

# КАК НА САМОМ ДЕЛЕ РАБОТАЮТ КОМПЬЮТЕРЫ

Мэтью Джастис



Мэттью Джастис

# КАК НА САМОМ ДЕЛЕ РАБОТАЮТ КОМПЬЮТЕРЫ

# HOW COMPUTERS REALLY WORK

by Matthew Justice



**no starch  
press**

San Francisco

# КАК НА САМОМ ДЕЛЕ РАБОТАЮТ КОМПЬЮТЕРЫ

Мэттью Джастис



Москва, 2022



УДК 004.382.7  
ББК 32.971.3  
Д40

**Мэттью Джастис**

**Д40** Как на самом деле работают компьютеры. Практическое руководство по внутреннему устройству машины / науч. ред. Ю. В. Ревич; пер. с англ. С. Л. Плехановой. – М.: ДМК Пресс, 2022. – 428 с.: ил.

**ISBN 978-5-97060-973-6**

В этом руководстве в доступной форме излагаются основы вычислительной техники. Рассматриваются принципы электронных вычислений и использование двоичных чисел; в общих чертах показано, как функционирует аппаратное обеспечение компьютера, для чего нужна операционная система и как передаются данные по интернету. Читатель получит базовое представление о языках программирования, изучая примеры кода на C и Python.

Каждая глава содержит упражнения и практические задания (проекты), позволяющие на практике применить полученные знания.

Книга будет полезна всем, кто хочет разобраться, как работает компьютер.

УДК 004.382.7  
ББК 32.971.3

Title of English-language original: How Computers Really Work: A Hands-On Guide to the Inner Workings of the Machine, ISBN 9781718500662, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103. The Russian-Language 1st edition Copyright © [year] by DMK Press Publishing under license by No Starch Press Inc. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-71850-066-2 (англ.)  
ISBN 978-5-97060-973-6 (рус.)

© 2021 by Matthew Justice  
© Оформление, издание, перевод,  
ДМК Пресс, 2022

Посвящается моей семье, которая верила в меня,  
когда я пробовал что-то новое

## Об авторе

Мэттью Джастис – инженер-программист. Он проработал 17 лет в компании Microsoft, где выполнял различные работы, включая отладку ядра Windows, автоматизацию исправления ошибок и руководство группой специалистов, ответственных за создание диагностических инструментов и сервисов. Он работал над низкоуровневым программным обеспечением (операционная система) и над программным обеспечением высокого уровня, удаленным от основного оборудования (например, веб-приложения). Мэттью имеет научную степень в области электротехники. Когда он не занят написанием кода или сбором схем, Мэттью любит проводить время с семьей, ходить в горы, читать, создавать музыкальные аранжировки и играть в старые видеоигры.

## О технических рецензентах

Доктор Билл Янг – доцент кафедры вычислительной техники Техасского университета в Остине. До поступления в Техасский университет в 2001 году он имел 20-летний опыт работы в данной отрасли. Он специализируется на формальных методах программирования и на компьютерной безопасности, но среди прочих курсов также часто преподает и компьютерную архитектуру.

Брайан Вильгельм – инженер-программист. Он имеет научные степени по математике и информатике и работает в Microsoft уже 20 лет, выполнив за это время самые разные работы – от отладки ядра Windows до разработки бизнес-приложений. Он любит читать, смотреть научно-фантастические фильмы и слушать классическую музыку.

Джон Хьюз начал собирать электрические цепи уже в раннем возрасте, а будучи подростком, перешел к проектам по электронике. Позже он получил научную степень по физике и продолжил развивать свой интерес к электронике, помогая школьникам с их проектами, пока работал ассистентом по науке. Джон преподавал электронику и физику вплоть до профессионального уровня в Великобритании и руководил школьным клубом электроники для детей в возрасте от 11 до 18 лет, создав сайт <http://www.electronicclub.info/> для поддержки клуба. Он считает, что каждый может получить удовольствие от создания проектов по электронике независимо от возраста и способностей.

# КРАТКОЕ СОДЕРЖАНИЕ

Благодарности.....	13
Введение.....	31
1. Принципы компьютерных вычислений.....	13
2. Двоичный код в действии .....	31
3. Электрические цепи.....	46
4. Цифровые схемы .....	68
5. Математика в цифровых схемах .....	92
6. Память и синхросигналы .....	111
7. Аппаратное обеспечение компьютера.....	143
8. Машинный код и язык ассемблера .....	168
9. Программирование высокого уровня .....	198
10. Операционные системы .....	243
11. Интернет.....	293
12. Всемирная паутина .....	326
13. Современные вычислительные технологии.....	359
Приложение А. Ответы на упражнения.....	388
Приложение В. Технические средства.....	404

# СОДЕРЖАНИЕ

Об авторе .....	6
О технических рецензентах.....	6
Благодарности.....	16
Введение.....	17

## 1

### **Принципы компьютерных вычислений .....23**

Определение компьютера.....	24
Аналоговый и цифровой .....	24
Аналоговый подход.....	24
Переход на цифровые технологии .....	26
Системы счисления .....	27
Десятичные числа .....	28
Двоичные числа.....	29
Биты и байты.....	31
Префиксы .....	32
Шестнадцатеричная система.....	35
Выводы .....	39

## 2

### **Двоичный код в действии .....40**

Представление данных в цифровом виде.....	40
Цифровой текст .....	41
ASCII .....	42
Цвета и изображения в цифровом формате.....	44
Подходы к представлению цветов и изображений .....	46
Интерпретация двоичных данных .....	48
Двоичная логика .....	49
Выводы .....	55

## 3

### **Электрические цепи.....56**

Определение электрических терминов.....	57
Электрический заряд.....	57
Электрический ток.....	58

Напряжение .....	58
Сопротивление .....	59
Аналогия с водой .....	59
Закон Ома.....	60
Схемы электрических цепей .....	61
Закон напряжения Кирхгофа .....	64
Электрические цепи в реальном мире .....	66
Светоизлучающие диоды.....	69
Выводы .....	71
<b>ПРОЕКТ № 1: Построение электрической цепи и измерения в ней.....</b>	<b>72</b>
<b>ПРОЕКТ № 2: Построение простой схемы со светодиодом.....</b>	<b>78</b>

## 4

### **Цифровые схемы .....81**

Что такое цифровая схема?.....	81
Логика с помощью механических выключателей .....	82
Удивительный транзистор .....	85
Логические вентили.....	88
Проектирование с помощью логических вентилях .....	91
Интегральные схемы .....	92
Выводы .....	95
<b>ПРОЕКТ № 3: Построение логических операторов (И, ИЛИ) с помощью транзисторов .....</b>	<b>96</b>
<b>ПРОЕКТ № 4: Построение схемы с логическими вентилями.....</b>	<b>98</b>

## 5

### **Математика в цифровых схемах ..... 104**

Двоичное сложение.....	105
Полусумматоры .....	107
Полные сумматоры.....	109
Четырехразрядный сумматор.....	111
Знаковые числа .....	112
Беззнаковые числа.....	117
Выводы .....	119
<b>ПРОЕКТ № 5: Построение полусумматора .....</b>	<b>120</b>

## 6

### **Память и синхросигналы ..... 122**

Последовательные логические схемы и память .....	122
--	-----

SR-защелка.....	123
Использование SR-защелки в схеме.....	127
Синхросигналы.....	130
JK-триггеры.....	132
Т-триггеры.....	133
Использование синхросигнала в трехбитном счетчике.....	134
Выводы.....	136
<b>ПРОЕКТ № 6: Построение SR-защелки с использованием вентилей НЕ-ИЛИ.....</b>	<b>137</b>
<b>ПРОЕКТ № 7: Построение базовой схемы торгового автомата.....</b>	<b>139</b>
<b>ПРОЕКТ № 8: Добавление отложенного сброса в схему торгового автомата.....</b>	<b>140</b>
<b>ПРОЕКТ № 9: Использование защелки в качестве ручного синхросигнала.....</b>	<b>143</b>
<b>ПРОЕКТ № 10: Тестирование JK-триггера.....</b>	<b>146</b>
<b>ПРОЕКТ № 11: Построение трехбитного счетчика.....</b>	<b>148</b>

## 7

### **Аппаратное обеспечение компьютера..... 151**

Обзор аппаратного обеспечения компьютера.....	151
Оперативная память.....	153
Центральный процессор (CPU).....	157
Архитектура набора команд.....	158
Внутреннее устройство процессора.....	161
Синхросигнал, ядра и кеш.....	162
За пределами памяти и процессора.....	166
Вторичное хранилище.....	166
Устройства ввода/вывода.....	168
Связь по шине.....	171
Выводы.....	172

## 8

### **Машинный код и язык ассемблера..... 173**

Определение программных терминов.....	173
Пример машинной инструкции.....	175
Вычисление факториала в машинном коде.....	177
Выводы.....	180
<b>ПРОЕКТ № 12: Факториал на ассемблере.....</b>	<b>181</b>
<b>ПРОЕКТ № 13: Исследование машинного кода.....</b>	<b>194</b>

## **Программирование высокого уровня ..... 199**

Обзор программирования высокого уровня.....	199
Введение в С и Python .....	201
Комментарии.....	202
Переменные .....	202
Переменные в С.....	203
Переменные в Python .....	204
Стек и куча .....	205
Стек.....	205
Куча.....	207
Математика .....	208
Логика.....	211
Побитовые операторы .....	212
Булевы операторы.....	213
Порядок выполнения программы.....	214
Операторы if .....	215
Циклы.....	216
Функции .....	217
Определение функций .....	218
Вызов функций .....	220
Использование библиотек.....	221
Объектно-ориентированное программирование .....	222
Компилируемый или интерпретируемый .....	223
Вычисление факториала в С.....	225
Выводы .....	228
<b>ПРОЕКТ № 14: Изучение переменных.....</b>	<b>229</b>
<b>ПРОЕКТ № 15: Изменение типа значения,</b>	
<b>на которое ссылается переменная в PYTHON .....</b>	<b>232</b>
<b>ПРОЕКТ № 16: Стек или куча.....</b>	<b>233</b>
<b>ПРОЕКТ № 17: Напишите игру-угадайку .....</b>	<b>236</b>
<b>ПРОЕКТ № 18: Использование класса банковского счета в PYTHON ....</b>	<b>237</b>
<b>ПРОЕКТ № 19: Факториал на С.....</b>	<b>239</b>

## **Операционные системы ..... 242**

Программирование без операционной системы .....	242
Обзор операционных систем .....	244
Семейства операционных систем.....	246
Режим ядра и режим пользователя.....	249



Процессы .....	251
Потоки.....	253
Виртуальная память.....	256
Интерфейс прикладного программирования (API) .....	259
Пользовательский режим и системные вызовы.....	262
API и системные вызовы .....	264
Программные библиотеки операционной системы.....	265
Двоичный интерфейс приложений .....	268
Драйверы устройств.....	268
Файловые системы .....	269
Службы и демоны.....	270
Безопасность .....	271
Выводы .....	272
<b>ПРОЕКТ № 20: Исследование</b>	
<b>запущенных процессов.....</b>	<b>272</b>
<b>ПРОЕКТ № 21: Создание потока выполнения и наблюдение за ним .....</b>	<b>275</b>
<b>ПРОЕКТ № 22: Исследование виртуальной памяти .....</b>	<b>277</b>
<b>ПРОЕКТ № 23: Исследование API операционной системы .....</b>	<b>280</b>
<b>ПРОЕКТ № 24: Наблюдение за системными вызовами.....</b>	<b>283</b>
<b>ПРОЕКТ № 25: Использование GLIBC.....</b>	<b>284</b>
<b>ПРОЕКТ № 26: Просмотр загруженных модулей ядра .....</b>	<b>287</b>
<b>ПРОЕКТ № 27: Исследование устройств хранения данных</b>	
<b>и файловых систем .....</b>	<b>288</b>
<b>ПРОЕКТ № 28: Просмотр служб .....</b>	<b>289</b>

## 11

<b>Интернет .....</b>	<b>290</b>
Определение сетевых терминов .....	290
Набор интернет-протоколов .....	292
Канальный уровень .....	295
Межсетевой уровень.....	297
Транспортный уровень.....	301
Прикладной уровень.....	303
Путешествие по интернету .....	304
Основополагающие возможности интернета.....	306
Протокол динамической настройки узла (DHCP).....	306
Частные IP-адреса и преобразование сетевых адресов .....	307
Система доменных имен .....	308
Сеть – это вычисления .....	312
Выводы .....	312

<b>ПРОЕКТ № 29: Изучение канального уровня.....</b>	<b>313</b>
<b>ПРОЕКТ № 30: Изучение межсетевого уровня .....</b>	<b>315</b>
<b>ПРОЕКТ № 31: Изучение использования портов .....</b>	<b>316</b>
<b>ПРОЕКТ № 32: Прослеживание маршрута до хоста в интернете .....</b>	<b>318</b>
<b>ПРОЕКТ № 33: Узнайте свой арендованный IP-адрес.....</b>	<b>319</b>
<b>ПРОЕКТ № 34: Является ли IP вашего устройства публичным или частным? .....</b>	<b>320</b>
<b>ПРОЕКТ № 35: Поиск информации в DNS.....</b>	<b>321</b>

## 12

<b>Всемирная паутина .....</b>	<b>323</b>
Обзор Всемирной паутины .....	323
Распределенная паутина .....	324
Адресуемая паутина.....	324
Связанная паутина.....	327
Веб-протоколы .....	327
Поиск в паутине .....	330
Языки Всемирной паутины .....	331
Структурирование веб с помощью HTML.....	331
Стилизация веб-страниц с помощью CSS .....	334
Создание скриптов с помощью JavaScript .....	337
Структурирование данных в веб с помощью JSON и XML .....	339
Веб-браузеры.....	342
Визуализация страницы .....	342
Строка агента пользователя (User Agent String ) .....	344
Веб-серверы .....	345
Выводы .....	348
<b>ПРОЕКТ № 36: Исследование трафика HTTP .....</b>	<b>349</b>
<b>ПРОЕКТ № 37: Запуск собственного веб-сервера .....</b>	<b>351</b>
<b>ПРОЕКТ № 38: Возврат HTML с вашего веб-сервера .....</b>	<b>353</b>
<b>ПРОЕКТ № 39: добавление CSS на ваш сайт .....</b>	<b>355</b>
<b>ПРОЕКТ № 40: Добавьте JavaScript на свой сайт.....</b>	<b>356</b>

## 13

<b>Современные вычислительные технологии .....</b>	<b>358</b>
Приложения .....	358
Нативные приложения.....	360
Веб-приложения .....	362
Виртуализация и эмуляция.....	363
Виртуализация .....	363
Эмуляция.....	365

Облачные вычисления.....	366
История удаленных вычислений.....	366
Категории облачных вычислений.....	367
Невидимый веб и темный веб .....	370
Биткойн.....	371
Основы биткойна .....	372
Биткойн-кошельки .....	372
Биткойн-транзакции.....	373
Майнинг биткойнов.....	374
Виртуальная и дополненная реальность.....	376
Интернет вещей.....	378
Выводы .....	380
<b>ПРОЕКТ № 41: Использование PYTHON для управления</b>	
<b>схемой торгового автомата .....</b>	<b>381</b>

## Приложение А

<b>Ответы на упражнения .....</b>	<b>391</b>
1-2: Двоичное в десятичное .....	391
1-3: Десятичное в двоичное .....	391
1-4: Из двоичной системы в шестнадцатеричную .....	392
1-5: Из шестнадцатеричной в двоичную .....	392
2-1: Создайте собственную систему представления текста.....	392
2-2: Кодировка и декодировка ASCII .....	393
2-3: Создание собственной системы представления градации серого .....	393
2-4: Создание собственного подхода к представлению простых	
изображений .....	394
2-5: Составление таблицы истинности для логической функции .....	396
3-1: Применение закона Ома .....	396
3-2: Определите падение напряжения .....	397
4-1: Реализация логического ИЛИ (OR) с транзисторами .....	397
4-2: Проектирование схемы с логическими вентилями.....	398
5-1: Практика двоичного сложения.....	398
5-2: Найдите дополнительный код .....	399
5-3: Сложите два двоичных числа и их интерпретируйте	
их как знаковые и беззнаковые .....	399
7-1: Вычислите необходимое количество битов .....	399
8-1: Используйте свой мозг в качестве процессора.....	400
9-1: Побитовые операторы .....	402
9-2: Выполните программу на C в уме.....	403
11-1: Какие IP находятся в одной подсети? .....	404
11-2: Исследование распространенных портов.....	404
12-1: Определение частей URL-адреса .....	405

## Приложение В

### Технические средства ..... 406

Покупка электронных компонентов для проектов .....	406
Названия микросхем серии 7400 .....	407
Покупка .....	409
Питание цифровых схем .....	410
Зарядное устройство USB .....	410
Питание для макетной платы .....	411
Питание от Raspberry Pi .....	412
Батарейки AA .....	413
Поиск и устранение неисправностей в электронных схемах .....	414
Raspberry Pi .....	416
Почему Raspberry Pi? .....	416
Необходимые детали .....	417
Настройка Raspberry Pi .....	418
Использование Raspberry Pi OS .....	419
Работа с файлами и папками .....	421

# БЛАГОДАРНОСТИ

Огромное спасибо моей жене Сьюзи, которая выступала в роли неформального редактора, предоставляя мне бесценную обратную связь. Она внимательно изучала каждое слово и каждую мысль в нескольких черновиках этой книги, помогая мне переосмыслить свои идеи и выразить их более четко. Она ободряла и поддерживала меня в этом начинании от его замысла до завершения.

Спасибо моим дочерям-подросткам, Эве и Айви, которые прочитали первые черновики и помогли увидеть мою работу глазами юных учеников. Они помогли избежать путаницы в формулировках и показали, где мне нужно уделить больше времени объяснению.

Я хочу выразить благодарность своим родителям, Расселу и Дебби Джастис, которые всегда верили в меня и предоставляли широкие возможности для обучения. Моя любовь к письменному слову – от мамы, а инженерный склад ума – от папы.

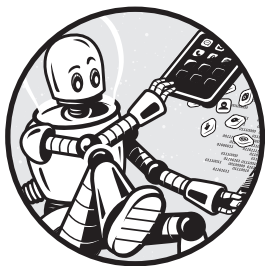
Спасибо всей команде издательства No Starch Press, особенно Алексу Фриду и Катрине Тейлор, а также моему редактору Ребекке Райдер. Это был мой первый опыт написания книги, и редакторы No Starch терпеливо вели меня через весь процесс. Они видели возможности для улучшения, которых я не замечал, и помогли мне ясно изложить свои идеи. Я по-новому оценил значимость издательской команды.

Я благодарен техническим рецензентам этой книги, Джону Хьюзу, Брайану Вильгельму и Биллу Янгу, которые внимательно изучили все детали моей работы.

Их вклад привел к созданию более точного и полного текста. Каждый из них привнес свою уникальную точку зрения и поделился своим ценным экспертным опытом.

Спасибо всем сотрудникам Microsoft, которые были моими наставниками и сотрудничали со мной на протяжении многих лет. Мне посчастливилось работать с невероятно талантливыми, умными и знающими людьми – вас слишком много, чтобы всех перечислить! Я смог написать эту книгу только потому, что некоторые замечательные люди в Microsoft нашли время поделиться своими знаниями.

# ВВЕДЕНИЕ



Вам интересно, как работают компьютеры? Обретение глубокого понимания вычислительной техники часто достигается долгим и извилистым путем. Проблема не в отсутствии информации. Быстрый поиск в интернете покажет, что существует очень много книг и веб-сайтов, посвященных объяснению работы вычислительной техники.

Программирование, информатика, электроника, операционные системы... так много информации. Это хорошо, но может напугать. С чего начать? Как одна тема связана с другой? Эта книга была написана, чтобы дать вам отправную точку для изучения ключевых концепций, касающихся вычислительной техники, и понимания того, как эти концепции связаны друг с другом.

Работая руководителем инженерного отдела, я регулярно проводил собеседования с людьми, претендующими на должности разработчиков программного обеспечения. Я разговаривал со многими кандидатами, которые умели писать код, но довольно многие из них, похоже, не знали, как на самом деле работают компьютеры. Они знали, как заставить компьютер выполнять их команды, но не понимали, что происходит за экраном. Размышления об этих интервью и воспоминания о моих собственных трудностях в попытках разобраться в компьютерах привели меня к написанию этой книги.

Моя цель – изложить основы вычислительной техники в доступной, практической форме, которая делает абстрактные понятия более понятными. В этой книге нет глубокого погружения в каждую из поднятых тем, но вместо этого в ней представлены основополагающие концепции вычислительной техники, а также связь между этими концепциями. Я хочу, чтобы вы смогли составить представление о том, как работают вычислительные машины, что позволит затем углубиться в интересующие вас темы.

Компьютеры уже повсюду, и поскольку наше общество все больше и больше зависит от технологий, нам нужны люди с широким пониманием вычислительной техники. Я надеюсь, что эта книга поможет вам обрести такое широкое понимание.

## Для кого эта книга?

Эта книга предназначена для всех, кто хочет разобраться, как работают компьютеры. Вам не нужно обладать какими-либо предварительными знаниями по рассматриваемым темам, так как мы начнем с основ. С другой стороны, если у вас уже есть знания в области программирования или электроники, эта книга поможет расширить понимание в других областях. Книга написана для тех, кто склонен к самообразованию, кто знает основы математики и естественных наук, пользуется компьютерами и смартфонами, но у кого все еще есть вопросы о том, как эти устройства работают. Учителя также найдут содержание книги полезным, так как, по моему мнению, проекты являются хорошим подспорьем в школьном образовании.

## Об этой книге

Эта книга рассматривает компьютеры как стек<sup>1</sup> (иерархию) технологий. Современное вычислительное устройство, например смартфон, состоит из нескольких технологических слоев. В нижней части этого множества слоев находится аппаратное обеспечение, в верхней части лежат приложения, а между ними имеется еще много уровней. Преимущество такой многослойной модели заключается в том, что каждый слой использует возможности всех нижних уровней, но каждый конкретный слой должен опираться только на тот слой, который находится непосредственно под ним. После изучения некоторых базовых концепций мы пройдем по технологическим уровням снизу вверх, начиная с электрических цепей и заканчивая технологиями, которые обеспечивают работу веб-сайтов и приложений. Вот что мы рассмотрим в каждой главе.

**Глава 1: Принципы компьютерных вычислений.** Охватывает основополагающие идеи, такие как понимание аналоговой и цифровой систем, двоичной системы счисления и кратных приставок в системе СИ<sup>2</sup>.

**Глава 2: Двоичная система в действии.** Рассмотрим, как двоичная система может использоваться для представления данных и логических состояний, а также познакомимся с логическими операторами.

**Глава 3: Электрические цепи.** Объясняет основные понятия, касающиеся электричества и электрических цепей, включая напряжение, ток и сопротивление.

**Глава 4: Цифровые схемы.** Знакомит с транзисторами и логическими вентилями, а также объединяет понятия из глав 2 и 3.

<sup>1</sup> Термин «stack» (стек) означает ряд вложенных друг в друга структур различного уровня, выполняющих каждая свою функцию (подробнее см. далее по тексту). – *Прим. ред.*

<sup>2</sup> СИ в данном случае не название языка программирования C, а общепринятое сокращение для самой распространенной международной системы единиц (SI, от фр. *Système International d'unités*). – *Прим. ред.*

**Глава 5: Математика в цифровых схемах.** Показывает, как можно выполнять сложение с помощью цифровых схем, и рассказывает о том, как числа представляются в компьютерах.

**Глава 6: Память и синхросигналы.** Представляет устройства памяти и последовательные схемы, а также демонстрирует взаимодействие посредством синхросигналов.

