroot использует для своей конфигурации Makefile и язык Kconfig[4]. Kconfig был разработан сообществом разработчиков ядра Linux[5] и широко используется в проектах с открытым исходным кодом, что делает его знакомым для многих разработчиков;

- В связи со способом сборки, заключающейся в отключении всех необязательных настроек для каждого отдельного пакета, Buildroot обычно создает наименьшие по объему возможные образы с использованием готовой конфигурации. Время сборки и сборки ресурсов хоста в целом также будет меньше, чем с использованием Yocto;

Недостатки использования Buildroot:

- Акцент на простоте и минимальных включенных параметрах сборки подразумевает, что разработчику может потребоваться выполнить объемную настройку сборки для пакета, если используется нестандартная конфигурация. Кроме того, все параметры конфигурации для пакета хранятся в одном файле[4], а это означает, что если проект поддерживает несколько аппаратных платформ, разработчику нужно будет вносить изменения в каждую настройку для каждой платформы;

- Любое изменение в файле конфигурации системы требует полной пересборки всех пакетов;

1.3.3 Системы адаптации дистрибутивов

Распространенный подход к проектированию встраиваемых систем на основе Linux заключается в том, чтобы начать с настольного дистрибутива, такого как Debian или Red Hat, и удалять ненужные компоненты, пока установленный образ не поместится в пространство целевого устройства[6]. Это подход, принятый для популярного дистрибутива Raspbian для платформы Raspberry Pi.

Преимущества использования системы адаптации дистрибутивов:

- Основным преимуществом этого подхода является ознакомленность разработчика с содержимым дистрибутива. Часто разработчики встраиваемых Linux-систем также являются пользователями настольных Linux-систем и хорошо разбираются в своем дистрибутиве. Использование аналогичной среды на цели может позволить разработчикам начать работу быстрее. В зависимости от выбранного дистрибутива, многие дополнительные инструменты могут быть установлены с использованием стандартных пакетных менеджеров, таких как apt и yum;

- Количество пакетов в репозиториях пакетных менеджеров, большинства настольных дистрибутивов, обычно больше, чем рецептов для Yocto или Buildroot;

- Благодаря встроенному менеджеру пакетов, такой дистрибутив удобно использовать во время разработки продукта в качестве отладочного окружения;

Недостатки использования системы адаптации дистрибутивов:

- Получить воспроизводимую среду таким способом сложно так как поставщики пакетов могут обновлять их версии;

- Добавление и удаление пакетов вручную подвержено ошибкам;

1.4 Постановка задач исследования

В рамках данной работы требуется спроектировать и реализовать программное обеспечение для сборки дистрибутивов операционных систем на основе Linux для встроенных систем.

Для достижения поставленной цели необходимо решить следующие задачи:

- определить требования к системе;
- спроектировать архитектуру системы;
- реализовать программное обеспечение системы в соответствии с разработанной архитектурой;
- провести тестирование и апробацию;

2 РАЗРАБОТКА АРХИТЕКТУРЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Требования к реализуемой системе

Разработка любого ПО должна начинаться с определения требований, которые предоставлены к нему[7]. Так, чтобы определить функциональные требования, нужно проанализировать задачи, на решение которых направлено разрабатываемое программное обеспечение.

Требования к разрабатываемому продукту можно разделить на функциональные и нефункциональные[7]. Функциональные требования определяют, требуемое поведение системы в определенных условиях. Нефункциональные требования определяют свойства или особенности, которым должна обладать система, или ограничение, которому должна соблюдать система. Другими словами, требования, отличные от функциональных, могут описывать не что система делает, а как хорошо она это делает[7].

Ниже приведены наиболее важные идеи, задачи и требования, предъявляемые к разрабатываемому продукту. К системе сборки были поставлены следующие функциональные требования:

- кроссплатформенность;
- возможность вручную конфигурировать необходимые предустановленные в образ пакеты;
- возможность одновременной сборки нескольких образов: минимального, серверного, расширенного;
- наличие загрузчика в образах;
- наличие пакетного менеджера в образах;
- пригодность для использования образов во время разработки ВС;

2.2 Проектирование архитектуры системы

Для построения архитектуры системы сборки необходимо поэтапно рассмотреть процедуру разработки образа для ВС.

- Начальным этапом при работе с ситемой является подготовка окружения. Так как сборка дистрибутива подразумевает выгрузку и компиляцию всех необходимых зависимостей, очевидна необходимость в наличии свободного пространства на жестком диске хост-системы.

- Зачастую, встраиваемые системы это системы, выполняемые на заказ [1], поэтому очевидно необхдимым при разаработке дистрибутива Linux для такой системы является конфигурация устанавливаемых в образ пакетов. По этой же причине, необходимым для внесения в результирующий образ являются патчи для ядра Linux, а также для загрузчика.

- Следующим этапом после внесения изменений является отладка файлов-рецептов и их версионирование доступной системой контроля версий для достижения воспроизводимости сборки.

- На следующем шаге происходит сборка дистрибутива. Пользователь дожен иметь возможность указать конфигурационные параметры для сборки, такие как максимальное количество параллельных потоков для утилиты make и т.п..

- Последним шагом является загрузка образа дистрибутива на устройство.

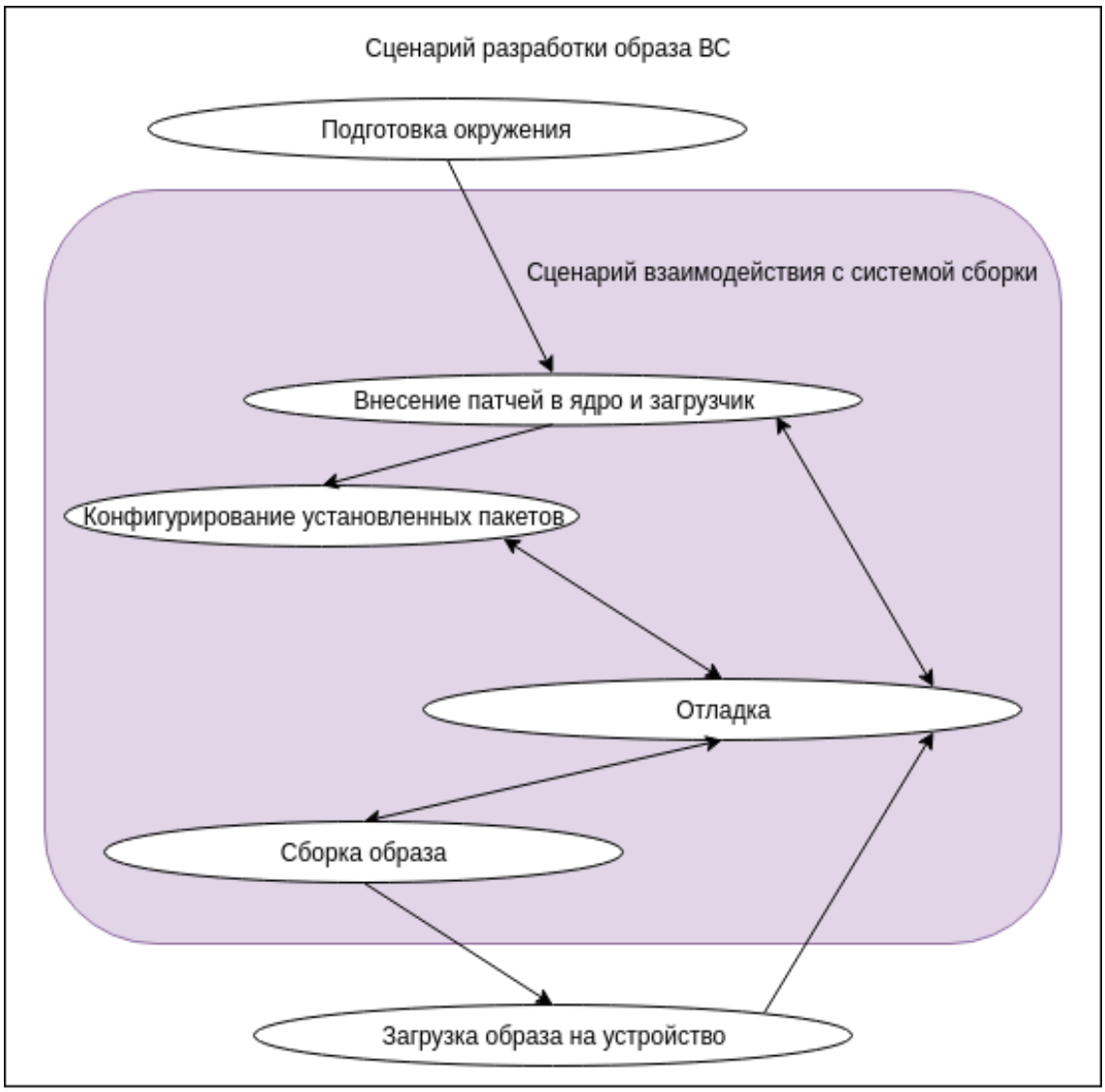

Рисунок 2.1 – Сценарий разработки образа ВС

Следует отметить, что этап подготовки окружения и загрузки образа на устройства остаются за рамками работы с системой сборки. Одним из основных вопросов, требующих рассмотрения на шаге подготовки окружения является обеспечение необходимого для сборки дискового простанства. Кроме того, этот шаг подразумевает установку необходимых для системы сборки зависимостей на хост-систему. Процесс загрузки образа на устройство может отличаться на различных устройствах, поэтому решение этой задачи также остается за рамками данной работы.

2.3 Проектирование фазы жизненного цикла системы

Для построения архитектуры жизненного цикла системы сборки необходимо поэтапно рассмотреть действия, необходимые для непосредственного создания образа. При создания архитектуры жизненного цикла используется модульная архитектура. В этом случае разрабока компонентов может быть проведена вне зависимости друг от друга.

Рисунок 2.2 – Этапы создания образа

Кроме того, исходя из требований к системе сборки, важно отметить что полученное решение должно быть кроссплатформенным. Так, чтобы обеспечить выполнение этого требования предлагается модифицировать общую архитектуру системы с использованием виртуализации на уровне операционной системы(контейнеризации):

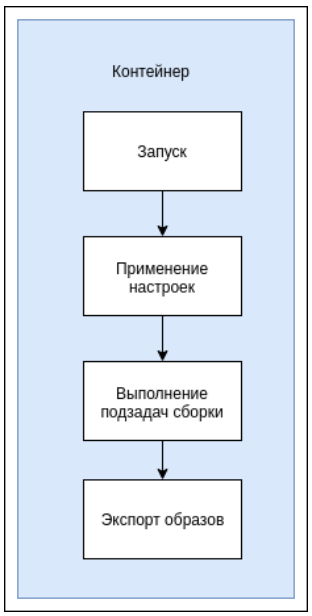

Рисунок 2.3 – Архитектура с использованием контейнризации

2.3.1 Этап запуска системы

Начальным этапом создания образа является заупск системы сборки (рис. 2.4). На этом этапе система должна выполнить проверку среды исполнения, удостовериться в наличии прав суперпользователя, а также поизвести считывание параметров из конфигурационного файла.

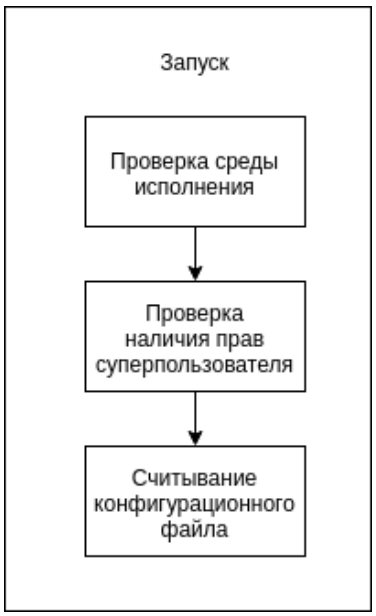

Рисунок 2.4 – Запуск системы сборки

2.3.2 Этап применения настроек

Применение настроек из конфигурационного файла (рис. 2.5). На этом этапе система должна выполнить экспорт глобальных констант для сборки. Кроме того система устанавливает флаги игнорирования стадий сборки и игнорирования стадий экспорта для образа.

Рисунок 2.5 – Применение настроек из конфигурационного файла

2.3.3 Этап выполнения подзадач

Выполнение подзадач сборки (рис. 2.6). На этом этапе система выполняет prerun-шаг, а после этого производит выполнение подшагов от 00 до 99. Передача созданной на предыдущих шагах коренвой файловой системы(rootfs) к следующим упрощает горизонтальное масштабирование.

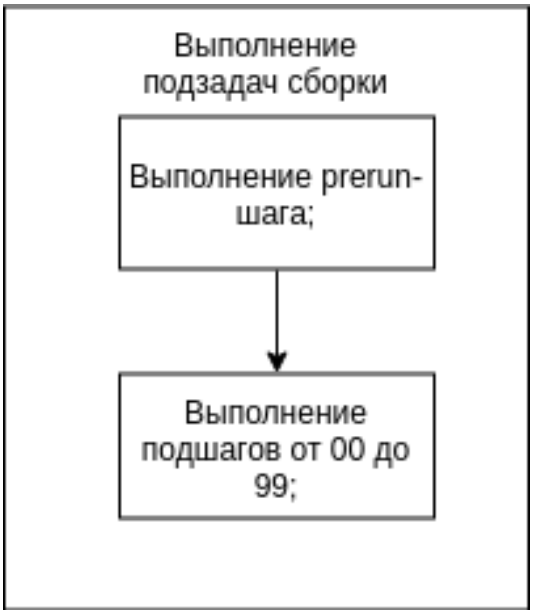

Рисунок 2.6 – Выполнение подзадач сборки

2.3.4 Этап экспорта образов

Экспорт образов (рис. 2.7). На этом этапе система выполняет ряд завершающих и обязательных действий среди которых разметка файла образа, установка основных пакетов, ядра Linux, загрузчика. После чего производит сохранение образа.