**Глава 7: Аппаратное обеспечение компьютера.** Рассматривает основные составные части компьютера: процессор, память, устройства ввода/вывода.

**Глава 8: Машинный код и язык ассемблера.** Представляет низкоуровневый машинный код, выполняемый процессорами, а также язык ассемблера – представление машинного кода в удобочитаемом виде.

**Глава 9: Программирование высокого уровня.** Знакомит с языками программирования, которые не зависят от конкретных процессоров, и включает примеры кода на языках C и Python.

**Глава 10: Операционные системы.** Рассматривает семейства операционных систем и основные возможности операционных систем.

**Глава 11: Интернет.** Показывает, как работает интернет, включая общий набор используемых сетевых протоколов.

**Глава 12: Всемирная паутина.** Объясняет, как функционирует Всемирная паутина, и рассматривает ее основные технологии: HTTP, HTML, CSS и JavaScript.

**Глава 13: Современные вычислительные технологии.** Содержит обзор нескольких направлений современных вычислительных технологий, таких как приложения, виртуализация и облачные сервисы.

По мере чтения этой книги вам будут встречаться схемы и исходные коды, используемые для иллюстрации концепций. Они предназначены для наглядности обучения, когда отдается предпочтение понятности, а не производительности, безопасности и другим факторам, которые учитывают инженеры-программисты при разработке аппаратного или программного обеспечения. Другими словами, схемы и коды в этой книге могут помочь вам узнать, как работают компьютеры, но они не обязательно являются примерами наилучшего способа выполнения задачи. Аналогичным образом в технических разделах книги предпочтение отдается простоте, а не полноте. Так я иногда опускаю некоторые детали, чтобы не завязнуть в сложных объяснениях.

## Об упражнениях и проектах

Во всех главах вы найдете упражнения и практические задания. Упражнения – это задачи, которые вы можете решить мысленно или с помощью карандаша и бумаги. Задания же выходят за рамки умственных упражнений и часто предполагают создание схем или программирование.



Для выполнения практических заданий (проектов) вам потребуется приобрести некоторое оборудование (список необходимых компонентов вы найдете в приложении В). Я включил такие проекты, потому что считаю, что лучший способ научиться – это попробовать самому, и я призываю вас выполнять их, если вы хотите получить максимальную пользу от этой книги.

Тем не менее я изложил материалы глав таким образом, чтобы вы могли следовать за повествованием, даже если не соберете ни одной схемы или не введете ни одной строки кода.

Ответы на упражнения вы найдете в приложении А, а подробное описание каждого проекта в конце соответствующей главы. Приложение В содержит информацию, которая поможет вам начать работу с проектами, а в тексте вы найдете отсылки к этой информации, когда она будет нужна.

Копия исходного кода, используемого в проектах, доступна по адресу <https://www.howcomputersreallywork.com/code/>. Вы также можете посетить страницу этой книги, <https://nostarch.com/how-computers-really-work/>, где мы будем предоставлять обновления.

## Мое путешествие в мир компьютеров

Мое увлечение компьютерами, вероятно, началось с видеоигр, в которые я играл в детстве. Когда я гостил у бабушки с дедушкой, то часами играл в Frogger, Pac-Man и Donkey Kong на Atari 2600 моей тети. Позже, когда я учился в пятом классе, родители подарили мне на Рождество игровую приставку Nintendo Entertainment System, и я был в восторге! Где-то в процессе увлеченной игры в Super Mario Bros. и Double Dragon я начал интересоваться, как работают видеоигры и компьютеры. К сожалению, игровая приставка Nintendo не очень-то помогала мне понять, что происходит внутри нее.

Примерно в то же время моя семья купила наш первый «настоящий» компьютер, Apple IIgs, открыв для меня новые возможности для изучения того, как именно работают эти машины. К счастью, в моей средней школе был организован класс по программированию на языке BASIC для компьютеров Apple II, и вскоре я понял, что не могу перестать программировать! Я писал код в школе, приносил домой копию своей работы на дискете и продолжал работать дома. На протяжении средней и старшей школы я узнавал все больше о программировании, и мне это очень нравилось. Я также начал понимать, что, хотя BASIC и другие подобные языки программирования позволяют относительно легко указывать компьютеру, что делать, они также во многом скрывают то, как работают компьютеры. Мне захотелось узнать больше.

В колледже я изучал электротехнику и начал разбираться в электронике и цифровых схемах. Я посещал занятия по программированию на языке C и языке ассемблера и, наконец, получил некоторое представление о том, как компьютеры выполняют инструкции. Эlemen-

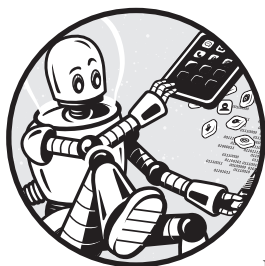
ты работы компьютеров на низком уровне начали обретать смысл. Во время учебы в колледже я также начал изучать новую штуку под названием Всемирная паутина – World Wide Web; я даже создал свою собственную веб-страницу (в то время это казалось большим событием)! Я начал программировать приложения для Windows, познакомился с Unix и Linux. Эти темы иногда казались весьма далекими от аппаратных цифровых схем и языка ассемблера, и мне было интересно понять, как все это сочетается друг с другом.

После колледжа мне посчастливилось получить работу в Microsoft. За 17 лет работы там я занимался различными вопросами разработки программного обеспечения, от отладки ядра Windows до разработки веб-приложений. Этот опыт помог мне получить более широкое и глубокое понимание компьютеров. Я работал со многими невероятно умными и знающими людьми и понял, что о компьютерах всегда можно узнать больше. Понимание того, как работают компьютеры, стало для меня путешествием длиною в жизнь, и я надеюсь передать вам часть того, чему я научился, посредством этой книги.



# 1

## ПРИНЦИПЫ КОМПЬЮТЕРНЫХ ВЫЧИСЛЕНИЙ



В наше время компьютеры повсюду: в домах, школах, офисах – вы можете найти компьютер в кармане, на запястье или даже в холодильнике. Найти и использовать компьютеры стало проще, чем когда-либо, но мало кто сегодня действительно понимает, как компьютеры работают. И это неудивительно, поскольку изучение сложных компьютерных технологий может оказаться непосильной задачей. Цель этой книги – изложить основные принципы работы вычислительной техники таким образом, чтобы любой любознательный человек с небольшими техническими способностями смог их понять. Прежде чем мы начнем погружаться в тонкости работы компьютеров, давайте уделим время знакомству с некоторыми основными принципами проведения вычислений в компьютерах.

В этой главе мы начнем с обсуждения определения компьютера. Далее рассмотрим различия между аналоговыми и цифровыми данными, а затем изучим системы счисления и терминологию, используемую для описания цифровых данных.

## Определение компьютера

Начнем с базового вопроса: что такое компьютер? Когда люди слышат слово «компьютер», большинство представляет себе ноутбук или настольный компьютер, иногда называемый персональным компьютером, или ПК. Это один из классов устройств, которые рассматриваются в данной книге, но давайте мыслить немного шире. Рассмотрим смартфоны. Смартфоны, безусловно, являются компьютерами; они выполняют те же операции, что и ПК. На самом деле для многих людей сегодня смартфон является основным вычислительным устройством. Большинство пользователей компьютеров сегодня также использует интернет, работа которого обеспечивается серверами, являющимися еще одним типом компьютеров. Каждый раз, когда вы посещаете веб-сайт или используете приложение, работающее через интернет, вы взаимодействуете с одним или несколькими серверами, подключенными к глобальной сети. Приставки для видеоигр, фитнес-трекеры, умные часы, умные телевизоры... все это – компьютеры!

Компьютер – это любое электронное устройство, которое может быть запрограммировано на выполнение набора логических инструкций. Учитывая это определение, становится понятно, что многие современные устройства на самом деле являются компьютерами!

### УПРАЖНЕНИЕ 1-1: Найдите компьютеры в своем доме

Потратьте немного времени и посмотрите, сколько компьютеров вы сможете найти в своем доме. Когда я выполнял это упражнение со своей семьей, мы быстро нашли около 30 устройств!

## Аналоговый и цифровой

Вы наверняка слышали, что компьютер называют цифровым устройством, в отличие от аналоговых устройств, таких как механические часы. Но что на самом деле означают эти два термина? Понимание различий между аналоговыми и цифровыми устройствами является основополагающим для понимания вычислительной техники, поэтому давайте рассмотрим эти два понятия подробнее.

### Аналоговый подход

Оглянитесь вокруг себя. Выберите какой-нибудь предмет. Спросите себя: какого он цвета? Какого он размера? Сколько он весит? Отвечая на эти вопросы, вы описываете атрибуты или *данные* этого объекта. Теперь возьмите другой объект и ответьте на те же вопросы. Если вы

повторите этот процесс для большего количества объектов, то обнаружите, что на каждый вопрос существует множество вариантов ответов. Вы можете взять красный, желтый или синий объект. Или цвет объекта может состоять из смеси основных цветов.

Такое разнообразие вариантов относится не только к цвету. Существует потенциально бесконечное количество вариантов определенного свойства среди всех объектов нашего мира.

Одно дело описать объект словами, но, допустим, вы хотите измерить один из его атрибутов более точно. Например, если вы хотите измерить вес объекта, можете положить его на весы. Обычные (не электронные!) весы, реагируя на поставленный на них вес, будут перемещать стрелку вдоль пронумерованной линии, останавливаясь, когда стрелка достигнет положения, соответствующего весу. Возьмите показанное на весах число, и вы получите вес объекта.

Такой способ измерения общераспространен, но давайте подумаем немного о том, как мы измеряем эти данные. Положение стрелки на весах на самом деле не является весом, это представление веса. Пронумерованная линия, на которую указывает игла, позволяет нам легко преобразовать положение иглы, представляющее вес, в числовое значение этого веса. Другими словами, хотя вес является атрибутом объекта, здесь мы можем определить этот атрибут через нечто другое: через положение иглы вдоль линии. Положение иглы изменяется пропорционально в зависимости от веса, помещенного на весы. Таким образом, весы являются *аналогией*, позволяющей нам определить вес объекта через положение иглы на линии. Вот почему мы называем этот метод измерения *аналоговым* подходом.

Другим примером аналогового измерительного прибора является жидкостный (например, ртутный или спиртовой) термометр. Объем жидкости увеличивается с ростом температуры. Производители термометров используют это свойство, помещая подкрашенную жидкость в стеклянную трубку с отметками, которые соответствуют ожидаемому объему жидкости при различных температурах. Таким образом, положение поверхности жидкости в трубке служит отображением температуры. Обратите внимание, что в обоих примерах (весы и термометр), когда производим измерение, мы можем использовать отметки на приборе для преобразования положения в конкретное числовое значение. Но значение, которое мы считываем с прибора, является лишь приблизительным. Истинное положение иглы или высота столбика жидкости могут быть любыми в пределах разрешающей способности прибора, и мы округляем значение в большую или меньшую сторону до ближайшего отмеченного значения. Поэтому, хотя может показаться, что подобные инструменты могут выполнять измерения из ограниченного набора значений (например, ртутный медицинский термометр – от примерно 35 до 42° через каждую десятую, всего около 70 значений), это ограничение связано с преобразованием в число, а не с самим аналоговым принципом.

На протяжении почти всей истории человечества люди измеряли разные вещи, используя аналоговый подход. Но люди используют аналоговый подход не только для измерений. Они также придумали хитроумные способы хранения данных в аналоговом виде. Граммофон-

ная пластинка использует извилистую канавку в качестве аналогового представления записанного звука. Форма канавки изменяется на всем ее протяжении в соответствии с изменениями формы звуковой волны во времени. Канавка – это не сам звук, а аналог формы волны оригинального звука. Пленочные камеры делают нечто подобное, кратко-временно подвергая пленку воздействию света через объектив камеры, что приводит к химическим изменениям на пленке. Химические свойства пленки – это не само изображение, а представление запечатленного изображения, аналогия изображения.

## **Переход на цифровые технологии**

Какое отношение все это имеет к вычислительной технике? Оказывается, компьютерам трудно оперировать всеми этими аналоговыми представлениями. Типы используемых аналоговых систем настолько разнообразны и изменчивы, что создать вычислительное устройство, способное понимать все из них, практически невозможно. Например, создание машины, способной измерить объем ртути, сильно отличается от создания машины, способной прочесть канавки на виниловом диске. Кроме того, компьютеры требуют высоконадежного и точного представления определенных типов данных, например числовых наборов данных и программного обеспечения. Аналоговые представления данных трудно поддаются точному измерению, имеют тенденцию разрушаться со временем и теряют точность при копировании. Компьютерам необходим способ представления всех типов данных в формате, который можно тщательно обрабатывать, хранить и копировать.

Что же мы можем сделать, если не хотим представлять данные в виде аналоговых значений, которые в принципе могут уточняться до бесконечности? Вместо этого мы можем использовать цифровой подход. *Цифровая* система представляет данные в виде последовательности символов, где каждый символ принадлежит ограниченному набору значений. Такое описание может показаться несколько формальным и не очень понятным, поэтому вместо того, чтобы углубляться в теорию цифровых систем, я объясню, что это означает на практике. Почти во всех современных компьютерах данные представлены комбинациями из двух символов: 0 и 1. Вот и все. Хотя цифровая система может использовать более двух символов, добавление большего их количества приведет к увеличению сложности и стоимости системы. Набор всего из двух символов позволяет упростить техническое обеспечение и повысить надежность. Все данные в большинстве современных вычислительных устройств представлены в виде последовательности из нулей и единиц. С этого момента в данной книге, когда я буду говорить о цифровых компьютерах, вы можете считать, что я говорю о системах, которые имеют дело только с нулями и единицами, а не с каким-то другим набором символов. Легко и просто!

Этот момент стоит повторить: все на вашем компьютере хранится в виде нулей и единиц. Последняя фотография, которую вы сделали на смартфон? Ваше устройство хранит ее как последовательность из нулей

и единиц. Песня, которую вы загрузили из интернета? Нули и единицы. Документ, который напечатали на компьютере? Нули и единицы. Приложение, которое вы установили? Это было множество из нулей и единиц. Веб-сайт, который посетили? Нули и единицы.

Может показаться сомнительным, что мы можем использовать только 0 и 1 для представления бесконечного числа значений, встречающихся в природе. Как можно свести музыкальную запись или детальную фотографию к нулям и единицам? Многим кажется парадоксальным тот факт, что столь ограниченный «словарный запас» может использоваться для выражения сложных идей. Ключевым моментом здесь является то, что цифровые системы используют *последовательность* из нулей и единиц. Цифровая фотография, например, обычно состоит из миллионов нулей и единиц.

Что же конкретно представляют собой эти нули и единицы? Вы можете встретить другие термины, используемые для описания этих нулей и единиц: ложь и истина, выключенное и включенное состояние, низкий и высокий уровень и т. д. Это происходит потому, что компьютер не хранит буквально числа 0 или 1. Он хранит последовательность состояний, где каждое состояние в последовательности может иметь только два возможных значения. Каждое состояние подобно выключателю света, который либо включен, либо выключен. На практике эти последовательности нулей и единиц хранятся по-разному. На CD-или DVD-дисках нули и единицы хранятся в виде выпуклостей (0) или ровных участков (1). На флеш-накопителях нули и единицы хранятся в виде электрических зарядов. На жестком диске нули и единицы хранятся с помощью ориентированных в разных направлениях магнитных зон (доменов). Как вы увидите в главе 4, цифровые схемы для представления нулей и единиц используют уровни электрического напряжения.

Прежде чем мы продолжим, последнее замечание по поводу термина «аналоговый» – он часто используется просто для обозначения «нецифровой». Например, инженеры могут говорить об аналоговом сигнале, имея в виду сигнал, который непрерывно изменяется и не соотносится с цифровыми значениями. Другими словами, это нецифровой сигнал, но он не обязательно представляет собой аналогию чего-либо другого. Поэтому, когда вы видите термин «аналоговый», имейте в виду, что он не всегда означает то, что вы думаете.

## Системы счисления

Итак, мы выяснили, что компьютеры – это цифровые машины, которые работают с нулями и единицами. Для многих людей эта концепция кажется странной; они привыкли, что для представления чисел в их распоряжении имеются символы от 0 до 9. Если мы ограничиваемся только двумя символами, а не десятью, как нам представлять большие числа? Чтобы ответить на этот вопрос, давайте вернемся в прошлое и рассмотрим тему из математики начальной школы о системах счисления.



## Десятичные числа

Обычно мы записываем числа, используя так называемую *десятичную позиционную систему счисления*. Давайте разберемся в этом. *Позиционная система счисления* означает, что каждая позиция в написанном числе представляет собой отдельный порядок величины; *десятичная*, или *по основанию 10*, означает, что порядки величины являются множителями 10, и на каждой позиции может быть один из десяти различных символов, от 0 до 9. Посмотрите на пример позиционной системы счисления на рис. 1-1.

2	7	5
Позиция сотен	Позиция десятков	Позиция единиц

Рис. 1-1. Число двести семьдесят пять, представленное в десятичной позиционной системе счисления

На рис. 1-1 число двести семьдесят пять записано в десятичной системе счисления как 275. Цифра 5 стоит на месте единиц, т. е. ее значение равно  $5 \times 1 = 5$ . 7 стоит на месте десятков, т. е. ее значение равно  $7 \times 10 = 70$ . 2 стоит на месте сотен, т. е. ее значение равно  $2 \times 100 = 200$ . Общее значение – это сумма всех позиций:  $5 + 70 + 200 = 275$ .

Легко, правда? Вы, наверное, поняли это еще в первом классе. Но давайте рассмотрим это немного подробнее. Почему крайнее правое место – это место единицы? И почему следующее место – место десятков и т. д.? Потому что мы работаем в десятичной системе, или с основанием 10, и поэтому каждая позиция – это степень десяти; другими словами, 10 умножается на себя определенное количество раз. Как видно на рис. 1-2, самая правая позиция – это  $10$ , возведенное в степень 0, что равно 1, так как любое число, возведенное в степень 0, равно 1. Следующая позиция – это  $10$ , возведенное в степень 1, что равно 10, а еще следующее место – это  $10$ , возведенное в степень 2 ( $10 \times 10$ ), что равно 100.

2	7	5
$10^2$	$10^1$	$10^0$
$10 \times 10$	10	1

Рис. 1-2. В десятичной позиционной системе счисления каждая позиция – это степень десяти

Если бы нам нужно было представить число больше 999 в десятичной системе, мы бы добавили еще одну позицию слева, позицию тысяч, и вес этой позиции был бы равен  $10$  в степени 3 ( $10 \times 10 \times 10$ ), т. е. 1000. По такой схеме мы можем представить любое большое целое число, добавляя по мере необходимости новые позиции.

Мы выяснили, почему различные позиции имеют определенный вес, но давайте продолжим копать дальше. Почему на каждой позиции используются символы от 0 до 9? При работе с десятичными системами у нас может быть только десять символов, потому что по определению каждая позиция может представлять только десять различных значений. В настоящее время используются символы от 0 до 9, но на самом деле можно использовать любой набор из десяти уникальных символов, каждый из которых соответствует определенному числовому значению.

Большинство людей предпочитает десятичную систему счисления, или систему с основанием 10. Некоторые считают, что это потому, что у нас десять пальцев на руках и десять на ногах, но какова бы ни была причина, в современном мире большинство людей читает, пишет и думает о числах в десятичной системе счисления. Конечно, это просто правило, которое мы коллективно выбрали для представления чисел. Как мы уже говорили ранее, это правило не относится к компьютерам, которые используют только два символа. Давайте посмотрим, как мы можем применять принципы позиционной системы счисления, ограничиваясь только двумя символами.

## Двоичные числа

Система счисления, состоящая только из двух символов, – это система с основанием 2, или двоичная система. Двоичная система по-прежнему является позиционной системой счисления, поэтому фундаментальная техника такая же, как и у десятичной системы, но есть несколько изменений. Во-первых, каждая позиция представляет собой степень 2, а не степень 10. Во-вторых, на каждой позиции может быть только один из двух символов, а не один из десяти. Эти два символа – 0 и 1. На рис. 1-3 приведен пример представления числа в двоичной системе счисления.

1	0	1
Позиция четверок	Позиция двоек	Позиция единиц
$2^2$	$2^1$	$2^0$
$2 \times 2 = 4$	2	1

Рис. 1-3. Число пять в десятичной системе, представленное в двоичной позиционной системе счисления

На рис. 1-3 представлено двоичное число: 101. Для вас это может выглядеть как сто один, но при работе с двоичными числами это на самом деле представление пяти! Если вы хотите назвать это число словами, то «один-ноль-один (двоичное)» будет хорошим способом передать написанное.

Как и в десятичной системе счисления, каждая позиция имеет вес, равный основанию, возведенному в различные степени. Так как у нас

основание 2, самая правая позиция – это 2 в степени 0, т. е. 1. Следующая позиция – это 2 в степени 1, т. е. 2, и еще следующая позиция – это 2, возведенное в степень 2 ( $2 \times 2$ ), т. е. 4. Так же как и в десятичной системе, для получения общего значения мы умножаем символ в каждой позиции на вес позиции и суммируем результаты. Итак, начиная справа, мы имеем  $(1 \times 1) + (0 \times 2) + (1 \times 4) = 5$ .

Теперь вы можете попробовать самостоятельно перевести двоичную систему счисления в десятичную.

### УПРАЖНЕНИЕ 1-2: Двоичное в десятичное

Переведите представленные в двоичной системе числа в их десятичные эквиваленты:

10 (двоичное) = \_\_\_\_\_ (десятичное);

111 (двоичное) = \_\_\_\_\_ (десятичное);

1010 (двоичное) = \_\_\_\_\_ (десятичное).

Вы можете проверить ответы в приложении А. У вас все получилось верно?

Последний пример может показаться немного сложным, поскольку он вводит еще одну позицию слева – позицию восьмерки. Теперь попробуйте пересчитать в обратном направлении, из десятичной системы счисления в двоичную.

### УПРАЖНЕНИЕ 1-3: Десятичное в двоичное

Переведите представленные в десятичной системе числа в их двоичные эквиваленты:

3 (десятичное) = \_\_\_\_\_ (двоичное);

8 (десятичное) = \_\_\_\_\_ (двоичное);

14 (десятичное) = \_\_\_\_\_ (двоичное).

Я надеюсь, что это вы тоже сделали без ошибок! Сразу можно увидеть, что одновременная работа с десятичной и двоичной системами счисления может сбить с толку, поскольку такое число, как 10, обозначает десять в десятичной системе счисления или два в двоичной. С этого момента в книге, там, где есть вероятность путаницы, двоичные числа будут записываться с префиксом 0b. Я выбрал префикс 0b, потому что несколько языков программирования использует этот подход. Ведущий символ 0 (ноль) обозначает числовое значение, а b – сокращение от binary (двоичный). Например, 0b10 означает два в двоичном исчислении, тогда как 10 без префикса означает десять в десятичном.

## Биты и байты

Одна позиция или символ в десятичном числе называется цифрой. Десятичное число 1247 – это четырехзначное число. Аналогично одна позиция или символ в двоичном числе называется битом (двоичной цифрой). Каждый бит может быть либо 0, либо 1. Такое двоичное число, как 0b110, является трехрядным.

Один бит не может передать много информации; он выключен либо включен, 0 или 1. Для представления чего-либо более сложного нам нужна последовательность битов. Чтобы облегчить работу с этими последовательностями битов, компьютеры объединяют биты в наборы по восемь штук, называемые байтами. Вот несколько примеров битов и байтов (без префикса 0b, поскольку все они двоичные).

**1** – это бит.

**0** – это тоже бит.

**11001110** – это один байт, или 8 бит.

**00111000** – это тоже байт!

**10100101** – еще один байт.

**0011100010100101** – это два байта, или 16 бит.

### ПРИМЕЧАНИЕ

*Забавный факт: четырехбитное число, половина байта, иногда называется ниббл<sup>1</sup>.*

Так сколько же данных мы можем хранить в байте? Другой способ посмотреть на этот вопрос – это найти, сколько уникальных комбинаций из нулей и единиц мы можем сделать с помощью наших 8 бит? Прежде чем мы ответим на этот вопрос, позвольте мне проиллюстрировать его на примере только 4 бит, так как это будет легче представить.

В табл. 1.1 я перечислил все возможные комбинации из нулей и единиц в четырехбитном числе. Я также включил соответствующее десятичное представление этих чисел.

Как видно из табл. 1.1, мы можем представить 16 уникальных комбинаций из нулей и единиц в четырехбитном числе, расположенных по порядку от 0 до 15 в десятичном исчислении. Представление списка комбинаций битов хорошо это иллюстрирует, но мы могли бы выяснить это также несколькими способами без перечисления всех возможных комбинаций.

<sup>1</sup> Nibble – огрызок, кусочек. В русскоязычной информатике для половины байта приняты термины «тетрада» или «полубайт». Англоязычное «ниббл» в русскоязычной практике употребляется очень редко, потому далее в этой книге мы будем заменять его более привычными и понятными русскими терминами. – *Прим. ред.*

**Таблица 1.1.** Все возможные значения четырехбитного числа

Двоичное	Десятичное
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10

Мы можем определить наибольшее возможное число, которое могут представлять 4 бита, установив все биты равными единице, что даст нам 0b1111. Это 15 в десятичной системе счисления; если мы прибавим 1 для учета представления 0, то получим наши 16 комбинаций. Другой короткий путь – увеличить 2 в количество раз, равное количеству битов, в данном случае 4, что дает нам  $2^4 = 2 \times 2 \times 2 \times 2 = 16$  комбинаций из нулей и единиц.

Рассмотрение 4 бит – это хорошее начало, но ранее мы говорили о байтах, которые содержат по 8 бит. Используя предыдущий подход, мы могли бы перечислить все комбинации из нулей и единиц, но давайте пропустим этот шаг и сразу перейдем к короткому пути. Возведем 2 в степень 8 и получим 256 – это и есть количество уникальных комбинаций битов в байте.

Теперь мы знаем, что четырехбитное число допускает 16 комбинаций из нулей и единиц, а байт допускает 256 комбинаций. Какое отношение это имеет к вычислительным технологиям? Допустим, в компьютерной игре 12 уровней; игра может легко хранить номер текущего уровня всего в 4 битах. С другой стороны, если в игре 99 уровней, 4 бит будет недостаточно... можно представить только 16 уровней! Байт, с другой стороны, легко справится с этим требованием 99 уровней. Компьютерным инженерам нередко приходится учитывать, сколько бит или байт потребуется для хранения данных.

## Префиксы

Для представления сложных типов данных требуется большое количество битов. Для такого простого числа, как 99, требуется не более байта; видео в цифровом формате, с другой стороны, может потребовать миллиарды бит. Чтобы проще передать размеры данных, мы используем такие префиксы, как гига- и мега-. Международная система единиц (СИ), также известная как метрическая система, определяет

набор стандартных префиксов. Эти префиксы используются для описания всего, что поддается количественной оценке, а не только битов. Мы еще встретимся с ними в последующих главах, посвященных электрическим цепям. В табл. 1.2 перечислены некоторые из распространенных префиксов СИ и их значения.

**Таблица 1.2.** Распространенные префиксы СИ

Название префикса		Обозначение префикса		Значение	Основание 10	Наименование
Англ.	Русск.	Англ.	Русск.			
tera	тера	T	Т	1 000 000 000 000	$10^{12}$	триллион
giga	гига	G	Г	1 000 000 000	$10^9$	миллиард
mega	мега	M	М	1 000 000	$10^6$	миллион
kilo	кило	k	к (K)	1000	$10^3$	тысяча
centi	санти	c	с	0,01	$10^{-2}$	сотый
milli	милли	m	м	0,001	$10^{-3}$	тысячный
micro	микро	$\mu$	мк	0,000001	$10^{-6}$	миллионный
nano	нано	n	н	0,000000001	$10^{-9}$	миллиардный
pico	пико	p	п	0,000000000001	$10^{-12}$	триллионный

С помощью этих префиксов, если мы хотим сказать «три миллиарда байт», мы можем использовать сокращение 3 ГБ. Или, если хотим обозначить четыре тысячи бит, можем сказать 4 Кбит. Обратите внимание, что байты обозначаются как «Б» (в англоязычном написании «b», иногда «B»), а биты как «бит» («bit»)<sup>1</sup>.

Вы увидите, что это правило часто используется для обозначения количества битов и байтов. К сожалению, оно также часто технически неверно. Вот почему: при работе с байтами большая часть программного обеспечения на самом деле работает с основанием 2, а не с основанием 10. Если ваш компьютер говорит вам, что размер файла составляет 1 МБ, на самом деле это 1 048 576 байт! Это приблизительно один миллион, но не совсем. Это число кажется странным, не так ли? Это потому, что мы смотрим на него в десятичной системе счисления. В двоичной системе это же число выражается как 0b1000000000000000000000. Это степень двух, а именно  $2^{20}$ . В табл. 1.3 показано, как интерпретировать префиксы СИ при работе с байтами.

<sup>1</sup> Согласно ГОСТ 8.417-2002, в русскоязычной практике приставка «кило» в физике сокращается до маленькой буквы (кг, а не Кг, кВт, а не КВт), а в компьютерной технике до заглавной (Кбит, КБ), чтобы подчеркнуть, что речь идет о множителе 1024, а не 1000 (см. далее в тексте). В англоязычной практике этого правила не сложилось, потому там «кило» принято сокращать до маленькой буквы (kbit). Путаться между этими обозначениями не следует: повторим, что «Кбит» правильно переводить на английский как «kbit», и наоборот. Ранее часто биты обозначали строчной «б», но это приводило к многочисленным ошибкам при чтении. Поэтому устоялось сокращение для бита полностью («бит»). Байт, согласно отечественным правилам (ГОСТ 8.417-2002), следует сокращать до заглавной буквы «Б», но часто во избежание путаницы его пишут также полностью (Кбайт, Мбайт). – *Прим. ред.*

**Таблица 1.3.** Значение префиксов при применении их к байтам

Название префикса		Обозначение префикса		Значение	Основание 10
Англ.	Русск.	Англ.	Русск.		
tera	тера	T	Т	1,099,511,627,776	2 <sup>40</sup>
giga	гига	G	Г	1,073,741,824	2 <sup>30</sup>
mega	мега	M	М	1,048,576	2 <sup>20</sup>
kilo	кило	k	К	1,024	2 <sup>10</sup>

Другой момент путаницы с битами и байтами связан со скоростью передачи данных в сети. Интернет-провайдеры обычно заявляют скорость передачи данных в битах в секунду по основанию 10. Так, если вы получаете 50 Мбит в секунду при подключении к интернету, это означает, что вы можете передавать только 6 МБ в секунду. То есть 50 000 000 бит в секунду, разделенных на 8 бит в одном байте, дают нам 6 250 000 байт в секунду. Разделив 6 250 000 на 2<sup>20</sup>, мы получим около 6 МБ в секунду.

### Префиксы СИ для двоичных данных

Для устранения путаницы, вызванной многозначностью префиксов, в 2002 году был введен новый набор префиксов (в стандарте под названием IEEE 1541) для использования в двоичных системах. Когда речь идет о степенях числа 2, вместо кило- следует использовать киби-, вместо мега- — меби- и т. д. Эти новые префиксы соответствуют значениям с основанием 2 и предназначены для использования в ситуациях, где старые префиксы ранее использовались неправильно. Например, поскольку килобайт может интерпретироваться как 1 000 или 1 024 байт, данный стандарт рекомендует использовать кибибайты для обозначения 1 024 байт, а кило- сохраняет свое первоначальное значение, так что килобайт равен 1 000 байт. Это кажется хорошей идеей, но на момент написания данной книги эти символы еще не были широко распространены. В табл. 1.4 перечислены новые префиксы и их значения.

**Таблица 1.4.** Префиксы для двоичных данных согласно IEEE 1541-2002

Название префикса		Обозначение префикса		Значение	Основание 2
Англ.	Русск.	Англ.	Русск.		
tebi	теби	Ti	Ти	1 099 511 627 776	2 <sup>40</sup>
gibi	гиби	Gi	Ги	1 073 741 824	2 <sup>30</sup>
mebi	меби	Mi	Ми	1 048 576	2 <sup>20</sup>
kibi	киби	Ki	Ки	1024	2 <sup>10</sup>

Это различие важно, потому что на практике большая часть программного обеспечения, отображающего размер файлов, использует старый префикс СИ, но вычисляет размер по основанию 2. Другими словами, если ваше устройство сообщает, что размер файла составляет 1 КБ, это означает 1024 байта. С другой стороны, производители запоминающих устройств, как правило, заявляют емкость своих устройств, используя основание 10. Это означает, что жесткий диск, емкость которого заявлена как 1 ТБ, вероятно, содержит один триллион байт, но, если вы подключите это устройство к компьютеру, компьютер покажет размер около 931 ГБ (один триллион, разделенный на  $2^{30}$ ). Учитывая отсутствие распространения новых префиксов, в этой книге я продолжу использовать старые префиксы СИ.

## Шестнадцатеричная система

Прежде чем мы оставим тему двоичной системы, я расскажу еще об одной системе счисления – шестнадцатеричной. Если кратко резюмировать, то наша «обычная» система счисления – десятичная, или с основанием 10. Компьютеры используют двоичную систему счисления, или с основанием 2. *Шестнадцатеричная* – это система с основанием 16! Учитывая то, что вы уже узнали в этой главе, вы, вероятно, понимаете, что это значит. Шестнадцатеричная система счисления – это позиционная система счисления, в которой каждая позиция представляет собой степень шестнадцати и на каждой позиции может быть один из шестнадцати символов.

Как и во всех позиционных системах счисления, крайняя правая позиция по-прежнему будет позицией единиц. Следующая позиция слева будет позицией 16, затем позиция 256 ( $16 \times 16$ ), затем – 4 096 ( $16 \times 16 \times 16$ ) и т. д. Достаточно просто. Но как насчет другого требования, что на каждой позиции может быть один из 16 символов? Обычно мы используем десять символов для представления чисел, от 0 до 9. Нам нужно добавить еще шесть символов для представления других значений. Мы могли бы выбрать несколько случайных символов, например & @ #, но эти символы не имеют очевидного порядка. Вместо этого стандартом является использование A, B, C, D, E и F (прописные или строчные – неважно!). В этой схеме A обозначает 10, B – 11 и т. д., вплоть до F, которая обозначает 15. Это имеет смысл, так как нам нужны символы, которые представляют значения от нуля до значения основания за вычетом единицы. Поэтому наши дополнительные символы – это буквы от A до F. Стандартной практикой является использование префикса 0x для обозначения шестнадцатеричной системы, когда это необходимо подчеркнуть. В табл. 1.5 перечислены все 16 шестнадцатеричных символов, а также их десятичные и двоичные эквиваленты.



**Таблица 1.5.** Шестнадцатеричные символы

Шестнадцатеричный	Десятичный	Двоичный (4-бит)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Что же происходит, когда вам нужно считать дальше, чем 15 в десятичной системе или 0xF? Как и в десятичной системе счисления, мы добавляем еще одну позицию. После 0xF идет 0x10, что равно 16 в десятичной системе. Затем 0x11, 0x12, 0x13 и т. д. Теперь посмотрите на рис. 1-4, где мы видим большее шестнадцатеричное число, 0x1A5.

<u>1</u>	<u>A</u>	<u>5</u>
Позиция 256	Позиция 16	Позиция единиц
$16^2$	$16^1$	$16^0$
$16 \times 16 = 256$	16	1

*Рис. 1-4. Число 0x1A5 в шестнадцатеричной системе, разбитое по значениям на позициях*

На рис. 1-4 мы имеем число 0x1A5 в шестнадцатеричной системе счисления. Каково значение этого числа в десятичной системе? Крайняя правая позиция имеет значение 5. Следующая позиция имеет вес 16, и там стоит буква A, что в десятичной системе равно 10, поэтому средняя позиция означает  $16 \times 10 = 160$ . Крайняя левая позиция имеет вес 256, и на этом месте стоит 1, поэтому эта позиция означает 256. Таким образом, суммарное значение равно  $5 + 160 + 256 = 421$  в десятичной системе.

Чтобы подчеркнуть вышесказанное, отметим, что этот пример показывает, что новые символы, такие как А, имеют разное значение в зависимости от позиции, на которой они находятся. 0xA – это 10 в десятичной системе, но 0xA0 – это 160 в десятичной системе, потому что А стоит на позиции шестнадцатых.

В этот момент вы, возможно, скажете себе: «Отлично, но зачем это нужно?» Я рад, что вы спросили. Компьютеры не используют шестнадцатеричную систему счисления, как и большинство людей. И все же шестнадцатеричная система очень полезна для людей, которым необходимо работать в двоичной системе.

Использование шестнадцатеричной системы помогает преодолеть два сложных момента, связанных с работой в двоичной системе. Во-первых, большинство людей с большим трудом читает длинные последовательности из нулей и единиц. Через некоторое время все биты сбиваются в кучу. Работа с 16 и более битами утомительна и чревата ошибками. Вторая проблема заключается в том, что, хотя люди хорошо умеют работать в десятичной системе счисления, преобразование между десятичной и двоичной системами не такое уж простое. Большинству людей трудно с одного взгляда на десятичное число быстро определить, какие биты будут единицами, а какие нулями, если бы это число было представлено в двоичном виде. Но в шестнадцатеричной системе счисления преобразование в двоичную систему гораздо проще. В табл. 1.6 приведена пара примеров 16-битных двоичных чисел и их соответствующих шестнадцатеричных и десятичных представлений. Обратите внимание, что для наглядности я вставил пробелы в двоичные значения.

**Таблица 1.6.** Примеры 16-битных двоичных чисел как десятичных и шестнадцатеричных

	Пример 1	Пример 2
Двоичное	1111 0000 0000 1111	1000 1000 1000 0001
Шестнадцатеричное	F00F	8881
Десятичное	61,455	34,945

Рассмотрим пример 1 в табл. 1.6. В двоичном исчислении существует четкая последовательность: первые четыре бита равны 1, следующие восемь бит равны 0, а последние четыре бита равны 1. В десятичной системе счисления эта последовательность не просматривается. При взгляде на число 61 455 совершенно неясно, какие биты могут быть нулевыми или единичными. С другой стороны, шестнадцатеричная система отражает последовательность в двоичной системе. Первый шестнадцатеричный символ – F (что в двоичном исчислении равно 1111), следующие два шестнадцатеричных символа – 0, а последний шестнадцатеричный символ – F.

Перейдем к примеру 2, первые три набора из четырех бит равны 1000, а последний набор из четырех бит – 0001. Это легко увидеть в двоичной системе, но довольно трудно в десятичной. Шестнадцатеричная

система дает более четкую картину: шестнадцатеричный символ 8 соответствует 1000 в двоичной системе, а шестнадцатеричный символ 1 так и соответствует 1!

Надеюсь, вы заметили закономерность: каждые четыре бита в двоичной системе соответствуют одному символу в шестнадцатеричной. Если вы помните, четыре бита – это половина байта (или тетрада). Поэтому байт можно легко представить двумя шестнадцатеричными символами. 16-битное число можно представить четырьмя шестнадцатеричными символами, 32-битное число – восемью шестнадцатеричными символами и т. д. В качестве примера возьмем 32-битное число на рис. 1-5.

# 8A52FF00

1000 1010 0101 0010 1111 1111 0000 0000

Рис. 1-5. Каждый шестнадцатеричный символ соответствует 4 битам

На рис. 1-5 мы можем переварить это довольно длинное число по полбайта за раз, что невозможно при использовании десятичного представления того же числа (2 320 695 040).

Поскольку переходить от двоичной системы к шестнадцатеричной относительно легко, многие инженеры часто используют эти две системы в тандеме, переходя к десятичным числам только в случае необходимости. Позже в этой книге я буду использовать шестнадцатеричную систему, когда это будет иметь смысл.

Попробуйте преобразовать двоичные числа в шестнадцатеричные, не проходя промежуточный этап преобразования в десятичные.

## УПРАЖНЕНИЕ 1-4: Из двоичной системы в шестнадцатеричную

Переведите эти числа, представленные в двоичном формате, в их шестнадцатеричные эквиваленты. Не переводите в десятичную систему, если это возможно! Цель состоит в том, чтобы перейти непосредственно от двоичной системы к шестнадцатеричной.

10 (двоичное) = \_\_\_\_\_ (шестнадцатеричное).

11110000 (двоичное) = \_\_\_\_\_ (шестнадцатеричное).

Вы можете проверить свои ответы в приложении А.

Как только вы освоите перевод из двоичной системы в шестнадцатеричную, попробуйте перейти в обратном направлении, из шестнадцатеричной системы в двоичную.

### УПРАЖНЕНИЕ 1-5: Из шестнадцатеричной в двоичную

Преобразуйте эти числа, представленные в шестнадцатеричной системе счисления, в их двоичные эквиваленты. Не переводите в десятичную систему счисления, если это возможно! Цель состоит в том, чтобы перейти непосредственно от шестнадцатеричной системы к двоичной.

1A (шестнадцатеричное) = \_\_\_\_\_ (двоичное).

C3A0 (шестнадцатеричное) = \_\_\_\_\_ (двоичное).

Вы можете проверить свои ответы в приложении А.

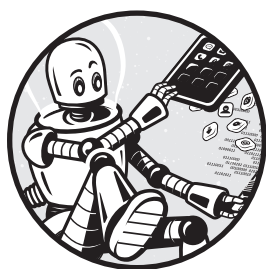
## Выводы

В этой главе мы рассмотрели некоторые основополагающие понятия вычислительной техники. Вы узнали, что компьютер – это любое электронное устройство, которое может быть запрограммировано на выполнение набора логических инструкций. Затем вы увидели, что современные компьютеры являются скорее цифровыми, чем аналоговыми устройствами, и узнали разницу между этими понятиями: аналоговые системы – это системы, использующие для представления данных широко варьирующиеся значения, в то время как цифровые системы представляют данные в виде последовательности символов. После этого мы рассмотрели, как современные цифровые компьютеры используют только два символа, 0 и 1, и узнали о системе счисления по основанию 2, состоящей только из двух символов, или двоичной системе. Мы разобрали биты, байты и стандартные префиксы СИ (гига-, мега-, кило- и т. д.), которые можно использовать для более удобного описания размера данных. Наконец, вы узнали, как шестнадцатеричная система счисления полезна для людей, которым необходимо работать в двоичной системе.

В следующей главе мы более подробно рассмотрим, как двоичная система используется в цифровых системах. Мы рассмотрим, как двоичная система может использоваться для представления различных типов данных, и увидим, как работает двоичная логика.

# 2

## ДВОИЧНЫЙ КОД В ДЕЙСТВИИ



В предыдущей главе мы определили компьютер как электронное устройство, которое может быть запрограммировано на выполнение набора логических инструкций. Затем разобрали довольно тщательно, что все в компьютере, от данных, которые он использует, до инструкций, которые он выполняет, хранится в двоичном виде, в виде нулей и единиц. В этой главе я проливаю свет на то, как именно нули и единицы могут использоваться для представления практически любых данных. Мы также рассмотрим, как двоичная система может применяться для логических операций.

### Представление данных в цифровом виде

До сих пор мы рассматривали хранение чисел в двоичном формате. Если более конкретно, мы рассмотрели, как хранить целые положительные числа и ноль. Однако компьютеры хранят все данные в виде битов: отрицательные числа, дробные числа, текст, цвета, изображения, аудио и видео и многое другое. Давайте рассмотрим, как различные типы данных могут быть представлены с помощью двоичной системы.

## Цифровой текст

Начнем с текста в качестве первого примера того, как биты, нули и единицы могут представлять что-то, помимо чисел. В контексте вычислительной техники текст означает набор буквенно-цифровых и относящихся к таковым символов, также называемых *знаками*. Текст обычно используется для представления слов, предложений, абзацев и т. д. Текст не включает форматирование (жирный шрифт, курсив). Для простоты пока ограничим наш набор знаков английским алфавитом и связанными с ним символами. В компьютерном программировании термин «строка» также обычно используется для обозначения последовательности текстовых символов.

Принимая во внимание это определение текста, что именно нам нужно представить? Нам нужны символы от А до Z, прописные и строчные, т. е. символ «А» должен отличаться от символа «а». Нам также нужны знаки препинания, такие как запятые и точки. Нам нужен способ обозначения пробелов. Нам также нужны цифры от 0 до 9. Требование к цифрам может сбить с толку; здесь я говорю о включении *символов* или *знаков*, которые используются для представления чисел от 0 до 9, что отличается от хранения *чисел* от 0 до 9.

Если сложить все уникальные символы, необходимые для представления всего вышеописанного, то получится около 100 знаков. Итак, если нам нужно иметь уникальную комбинацию битов для представления каждого символа, сколько битов нам нужно на один символ? 6-битное число дает 64 уникальные комбинации, что не совсем достаточно. Но 7-битное число дает нам 128 комбинаций, что достаточно для представления ста или около того символов, которые нам нужны. Однако, поскольку компьютеры обычно работают в байтах, имеет смысл просто округлить и использовать целых 8 бит, один байт для представления каждого символа. С помощью байта мы можем представить 256 уникальных символов.

Как же нам использовать 8 бит для представления каждого символа? Как вы, возможно, и ожидаете, уже существует стандартный способ представления текста в двоичном виде, и мы перейдем к нему через минуту. Но прежде важно понять, что мы можем придумать любую схему для представления каждого символа, если только программное обеспечение, работающее на компьютере, знает о нашей схеме. При этом некоторые схемы лучше других подходят для представления определенных типов данных. Разработчики программного обеспечения предпочитают схемы, которые упрощают выполнение обычных операций.

Представьте, что вы отвечаете за создание собственной системы, которая представляет каждый символ в виде набора битов. Вы можете выбрать 0b000000 для представления символа А, 0b00000001 для представления символа В и т. д. Этот процесс перевода данных в цифровой формат называется *кодированием*; а когда вы интерпретируете эти цифровые данные, это называется *декодированием*.

## УПРАЖНЕНИЕ 2-1: Создайте собственную систему представления текста

Определите способ представления заглавных букв от A до D в виде 8-битных чисел, а затем закодируйте слово DAD в 24 бита с помощью вашей системы. Здесь нет единственно правильного ответа; пример ответа см. в приложении А. Бонус: представьте ваше закодированное 24-битное число в шестнадцатеричном виде.

## ASCII

К счастью, у нас уже есть несколько стандартных способов представления текста в цифровом виде, поэтому нам не нужно изобретать свой собственный! *Американский стандартный код для обмена информацией* (*American Standard Code for Information Interchange* – ASCII) – это формат, который представляет 128 символов, используя 7 бит на символ, хотя обычно каждый символ хранится в полном байте из 8 бит. Использование 8 бит вместо 7 означает лишь, что у нас есть дополнительный старший (наиболее значимый) бит, равный 0. ASCII обрабатывает символы, необходимые для английского языка, а другой стандарт, называемый *Unicode*, обрабатывает символы, используемые почти во всех языках, включая английский. Пока что для простоты изложения остановимся на ASCII. В табл. 2.1 показаны двоичные и шестнадцатеричные значения для подмножества символов ASCII. Первые 32 символа не показаны; это управляющие символы, такие как возврат каретки и прогон листа, изначально предназначенные для управления устройствами, а не для хранения текста.