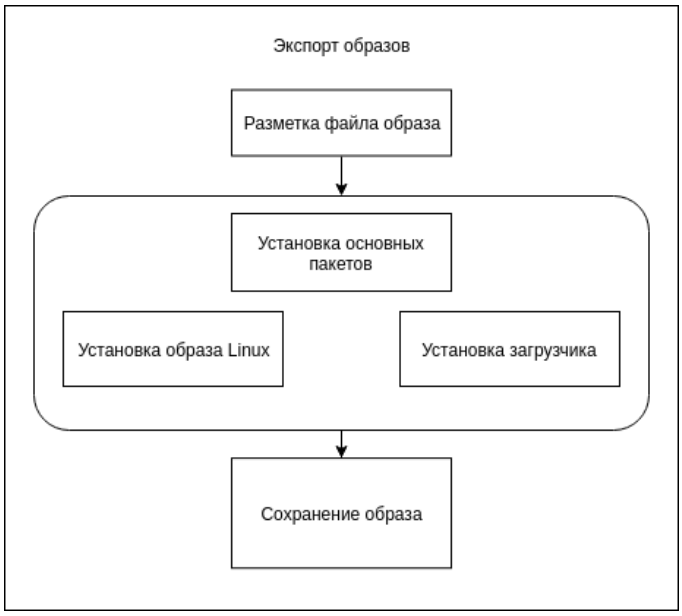

Рисунок 2.7 – Экспорт образов

3 ОПИСАНИЕ РЕАЛИЗАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

3.1 Подготовка окружения для разработки

Основным средсвом для реализации системы сборки является командная оболочка bash. Система также основана на утилитах debootstrap и chroot.

Bash - это командный процессор, соответвующий стандарту POSIX и предоставляющий пользователям интерфейс для взаимодействия с другими программами. Bash также обладает возможностью читать команды из файлов скриптов, состоящих из списка команд. Как и многие другие оболочки, в bash поддерживается автодополнение имён дерева каталогов. Также в bash поддерживаются переменные, существуют стандартные конструкции для ветвления, циклов, объявления функций и т. п.. Многие лексические особенности были заимствованы из оболочки sh. Кроме того, некоторые функции были заимствованы из csh и ksh.

Debootstrap - инструмент, позволяющий установить базовую систему Debian в поддиректорию уже существующей системы. Для этого не требуется установочный компакт-диск, необходимо лишь обеспечить доступность к репозитория Debian. Его также можно установить и запустить из другой ОС, поэтому, например, существует возможность использовать debootstrap для инсталяции Debian на свободный раздел работающей системы Gentoo. Инструмент также может использоваться при создании rootfs для машины с другой архитектурой, которая называется «перекрестная перезагрузка»[8].

Chroot - Unix системах это операция, которая изменяет корневой каталог для текущего запущенного процесса и всех его дочерних процессов. Программа, которая запускается в такой измененной среде, не имеет доступа к файлам за пределами указанного дерева каталогов.

3.2 Контейнеризация

Для дострижения требования кроссплатформенности предлагается обернуть систему сборки в контейнер. Контейнеры - это способ инкапсуляции приложения с его зависимостями. На первый взгляд, они кажутся просто облегченной формой виртуальных машин (ВМ) - подобно виртуальной машине, контейнер содержит изолированный экземпляр операционной системы (ОС), который можно использовать для запуска приложений [9]. Однако контейнеры имеют ряд преимуществ, которые позволяют использовать их в задачах, которые являются сложными или невозможными для традиционных виртуальных машин:

- Контейнеры совместно используют ресурсы с операционной системой хоста, что делает их на порядок более эффективными.
- Приложения, работающие в контейнерах, требуют минимальных или нулевых накладных расходов по сравнению с приложениями, работающими на основной операционной системе.
- Портативность контейнеров может устранить целый класс ошибок, которые могут вызываться небольшими изменениями в рабочей среде.
- Пользователи могут загружать и запускать сложные приложения без необходимости тратить часы на конфигурацию и установку.

Для разрабатываемой системы сборки виртуализации на уровне операционной системы необходима по причине использования небезопасных вызовов chroot с динамически определяемыми аргументами. Это может повлечь нарушение целостности хост-системы. В качестве средства управления контейнерами был выбран docker как один из самых распространенных и известных автору. Для менеджера контейнеров был написан конфигурационный файл, создающий минимально необходимое для сборки окружение.

3.3 Структура проекта

Сборка образов разделена на несколько стадий, а стадии на несколько шагов для логической ясности и модульности. Структура системы подробно описана на рисунке 3.1.

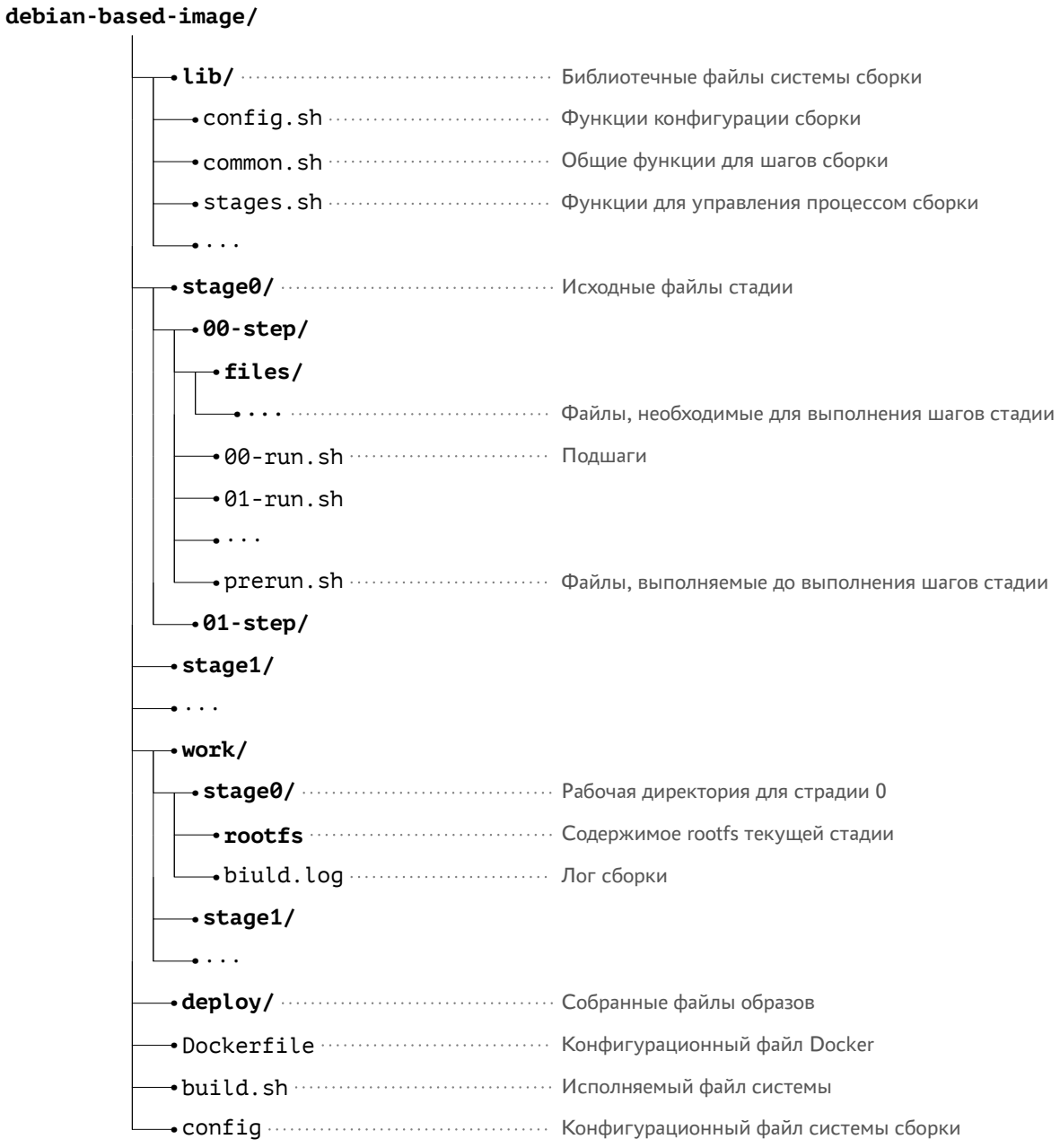

Рисунок 3.1 – Структура проекта

3.4 Стадии сборки

3.4.1 Создание файловой системы

Основная цель этого этапа - создать удобную файловую систему. Для генерации базовой системы при создании системы объявляется функция bootstrap. В случае, если арихтектура операционной системы-хоста arm64, используется классическая утилита debootstrap. В противном случае используется оболочка debootstrap для qemu.

Листинг 3.1 – Реализация функции bootstrap

```
bootstrap(){
  local ARCH=$(dpkg --print-architecture)

  if [ "$ARCH" !=  "arm64" ]; then
    local BOOTSTRAP_CMD=qemu-debootstrap
  else
    local BOOTSTRAP_CMD=debootstrap
  fi

  capsh --drop=cap_setfcap -- -c "${BOOTSTRAP_CMD} \
    --components=main,contrib,non-free \
    --arch arm64 \
    --variant=minbase \
    $1 $2 $3" || rmdir "$2/debootstrap"
}
export -f bootstrap
```

После получения базовой файловой системы, необходимо добавить конфигурирование пакетного менеджера, установку пакетов локалей. Кроме того, необходимо выполнить деинсталяцию окружения рабочего стола gnome, так как пакеты gnome являются достаточно объемными и нет необходимости включать их в результирующий образ. Для работы с .deb пакетами в базовой файловой систем используется утилита chroot, для проведения манипуляций с пакетами внутри базовой файловой системы используется пакетный менеджер apt.

Листинг 3.2 – Деинсталяция окружения gnome

```
#!/bin/bash -e

on_chroot << EOF
apt-get purge `dpkg --get-selections | grep gnome | cut -f 1`
apt-get purge `dpkg --get-selections | grep deinstall | cut -f 1`
apt autoremove
EOF
```

3.4.2 Кросс-компиляция зависимостей

На этом этапе накладываются патчи и компилируются важные зависимости для будущего образа, такие как Linux, u-boot, ARM Trusted Firmware и другие. Для обеспечения кросс-компиляции в системе сборки используется набор утилит для компиляции и отладки программ компании Linaro.

Листинг 3.3 – Загрузка утилит компиляции Linaro

```
#!/bin/bash -e

GCC_LINARO_ARCHIVE_NAME="gcc-linaro.tar.gz"

if [ ! -f "${DOWNLOADS_DIR}/${GCC_LINARO_ARCHIVE_NAME}.done" ]; then
    wget ${GCC_SOURCE_LINK} \
    --output-document=${DOWNLOADS_DIR}/${GCC_LINARO_ARCHIVE_NAME} \
     --no-verbose
    touch "${DOWNLOADS_DIR}/${GCC_LINARO_ARCHIVE_NAME}.done"
fi

if [ ! -f "${GCC_LINARO_SRC}.done" ]; then
    tar -xf ${DOWNLOADS_DIR}/${GCC_LINARO_ARCHIVE_NAME} \
        --directory ${GCC_LINARO_SRC} \
        --strip-components=1
    touch "${GCC_LINARO_SRC}.done"
fi
```

После выполнения этого этапа результирующий образ системы способен загружаться, настраивается загрузчик, обеспечивается работоспособ-

ность сети. На этом этапе результирующая система может загрузиться с консоли, в которой у пользователя есть средства для выполнения основных задач, необходимых для настройки и использования дистрибутива. Эта система настолько минимальна, насколько это возможно, и ее пока нельзя реально использовать в традиционном смысле, так как все еще остается необходимость добавить ряд пакетов, необходимых для разработки и отладки встроенных систем.

3.4.3 Создание серверного образа

Система устанавливает некоторые инструменты разработки, добавляет поддержку Wi-Fi и Bluetooth и другие основные пакеты для управления оборудованием. Происходит инсталяция таких пакетов как ssh, python3, i2c-tools, alsa-utils и других. Кроме того, на этом этапе накладывается патч на конфигурационный файл демона sshd, добавляющий возможность доступа по ssh для суперпользователя. Наложение патчей происходит с использованием функции apply_patches из файла lib/common.sh.

Листинг 3.4 – Реализация функции apply_patches

```
apply_patches()
{
  local patches_dir=$1
  local patchfiles=`ls $patches_dir | grep .patch`
  if [ -e "PATCHED" ]; then
    echo "${patches_dir} is patched already. Nothing to do."
  else
    for patch in $patchfiles; do
      echo "Applying $patch patchfile."
      patch --batch --silent -p1 -N < "${patches_dir}/${patch}"
    done;
    touch "PATCHED"
  fi
}
export -f apply_patches
```

3.4.4 Экспорт образов

На этом этапе система размечает файл образа и создает файловую систему. После этого происходит упаковка и установка скомпилировнных на предыдущих шагах пакетов.