## УПРАЖНЕНИЕ 2-2: Кодировка и декодировка ASCII

Используя табл. 2.1, закодируйте следующие слова в двоичном и шестнадцатеричном коде ASCII, взяв по одному байту для каждого символа. Помните, что для заглавных и строчных букв существуют разные значения.

- Hello
- 5 cats

Используя табл. 2.1, декодируйте следующие слова. Каждый символ представлен в виде 8-битного значения ASCII с пробелами для наглядности.

- 01000011 01101111 01100110 01100110 01100101 01100101
- 01010011 01101000 01101111 01110000

Используя табл. 2.1, расшифруйте следующее слово. Каждый символ представлен в виде 8-битного шестнадцатеричного значения с пробелами для наглядности.

- 43 6C 61 72 69 6E 65 74

Ответы приведены в приложении А.

**Таблица 2.1.** Символы ASCII от 0x20 до 0x7F

Двоичное	Шест- надца- терич- ное	Символ	Двоичное	Шест- надца- терич- ное	Символ	Двоичное	Шест- надца- терич- ное	Символ
00100000	20	[Space]	01000000	40	@	01100000	60	`
00100001	21	!	01000001	41	A	01100001	61	a
00100010	22	"	01000010	42	B	01100010	62	b
00100011	23	#	01000011	43	C	01100011	63	c
00100100	24	\$	01000100	44	D	01100100	64	d
00100101	25	%	01000101	45	E	01100101	65	e
00100110	26	&	01000110	46	F	01100110	66	f
00100111	27	'	01000111	47	G	01100111	67	g
00101000	28	(	01001000	48	H	01101000	68	h
00101001	29	)	01001001	49	I	01101001	69	i
00101010	2A	*	01001010	4A	J	01101010	6A	j
00101011	2B	+	01001011	4B	K	01101011	6B	k
00101100	2C	,	01001100	4C	L	01101100	6C	l
00101101	2D	-	01001101	4D	M	01101101	6D	m
00101110	2E	.	01001110	4E	N	01101110	6E	n
00101111	2F	/	01001111	4F	O	01101111	6F	o
00110000	30	0	01010000	50	P	01110000	70	p
00110001	31	1	01010001	51	Q	01110001	71	q
00110010	32	2	01010010	52	R	01110010	72	r
00110011	33	3	01010011	53	S	01110011	73	s
00110100	34	4	01010100	54	T	01110100	74	t
00110101	35	5	01010101	55	U	01110101	75	u
00110110	36	6	01010110	56	V	01110110	76	v
00110111	37	7	01010111	57	W	01110111	77	w
00111000	38	8	01011000	58	X	01111000	78	x
00111001	39	9	01011001	59	Y	01111001	79	y
00111010	3A	:	01011010	5A	Z	01111010	7A	z
00111011	3B	;	01011011	5B	[	01111011	7B	{
00111100	3C	<	01011100	5C	\	01111100	7C	
00111101	3D	=	01011101	5D	]	01111101	7D	}
00111110	3E	>	01011110	5E	^	01111110	7E	~
00111111	3F	?	01011111	5F	_	01111111	7F	[Delete]



Представить текст в цифровом формате довольно просто. Система, подобная ASCII, отображает каждый символ, или знак, в уникальную последовательность битов. Затем вычислительное устройство интерпретирует эту последовательность битов и показывает пользователю соответствующий символ.

## Цвета и изображения в цифровом формате

Теперь, когда мы рассмотрели, как представлять числа и текст в двоичном формате, давайте изучим другой тип данных: цвет. Любое вычислительное устройство, имеющее цветной графический дисплей, должно иметь определенную систему для описания цветов. Как и следовало ожидать, аналогично случаю с текстом у нас уже есть стандартные способы хранения данных о цвете. Мы еще вернемся к ним, но сначала давайте разработаем нашу собственную систему цифрового описания цветов.

Ограничим наш диапазон цветов черным, белым и оттенками серого. Этот ограниченный набор цветов известен как *градации серого*. Как и в случае с текстом, начнем с того, что определим, сколько уникальных оттенков серого мы хотим представить. Давайте не будем усложнять и остановимся на черном, белом, темно-сером и светло-сером. Это всего четыре цвета серой шкалы, так сколько же битов нам нужно для представления четырех цветов? Необходимо всего 2 бита. Двухбитное число может представлять четыре уникальных значения, поскольку  $2$  в степени  $2$  равно  $4$ .

### УПРАЖНЕНИЕ 2-3: Создание собственной системы представления градации серого

Определите способ цифрового представления черного, белого, темно-серого и светло-серого цветов. Здесь нет единственно правильного ответа; пример ответа см. в приложении А.

Разработав систему представления градации серого в двоичном виде, вы можете развить этот подход и создать собственную систему для описания простого изображения в градациях серого. Изображение, по сути, – это расположение цветов на двумерной плоскости. Эти цвета обычно располагаются в виде сетки, состоящей из одноцветных квадратов, называемых *пикселями*. Простой пример приведен на рис. 2-1.

Изображение на рис. 2-1 имеет ширину 4 пикселя и высоту 4 пикселя, что дает нам в общем 16 пикселей. Если прищуриться и использовать воображение, то можно увидеть белый цветок и темное небо за ним. Изображение состоит только из трех цветов: белого, светло-серого и темно-серого.

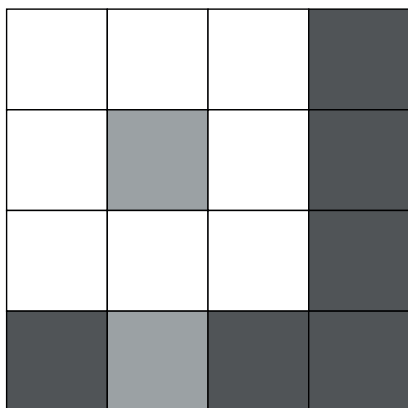


Рис. 2-1. Простое изображение

#### ПРИМЕЧАНИЕ

Рисунок 2-1 состоит из нескольких действительно больших пикселей, чтобы проиллюстрировать задачу. Об экранах современных телевизоров, о компьютерных мониторах и экранах смартфонов также можно говорить как о сетке из пикселей, но каждый пиксель там очень мал. Например, экран с HD-разрешением<sup>1</sup> обычно имеет размер 1920 пикселей (по ширине) на 1080 пикселей (по высоте), что в общей сложности составляет около 2 млн пикселей! Другой пример: цифровые фотографии часто содержат более 10 млн пикселей в одном изображении.

### УПРАЖНЕНИЕ 2-4: Создание собственного подхода к представлению простых изображений

**Часть 1.** На основе вашей предыдущей системы представления цветов градации серого разработайте подход к представлению изображения, состоящего из этих цветов. Если вы хотите упростить задачу, можете предположить, что изображение всегда будет иметь размер 4 на 4 пикселя, как на рис. 2-1.

**Часть 2.** Используя свой подход из части 1, запишите двоичное представление изображения цветка на рис. 2-1.

**Часть 3.** Объясните другу свой подход к представлению изображений. Затем дайте другу свои двоичные данные и посмотрите, сможет ли он получить изображение, представленное выше, не видя исходной картинки!

Здесь нет единственно правильного ответа; пример ответа см. в приложении А.

<sup>1</sup> От *high definition*, что и означает «высокое разрешение». – Прим. ред.

В части 2 упражнения 2-4 вы действовали как компьютерная программа, отвечающая за кодирование изображения в двоичные данные. В части 3 ваш друг действовал как компьютерная программа, отвечающая за обратное: декодирование двоичных данных в изображение. Надеюсь, ваш друг смог расшифровать двоичные данные и нарисовать цветок! Если он справился с этим, то отлично: вместе вы продемонстрировали, как программное обеспечение кодирует и декодирует данные! Если все пошло не так хорошо, и в итоге он нарисовал что-то больше похожее на маринованный огурец, чем на цветок, это тоже нормально; вы продемонстрировали, что бывает, если программное обеспечение имеет недостатки, способные привести к самым неожиданным результатам.

## Подходы к представлению цветов и изображений

Как упоминалось ранее, уже существуют стандартные подходы к представлению цветов и изображений в цифровом виде. Для изображений в градациях серого одним из распространенных подходов является использование 8 бит на пиксель, что позволяет получить 256 оттенков серого.

Значение каждого пикселя обычно представляет интенсивность света, поэтому 0 означает отсутствие интенсивности света (черный), 255 представляет полную интенсивность (белый), а значения между ними представляют собой различные оттенки серого, от темного до светлого. На рис. 2-2 показаны различные градации серого с использованием 8-битной схемы кодирования.

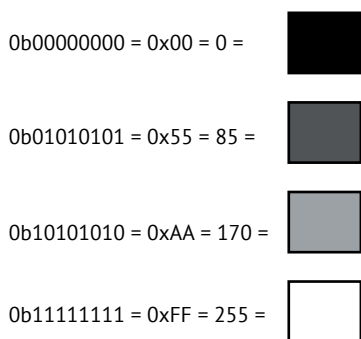


Рис. 2-2. Градации серого, представленные 8 битами в двоичной, шестнадцатеричной и десятичной системах

Представление цветов за пределами градации серого работает аналогичным образом. Хотя градации серого могут быть представлены одним 8-битным числом, подход, известный как RGB, использует три 8-битных числа для представления интенсивности красного, зеленого и синего цветов, которые в совокупности образуют один цвет. Выделение 8 бит на каждый из трех составляющих цветов означает, что для представления полного цвета требуется 24 бита.

### ПРИМЕЧАНИЕ

*RGB основан на аддитивной цветовой модели, в которой цвета состоят из смеси красного, зеленого и синего цвета. Это отличается от субтрактивной цветовой модели, используемой в живописи, где смешиваются красный, желтый и синий цвета<sup>1</sup>.*

Например, красный цвет представлен в RGB со всеми 8 красными битами, равными 1, где остальные 16 бит для двух других цветов установлены равными 0. Или, если вы хотите представить желтый цвет, который является комбинацией красного и зеленого, но без участия синего, можете установить красные и зеленые биты равными единицам, а синие биты оставить нулевыми. Это показано на рис. 2-3.

111111100000000000000000 = 0xFF0000 = Red

Красный Зеленый Синий

111111111111111100000000 = 0xFFFF00 = Yellow

Красный Зеленый Синий

Рис. 2-3. Красный и желтый, представленные с использованием RGB

В обоих примерах на рис. 2-3 «включенные» цвета полностью представлены единицами, но система RGB позволяет красному, синему и зеленому компонентным цветам также иметь частичную интенсивность.

Каждый компонентный цвет может изменяться от 00000000 (0 десятичное / 0 шестнадцатеричное) до 11111111 (255 десятичное / FF шестнадцатеричное). Меньшее значение означает более темный оттенок этого цвета, а большее значение – более яркий оттенок этого цвета. Благодаря такой гибкости смешивания цветов мы можем изобразить практически любой оттенок, который только можно себе представить.

Существуют не только стандартные способы представления цветов, но и множество общепринятых подходов к представлению полного изображения. Как вы видели на рис. 2-1, мы можем создавать изображения с помощью сетки пикселей, каждый из которых имеет определен-

Аддитивная (от слова *addition* – сложение) модель складывает самосветящиеся прозрачные цвета и обычно относится к экранам как устройствам, обладающим собственным свечением. Субтрактивная (от слова *subtract* – вычитание) модель вычитает друг из друга отраженные цвета красок, нанесенных на какой-либо непрозрачный носитель (например, бумагу). В стандартной субтрактивной модели используют не чистые красный-синий-желтый, а бирюзовый (Cyan), малиновый (Magenta) и желтый (Yellow). К этим трем цветам часто прибавляют черный (black) для получения более насыщенных темных оттенков, потому полностью такая цветовая модель носит название CMYK. Обычно компьютерщики имеют дело с экранами, потому в основном работают с аддитивной моделью RGB, обращаясь к субтрактивной модели CMYK только при необходимости печати изображений в типографии (при печати на цветном принтере все преобразование производится программным обеспечением принтера автоматически) – *Прим. ред.*

ный цвет. За прошедшие годы было разработано множество форматов изображений, позволяющих сделать именно это. Упрощенный подход к представлению изображения называется *растровым*. Растровые изображения хранят данные о цвете RGB для каждого отдельного пикселя. Некоторые растровые форматы, такие как JPEG и PNG, используют методы сжатия, чтобы уменьшить количество байтов, необходимых для хранения изображения, по сравнению с «чистым растром» (пример – формат BMP).

## Интерпретация двоичных данных

Давайте рассмотрим еще одно двоичное значение: 011000010110001001100011. Как вы думаете, что оно представляет? Если мы предположим, что это текстовая строка ASCII, то она представляет «abc». С другой стороны, возможно, оно представляет 24-битный цвет RGB (0x61, 0x62, 0x63), что делает его оттенком, близким к темно-серому. А может быть, это целое положительное число, в таком случае в десятичном исчислении оно равно 6 382 179. Эти различные интерпретации показаны на рис. 2-4.

011000010110001001100011

abc

по ASCII

6,382,179

как 32-битное целое число



как цвет в RGB

Рис. 2-4. Интерпретации 011000010110001001100011

Так что же оно означает? Это может быть любая из этих интерпретаций или что-то совсем другое. Все зависит от контекста, в котором интерпретируются данные. Программа текстового редактора будет считать, что данные – это текст, в то время как программа для просмотра изображений может считать, что это цвет пикселя изображения, а калькулятор будет считать, что это число. Каждая программа написана так, что ожидает данные в определенном формате, поэтому одно двоичное значение имеет разное значение в разных контекстах.

Мы показали, как двоичные данные могут использоваться для представления чисел, текста, цветов и изображений. На основании этого вы можете сделать некоторые предположения о том, как можно хранить другие типы данных, например видео или аудио. Нет никаких ограничений на то, какие типы данных могут быть представлены в цифровом виде. Цифровое представление не всегда является точной копией ис-

ходных данных, но во многих случаях это не является проблемой. Возможность представить что-либо в виде последовательности из нулей и единиц чрезвычайно полезна, поскольку, создав устройство, работающее с двоичными данными, мы можем адаптировать его с помощью программного обеспечения для работы с любыми данными!

## Двоичная логика

Мы убедились в полезности использования двоичной системы для представления данных, но компьютеры способны на большее, чем просто хранение данных. Они позволяют нам также работать с данными. С помощью компьютера мы можем читать, редактировать, создавать, преобразовывать, обмениваться и иным образом манипулировать данными. Компьютеры дают нам возможность обрабатывать данные с помощью аппаратного обеспечения, которое мы можем запрограммировать на выполнение последовательности простых инструкций, таких как «сложить два числа вместе» или «проверить, равны ли два значения». Компьютерные процессоры, выполняющие эти инструкции, основаны на *двоичной логике* – системе описания логических утверждений, в которой переменные могут иметь только одно из двух значений – истину или ложь. Давайте разберем двоичную логику, и в процессе мы снова увидим, как все в компьютере сводится к единицам и нулям.

Давайте убедимся, насколько естественно двоичная система подходит для логики. Обычно, когда кто-то говорит о логике, он имеет в виду аргументацию или обдумывание того, что известно, чтобы прийти к обоснованному выводу. Когда нам предоставляется набор фактов, логика позволяет определить, является ли другое связанное с ними утверждение также фактическим. Логика – это все о правде: о том, что истинно, а что ложно. Аналогичным образом бит может иметь только одно из двух значений – 1 или 0. Поэтому один бит может быть использован для представления логического состояния – истинного (1) или ложного (0).

Рассмотрим пример логического утверждения.

---

ДАНО, что фигура имеет четыре прямые стороны,  
И ДАНО, что фигура имеет четыре прямых угла,  
Я ДЕЛАЮ ВЫВОД, что фигура является прямоугольником.

---

В этом примере есть два условия (четыре стороны, четыре прямых угла), *оба* из которых должны быть истинными, чтобы заключение также было истинным. Для такой ситуации мы используем логический оператор (логическую функцию) И (AND), чтобы соединить два утверждения вместе. Если одно из условий ложно, то заключение также ложно. Я представил ту же логику в табл. 2-2.

**Таблица 2-2.** Логическое утверждение для прямоугольника

Четыре стороны	Четыре прямых угла	Является прямоугольником
Ложь	Ложь	Ложь
Ложь	Истина	Ложь
Истина	Ложь	Ложь
Истина	Истина	Истина

Используя табл. 2-2, мы можем интерпретировать каждую строку следующим образом.

1. Если фигура *не имеет* четырех сторон и *не имеет* четырех прямых углов, то это *не* прямоугольник.
2. Если у фигуры *нет* четырех сторон, но *есть* четыре прямых угла, она *не является* прямоугольником.
3. Если фигура *имеет* четыре стороны и *не имеет* четырех прямых углов, она *не является* прямоугольником.
4. Если у фигуры *есть* четыре стороны и *есть* четыре прямых угла, то она *является* прямоугольником!

Этот тип таблицы известен как *таблица истинности*: таблица, которая показывает все возможные комбинации условий (входы) и их логические выводы (выходы). Таблица 2-2 была составлена специально для нашего утверждения о прямоугольнике, но на самом деле эта же таблица применима к любому логическому утверждению, соединенному с помощью оператора И (AND).

Таблицу 2-3 я сделал более общей, используя А и В для представления наших двух входных условий, а Выход (Output) – для представления логического результата. В частности, в этой таблице Выход (Output) – это результат функции А И В (A AND B).

**Таблица 2-3.** Таблица истинности с И (AND)

А	В	Выход (Output)
Ложь	Ложь	Ложь
Ложь	True	Ложь
Истина	Ложь	Ложь
Истина	Истина	Истина

В табл. 2-4 я внес еще одно изменение. Поскольку эта книга посвящена вычислительной технике, я представил ложь как 0, а истину как 1, так же как это делают компьютеры.

Таблица 2-4 – это стандартная форма таблицы истинности для логической функции И (AND), когда вы имеете дело с цифровыми системами, использующими 0 и 1. Компьютерные инженеры используют такие таблицы, чтобы показать, как будут вести себя компоненты при предъявлении им определенного набора входных данных. Теперь давайте

рассмотрим, как это работает с другими логическими операторами и более сложными логическими утверждениями.

**Таблица 2-4.** Таблица истинности с И (AND)

A	B	Выход (Output)
0	0	0
0	1	0
1	0	0
1	1	1

Допустим, вы работаете в магазине, который предоставляет скидку только двум типам покупателей: детям и людям в солнцезащитных очках. Больше никто не имеет права на скидку. Если бы вы хотели изложить политику магазина в виде логического выражения, вы могли бы сказать следующее:

---

ДАНО, что покупатель – ребенок,  
ИЛИ ДАНО, что покупатель носит солнцезащитные очки,  
Я ДЕЛАЮ ВЫВОД, что покупатель имеет право на скидку.

---

Здесь у нас есть два условия (ребенок, носит солнцезащитные очки), где *хотя бы одно* условие должно быть истинным, чтобы заключение было истинным. В этой ситуации мы используем логический оператор ИЛИ (OR), чтобы соединить два утверждения вместе. Если одно из условий истинно, то заключение также истинно. Мы можем выразить это в виде таблицы истинности, как показано в табл. 2-5.

**Таблица 2-5.** Таблица истинности с ИЛИ (OR)

A	B	Выход (Output)
0	0	0
0	1	1
1	0	1
1	1	1

Смотря на входные (inputs) и выходные (outputs) данные в табл. 2-5, мы можем быстро увидеть, что скидка будет предоставлена (Output = 1), когда либо клиент является ребенком (A = 1), либо клиент носит солнцезащитные очки (B = 1). Обратите внимание, что значения столбцов ввода для A и B абсолютно одинаковы как в табл. 2-4, так и в табл. 2-5. Это логично, поскольку в обеих таблицах два входа и, следовательно, один и тот же возможный набор комбинаций входов. Отличие заключается в столбце «Выход» (Output).

Давайте используем функции И (AND) и ИЛИ (OR) для построения более сложного логического утверждения. Для примера предположим,



что я хожу на пляж каждый солнечный и теплый день, а также предположим, что я хожу на пляж каждый год в свой день рождения. На самом деле я всегда хожу на пляж только при этих конкретных обстоятельствах – моя жена говорит, что я слишком упрям в этом отношении. Комбинируя эти условия, мы получаем следующее логическое утверждение:

---

ДАНО, что сейчас солнечно, И ДАНО, что тепло,  
ИЛИ ДАНО, что сегодня мой день рождения,  
Я ДЕЛАЮ ВЫВОД, что иду на пляж.

---

Давайте подпишем наши входные условия, а затем нарисуем таблицу истинности для этого выражения.

**Условие А.** Солнечно.

**Условие В.** Тепло.

**Условие С.** Сегодня мой день рождения.

Наше логическое выражение будет выглядеть следующим образом:

---

(А И В) ИЛИ С  
((A AND B) OR C)

---

Как и в алгебраическом выражении, круглые скобки вокруг А И (AND) В означают, что эта часть выражения должна быть оценена первой. В табл. 2-6 приведена таблица истинности для этого логического выражения.

**Таблица 2-6.** Таблица истинности для функции (А И В) ИЛИ С

А	В	С	Выход (Output)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Таблица 2-6 немного сложнее, чем простая таблица истинности для И (AND), но она все еще понятна. Формат таблицы позволяет легко найти определенное условие и увидеть результат. Например, третья строка говорит нам, что если А = 0 (*не* солнечно), В = 1 (тепло), С = 0 (сегодня *не* мой день рождения), то Выход = 0 (*я не* пойду сегодня на пляж).

Подобная логика – это то, с чем регулярно приходится работать компьютерам. На самом деле, как уже упоминалось ранее, фундаментальные

возможности компьютера сводятся к наборам логических операций. Хотя простой оператор И (AND) может показаться далеким от возможностей смартфона или ноутбука, эти логические операторы служат концептуальными строительными блоками всех цифровых компьютеров.

### УПРАЖНЕНИЕ 2-5: Составление таблицы истинности для логической функции

В табл. 2-7 показаны состояния трех входов для некоего логического выражения. Заполните таблицу истинности для функции (А ИЛИ В) И С ((A OR B) AND C). Ответ приведен в приложении А.

**Таблица 2-7.** Таблица истинности для функции (А ИЛИ В) И С

A	B	C	Выход (Output)
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Кроме операторов И (AND) и ИЛИ (OR), при разработке цифровых систем используются еще несколько распространенных логических операторов. На следующих страницах я рассмотрю каждый оператор и приведу таблицу истинности для каждого из них. Мы снова используем их в главе 4, посвященной цифровым схемам.

Логический оператор НЕ (NOT) – это именно то, что написано: выходное значение противоположно входному условию. То есть если А истинно, то выход *не* истинен, и наоборот. Как видно из табл. 2-8, НЕ (NOT) принимает только одно значение на вход, а не два.

**Таблица 2-8.** Таблица истинности для НЕ (NOT)

A	Выход (Output)
0	1
1	0

Оператор НЕ-И (NAND) означает НЕ И, поэтому выходной сигнал будет обратным логическому И. Если оба входа истинны, результат бу-

дет ложным. В противном случае результат будет истинным. Это показано в табл. 2-9.

**Таблица 2-9.** Таблица истинности для НЕ-И (NAND)

A	B	Выход (Output)
0	0	1
0	1	1
1	0	1
1	1	0

Оператор НЕ-ИЛИ (NOR) означает НЕ ИЛИ, поэтому на выходе получается результат, обратный оператору ИЛИ. Если оба входа ложны, результат будет истинным. В противном случае результат будет ложным. В табл. 2-10 это представлено в виде таблицы истинности.