Листинг 3.5 – Реализация функций упаковки и установки .deb пакетов

```
build_deb_package()
{
  local package_name=${PACKAGE_DEB_DIR}.deb

  cp files/control ${PACKAGE_DEB_DIR}/DEBIAN
  fakeroot dpkg-deb --build ${PACKAGE_DEB_DIR}
  cp ${package_name} ${ROOTFS_DIR}/var/cache/apt/archives
}
export -f build_deb_package

install_deb_package()
{
  local package_name=${PACKAGE_NAME}.deb
  on_chroot << EOF
dpkg -i /var/cache/apt/archives/${package_name}
EOF
}
```

После окончания установки пакетов на этой стадии созданный файл образа маркируется датой создания и перемещается в директорию доступную для дальнейшей выгрузки.

3.5 Сборка образов с различными конфигурациями

В зависимости от заданных пользователем параметров, в процесс создания образа могут быть включены дополнительные этапы, что может повлечь сборку нескольких образов: минимального, серверного и полного.

3.5.1 Минимальный образ

Этот образ содержит минимальную файловую систему для дальнейших манипуляций с ней. Не установлено никаких дополнительных пакетов. Это достигается в основном за счет использования debootstrap, который создает минимальную файловую систему, подходящую для использования в качестве основы rootfs в системах Debian.

3.5.2 Серверный образ

Этот образ расширяет количество предустановленных пакетов, по умолчанию добавляются пакеты ssh, vim, python3, parted и другие. Он также обладает сконфигурированными польщовательскими группами и предоставляет пользователю доступ к sudo и стандартным группам полномочий.

3.5.3 Полный образ

Полный образ является расширенным серверным образом и включает в себя среду рабочего стола xfce4.

4 ТЕСТИРОВАНИЕ И АПРОБАЦИЯ РАЗРАБОТАННОЙ СИТЕМЫ

4.1 Функциональное тестирование

Для тестирования было выбрано устройство Emlid Neutis N5. Устройство представляет собой систему на модуле (System on Module, SoM), рассчитанную на разработчиков, проектирующих различные устройства для Интернета вещей и «умного» дома [10]. Это устройство объединяет четыре ядра ARM Cortex-A53 и графический контроллер ARM Mali-450MP4 с поддержкой OpenGL ES 2.0/1.1, что позволит провести также тестирование полной версии образа [10].

Функциональное тестирование приложения проводилось в ручном режиме.

4.2 Конфигурация пакетов

Для тестирования выходных образов системы сборки на выбранном устройстве необходимо сконфигурировать загрузчик и ядро Linux под выбранную платформу. В данной главе в силу объемности конфигурационных файлов будет рассмотрена лишь процедура обновления пакета загрузчика для выбранной платформы. Примеры конфигурирования и обновления прочих пакетов доступны в приложении к работе. Обновление загрузчика осуществляется следующим образом:

1) Загрузка yocto слоя с патчами для загрузчика и ядра.

Листинг 4.1 – Загрузка yocto слоя

```
1 #!/bin/bash -e
2
3 META_EMLID_NEUTIS_SOURCE_LINK="https://github.com/Neutis/meta-emlid-neutis.git"
4 META_EMLID_NEUTIS_BRANCH="sumo"
5 META_EMLID_NEUTIS_COMMIT_HASH=e020b06b5c5b36581ae72ab7a4b932913cdf96a0
6
```

```
if [ ! -d ${META_EMLID_NEUTIS_SRC} ]; then
  git clone -b ${META_EMLID_NEUTIS_BRANCH} \
  --single-branch ${META_EMLID_NEUTIS_SOURCE_LINK} ${META_EMLID_NEUTIS_SRC}
  git -C ${META_EMLID_NEUTIS_SRC} checkout ${META_EMLID_NEUTIS_COMMIT_HASH} >-
fi
```

2) Загрузка и компиляция U-boot.

Листинг 4.2 – Загрузка U-boot

```
#!/bin/bash -e

U_BOOT_ARCHIVE_NAME="u-boot-${U_BOOT_VERSION}.tar.gz"
U_BOOT_SOURCE_LINK=\
"https://github.com/u-boot/u-boot/archive/v${U_BOOT_VERSION}.tar.gz"

if [ ! -f "${DOWNLOADS_DIR}/${U_BOOT_ARCHIVE_NAME}.done" ]; then
    wget ${U_BOOT_SOURCE_LINK} \
    --output-document=${DOWNLOADS_DIR}/${U_BOOT_ARCHIVE_NAME} --no-verbose
    touch "${DOWNLOADS_DIR}/${U_BOOT_ARCHIVE_NAME}.done"
fi

if [ ! -f "${U_BOOT_SRC}.done" ]; then
    tar -zxf ${DOWNLOADS_DIR}/${U_BOOT_ARCHIVE_NAME} \
        --directory ${U_BOOT_SRC} \
        --strip-components=1
    touch -f "${U_BOOT_SRC}.done"
fi
```

Листинг 4.3 – Компиляция U-boot

```
#!/bin/bash -e

export PATH="${GCC_LINARO_SRC}/bin:$PATH"

cp $ATF_SUNXI_SRC/build/sun50i_a64/debug/bl31.bin $U_BOOT_SRC
pushd $U_BOOT_SRC
apply_patches $META_EMLID_NEUTIS_SRC/meta-neutis-bsp/recipes-bsp/u-boot/u-boot
make CROSS_COMPILE=aarch64-linux-gnu- emlid_neutis_n5_defconfig
make CROSS_COMPILE=aarch64-linux-gnu- -j $CPU_CORES
popd
```

2) Создание и установка .deb пакета с initramfs, установка пакетов загрузчика в образ.