**Таблица 2-10.** Таблица истинности для НЕ-ИЛИ (NOR)

A	B	Выход (Output)
0	0	1
0	1	0
1	0	0
1	1	0

XOR – это Исключающее ИЛИ, что означает, что только один (исключительный) вход может быть истинным, чтобы результат был истинным. То есть результат будет истинным, если истинно только А или только В, в то время как результат будет ложным, если оба входа истинны. Это подробно описано в табл. 2-11.

**Таблица 2-11.** Таблица истинности для XOR

A	B	Выход (Output)
0	0	0
0	1	1
1	0	1
1	1	0

Учение о логических функциях переменных с двумя значениями (истина или ложь) известно как *булева алгебра* или *булева логика*. Джордж Буль описал этот подход к логике в 1800-х годах, задолго до появления цифровых компьютеров. Его работа оказалась основополагающей для развития цифровой электроники, включая компьютеры.

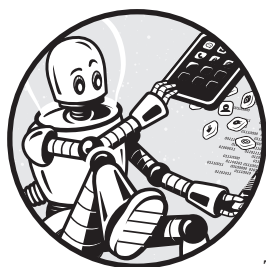
## Выводы

В этой главе мы рассмотрели, как двоичная система используется для представления данных и логических состояний. Вы узнали, как нули и единицы могут быть использованы для представления практически любых данных. Мы рассмотрели текст, цвета и изображения как примеры данных в двоичном формате. Вас познакомили с различными логическими операторами, такими как И (AND) и ИЛИ (OR), и вы узнали об использовании таблиц истинности для представления логических утверждений. Понимание этого важно, поскольку сложные процессоры, используемые в современных компьютерах, основаны на сложной логической системе.

Мы вернемся к теме двоичной системы при обсуждении цифровых схем в главе 4, но сначала, чтобы подготовить вас к этой теме, сделаем экскурс в главу 3, чтобы рассмотреть основы электрических цепей. Мы изучим законы электричества, посмотрим, как работают электрические цепи, и познакомимся с некоторыми основными компонентами, встречающимися во многих цепях. У вас даже будет возможность собрать собственную электрическую цепь!

# 3

## ЭЛЕКТРИЧЕСКИЕ ЦЕПИ



Мы рассмотрели некоторые аспекты вычислительных технологий на концептуальном уровне; теперь давайте сменим направление и рассмотрим физические основы вычислительной техники. Для начала еще раз рассмотрим наше определение компьютера. Компьютер – это электронное устройство, которое может быть запрограммировано на выполнение набора логических инструкций.

Компьютеры – это устройства, которые следуют правилам, разработанным людьми, но в конечном итоге компьютеры должны вести себя в соответствии с другим набором правил: законами природы. Компьютер – это просто машина, и, как и все машины, он использует законы природы для выполнения задачи. В частности, современные компьютеры – это электронные устройства, поэтому законы электричества являются естественным фундаментом, на котором созданы эти устройства. Чтобы получить всестороннее понимание о вычислительной технике, необходимо иметь представление об электричестве и электрических цепях; именно это мы и рассмотрим в данной главе. Давайте начнем с электрических терминов и понятий, изучим некоторые законы электричества, рассмотрим электрические схемы и, наконец, построим несколько простых электрических цепей.

# Определение электрических терминов

Для того чтобы мы могли обсуждать электрические цепи, вам сначала необходимо ознакомиться с несколькими ключевыми понятиями и терминами. Сейчас я расскажу об этих понятиях электричества и объясню, как они связаны друг с другом. Этот раздел полон погружения в детали, поэтому давайте начнем с обзора, приведенного в табл. 3-1, прежде чем перейти к конкретике.

Таблица 3-1. Сводка основных электрических терминов

Термин	Объяснение	Единицы измерения	Аналогия с водой
Электрический заряд	Заставляет материю испытывать силу	Кулоны	Вода
Электрический ток	Поток заряда	Амперы	Поток воды по трубе
Напряжение	Разность электрических потенциалов между двумя точками	Вольты	Давление воды
Сопротивление	Мера трудности прохождения тока через материал	Омы	Диаметр трубы

В табл. 3-1 дано простое объяснение каждого термина, указаны единицы измерения, и каждый термин соотнесен с его аналогией в водной системе (что показано на рис. 3-2 далее в этой главе). Если это не сразу понятно, не волнуйтесь! Мы рассмотрим каждый термин более подробно на следующих страницах.

## Электрический заряд

В школе вы могли узнать, что атомы состоят из положительно заряженных протонов, отрицательно заряженных электронов и незаряженных нейтронов. *Электрический заряд* заставляет материю испытывать силу: неодинаковые заряды притягиваются, в то время как одинаковые отталкиваются. При объяснении концепции электрических цепей мне нравится использовать аналогию с водой, текущей по трубе. В этой аналогии электрический заряд похож на воду, а провод – на трубу.

Единицей измерения заряда является *кулон*. Заряд одного протона или электрона составляет крошечную долю заряда, представленного одним кулоном.

## Электрический ток

Особое значение для нашего обсуждения имеет передача или перемещение электрического заряда, известное как *электрический ток*. Перемещение заряда по проводу аналогично перемещению воды по трубе. В повседневном обиходе, в том числе и в этой книге, мы говорим, что «ток течет», хотя если быть точным, то это заряд течет, а ток – это результат измерения интенсивности этого потока заряда.

При представлении тока в уравнениях используется символ  $I$  или  $i$ . Ток измеряется в амперах, сокращенно А.

Один ампер соответствует одному кулону в секунду. Допустим, у вас есть два провода, по первому из которых течет ток 5 А, а по второму – 1 А. Поскольку амперы представляют собой скорость изменения количества заряда («расход заряда», аналогично «расходу воды»), по первому проводу за то же время движется в пять раз больше зарядов, чем по второму.

## Напряжение

Поскольку заряд течет по проводу, как вода по трубе, давайте расширим эту аналогию. Пока что у нас есть вода (электрический заряд), простая труба (медный провод) и интенсивность (расход) потока воды (электрический ток). Давайте добавим к этому насос, который перемещает воду по трубе. Чем больше давление воды, тем быстрее вода течет по трубе, т. е. тем больше ее расход. Применительно к электрическим цепям водяной насос представляет собой *источник питания*, источник электрической энергии, например батарею.

Давление воды в этой аналогии представляет собой новое понятие: *напряжение*. Подобно тому как давление воды влияет на количество воды, протекающей по трубе в единицу времени, напряжение влияет на ток – количество заряда, протекающее по цепи. Чтобы понять, что такое напряжение, вспомните из курса естественных наук, что *потенциальная энергия*, измеряемая в *джоулях*, – это способность совершать работу. В случае с электричеством *работа* означает перемещение заряда из одной точки в другую. *Электрический потенциал* – это потенциальная энергия на единицу электрического заряда, измеряемая в джоулях на кулон. Напряжение определяется как разность электрических потенциалов между двумя точками. То есть напряжение – это работа, необходимая для перемещения одного кулона заряда из одной точки в другую.

При представлении напряжения в уравнении используется символ  $V$  или  $v^1$ . Напряжение измеряется в *вольтах*, сокращенно В. Напряжение всегда измеряется между двумя точками, например положительной и отрицательной клеммами батареи. *Клемма* в данном контексте озна-

---

<sup>1</sup> В русскоязычной технической литературе принято обозначение напряжения буквой  $U$ , но здесь будем придерживаться авторских обозначений. – *Прим. ред.*

чает точку электрического соединения. Чем выше напряжение, тем большее «давление» перемещения заряда по цепи от одной клеммы к другой и, следовательно, тем выше ток при подключении источника напряжения к цепи. Однако напряжение может существовать даже при отсутствии тока. Например, 9-вольтовая батарейка имеет напряжение 9 В на своих клеммах, даже если она ни к чему не подключена.

## Сопrotивление

Давайте вернемся к нашей аналогии с водой. Еще одним фактором, влияющим на скорость движения воды, является диаметр трубы. Широкая труба позволяет воде течь беспрепятственно, а узкая труба затрудняет течение. Если применить эту аналогию к электрическим цепям, то диаметр трубы представляет собой *электрическое сопротивление* в цепи. Электрическое сопротивление зависит и от толщины проводника и от материала, из которого он изготовлен. Чем выше сопротивление материала, тем труднее току пройти через него.

При обозначении сопротивления в уравнении используется символ  $R$ . Сопротивление измеряется в *омах*, сокращенно Ом (международное обозначение:  $\Omega$ , греческая буква омега). Медный провод имеет очень низкое сопротивление; для наших целей будем считать, как будто он вообще не имеет сопротивления, т. е. ток может свободно проходить по нему.

Электрический компонент, называемый *резистором*, используется в электрических цепях для создания определенного сопротивления там, где это необходимо. Фотография типичного резистора приведена на рис. 3-1.

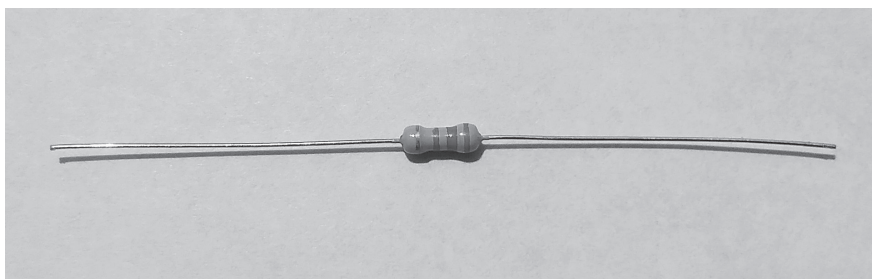


Рис. 3-1. Резистор

## Аналогия с водой

Теперь, когда мы рассмотрели основные электрические понятия, давайте вернемся к аналогии с водой, которую мы использовали для объяснения работы электрических цепей, как показано на рис. 3-2.



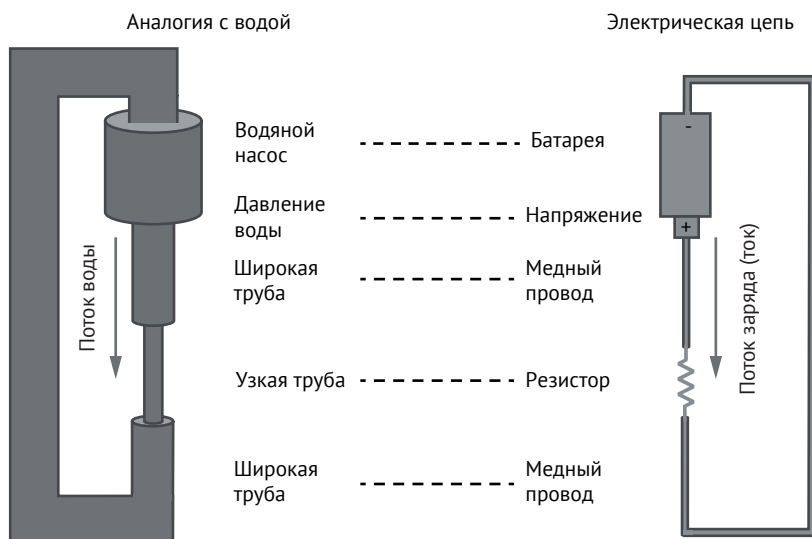


Рис. 3-2. Описание электрической цепи посредством аналогии с водой

Водяной насос перемещает воду по трубе, как батарея перемещает заряд по электрической цепи. Как течет вода, так течет и электрический заряд, мы называем это током. Давление воды влияет на интенсивность ее течения, и точно так же напряжение влияет на ток – большее напряжение означает больший ток. Вода свободно течет по широкой трубе, как и ток по медному проводу. Узкая труба ограничивает движение воды, так же как резистор ограничивает ток.

Теперь давайте применим полученные знания к электрической цепи, состоящей из батареи с проводником, присоединенным к ее клеммам. Потенциальная энергия хранится в батарее. Напряжение на клеммах батареи составляет определенное число вольт и представляет собой разность электрических потенциалов. Когда проводник присоединен к клеммам батареи, напряжение действует как давление, которое перемещает заряд по проводнику, создавая ток. Проводник имеет определенное сопротивление; низкое сопротивление приводит к большему току, а высокое сопротивление – к меньшему.

## Закон Ома

Особенности взаимосвязи между током, напряжением и сопротивлением определяются *законом Ома*, который гласит, что сила тока, текущего от одной точки к другой, равна напряжению на этих точках, деленному на сопротивление между ними. Или в виде уравнения:

$$I = V/R.$$

Допустим, у вас есть 9-вольтовая батарея с резистором 10 000 Ом, подключенным к ее клеммам. Закон Ома говорит нам, что ток, проходящий через этот резистор, будет равен  $9 \text{ В} / 10\,000 \text{ Ом} = 0,0009 \text{ ампер}$ , или 0,9 миллиампер (мА). Обратитесь к табл. 1-2, чтобы вспомнить, почему мы используем здесь приставку милли-. Обратите внимание, что в этой главе мы снова используем систему исчисления по основанию 10, так что вы можете забыть пока об основании 2 и некоторое время думать о числах как любой нормальный человек!

## АС и DC

Стоит сделать небольшую паузу, чтобы рассказать об АС и DC. Нет, не об австралийской рок-группе. АС означает *переменный ток* (*alternating current*) – электрический ток, который периодически меняет направление в противоположность DC – *постоянному току* (*direct current*), где ток течет только в одном направлении. Переменный ток используется для передачи электроэнергии от электростанций к домам и предприятиям. Приборы, лампы, телевизоры и другие устройства, которые подключаются непосредственно к розетке на стене без использования адаптера, работают от переменного тока. Более мелкая электроника, такая как ноутбуки и смартфоны, работает на постоянном токе. Когда вы заряжаете такое устройство, как смартфон, адаптер преобразует переменный ток из розетки в постоянный, необходимый вашему устройству. Батареи также обеспечивают постоянный ток. Термины АС и DC также применяются к напряжению (например, *источник постоянного напряжения*), в этом случае они означают «переменное напряжение» или «постоянное напряжение». Все схемы, с которыми мы имеем дело в этой книге, работают на постоянном токе, поэтому вам не нужно погружаться в детали, касающиеся переменного тока, кроме как знать разницу между этими понятиями.

## Схемы электрических цепей

Когда мы описываем электрические цепи, схема может быть очень полезным наглядным пособием. Электрические схемы рисуются с использованием стандартных символов для представления различных элементов цепи. Линии, соединяющие эти символы, представляют собой провода. Давайте рассмотрим, как на схемах изображаются некоторые распространенные элементы электрических цепей.

На рис. 3-3 показаны символы резистора и источника напряжения, например батареи. Знак «+» обозначает положительную клемму источника напряжения, а знак «-» – отрицательную. Другими словами, на клемме «+» напряжение положительное по отношению к клемме «-».

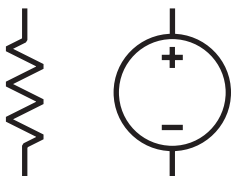


Рис. 3-3. Символы резистора (слева) и источника напряжения (справа)

Используя эти два символа, мы можем нарисовать схему, которая представляет собой наш пример электрической цепи из предыдущего раздела (9-вольтовая батарея с резистором 10 000 Ом, подключенным к ее клеммам). Эта схема показана на рис. 3-4. Обратите внимание на обозначение 10 кОм на резисторе, это сокращение для 10 000 Ом (другими словами, «к» — это кило-, или тысяча<sup>1</sup>). Учитывая наш предыдущий расчет по закону Ома, мы знаем, что через резистор протекает ток 0,9 мА, поэтому он также указан на схеме.

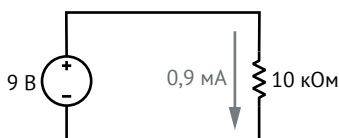


Рис. 3-4. 9-вольтовая батарея с резистором 10 000 Ом, подключенным к ее клеммам

Мы также можем представить ток в виде петли, как показано на рис. 3-5. Подобная иллюстрация помогает передать идею о том, что ток течет через всю цепь, а не только через резистор.

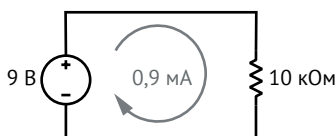


Рис. 3-5. Поток тока, представленный в виде петли

Говоря о петлях, сейчас самое время сделать шаг назад и вернуться к термину, который уже несколько раз упоминался, но я еще не дал ему определения. *Электрическая цепь* — это набор электрических компонентов, соединенных таким образом, что ток течет по контуру от источника питания через элементы цепи и обратно к источнику. Если отложить пока электричество, то общий термин «цепь» означает маршрут, который начинается и заканчивается в одном и том же месте. Это важное понятие, которое необходимо запомнить, потому что без цепи ток не будет протекать. Цепь с разрывом в контуре называется *разомкнутой*, и когда цепь разомкнута, ток не течет.

<sup>1</sup> См. табл. 1-2. — Прим. ред.

С другой стороны, *короткое замыкание* – это путь в цепи, который пропускает ток с небольшим или нулевым сопротивлением, обычно не-преднамеренно.

### УПРАЖНЕНИЕ 3-1: Применение закона Ома

Посмотрите на электрическую цепь на рис. 3-6. Каково здесь значение тока  $I$ ? Ответ приведен в приложении А.

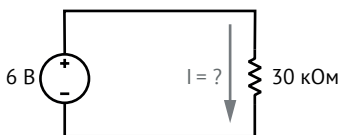


Рис. 3-6. Найдите ток, используя закон Ома

В контексте электрических цепей постоянного тока *земля* (нередко обозначаемая сокращенно как GND<sup>1</sup>) означает точку, относительно которой мы измеряем все другие напряжения в цепи. Другими словами, потенциал земли считается равным 0 В, и мы отсчитываем другие напряжения в цепи относительно нее. Как уже говорилось ранее, мы всегда измеряем напряжение между двумя точками, поэтому важна разность потенциалов, а не потенциал в одной точке. Назначив землю точкой отсчета в 0 В, мы упрощаем себе задачу об относительном напряжении в других точках цепи. В простых цепях постоянного тока, подобных тем, которые мы обсуждаем здесь, обычно землей считается отрицательная клемма батареи или другого источника питания.

Термин «земля» связан с тем, что некоторые электрические цепи физически соединены с землей. Они буквально соединены с землей, и это соединение служит точкой отсчета в 0 В. Портативные устройства или устройства с питанием от батареи обычно не имеют физического соединения с землей, но мы все равно называем в таких электрических цепях назначенную точку отсчета в 0 В «землей».

Иногда инженеры не изображают электрические схемы в виде контура, а вместо этого специально обозначают соединения земли и источника напряжения символами, показанными на рис. 3-7. Это делает электрические схемы более компактными, но не меняет физического устройства цепи: ток по-прежнему течет от положительной клеммы источника питания к отрицательной.

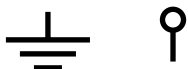


Рис. 3-7. Символы для земли (слева) и источника напряжения относительно земли (справа)

<sup>1</sup> От англ. *ground* – земля, почва. – Прим. ред.

В качестве примера на рис. 3-8 слева показана схема, которую мы обсуждали ранее, а справа та же схема представлена с использованием символов земли и напряжения источника питания, введенных на рис. 3-7.

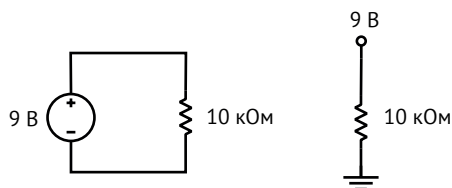


Рис. 3-8. Эти две схемы эквивалентны

Две схемы на рис. 3-8 функционально эквивалентны, различаются только представлением на схемах.

## Закон напряжения Кирхгофа<sup>1</sup>

Еще один принцип, объясняющий поведение электрических цепей, – это закон *напряжения Кирхгофа*, который говорит нам, что сумма напряжений в цепи равна нулю. Это означает, что если источник напряжения подает на цепь напряжение 9 В, то различные элементы этой цепи должны в совокупности «использовать» 9 В. Каждый элемент в контуре электрической цепи уменьшает электрический потенциал. Когда это происходит, мы говорим, что напряжение *падает* на каждом элементе. В качестве примера рассмотрим электрическую цепь на рис. 3-9.

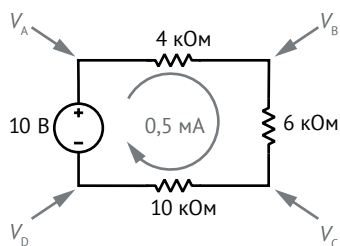


Рис. 3-9. Схема, иллюстрирующая закон напряжения Кирхгофа

На рис. 3-9 показан источник питания напряжением 10 В, подключенный к трем резисторам в электрической цепи. Когда резисторы соединены вдоль одного пути (*последовательно*), общее сопротивление просто равно сумме значений отдельных сопротивлений. В данном случае это означает, что общее сопротивление равно  $4\text{ кОм} + 6\text{ кОм} + 10\text{ кОм} = 20\text{ кОм}$ . Используя закон Ома, мы можем рассчитать ток, проходящий через эту цепь, как  $10\text{ В} / 20\text{ кОм} = 0,5\text{ мА}$ . В электриче-

<sup>1</sup> В русскоязычной литературе известен, как 2-й закон Кирхгофа. – *Прим. ред.*

ской цепи есть четыре точки, в которых мы можем измерить напряжение, обозначенные как  $V_A$ ,  $V_B$ ,  $V_C$  и  $V_D$ . Мы определим напряжение в каждой из этих точек относительно отрицательной клеммы нашего источника питания.

Начнем с самого простого:  $V_D$  напрямую подключена к отрицательной клемме источника питания, поэтому  $V_D = 0$  В, т. е. это – земля. Аналогично  $V_A$  подключена к положительной клемме источника питания, поэтому  $V_A = 10$  В. Теперь мы знаем из закона напряжения Кирхгофа, что на каждом из резисторов напряжение падает, поэтому  $V_B$  должно быть меньше 10 В, а  $V_C$  должно быть меньше  $V_B$ .

На сколько же упадет напряжение на резисторе 4 кОм? Закон Ома гласит, что  $V = I \times R$ , поэтому падение напряжения составляет  $0,5 \text{ мА} \times 4 \text{ кОм} = 2$  В. Это означает, что  $V_B$  будет на 2 В меньше, чем  $V_A$ . Итак,  $V_B = 10 - 2 = 8$  В.

Таким же образом падение напряжения на резисторе 6 кОм будет равно 3 В. Следовательно,  $V_C = 8 - 3 = 5$  В. Даже не выполняя математических расчетов, мы знаем из закона напряжения Кирхгофа, что 5 В должно падать на резисторе 10 кОм, поскольку это последний элемент цепи перед отрицательной клеммой с 0 В. На рис. 3-10 наша схема обновлена с учетом напряжений и падений напряжения.

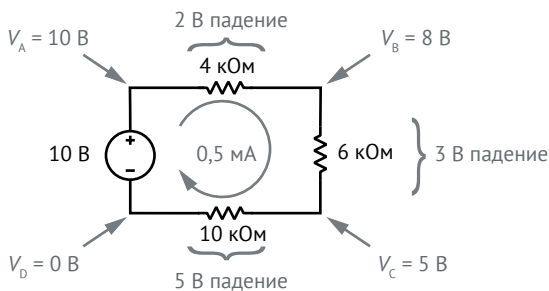


Рис. 3-10. Падение напряжения в простой электрической цепи

Подведем итоги для данного примера: источник напряжения выдает 10 В, которые мы считаем положительным напряжением. Каждый резистор вызывает падение напряжения, и мы считаем это падение отрицательным. Если сложить положительное напряжение источника питания с отрицательным падением напряжения, то мы получим:  $10 \text{ В} - 2 \text{ В} - 3 \text{ В} - 5 \text{ В} = 0 \text{ В}$ . Сумма напряжений в электрической цепи равна 0, что соответствует закону напряжения Кирхгофа.