Листинг 4.4 – Создание и установка пакета с initramfs

```
#!/bin/bash -e

ROCKO_META_EMLID_NEUTIS_HASH=858be875304f728db22a7f921dfd2ee635b9a191

PACKAGE_NAME=neutis-n5-initramfs
PACKAGE_DEB_DIR=${STAGE_WORK_DIR}/packages/${PACKAGE_NAME}
META_EMLID_NEUTIS_BSP=${META_EMLID_NEUTIS_SRC}/meta-neutis-bsp

mkdir -p ${PACKAGE_DEB_DIR}/boot
mkdir -p ${PACKAGE_DEB_DIR}/DEBIAN

pushd ${STAGE_WORK_DIR}
cp \
${META_EMLID_NEUTIS_SRC}/meta-neutis-bsp/recipes-bsp/u-boot/u-boot/boot.cmd .
apply_patches ${SUB_STAGE_DIR}/files
mkimage -C none -A arm -T script -d boot.cmd boot.scr
cp boot.scr ${PACKAGE_DEB_DIR}/boot
rm PATCHED
popd

cp $META_EMLID_NEUTIS_SRC/meta-neutis-bsp/recipes-bsp/u-boot/u-boot/Env.txt\
 $PACKAGE_DEB_DIR/boot

git -C ${META_EMLID_NEUTIS_SRC} checkout ${ROCKO_META_EMLID_NEUTIS_HASH} >-
cp \
${META_EMLID_NEUTIS_BSP}/recipes-bsp/neutis-initramfs/files/uInitrd \
${PACKAGE_DEB_DIR}/boot
git -C ${META_EMLID_NEUTIS_SRC} checkout - >-

build_deb_package
install_deb_package
```

Листинг 4.5 – Установка загрузчика

```
#!/bin/bash -e

```

```
3 IMG_FILE="${STAGE_WORK_DIR}/${IMG_NAME}${IMG_SUFFIX}-${IMG_DATE}.img"
4
5 dd if=${U_BOOT_SRC}/u-boot-sunxi-with-spl.bin \
6 of=${IMG_FILE} bs=1024 seek=8 conv=notrunc
```

4.3 Запуск и апробация образа системы

С помощью разработанной системы сборки были созданы три образа. Были проверены следующие свойства полученных дистрибутивов:

- Загрузка и работоспособность минимального образа;
- Загрузка и работоспособность серверного образа;
- Загрузка и работоспособность полного образа;

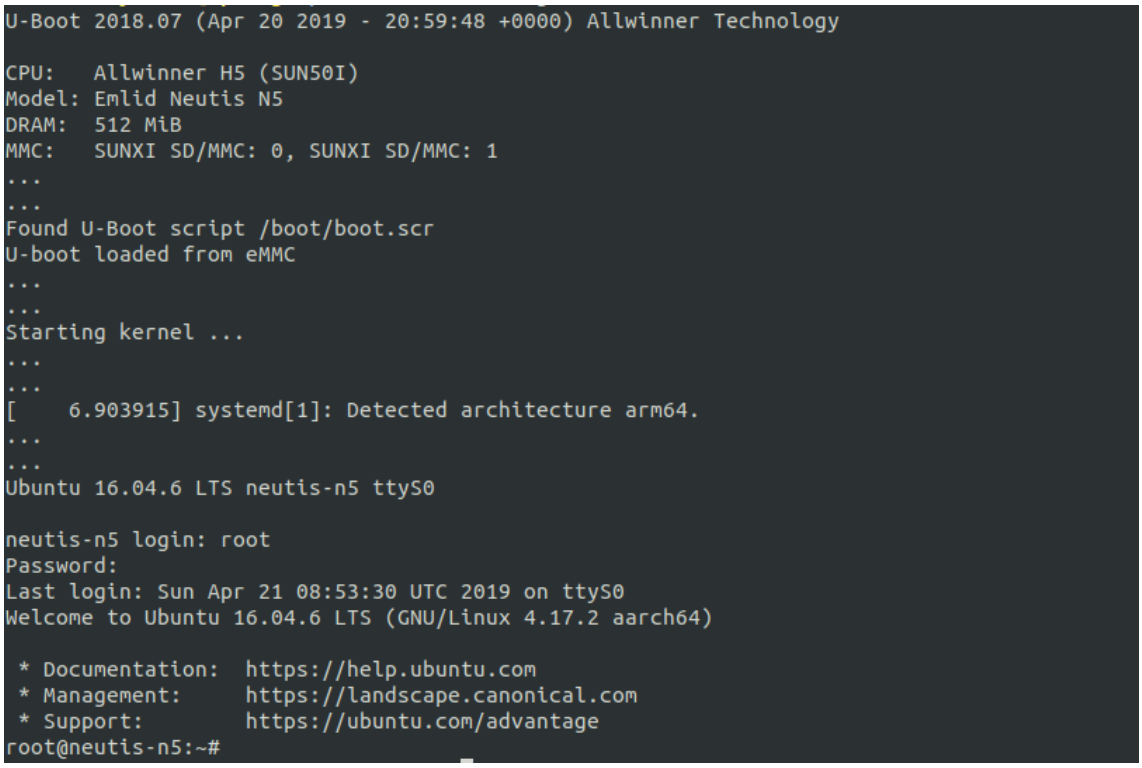

Рисунок 4.1 – Лог загрузки образа системы

Кроме того, была проверена работоспособность системы сборки на Windows 10 Pro.

ЗАКЛЮЧЕНИЕ

В рамках проведённой работы были получены следующие результаты:

– Проведен обзор и сравнение существующих систем сборки Linux дистрибутивов для встроенных сиетем.

– Создана архитектура системы, на основе которой было разработана система сборки, полностью удовлетворяющяя всем заявленным требованиям.

– Проведено функциональное тестирование системы сборки.

– Проведена апробация результатов работы. Система сборки была протестирована сотрудниками компании Emlid.

– На основе полученных в ходе апробации отчётов об ошибках и пожеланий произведены доработки системы. Разработанный продукт был успешно внедрён в качестве системы сборки debian дистрибутивов для устройств Neutis N5.

Автор планирует продолжать работу над проектом и развивать его, добавляя новые функции и исправляя возможные ошибки, которые могут быть найдены во время эксплуатации системы.

Библиографический список

1. Ключев А.О., Ковязина Д.Р., Кустарев П.В. [и др.]. Аппаратные и программные средства встраиваемых систем [Электронный ресурс]. СПб, Университет ИТМО. 2010. URL: https://books.ifmo.ru/file/pdf/686.pdf (дата обращения: 12.04.2019).

2. Andrew S. Tanenbaum Herbert Bos. Современные операционные системы. 4-е изд. СПб: Питер, 2015. 1118 с.

3. Yocto Project Mega-Manual [Электронный ресурс]. Официальный сайт проекта yocto. 2019. URL: https://www.yoctoproject.org (дата обращения: 12.04.2019).

4. The Buildroot user manual [Электронный ресурс]. Официальный сайт проекта buildroot. 2019. URL: https://buildroot.org/ (дата обращения: 12.04.2019).

5. kconfig-language [Электронный ресурс]. Официальный сайт проекта linux. 2019. URL: https://www.kernel.org (дата обращения: 12.04.2019).

6. Armbian Documentation [Электронный ресурс]. Официальный сайт проекта armbian. 2019. URL: https://docs.armbian.com (дата обращения: 12.04.2019).

7. Вигерс К. Битти Д. Разработка требований к программному обеспечению. 3-е изд., дополненное. М.: Русская редакция, 2014. 736 с.

8. Debootstrap manpage [Электронный ресурс]. Официальный сайт проекта Debian. 2017. URL: https://wiki.debian.org/Debootstrap (дата обращения: 12.04.2019).