Вы можете задаться вопросом, работает ли это только с определенными значениями резисторов? В конце концов в приведенном примере математические вычисления получились очень уж гладко, возможно, даже слишком! В следующем упражнении мы поменяем один из резисторов в схеме примера с 4 на 24 кОм, и вы сможете увидеть, что закон напряжения Кирхгофа по-прежнему выполняется.

### УПРАЖНЕНИЕ 3-2: Определите падение напряжения

Дана электрическая цепь, как на рис. 3-11. Каково будет значение тока  $I$ ? Каково будет падение напряжения на каждом резисторе? Найдите обозначенные напряжения:  $V_A$ ,  $V_B$ ,  $V_C$  и  $V_D$ , каждое из которых измеряется относительно отрицательной клеммы источника питания. Ответ в приложении А.

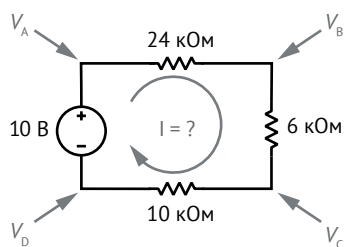


Рис. 3-11. Другая схема, иллюстрирующая закон напряжения Кирхгофа

## Электрические цепи в реальном мире

Давайте посмотрим, как наша простая электрическая цепь с 9 В / 10 кОм (с рис. 3-4) может быть построена в реальном мире. На фотографии, представленной на рис. 3-12, я прикрепил резистор 10 кОм к 9-вольтовой батарее с помощью зажимов типа «крокодил».

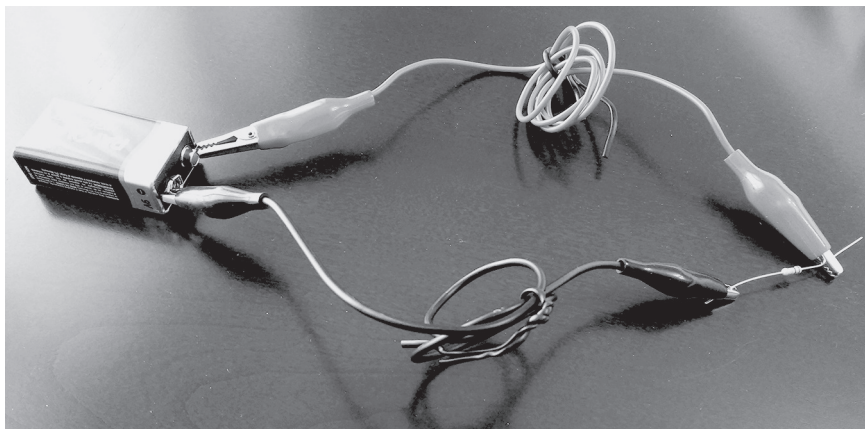


Рис. 3-12. Резистор 10 кОм, подключенный к 9-вольтовой батарее с помощью зажимов типа «крокодил»

Такой подход работает, но есть и более эффективные способы создания электрических цепей. *Макетная плата* (англ. *breadboard* – хлебная доска) – это основа для создания прототипов электрических цепей. Исторически для этих целей использовались доски, подобные тем, что используются для выпечки хлеба, но, к сожалению, современные макетные платы не имеют ничего общего с хлебом! Макетная плата (рис. 3-13) позволяет легко соединять различные электрические компоненты.

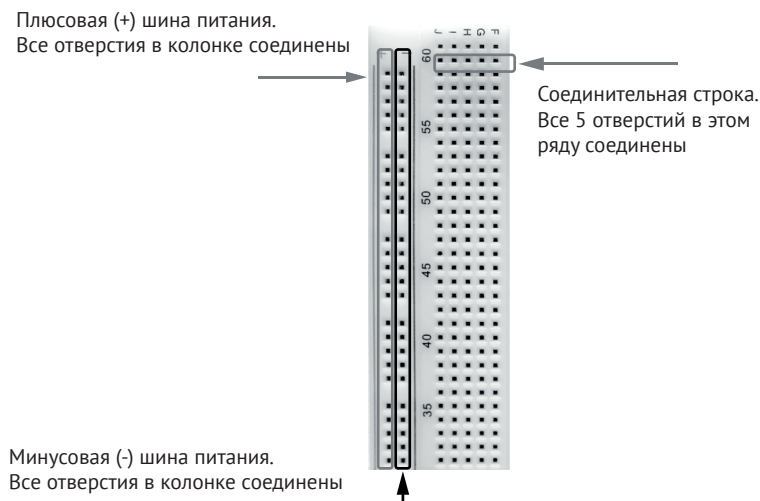


Рис. 3-13. Участок макетной платы

Как показано на рис. 3-13, по краям макетной платы расположены длинные колонки (шины), обычно обозначенные + и -. Эти шины часто имеют цветовую маркировку: красный для плюса, синий для минуса. Все отверстия вдоль таких крайних колонок электрически соединены, и эти шины предназначены для подачи питания на схему, поэтому обычно к ним подключается батарея или другой источник питания. Аналогично ряды, состоящие обычно из пяти отверстий (на рис. 3-13 они расположены справа от крайних колонок), также соединены. Два компонента можно соединить, просто поместив вывод каждого компонента в один и тот же ряд. Не требуется ни пайка, ни зажимы, ни электроизоляционная лента!

На рис. 3-14 представлена фотография схемы с рис. 3-4, собранной на макетной плате.



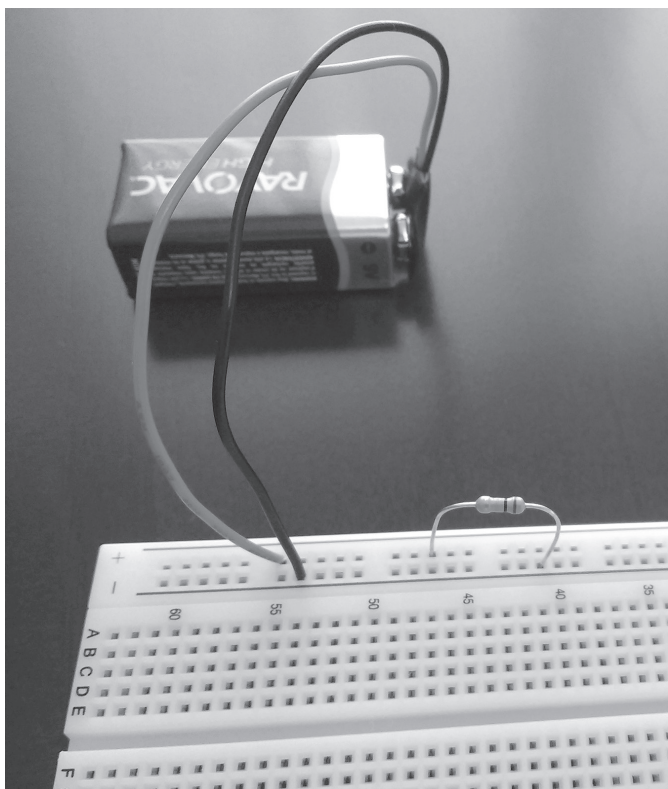


Рис. 3-14. Простая схема, построенная на макетной плате

Как видите, это более аккуратный и простой способ соединения электрических компонентов. Я немного оптимизировал схему, вставив концы резистора в шины питания, чтобы подключить его непосредственно к батарее.

#### ПРИМЕЧАНИЕ

*Чтобы выполнить первый проект из этой книги, обратитесь к проекту № 1 на стр. 72. В предыдущих упражнениях вас просили решить задачу теоретически, в то время как в проектах вам нужно будет сделать немного больше, и для этого понадобится приобрести некоторое оборудование. Конечно, приобретение потребует определенных усилий и затрат, но я считаю, что, как говорится, «пощупать руками» — это лучший способ действительно понять принципы, рассматриваемые в этой книге. Перейдите в конец этой главы, чтобы найти раздел проектов. Там вы сможете собрать свою собственную электрическую схему!*

## Светоизлучающие диоды

Простые электрические цепи, которые мы обсуждали до сих пор, иллюстрируют основы их функционирования, но они не делают ничего интересного. Я обнаружил, что самый простой способ превратить скучную схему в интересную – это добавить в нее *светоизлучающий диод (LED)*. Фотография типичного светодиода показана на рис. 3-15.

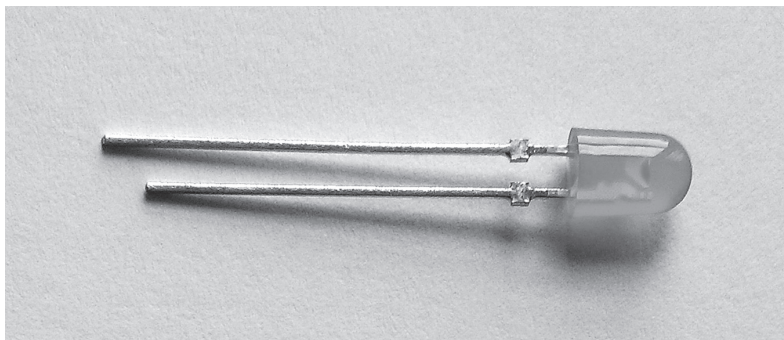


Рис. 3-15. Светоизлучающий диод

Давайте рассмотрим основы работы со светодиодами, а затем добавим один из них в схему. Часть названия «светоизлучающий» говорит сама за себя – это элемент схемы, который излучает свет. Точнее, это диод, излучающий свет. *Диод* – это электронный компонент, который позволяет току проходить через него только в одном направлении. В отличие от резистора, который пропускает ток в любом направлении, диод имеет очень низкое сопротивление в одном направлении (пропускает ток) и очень высокое сопротивление в другом (препятствует прохождению тока). Светодиод – это особый вид диода, который еще и светит, когда через него течет ток. Светодиоды бывают разных цветов. Условное обозначение светодиода показано на рис. 3-16.

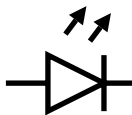


Рис. 3-16. Условное обозначение светодиода

Чтобы заставить светодиод излучать свет, необходимо обеспечить протекание через него определенной величины тока.

Стандартный красный светодиод рассчитан на максимальный ток около 25 мА, и мы не хотим превышать этот максимальный ток, так как это может привести к повреждению светодиода. Давайте определим

20 мА в качестве величины тока, который мы хотим пропустить через наш светодиод. Меньшее значение тока тоже сработает, но светодиод будет не таким ярким<sup>1</sup>.

Как же обеспечить нужную величину тока для светодиода? Нам просто нужно подобрать подходящий резистор, чтобы ограничить ток в нашей цепи. Но прежде чем мы сможем это сделать, необходимо узнать еще об одном свойстве светодиода – *прямом напряжении*, которое описывает, насколько напряжение падает на светодиоде, когда через него течет ток. Типичный красный светодиод имеет прямое напряжение около 2 В. Прямое напряжение часто обозначается как  $V_f$ .

Схема нашей электрической цепи с батареей, светодиодом и резистором для ограничения тока показана на рис. 3-17. На схеме также показан желаемый ток 20 мА.

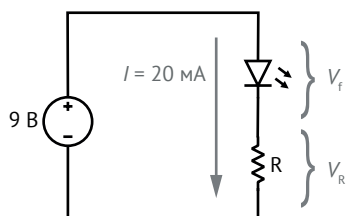


Рис. 3-17. Базовая схема со светодиодом

На рис. 3-17 у нас имеется 9-вольтовая батарея, светодиод с прямым напряжением  $V_f$  и резистор со значением сопротивления  $R$ . Падение напряжения на резисторе равно  $V_R$ . Имейте в виду, что падение напряжения на резисторе зависит от величины тока, протекающего через него, в отличие от светодиода, где падение напряжения определяется его характеристикой прямого напряжения<sup>2</sup>. В нашей предыдущей схеме, в которой были только батарея и резистор (рис. 3-4), все 9 В падали на резисторе. Теперь, когда у нас есть два элемента в цепи, подключенных к нашей батарее, закон Кирхгофа говорит, что часть напряжения будет падать на светодиоде, а остальная часть – на резисторе. Напоминаем, что вы можете думать о батарее как об источнике напряжения, а о других элементах как о потребителях напряжения. Если мы применим это к нашей схеме (рис. 3-17), то  $V_f + V_R = 9 \text{ В}$ .

<sup>1</sup> Рекомендуемое рабочее значение тока через стандартный сигнальный светодиод (не повышенной яркости) – 5–7 мА. Яркость при этом достаточная (еще большая яркость будет только раздражать), а общее потребление схемы падает. 20 мА следует применять только в иллюстративных целях, как в этом примере. – *Прим. ред.*

<sup>2</sup> Это не совсем так – прямое падение напряжения на светодиоде также зависит от тока, но зависимость эта нелинейна и должна приниматься во внимание только при низких напряжениях источника питания (3 В и менее). Во всех остальных случаях для практических расчетов можно принимать величину прямого напряжения любых стандартных сигнальных светодиодов (т. е. не осветительных и не повышенной яркости) от 1,5 до 2,5 В (большая величина у синих, меньшая у красных). – *Прим. ред.*

Предполагая, что мы используем светодиод со стандартным прямым напряжением 2 В, получаем:  $V_R = 9 \text{ В} - 2 \text{ В} = 7 \text{ В}$ . Давайте обновим нашу схему этими значениями напряжения, как показано на рис. 3-18.

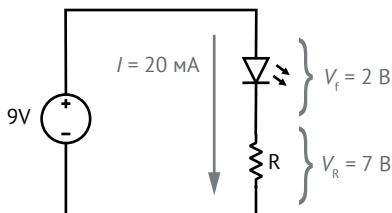


Рис. 3-18. Базовая схема со светодиодом, где показаны падения напряжения

Остается только одно неизвестное,  $R$  – сопротивление резистора. Мы можем рассчитать его по закону Ома:  $I = V / R$  или  $R = V / I$ . Таким образом, получаем  $R = 7 \text{ В} / 20 \text{ mA} = 350 \text{ Ом}$ . И с этим мы получаем последний элемент для нашей задачи о том, как обеспечить нужное количество тока, протекающего через наш светодиод. Нам нужен резистор около 350 Ом, подключенный к нашей батарее и светодиоду.

#### ПРИМЕЧАНИЕ

*Посмотрите проект № 2 на стр. 78, где вы сможете сами собрать светодиодную цепь и увидеть, как он загорается!*

## Выводы

В этой главе мы рассмотрели электрические цепи – физическую основу современных вычислительных устройств. Вы узнали об электрических понятиях, таких как заряд, ток, напряжение и сопротивление. Вы познакомились с двумя законами, которые управляют поведением электрических цепей – законом Ома и законом напряжения Кирхгофа. Вы узнали о схемах электрических цепей и о том, как построить свои собственные цепи. Понимание основ электрических цепей поможет вам получить фундаментальное представление о том, как работают компьютеры. В следующей главе представлены цифровые схемы, объединяющие концепции двоичной логики и электрических схем.

## ПРОЕКТ № 1: Построение электрической цепи и измерения в ней

Теперь вы знаете достаточно, чтобы построить свою собственную электрическую цепь. Нет лучшего способа научиться, чем пробовать все самому! Чтобы начать работу, вам понадобится некоторое оборудование, которое можно приобрести в интернете или в местном магазине электроники, если вам повезло иметь его поблизости. Смотрите раздел «Покупка электронных компонентов» на стр. 406, который поможет в приобретении этих деталей. Вот что вам понадобится для этого и следующего проектов:

- макетная плата (либо 400-точечная, либо 830-точечная модель);
- резисторы (набор резисторов). В этом проекте вы будете использовать резистор 10 кОм. Убедитесь, что используете резистор 10 кОм, а не 10 Ом. Использование резистора со слишком низким значением приведет к чрезмерному току и может стать причиной перегрева схемы!
- цифровой мультиметр (понадобится вам для проверки напряжения, тока и сопротивления вашей цепи);
- 9-вольтовая батарея<sup>1</sup>;
- разъем для 9-вольтовой батареи (это значительно облегчает подключение батареи);
- по крайней мере один красный LED (светоизлучающий диод) размером 5 или 3 мм;
- необязательно: инструмент для зачистки проводов (может понадобиться, чтобы снять пластик с концов проводов и обнажить медь);
- необязательно: зажимы типа «крокодил» (облегчают подключение батареи к макетной плате или мультиметра к цепи);
- необязательно: провода-перемычки для макетной платы (подсоедините их к концам проводов с зажимами для 9-вольтовой батареи, чтобы ее было легче подключить к макетной плате).

Даже при низких уровнях напряжения, с которыми мы работаем, существует небольшой риск того, что какой-либо компонент схемы нагреется. Учитывая это, я рекомендую вам отключать источник питания, в данном случае батарею, во время подсоединения компонентов и подключать питание только на последнем этапе сборки схемы<sup>2</sup>.

После того как у вас есть все компоненты, соедините их вместе.

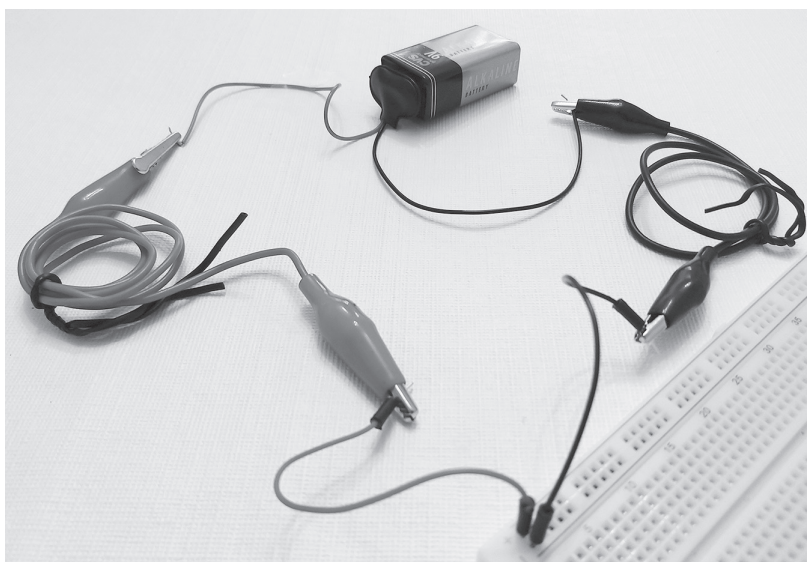
1. Подключите любой конец резистора 10 кОм к плюсовой шине питания.
2. Подключите другой конец резистора к минусовой шине питания.

<sup>1</sup> В отечественных магазинах для батареек, которую имеет в виду автор, часто используется название типа «Крона». – *Прим. ред.*

<sup>2</sup> Существует большой риск неправильной сборки схемы, потому любые схемы *всегда* собираются при отключенном источнике. – *Прим. ред.*

3. Подключите красный/плюсовой провод разъема батареи к плюсовой шине питания на макетной плате.
4. Подключите черный/минусовой провод разъема батареи к минусовой шине питания на макетной плате.
5. Подключите разъем батареи к клеммам 9-вольтовой батареи.

Провода разъема 9-вольтовой батареи обычно гибкие, что затрудняет их вставку в макетную плату. Если вы столкнулись с этой проблемой, попробуйте подключить один конец провода-перемычки к гибкому проводу разъема батареи, а другой конец перемычки – к макетной плате. Вы можете соединить два провода электроизоляционной лентой или зажимами «крокодил» (см. рис. 3-19) или даже спаять их вместе, если умеете пользоваться паяльником. Если вы попытаетесь использовать любой из этих методов, следите за тем, чтобы металлические части плюсового и минусового проводов не касались друг друга. Случайное их соединение приведет к короткому замыканию батареи, в результате чего провода сильно нагреются и быстро разрядят батарею.



*Рис. 3-19. Использование зажимов «крокодил» и проводов-перемычек для удобного подключения гибких проводов разъема 9-вольтовой батареи*

Вам может быть интересно, как определить значение резистора. Резисторы имеют цветовую маркировку: полосы отображают множители, а цвета – числовые значения. В интернете можно найти множество бесплатных калькуляторов и таблиц цветовых кодов резисторов, поэтому здесь я не буду вдаваться в подробности. Для резистора 10 кОм ищите резистор с коричневой, черной и оранжевой полосами в таком порядке. Четвертая полоса обычно золотая или серебряная, что указывает на допуск, т. е. на допустимое отклонение от заявленного значения.



Теперь вы собрали схему, но как определить, что что-то происходит? К сожалению, эта схема никак не показывает визуально, что она работает, поэтому пришло время достать мультиметр и измерить ее различные характеристики. Для использования мультиметра вам понадобятся два прилагаемых к мультиметру тестовых провода-щупа. Если тестовые щупы мультиметра не закреплены в корпусе, то он, скорее всего, будет иметь две или три входные клеммы для подключения щупов, как показано на рис. 3-20.

Как показано на рис. 3-20, подключите один провод к входной клемме с надписью «COM» (что означает «common» – «общий»). Обычно мы подключаем черный провод к клемме COM. Если ваш измерительный прибор имеет только две входные клеммы, просто подключите второй провод, обычно окрашенный в красный цвет, ко второй клемме. Измерительные приборы с тремя клеммами обычно имеют вход COM, универсальный вход для измерения напряжений, сопротивлений, небольшого тока и т. д., и отдельный вход для измерения больших значений величины тока. В этом случае вам нужно подключить второй провод к универсальному входу; он обычно маркирован различными типами измерений, которые он поддерживает, например «V Ом mA». Обычно входная клемма большого тока маркируется как «A», «10 A», «10 A max» или что-то в этом роде. Еще раз: это не та клемма, которую здесь следует использовать. Некоторые мультиметры имеют четыре клеммы; в этом случае для измерения слабого тока необходимо использовать другую входную клемму относительно измерения напряжения и сопротивления. Независимо от того, какой у вас прибор, при возникновении вопросов изучите инструкцию по эксплуатации.

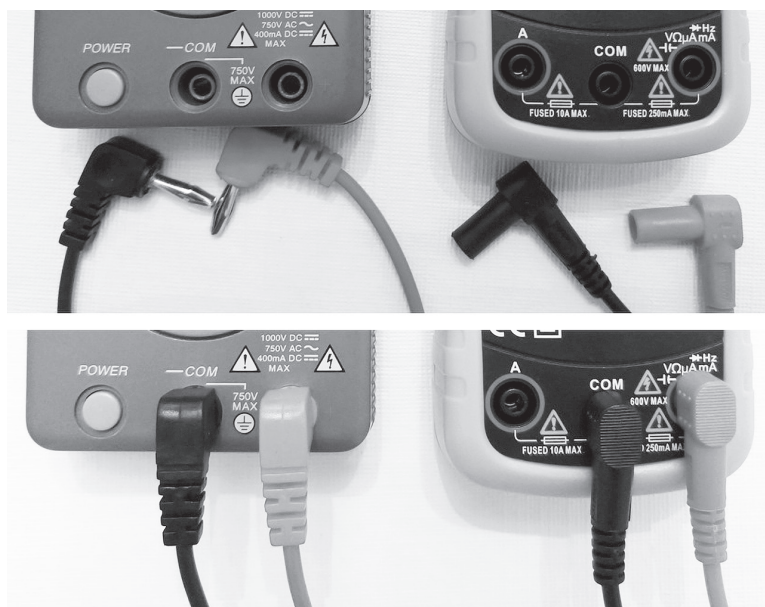


Рис. 3-20. Подключение тестовых проводов к мультиметру. Прибор с двумя клеммами (слева) и прибор с тремя клеммами (справа)

В мультиметре есть возможность выбрать, что вы измеряете: напряжение, ток или сопротивление. Начнем с напряжения. Настройте мультиметр на измерение напряжения постоянного тока. Оно может быть обозначено символом «V» и буквами «DC» рядом с ним, или вы можете увидеть прерывистую линию рядом с «V», что указывает на постоянный ток (в отличие от волнистой линии, которая указывает на переменный ток).

После того как мультиметр настроен на измерение постоянного напряжения, измерьте напряжение на резисторе, прикоснувшись одним щупом измерительного провода к левой стороне резистора (металл к металлу), а другим щупом – к правой. Помните, что напряжение всегда измеряется между двумя точками, поэтому логично, что измерять нужно с обеих сторон резистора. Поскольку мы питаем эту схему от 9-вольтовой батарейки, а единственным элементом схемы является резистор, ожидайте увидеть результат примерно в 9 В. Во время измерения вы можете заметить, что значение, показываемое измерительным прибором, немного меняется (обычно это только наименьшая по значению цифра справа). Это результат работы цифровых измерительных приборов и не означает, что ваш прибор или схема неисправны.

Теперь попробуйте поменять местами щупы, поместив левый щуп на правую сторону резистора и наоборот. Вы должны увидеть, что значение напряжения стало отрицательным (или положительным, если до этого оно было отрицательным). Это происходит потому, что измерительный прибор измеряет разность потенциалов и рассматривает щуп, подключенный к COM, как 0 В; показанное значение измерения – это разность напряжений относительно другого щупа. На рис. 3-21 показано измерение напряжения в моей цепи, равное 9,56 В. Это похоже на правду, поскольку напряжение новой батареи часто немного выше, чем заявлено.

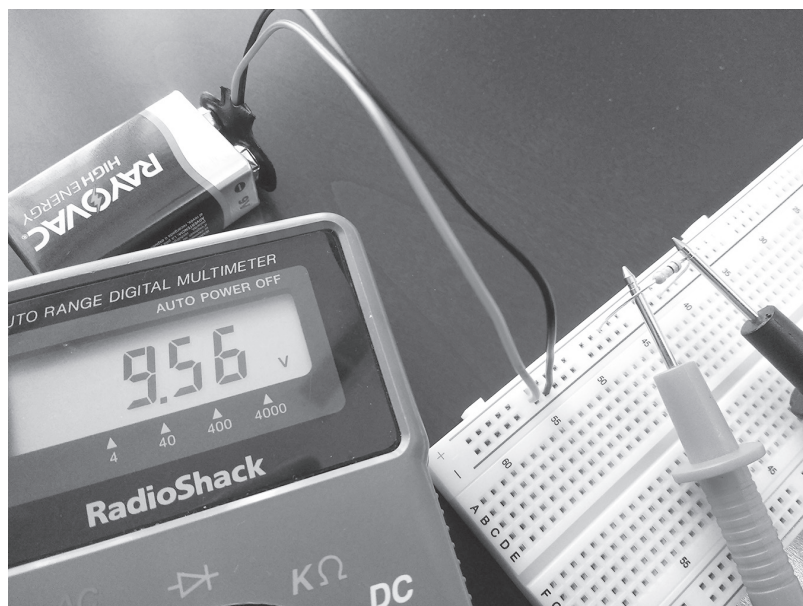


Рис. 3-21. Измерение напряжения



Далее давайте измерим сопротивление резистора. Сначала отключите мультиметр от электрической цепи, чтобы избежать случайного повреждения при изменении настройки. Затем установите мультиметр на настройку сопротивления, которая может быть обозначена как Ом. Если ничего не подключено, ваш мультиметр, скорее всего, покажет слева 1 или OL. Это означает, что значение сопротивления слишком велико для отображения – сопротивление воздуха очень высокое! Соедините щупы мультиметра вместе, и на дисплее должен появиться ноль. Чтобы измерить сопротивление резистора, его нужно отсоединить от схемы, но можно оставить его прикрепленным к макетной плате, если это удобно. Для получения точных показаний избегайте прикосновения пальцами к компонентам схемы и металлической части щупов во время измерения, так как сопротивление вашего тела может изменить значение. Мое измерение сопротивления показано на рис. 3-22; в моем случае значение составило 9,88 кОм. Резистор, который я использовал, имеет полосы коричневого, черного и оранжевого цветов (обозначающие 10 000 Ом), за которыми следует золотая полоса (обозначающая 5%-ный допуск), так что полученное измерение выглядит неплохо: 9,88 кОм находится в пределах 5 % от 10 кОм.

На данном этапе вы знаете измеренные значения напряжения и сопротивления, поэтому можете рассчитать ожидаемый ток, используя закон Ома, – ток равен напряжению, деленному на сопротивление. Для моей цепи это значение будет  $9,56 \text{ В} / 9,880 \text{ кОм} = 0,97 \text{ мА}$ . Прежде чем измерять ток в цепи, выполните такой же расчет, используя полученные значения напряжения и сопротивления, чтобы понять, какой ток вы должны ожидать в электрической цепи.

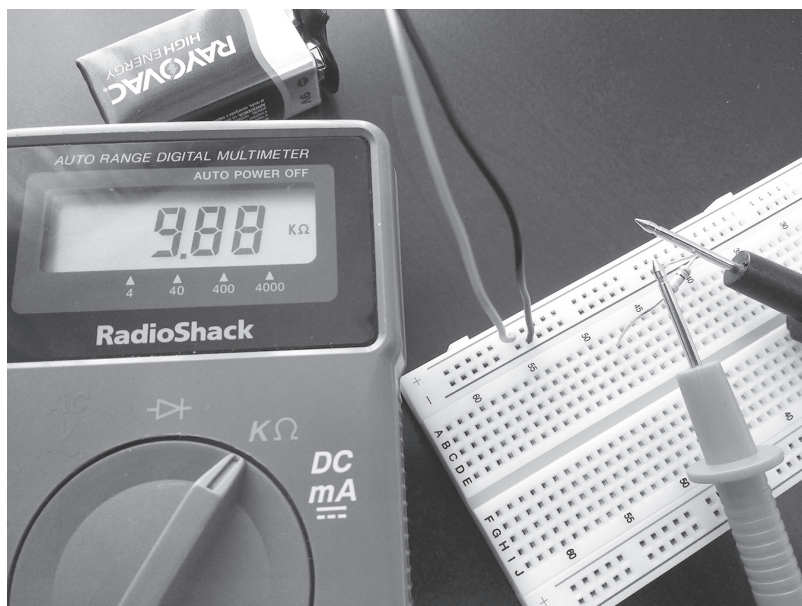


Рис. 3-22. Измерение сопротивления

Теперь измерьте ток, протекающий через вашу цепь, и посмотрите, насколько точно он соответствует вашим расчетам. Измерение тока немного отличается от измерения напряжения или сопротивления. Чтобы мультиметр мог измерять ток, ток должен через него протекать. Другими словами, измерительный прибор должен стать частью цепи, как показано на рис. 3-23.

Не забудьте отключить мультиметр, прежде чем настроить его на измерение постоянного тока. Обозначения постоянного тока могут быть представлены буквами А или mA с DC или прерывистой линией. Некоторые измерительные приборы имеют одну настройку для измерения постоянного и переменного тока, в этом случае символ может показывать как прямую, так и волнистую линию. После того как мультиметр настроен правильно, подключите его к цепи, как показано на рис. 3-23.

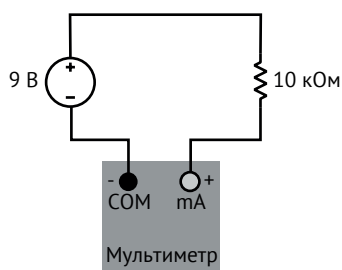


Рис. 3-23. Как подключить мультиметр, чтобы измерить ток

Надеюсь, ваши измерения оказались близки к расчетному значению. Как показано на рис. 3-24, мое измерение тока составило 0,97 мА, что совпадает с моим предыдущим расчетом.



Рис. 3-24. Измерение тока. Правую часть резистора и черный провод от батареи не нужно подключать к макетной плате

Если до сих пор у вас все получилось, то вас можно поздравить! Вы только что собрали электрическую цепь и сделали замеры или, по крайней мере, прочитали о том, как я это сделал. И все же, возможно, это не очень-то вас впечатлило. Я понимаю. Честно говоря, помимо отличной образовательной ценности, эта электрическая цепь несколько скучновата и бесполезна! В следующем проекте давайте сделаем что-то более интересное.

## ПРОЕКТ № 2: Построение простой схемы со светодиодом

Теперь ваша очередь собрать электрическую цепь со светодиодом и увидеть, как он загорится! Если вы делали предыдущий проект, вы можете оставить 9-вольтовую батарею, подключенную к макетной плате, но давайте заменим резистор 10 кОм на резистор с более подходящим для этой схемы значением сопротивления (номиналом). Из расчетов, проведенных ранее в этой главе в разделе «Светоизлучающие диоды» на стр. 69, мы выяснили, что для обеспечения тока 20 мА нам нужен резистор 350 Ом. Если я загляну в свою большую кучу резисторов, то вряд ли найду резистор на 350 Ом, да и вы тоже. Ближайший по значению резистор, который вы, скорее всего, найдете, будет 330 Ом<sup>1</sup>.

Мы можем использовать более одного резистора, чтобы получить точно 350 Ом, но нужно ли это? Достаточно ли будет 330 Ом? Давайте выясним.

Если мы оставим 9-вольтовую батарею и будем продолжать считать, что на резисторе нужно снять 7 В, то по закону Ома ток, проходящий через нашу цепь, будет равен  $I = V / R = 7 \text{ В} / 330 \text{ Ом} = 21,2 \text{ мА}$ . И это будет как раз то, что нужно, так как типичный красный светодиод рассчитан на максимальный ток около 25 мА.

Стоит отметить, что любые приобретенные вами светодиоды могут иметь характеристики, отличные от описанных мной. Если ваш светодиод поставляется с техническим паспортом или если его характеристики указаны в интернете, проверьте их и сделайте собственные расчеты. Каков фактический максимальный или желаемый ток для вашего конкретного светодиода? Каково фактическое прямое напряжение для вашего конкретного светодиода? Обратите внимание, что это обычно зависит от цвета светодиода и протекающего тока.

Перед созданием схемы необходимо знать еще одну вещь о светодиодах. В отличие от резистора, где ток может течь в любом направлении, светодиод предназначен для пропускания тока только в одном направлении, поэтому вам нужно знать, с какого конца будет течь ток. Положительная сторона (анод) обычно имеет более длинный вывод, а отрицательная сто-

<sup>1</sup> В ряду резисторов с допуском 5 % найдется также номинал 360 Ом, что ближе к заданной величине. – *Прим. ред.*

рона (катод) – более короткий, как показано на рис. 3-25<sup>1</sup>. Ток течет от положительного конца к отрицательному.

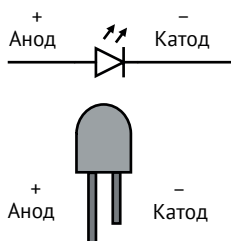


Рис. 3-25. Определение анода и катода на светодиоде

После того как вы разобрались с числами и знаете, какое значение резистора вы хотите использовать, подключите все, как показано на схеме на рис. 3-26.

1. Временно отсоедините 9-вольтовую батарею от макетной платы.
2. Подключите более длинный вывод светодиода к плюсовой шине питания.
3. Подключите более короткий вывод светодиода к любой неиспользуемой строке на макетной плате.
4. Подключите один конец резистора к той же строке, что и короткий вывод светодиода.
5. Подключите другой конец резистора к минусовой шине питания.
6. Снова подключите 9-вольтовую батарею к макетной плате.

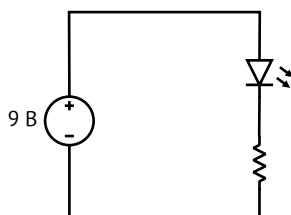


Рис. 3-26. Простая LED-цепь

На рис. 3-27 представлена фотография моего светодиода, подключенного в соответствии с описанием. Посмотрите на свечение! Надеюсь, ваш светодиод также загорелся после того, как вы полностью собрали электрическую цепь.

<sup>1</sup> У круглых светодиодов диаметром 5 мм обычно также бывает сточен бортик со стороны анода (положительного вывода). – Прим. ред.

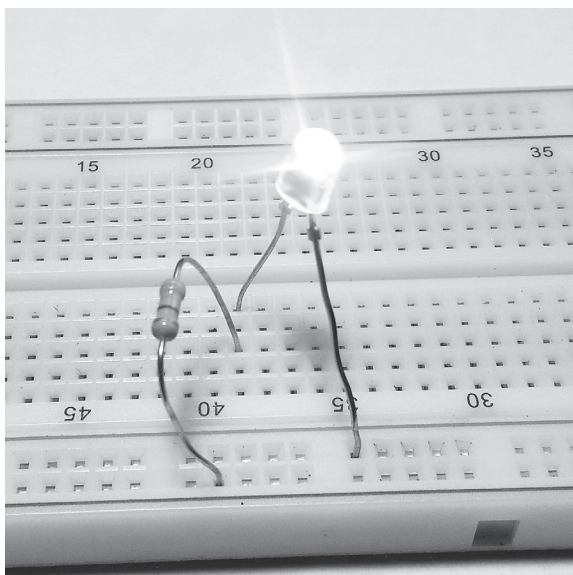
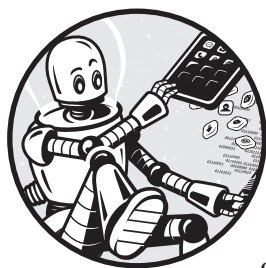


Рис. 3-27. Светящийся светодиод; подключенная батарея не показана

# 4

## ЦИФРОВЫЕ СХЕМЫ



До сих пор мы рассмотрели два аспекта вычислительной техники. Во-первых, компьютеры работают в двоичной системе из нулей и единиц. Во-вторых, компьютеры – это электронные устройства, построенные на основе электрических цепей. Теперь пришло время объединить эти два аспекта вычислительной техники. В этой главе мы определим, что значит для электрической цепи быть цифровой, разберем подходы к реализации цифровых схем, включая роль, которую играют транзисторы. Наконец, мы рассмотрим логические вентили и интегральные схемы – строительные блоки более сложных компонентов, о которых поговорим в следующих главах.

### Что такое цифровая схема?

Вы, наверное, заметили, что схемы, которые мы создавали в предыдущей главе, не были цифровыми – они были аналоговыми. В этих схемах напряжение, ток и сопротивление могли изменяться в широком диапазоне значений. Это неудивительно: наш мир по своей природе аналоговый! Однако компьютеры работают в цифровой реальности, и, чтобы понять компьютеры, нужно разобраться с цифровыми схемами. Если мы хотим, чтобы наши схемы были цифровыми, следует

сначала определить, что это значит в контексте электроники, и затем мы сможем использовать аналоговые компоненты для построения цифровых схем.

*Цифровые схемы* имеют дело с сигналами, число возможных состояний которых ограничено. В этой книге рассматриваются двоичные цифровые схемы, поэтому 0 и 1 – это единственные два состояния, которые будут приниматься во внимание. Для обозначения 0 или 1 в цифровой схеме обычно используется напряжение, где 0 – это низкое напряжение, а 1 – высокое напряжение. Обычно низкое напряжение означает 0 В, а высокое, как правило, равно 5 В, 3,3 В или 1,8 В, в зависимости от конструкции схемы. В действительности цифровым схемам не требуется точное значение напряжения для установления высокого или низкого уровня. Вместо этого обычно диапазон напряжений отмечается как высокий или низкий. Например, в условной 5-вольтовой цифровой схеме входное напряжение значением между 2 и 5 В относится к высокому, а значением между 0 и 0,8 В – к низкому. Любой другой уровень напряжения приводит к неопределенному поведению схемы, и его следует избегать.

Обычно земля является самым низким напряжением в цифровой схеме, а все остальные напряжения в этой схеме положительны по сравнению с землей. Если цифровая схема питается от батареи, мы считаем отрицательную клемму батареи землей. То же самое относится и к другим видам источников питания постоянного тока; отрицательная клемма или провод считается землей.

Когда речь идет о состояниях 0 и 1 в цифровых схемах, используется множество терминов и сокращений, все из которых означают одно и то же. Эти термины часто используются как взаимозаменяемые. Вот некоторые общие термины для обозначения низкого и высокого уровней:

**Низкий уровень (низкое напряжение):** Низкое, LO, выкл, земля, GND, ложь, ноль, 0;

**Высокий уровень (высокое напряжение):** Высокое, HI, вкл, V+, истина, один, 1<sup>1</sup>.

## Логика с помощью механических выключателей

Теперь, когда мы установили, что высокое и низкое напряжение представляют 1 и 0 в нашей цифровой схеме, давайте рассмотрим, как мы можем построить такую цифровую схему. Нам нужна схема, в которой входное и выходное напряжения всегда имеют заданное высокое или низкое значение или, по крайней мере, находятся в допустимом диапазоне. Чтобы помочь нам это сделать, давайте применим очень простой и хорошо знакомый элемент – механический выключатель. *Выключатель*

---

<sup>1</sup> Соответствующие англоязычные термины:

- для низкого уровня: Low, LO, off, ground, GND, false, zero, 0;
- для высокого уровня: High, HI, on, V+, true, one, 1. – *Прим. ред.*



полезен тем, что он цифровой по своей природе. Он включен или выключен. Когда выключатель включен, он действует как простой медный провод, по которому свободно течет ток. Когда выключатель выключен, он действует как разомкнутая цепь, и ток не может течь. Мы изображаем выключатель с помощью символа, показанного на рис. 4-1.

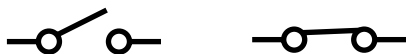


Рис. 4-1. Условное обозначение выключателя или кнопки на электрической схеме — разомкнут/выключен (слева), замкнут/включен (справа)

Символ выключателя передает идею о том, что в выключенном положении выключатель представляет собой разомкнутую цепь, а во включенном положении — замкнутую цепь. Символ выключателя и сам выключатель можно представить как ворота в заборе, которые либо открыты, либо закрыты. Ток течет через выключатель, когда он закрыт. В реальном мире выключатели бывают разных форм и размеров, как показано на рис. 4-2.

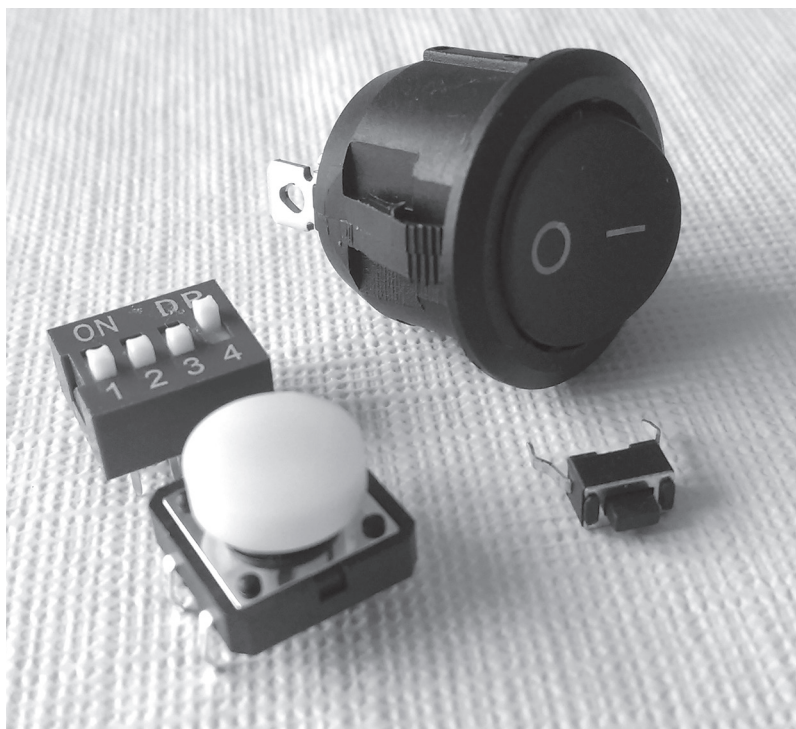


Рис. 4-2. Некоторые электрические выключатели

Обратите внимание, что на рис. 4-2 два ближайших к нам выключателя — это *кнопочные выключатели*, которые вы обычно не считаете выключателями. Кнопочные выключатели также известны как *кнопки без фиксации*, поскольку такой выключатель замкнут только тогда, когда



кнопка нажата. Когда давление на кнопку снимается, выключатель размыкается<sup>1</sup>.

Теперь, когда мы представили элемент схемы, который можно легко включать и выключать, давайте используем выключатели для построения цифровой схемы, которая действует как логический оператор И (AND). Если вы помните из главы 2, логическое И с двумя входами выдает 1, если оба входа равны 1, и 0 в противном случае. В качестве напоминания таблица истинности для оператора И повторена в табл. 4-1.

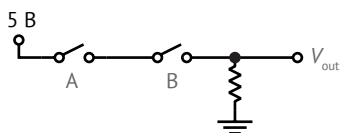
**Таблица 4-1.** Таблица истинности для оператора И (AND)

A	B	Выход
0	0	0
0	1	0
1	0	0
1	1	1

Давайте теперь переведем это в схему, руководствуясь следующими указаниями.

1. Входы A и B представлены механическими выключателями. Мы представляем 0 с помощью разомкнутого выключателя, а для представления 1 используем замкнутый выключатель.
2. Выход определяется напряжением в определенной точке нашей цепи, обозначенной  $V_{out}$ .
3. Если напряжение  $V_{out}$  равно примерно 5 В, то на выходе логическая 1, а если  $V_{out}$  равно примерно 0 В, то на выходе логический 0.

Рассмотрим схему, показанную на рис. 4-3. Это логическое И (AND), реализованное с помощью выключателей.



**Рис. 4-3:** Логическое И (AND), реализованное с помощью выключателей

Если один из выключателей на рис. 4-3 выключен (разомкнут/0), ток не течет и напряжение на  $V_{out}$  равно 0 В. Если оба выключателя включены (замкнуты/1), это создает путь к земле, ток течет и напряжение на  $V_{out}$  равно 5 В. Другими словами, если A и B равны 1, то на выходе будет 1.

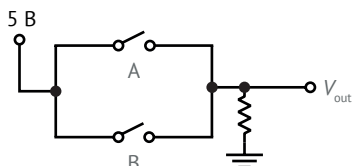
Давайте применим тот же подход к логическому ИЛИ (OR). Таблица истинности для оператора ИЛИ (OR), впервые упомянутая в главе 2, показана в табл. 4-2.

<sup>1</sup> Существуют, конечно, кнопочные выключатели с фиксацией, которые работают аналогично обычным – при каждом нажатии контакты перебрасываются в противоположное состояние. В дальнейшем описании автор под выключателями почти всегда, за редким исключением, подразумевает именно кнопки без фиксации. – *Прим. ред.*

**Таблица 4-2.** Таблица истинности для оператора ИЛИ (OR)

A	B	Выход
0	0	0
0	1	1
1	0	1
1	1	1

Посмотрите на схему на рис. 4-4, где представлено логическое ИЛИ (OR), реализованное с помощью выключателей.



*Рис. 4-4. Логическое ИЛИ (OR), реализованное с помощью выключателей*

На рис. 4-4, когда оба выключателя выключены (разомкнуты/0), ток не течет и напряжение на  $V_{out}$  равно 0 В, т. е. логическому 0. Когда любой из выключателей включен (замкнут/1), ток течет и напряжение на  $V_{out}$  равно 5 В. Другими словами, если A или B равны 1, то на выходе будет 1.