9. Mouat A. Using Docker. 1004 Gravenstein Highway North, Sebastopol, CA 95472.: Published by O'Reilly Media, Inc., 2018. 354 p.

10. Neutis N5 Documentation [Электронный ресурс]. Официальный сайт проекта Neutis N5. 2018. URL: https://docs.neutis.io (дата обращения: 12.04.2019).

11. Simmonds C. Mastering Embedded Linux Programming. Birmingham B3 2PB, UK.: Packt Publishing, 2015. 416 p.

Приложение А

Листинг А.1 – lib/common.sh: Модуль общих библиотечных функций

```
log (){
  date +"[%T] $@" | tee -a "${LOG_FILE}"
}
export -f log

check_build_environment(){
  if [[ $(systemd-detect-virt) == "none" ]]; then
    echo "Execution of build.sh directly is deprecated.
It may break your system.
Please, run ./build inside the docker container." 1>&2
    exit 1
  fi
}
export -f check_build_environment

check_root_rights(){
  if [ "$(id -u)" != "0" ]; then
    echo "Please run as root" 1>&2
    exit 1
  fi
}
export -f check_root_rights

bootstrap(){
  local ARCH=$(dpkg --print-architecture)

  if [ "$ARCH" !=  "arm64" ]; then
    local BOOTSTRAP_CMD=qemu-debootstrap
  else
    local BOOTSTRAP_CMD=debootstrap
  fi

  capsh --drop=cap_setfcap -- -c "${BOOTSTRAP_CMD} \
    --components=main,contrib,non-free \
    --arch arm64 \
    --variant=minbase \
```

```
37      $1 $2 $3" || rmdir "$2/debootstrap"
38  }
39  export -f bootstrap
40
41  copy(){
42    source=$1
43    if [ ! -d "${source}" ]; then
44      echo "Stage rootfs ${source} not found."
45      false
46    fi
47    mkdir -p "${ROOTFS_DIR}"
48    rsync -aHAXx --exclude var/cache/apt/archives "${source}/" "${ROOTFS_DIR}/"
49  }
50  export -f copy
51
52  copy_previous(){
53    copy ${PREV_ROOTFS_DIR}
54  }
55  export -f copy_previous
56
57  copy_previous_for_export(){
58    copy ${EXPORT_ROOTFS_DIR}
59  }
60  export -f copy_previous_for_export
61
62  unmount(){
63    if [ -z "$1" ]; then
64      DIR=$PWD
65    else
66      DIR=$1
67    fi
68
69    while mount | grep -q "$DIR"; do
70      local LOCS
71      LOCS=$(mount | grep "$DIR" | cut -f 3 -d ' ' | sort -r)
72      for loc in $LOCS; do
73        umount "$loc"
74      done
75    done
76  }
```

```
77 export -f unmount
78
79 unmount_image(){
80   sync
81   sleep 1
82   local LOOP_DEVICES
83   LOOP_DEVICES=$(losetup -j "${1}" | cut -f1 -d':')
84   for LOOP_DEV in ${LOOP_DEVICES}; do
85     if [ -n "${LOOP_DEV}" ]; then
86       local MOUNTED_DIR
87       MOUNTED_DIR=$(mount | grep "$(basename "${LOOP_DEV}")" \
88       | head -n 1 | cut -f 3 -d ' ')
89       if [ -n "${MOUNTED_DIR}" ] && [ "${MOUNTED_DIR}" != "/" ]; then
90         unmount "$(dirname "${MOUNTED_DIR}")"
91       fi
92       sleep 1
93       losetup -d "${LOOP_DEV}"
94     fi
95   done
96 }
97 export -f unmount_image
98
99 on_chroot() {
100   if [[ -z "${ROOTFS_DIR}" ]]; then
101     echo "ROOTFS_DIR variable is empty, unable to mount."
102     exit 1
103   fi
104
105   if ! mount | grep -q "$(realpath "${ROOTFS_DIR}"/proc)"; then
106     mount -t proc proc "${ROOTFS_DIR}/proc"
107   fi
108
109   if ! mount | grep -q "$(realpath "${ROOTFS_DIR}"/dev)"; then
110     mount --bind /dev "${ROOTFS_DIR}/dev"
111   fi
112
113   if ! mount | grep -q "$(realpath "${ROOTFS_DIR}"/dev/pts)"; then
114     mount --bind /dev/pts "${ROOTFS_DIR}/dev/pts"
115   fi
116
```

```
117   if ! mount | grep -q "$(realpath "${ROOTFS_DIR}"/sys)"; then
118     mount --bind /sys "${ROOTFS_DIR}/sys"
119   fi
120
121   capsh --drop=cap_setfcap "--chroot=${ROOTFS_DIR}/" -- "$@"
122 }
123 export -f on_chroot
124
125 apply_patches()
126 {
127   local patches_dir=$1
128   local patchfiles=`ls $patches_dir | grep .patch`
129   if [ -e "PATCHED" ]; then
130     echo "${patches_dir} is patched already. Nothing to do."
131   else
132     for patch in $patchfiles; do
133       echo "Applying $patch patchfile."
134       patch --batch --silent -p1 -N < "${patches_dir}/${patch}"
135     done;
136     touch "PATCHED"
137   fi
138 }
139 export -f apply_patches
140
141 build_deb_package()
142 {
143   local package_name=${PACKAGE_DEB_DIR}.deb
144
145   cp files/control ${PACKAGE_DEB_DIR}/DEBIAN
146   fakeroot dpkg-deb --build ${PACKAGE_DEB_DIR}
147   cp ${package_name} ${ROOTFS_DIR}/var/cache/apt/archives
148 }
149 export -f build_deb_package
150
151 install_deb_package()
152 {
153   local package_name=${PACKAGE_NAME}.deb
154   on_chroot << EOF
155 dpkg -i /var/cache/apt/archives/${package_name}
156 EOF
```

```
157 }
158 export -f install_deb_package
```

Приложение Б

Листинг Б.1 – lib/config.sh: Модуль конфигурационных библиотечных функций

```
apply_config_file()
{
  if [ -f config ]; then
    source config
  fi
  if [ -z "${IMG_NAME}" ]; then
    IMAGE_NAME="neutis-n5-xenial"
  fi
  if [ -z "${CPU_CORES}" ]; then
    CPU_CORES=1
  fi
}

set_up_stages_skip()
{
  log "Begin set_up_stages_skip"
  find . -name "SKIP" | xargs rm -f
  if [ -n "$SKIP_STAGES" ]; then
    for STAGE_IDENTITIES in ${SKIP_STAGES}; do
      STAGE_SUFFIX=`echo ${STAGE_IDENTITIES} | awk -F: '{print $1}'`
      SUBSTAGE_PREFIX=`echo ${STAGE_IDENTITIES} | awk -F: '{print $2}'`
      stage_dir=${BASE_DIR}/stage${STAGE_SUFFIX}
      if [ -z "${SUBSTAGE_PREFIX}" -a -d ${stage_dir} ]; then
        log "Create SKIP file for directory ${stage_dir}"
        touch $stage_dir/SKIP
      fi
      if  [ -n "${SUBSTAGE_PREFIX}" ]; then
        for substage_dir in $stage_dir/${SUBSTAGE_PREFIX}*; do
          log "Create SKIP file for directory ${substage_dir}"
          touch ${substage_dir}/SKIP
        done
      fi
    done
  fi
  log "End set_up_stages_skip"
```

```
}

set_up_export_stages()
{
  log "Begin set_up_export_stages"
  find . -name "EXPORT_IMAGE" | xargs rm -f
  if [ -n "$EXPORT_STAGES" ]; then
      for STAGE_IDENTITIES in ${EXPORT_STAGES}; do
      STAGE_SUFFIX=`echo ${STAGE_IDENTITIES} | awk -F: '{print $1}'`
      IMG_SUFFIX=`echo ${STAGE_IDENTITIES} | awk -F: '{print $2}'`
      stage_dir=${BASE_DIR}/stage${STAGE_SUFFIX}
      if [ -z "${IMG_SUFFIX}" ]; then
        IMG_SUFFIX="${DEFAULT_DEVICE_NAME}-stage${STAGE_SUFFIX}"
      fi
      if [ -d ${stage_dir} ]; then
        log "Create EXPORT_IMAGE file for directory ${stage_dir}"
        printf "IMG_SUFFIX=\"${IMG_SUFFIX}\"" > ${stage_dir}/EXPORT_IMAGE
      fi
    done
  fi
  log "End set_up_export_stages"
}
```

Приложение В

Листинг С.1 – lib/stages.sh: Модуль функций контроля порядка выполнения

```
run_sub_stage()
{
  log "Begin ${SUB_STAGE_DIR}"
  pushd ${SUB_STAGE_DIR} > /dev/null
  for i in {00..99}; do
    if [ -f ${i}-debconf ]; then
      log "Begin ${SUB_STAGE_DIR}/${i}-debconf"
      on_chroot << EOF
debconf-set-selections <<SELEOF
`cat ${i}-debconf`
SELEOF
EOF
    log "End ${SUB_STAGE_DIR}/${i}-debconf"
    fi
    if [ -f ${i}-packages-nr ]; then
      log "Begin ${SUB_STAGE_DIR}/${i}-packages-nr"
      PACKAGES="$(sed -f "${LIB_DIR}/remove-comments.sed" < ${i}-packages-nr)"
      if [ -n "$PACKAGES" ]; then
        on_chroot << EOF
apt-get install --no-install-recommends -y $PACKAGES
EOF
      fi
      log "End ${SUB_STAGE_DIR}/${i}-packages-nr"
    fi
    if [ -f ${i}-packages ]; then
      log "Begin ${SUB_STAGE_DIR}/${i}-packages"
      PACKAGES="$(sed -f "${LIB_DIR}/remove-comments.sed" < ${i}-packages)"
      if [ -n "$PACKAGES" ]; then
        on_chroot << EOF
apt-get install -y $PACKAGES
EOF
      fi
      log "End ${SUB_STAGE_DIR}/${i}-packages"
    fi
    if [ -d ${i}-patches ]; then
      log "Begin ${SUB_STAGE_DIR}/${i}-patches"
```

```
37        pushd ${STAGE_WORK_DIR} > /dev/null
38        if [ "${CLEAN}" = "1" ]; then
39          rm -rf .pc
40          rm -rf *-pc
41        fi
42        QUILT_PATCHES=${SUB_STAGE_DIR}/${i}-patches
43        SUB_STAGE_QUILT_PATCH_DIR="$(basename $SUB_STAGE_DIR)-pc"
44        mkdir -p $SUB_STAGE_QUILT_PATCH_DIR
45        ln -snf $SUB_STAGE_QUILT_PATCH_DIR .pc
46        if [ -e ${SUB_STAGE_DIR}/${i}-patches/EDIT ]; then
47          echo "Dropping into bash to edit patches..."
48          bash
49        fi
50        quilt upgrade
51        RC=0
52        quilt push -a || RC=$?
53        case "$RC" in
54          0|2)
55            ;;
56          *)
57            false
58            ;;
59        esac
60        popd > /dev/null
61        log "End ${SUB_STAGE_DIR}/${i}-patches"
62      fi
63      if [ -x ${i}-run.sh ]; then
64        log "Begin ${SUB_STAGE_DIR}/${i}-run.sh"
65        ./${i}-run.sh
66        log "End ${SUB_STAGE_DIR}/${i}-run.sh"
67      fi
68      if [ -f ${i}-run-chroot.sh ]; then
69        log "Begin ${SUB_STAGE_DIR}/${i}-run-chroot.sh"
70        on_chroot < ${i}-run-chroot.sh
71        log "End ${SUB_STAGE_DIR}/${i}-run-chroot.sh"
72      fi
73    done
74    popd > /dev/null
75    log "End ${SUB_STAGE_DIR}"
76  }
```

```
77
78 run_stage(){
79   log "Begin ${STAGE_DIR}"
80   STAGE=$(basename ${STAGE_DIR})
81   pushd ${STAGE_DIR} > /dev/null
82   unmount ${WORK_DIR}/${STAGE}
83   STAGE_WORK_DIR=${WORK_DIR}/${STAGE}
84   ROOTFS_DIR=${STAGE_WORK_DIR}/rootfs
85   if [ -f ${STAGE_DIR}/EXPORT_IMAGE ]; then
86     EXPORT_DIRS="${EXPORT_DIRS} ${STAGE_DIR}"
87   fi
88   if [ ! -f SKIP ]; then
89     if [ "${CLEAN}" = "1" ]; then
90       if [ -d ${ROOTFS_DIR} ]; then
91         rm -rf ${ROOTFS_DIR}
92       fi
93     fi
94     if [ -x prerun.sh ]; then
95       log "Begin ${STAGE_DIR}/prerun.sh"
96       ./prerun.sh
97       log "End ${STAGE_DIR}/prerun.sh"
98     fi
99     for SUB_STAGE_DIR in ${STAGE_DIR}/*; do
100      if [ -d ${SUB_STAGE_DIR} ] &&
101         [ ! -f ${SUB_STAGE_DIR}/SKIP ]; then
102        run_sub_stage
103      fi
104    done
105  fi
106  unmount ${WORK_DIR}/${STAGE}
107  PREV_STAGE=${STAGE}
108  PREV_STAGE_DIR=${STAGE_DIR}
109  PREV_ROOTFS_DIR=${ROOTFS_DIR}
110  popd > /dev/null
111  log "End ${STAGE_DIR}"
112 }
```