## Удивительный транзистор

В нашем стремлении разработать цифровые схемы только что рассмотренные устройства на основе выключателей являются хорошим концептуальным началом. Однако в вычислительных устройствах мы практически не можем использовать механические выключатели. Входы в компьютер многочисленны, и щелканье выключателями для управления этими входами не лучший вариант. Кроме того, вычислительные устройства должны соединять несколько логических схем вместе, выход одной схемы должен стать входом другой. Для этого наши выключатели должны управляться электрически, а не механически. Нам не нужен механический выключатель, нам нужен электронный выключатель. К счастью, есть компонент схемы, который может работать как электронный выключатель, – это транзистор<sup>1</sup>!

<sup>1</sup> Ради сокращения изложения автор пропускает один важный компонент, в котором электрически управляются механические контакты – электромагнитное реле. На основе реле строились исторически первые компьютеры в 1940-х годах. Причем именно построение логических элементов на электромагнитных реле наиболее наглядно с познавательной точки зрения (см., например, Чарльз Петцольд: Код. Тайный язык информатики. Изд-во Microsoft Press, 2001, 2004 г.). – *Прим. ред.*

Транзистор – это устройство, используемое для включения/выключения или усиления тока. Для наших целей давайте сосредоточимся на включательных/выключательных (ключевых) возможностях транзистора. Транзистор является основой современной электроники, включая вычислительные устройства. Существует два основных типа транзисторов: биполярные и полевые транзисторы<sup>1</sup>. Различия между этими двумя типами не имеют значения для нашего обсуждения здесь, поэтому для простоты давайте сосредоточимся только на одном типе – биполярном.

Биполярные транзисторы имеют три вывода: базу, коллектор и эмиттер. Существует два типа биполярных транзисторов: NPN и PNP. Эти два типа отличаются реакцией на подачу тока на базовый вывод. Для наших целей мы сосредоточимся на типе NPN. Схема и фотография маломощного биполярного транзистора NPN приведены на рис. 4-5.

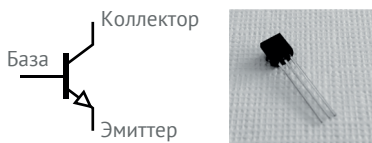


Рис. 4-5. Символ (слева) и фотография (справа) биполярного транзистора

В биполярном транзисторе типа NPN подача слабого тока на базу позволяет большему току течь от коллектора к эмиттеру. Другими словами, если рассматривать транзистор как ключ, то подача тока на базу подобна включению транзистора, а прекращение тока выключает транзистор. Давайте посмотрим, как транзистор может быть подключен в качестве электронного выключателя. Схема показана на рис. 4-6.

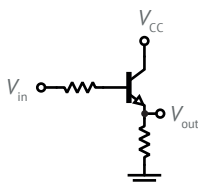


Рис. 4-6. NPN-транзистор в качестве выключателя

На рис. 4-6 транзистор NPN подключен к паре резисторов с различными напряжениями на них.  $V_{cc}$  – это положительное напряжение питания схемы, подаваемое на вывод коллектора. Если вам интересно, то «cc» в  $V_{cc}$  означает «общий коллектор» (common collector), и  $V_{cc}$  – это типичное обозначение положительного напряжения питания в схемах на основе NPN.  $V_{out}$  – это напряжение, которым мы хотим управлять; мы хотим, чтобы это напряжение имело высокий уровень, когда наш транзистор-как-выключатель включен, и низкий, когда наш выключа-

<sup>1</sup> В англоязычной литературе для биполярных транзисторов принято сокращение BJT, для полевых – FET. – Прим. ред.

тель выключен.  $V_{in}$  действует как входное напряжение, которое электрически управляет нашим выключателем<sup>1</sup>. Вместо того чтобы выключать механическое устройство, мы можем использовать напряжение  $V_{in}$  для управления нашим выключателем.

Давайте рассмотрим, что произойдет, если мы установим  $V_{in}$  на низкий или высокий уровень напряжения. Если  $V_{in}$  имеет низкий уровень, соединенный с землей, то ток на базу транзистора не течет. При отсутствии тока в базе транзистор работает как разомкнутая цепь между коллектором и эмиттером. Это означает, что напряжение  $V_{out}$  также имеет низкий уровень.

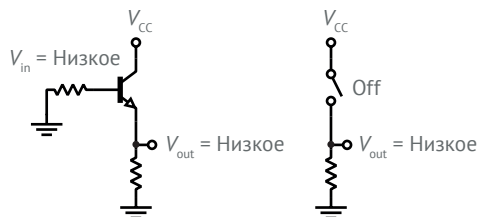


Рис. 4-7. NPN-транзистор в качестве выключателя в выключенном состоянии

Рисунок 4-7 иллюстрирует эту ситуацию. Слева на нем показана схема на базе транзистора, которую мы сейчас обсуждаем, а справа – схема на базе выключателя, которая представляет то же состояние. Другими словами, схема слева фактически такая же, как и схема справа; я просто заменил транзистор выключателем, чтобы было понятно, что транзистор в этом состоянии подобен разомкнутому выключателю.

Если  $V_{in}$  переключить в высокий уровень, то ток потечет на базу транзистора. Это заставляет транзистор проводить ток от коллектора к эмиттеру.  $V_{out}$  фактически соединится с  $V_{cc}$ , и поэтому на выходе будет высокий уровень напряжения, как показано на рис. 4-8.

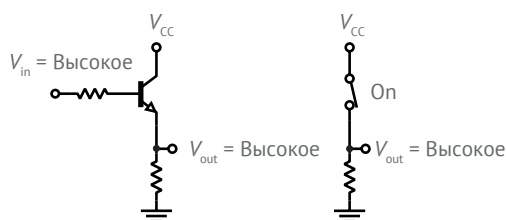


Рис. 4-8. NPN-транзистор в качестве выключателя во включенном состоянии

Слева на рис. 4-8 изображена схема на базе транзистора, которую мы обсуждаем, а справа – схема на базе механического выключателя, которая представляет то же состояние. В этом состоянии транзистор похож на замкнутый выключатель.

<sup>1</sup> «In» и «out» переводятся как «вход» и «выход». Соответственно,  $V_{in}$  означает входное напряжение, а  $V_{out}$  – выходное. – Прим. ред.

## Логические вентили<sup>1</sup>

Теперь, когда мы убедились, что транзистор может работать как электрически управляемый выключатель, можем создавать элементы схемы, реализующие логические функции, где входами и выходами является высокое и низкое напряжение. Такие компоненты называются *логическими вентилями*. Давайте начнем с того, что возьмем наш предыдущий проект схемы, выполняющей функцию И (AND), и заменим механические выключатели транзисторами. Преимуществом этого является то, что мы можем менять состояние входа, просто изменяя напряжение, нам не нужно для этого переключать механические выключатели. Хотя механические выключатели – это хороший способ взаимодействия человека со схемами, электронные выключатели позволяют нескольким схемам взаимодействовать друг с другом – выход одной схемы может легко стать входом другой.

Ранее мы построили схему И (AND), используя механические выключатели (см. рис. 4-3). Давайте используем транзисторы в качестве выключателей, чтобы добиться того же самого, как показано на рис. 4-9.

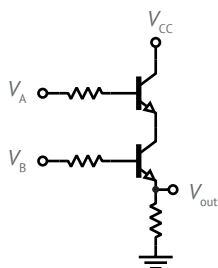


Рис. 4-9. Логическое И, реализованное с помощью транзисторов

На рис. 4-9 если  $V_A$  и  $V_B$  высокого уровня (логическая 1), то ток течет через оба транзистора и  $V_{out}$  также имеет высокий уровень (логическая 1). Если  $V_A$  или  $V_B$  низкого уровня (логический 0), то ток не течет и  $V_{out}$  будет также низкого уровня (логический 0). Эта схема реализует логическое И (AND).

Аналогичный подход может быть использован для реализации логического ИЛИ (OR) с помощью транзисторов. Я оставляю это как упражнение и проект для вас.

### УПРАЖНЕНИЕ 4-1: Реализация логического ИЛИ (OR) на транзисторах

Нарисуйте схему логической функции ИЛИ (OR), в которой для входов А и В используются транзисторы. Адаптируйте схему на рис. 4-4, в которой используются механические выключатели, но вместо них используйте NPN-транзисторы. Решение см. в приложении А.

<sup>1</sup> В отечественной литературе логические вентили чаще называют логическими элементами. – *Прим. ред.*

*Обратитесь к проекту № 3 на стр. 96, где вы сможете построить схемы логического И и логического ИЛИ с использованием транзисторов.*

Мы только что увидели, как с помощью транзисторов и резисторов можно построить логический вентиль – схему, реализующую логическую функцию. С этого момента я не стану показывать детали того, как реализуются логические вентиля, вместо этого мы будем рассматривать весь вентиль как единый элемент схемы. Это не просто теоретический способ рассмотрения логических вентилях, он соответствует тому, как эти элементы схемы используются в реальном мире. Логические вентиля можно приобрести уже собранными и упакованными как компоненты схем, поэтому обычно нет необходимости собирать их самостоятельно из транзисторов, разве что в качестве учебного упражнения. Для отображения различных логических вентилях на схемах определены стандартные символы. Некоторые из наиболее распространенных вы можете увидеть на рис. 4-10.

Рассматривая различные логические вентиля на рис. 4-10, обратите внимание на маленькие кружочки, добавленные к различным символам для обозначения функции НЕ (NOT) или инверсии. Простейшим примером этого является вентиль НЕ (NOT): он имеет один вход, а его выход является инверсией этого входа. Таким образом, 1 становится 0, а 0 становится 1. Выход вентиля НЕ-И (NAND) – это то же самое, что и НЕ И (NOT AND); его выход – это инверсия выхода обычного вентиля И (AND). То же самое верно и для НЕ-ИЛИ (NOR)<sup>1</sup>. Вы увидите, что маленький кружок встречается и в других местах в логических символах для обозначения НЕ или инверсии.

Прежде чем двигаться дальше, давайте сделаем паузу и поразмышляем над некоторыми аспектами того, что мы только что рассмотрели. Сначала мы рассмотрели, как логический вентиль работает изнутри. Затем взяли эту схему, упаковали ее, дали ей имя и символ. Мы намеренно скрыли реализацию схемы, продолжая документировать ее ожидаемое поведение. Мы поместили детали конструкции логического вентиля в так называемый *черный ящик* – элемент, в котором известны входы и выходы, но внутреннее устройство скрыто. Другой термин для этого подхода – *инкапсуляция*<sup>2</sup>, выбор конструкции, которая скрывает внутренние детали компонента, в то же время описывая, как взаимодействовать с данным компонентом.

<sup>1</sup> В отечественной литературе принято обратное написание ИЛИ-НЕ и И-НЕ, так как инверсия сигнала происходит на выходе элемента (вентиля, см. рис. 4-10). Этот разнобой в обозначениях не должен вас смущать, так как выполняемые логические операции во всех случаях одинаковые. – *Прим. ред.*

<sup>2</sup> Инкапсуляция – размещение в оболочке, изоляция, закрытие внутренних деталей. Принцип «черного ящика» с сокрытием деталей внутреннего устройства широко применяется в программировании и электронике, см. далее. – *Прим. ред.*

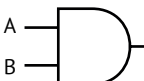

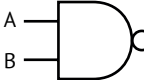
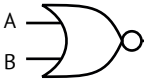
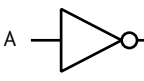

Тип	Символ	Таблица истинности															
И		<table><tr><th>A</th><th>B</th><th>Выход</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Выход	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Выход															
0	0	0															
0	1	0															
1	0	0															
1	1	1															
ИЛИ		<table><tr><th>A</th><th>B</th><th>Выход</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Выход	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Выход															
0	0	0															
0	1	1															
1	0	1															
1	1	1															
НЕ-И		<table><tr><th>A</th><th>B</th><th>Выход</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Выход	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Выход															
0	0	1															
0	1	1															
1	0	1															
1	1	0															
НЕ-ИЛИ		<table><tr><th>A</th><th>B</th><th>Выход</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Выход	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Выход															
0	0	1															
0	1	0															
1	0	0															
1	1	0															
НЕТ		<table><tr><th>A</th><th>Выход</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Выход	0	1	1	0									
A	Выход																
0	1																
1	0																
Исключающее ИЛИ		<table><tr><th>A</th><th>B</th><th>Выход</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Выход	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Выход															
0	0	0															
0	1	1															
1	0	1															
1	1	0															

Рис. 4-10. Распространенные логические вентили<sup>1</sup>

Принцип «черного ящика» – прием проектирования, который встречается во всех современных технологиях. Он используется, когда разработчики компонентов хотят, чтобы другие могли легко использовать плоды их разработки и создавать что-либо на их основе, не нуждаясь в полном понимании деталей реализации компонентов. Такой подход также позволяет улучшать внутреннюю конструкцию «черного ящика», и пока входы и выходы ведут себя одинаково, «ящик» можно продолжать использовать, как и раньше. Еще одно преимущество принципа «черного ящика» заключается в том, что команда может совместно работать над большим проектом, при этом отдельные детали проекта могут быть скрыты (инкапсулированы). Это освобождает отдельных членов команды от необходимости понимать каждую деталь каждого компонента. Инкапсуляция транзисторов в логических вентилях – это первый пример применения принципа «черного ящика» в этой книге, но вы увидите такие примеры еще много раз в дальнейшем.

<sup>1</sup> На рис. 4-10 и далее в этой книге используется американский вариант графического исполнения логических элементов (стандарт ANSI). В отечественной литературе приняты иные обозначения – по международному стандарту IEC и соответствующему ему ГОСТ 2.743-91. – *Прим. ред.*

## Проектирование с помощью логических вентилях

В главе 2 мы рассмотрели, как можно комбинировать несколько логических операторов для создания более сложных логических утверждений. Теперь давайте применим эту идею к логическим вентилям.

После того как логическое утверждение или таблица истинности написаны, эта логика может быть физически реализована в оборудовании с помощью логических вентилях. Применим этот принцип к таблице истинности, которую мы ранее создали (табл. 2-6) для следующего утверждения:

---

ЕСЛИ сегодня солнечно И тепло ИЛИ сегодня мой день рождения, ТОГДА я иду на пляж.

---

Мы упростили это утверждение до:

---

(А И В) ИЛИ С.

---

Давайте теперь представим это утверждение с помощью диаграммы с логическими вентилями, как показано на рис. 4-11.

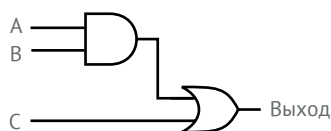


Рис. 4-11. Схема соединения логических вентилях для функции (А И В) ИЛИ С

Если оба А и В равны 1, то на выходе вентиля И будет 1. Выход И подключается ко входу в вентиль ИЛИ вместе с С. Если выход И вентиля или С равен 1, то общий выход будет равен 1.

Когда мы объединяем логические вентиля таким образом, что выход является функцией от текущих входов, схема называется *комбинационной логической схемой*. То есть определенный набор текущих входов всегда будет давать один и тот же выход. В отличие от *секвенциальной (последовательной) логики*<sup>1</sup>, в которой выход является функцией как настоящих, так и прошлых состояний входов. Мы рассмотрим последовательную логику позже в этой книге. А пока попробуйте спроектировать схему, представляющую логическое выражение, описанное в упражнении 4-2.

---

<sup>1</sup> *Sequential* – последовательный, являющийся продолжением. К комбинационной логике относятся простые логические схемы (наборы элементов И, ИЛИ, И-НЕ, ИЛИ-НЕ, Исключающее ИЛИ, инверторов и т. п.), а также устройства на их основе: мультиплексоры/демультиплексоры, шифраторы/дешифраторы, сумматоры и т. п. Последовательная логика представлена схемами, имеющими в своем составе устройства с памятью: триггеры, счетчики, регистры и т. п. – *Прим. ред.*



## УПРАЖНЕНИЕ 4-2: Проектирование схемы с логическими вентилями

В упражнении 2-5 главы 2 вы создали таблицу истинности для функции (А ИЛИ В) И С. Теперь, основываясь на этой работе, переведите таблицу истинности и логическое выражение в цифровую схему. Нарисуйте схему логических вентилях (подобную той, что показана на рис. 4-11). Ответ см. в приложении А.

## Интегральные схемы

Как упоминалось ранее, компании производят и продают готовые к использованию цифровые логические вентили. Разработчики оборудования могут купить эти вентили и начать строить свою логику, не заботясь о том, как вентили работают внутри. Эти логические вентили являются примером интегральной схемы. *Интегральная схема (ИС)* содержит множество компонентов на одном куске кремния в корпусе с внешними электрическими контактами или *выводами*. ИС также часто называют *микросхемами* или *чипами*.

В этой книге мы в основном рассматриваем ИС, которые упакованы в корпус с двухрядным расположением выводов (*DIP – dual in-line package*) – прямоугольный корпус с двумя параллельными рядами выводов. Эти выводы расположены на стандартном расстоянии друг от друга, так что их можно легко использовать на макетной плате. Производители собирают микросхемы из крошечных транзисторов, гораздо меньших, чем дискретные транзисторы, вроде показанного ранее на рис. 4-5. *Дискретный* компонент – это электронное устройство, содержащее только один элемент, например резистор или транзистор. ИС позволяют создавать небольшие схемы, которые могут работать быстрее и стоят дешевле, чем такие же схемы, построенные из дискретных транзисторов.

Логические схемы, которые мы обсуждали ранее и в которых использовались резисторы и транзисторы, известны как резисторно-транзисторные логические схемы (РТЛ). Производители создавали таким образом ранние цифровые логические схемы, но позже они стали использовать другие подходы, включая диодно-транзисторную логику (ДТЛ) и транзисторно-транзисторную логику (ТТЛ). Серия 7400 – самая популярная линейка логических схем ТТЛ. Эта линейка интегральных схем включает логические вентили и другие цифровые компоненты. Появившись в 1960-х годах, серия 7400 и ее последователи до сих пор являются стандартом для цифровых схем. Я собираюсь сосредоточиться на классической серии 7400, чтобы дать вам примеры использования интегральных схем в реальном мире.

Давайте рассмотрим конкретную интегральную схему серии 7400. Микросхема 7432, показанная на рис. 4-12, содержит четыре вентиля ИЛИ.

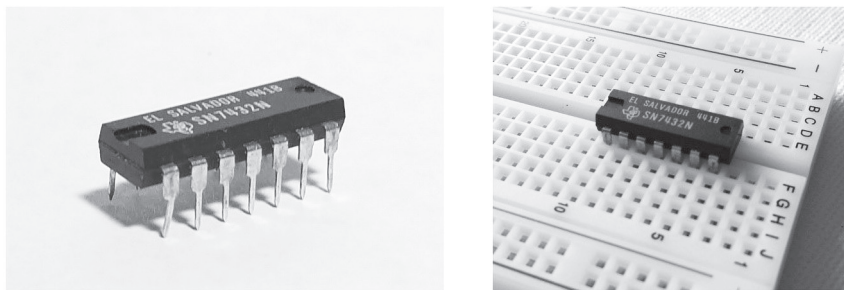


Рис. 4-12. Интегральная схема SN7432N в корпусе с двухрядным расположением выводов, размещенная на макетной плате (справа)

Микросхема 7432 поставляется в корпусе с 14 выводами. Каждый из четырех вентилях ИЛИ требует 3 вывода, что дает 12 выводов плюс 1 вывод для положительного напряжения ( $V_{cc}$ ) и 1 вывод для земли (GND) – итого 14 выводов. Серия 7400 работает с ожидаемым напряжением питания ( $V_{cc}$ ), равным 5 В. То есть высокий уровень, логическая 1, в идеале равен 5 В, а низкий уровень равен 0 В. Однако на практике входное напряжение от 2 до 5 В считается высоким уровнем, а от 0 до 0,8 В – низким.

На рис. 4-12 видно, что корпус 7432 имеет 7 выводов с каждой стороны и аккуратно помещается в макетную плату<sup>1</sup>. При размещении такого чипа на макетной плате убедитесь, что он расположена так, чтобы накрывать промежуток в центре макетной платы, чтобы исключить случайное соединение выводов, расположенных прямо напротив друг друга (например, выводы 1 и 14). Обратите внимание на полукруглую выемку на корпусе, она подскажет вам, в какую сторону ориентировать микросхему при определении выводов<sup>2</sup>.

Расположение схемы внутри корпуса показано на рис. 4-13. Это *схема разводки выводов (цоколевка)*, т. е. чертеж, на котором обозначены электрические контакты компонента. Цель такой схемы – показать внешние точки подключения компонента. Обычно чертеж разводки выводов не демонстрирует внутреннюю конструкцию интегральной схемы.

<sup>1</sup> Отметим, что в некоторые макетные платы микросхемы в DIP-корпусе непосредственно не могут быть установлены – выводы у них не достают до подпружиненных заглубленных контактов платы. Для установки таких микросхем следует дополнительно приобрести переходные панельки так называемого цангового типа. – Прим. ред.

<sup>2</sup> При взгляде на микросхему со стороны корпуса (сверху), вывод номер 1 – первый снизу от выемки или иной отметки на корпусе. Остальные выводы нумеруются, начиная от первого, при обходе микросхемы против часовой стрелки (см. рис. 4-13). Обратите внимание, что у игольчатых разъемов нумерация выводов производится по иному принципу, чем у микросхем (см. рис. 13-11). – Прим. ред.

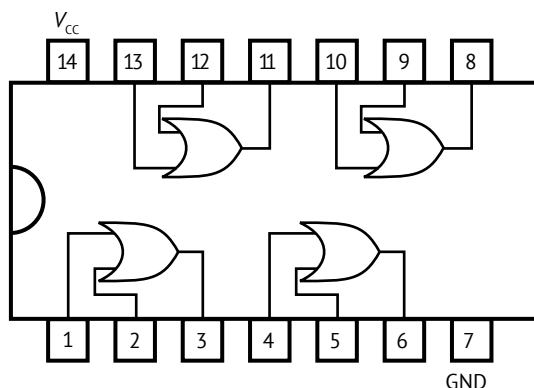


Рис. 4-13. Разводка выводов микросхемы 7432

Допустим, вы хотите использовать один из четырех вентилях ИЛИ чипа 7432 и выбираете вентиль в левом нижнем углу схемы разводки выводов на рис. 4-13, подключенный к выводам 1, 2 и 3. Чтобы использовать этот вентиль, нужно подключить выводы, как показано в табл. 4-3.

**Таблица 4-3.** Подключение одного вентиля ИЛИ в микросхеме 7432

Вывод	Подключение
1	Это вход А вентиля ИЛИ. Подключите к 5 В или GND для 1 или 0 соответственно
2	Это вход В вентиля ИЛИ. Подключите к 5 В или GND для 1 или 0 соответственно
3	Это выход вентиля ИЛИ. Ожидайте, что он будет либо 5 В, либо GND для 1 или 0 соответственно
7	Соедините с землей
14	Подключите к 5-вольтовому источнику энергии

Серия 7400 содержит сотни компонентов. Мы не будем рассматривать их все здесь, но на рис. 4-14 вы можете увидеть разводку выводов четырех распространенных логических вентилях, доступных в этой серии. Вы можете провести быстрый поиск в интернете, чтобы найти цоколевку других микросхем серии 7400.

Вооружившись схемами разводки выводов для каждой из этих микросхем, вы теперь обладаете знаниями, необходимыми для физического построения схемы, которую вы ранее спроектировали в упражнении 4-2.

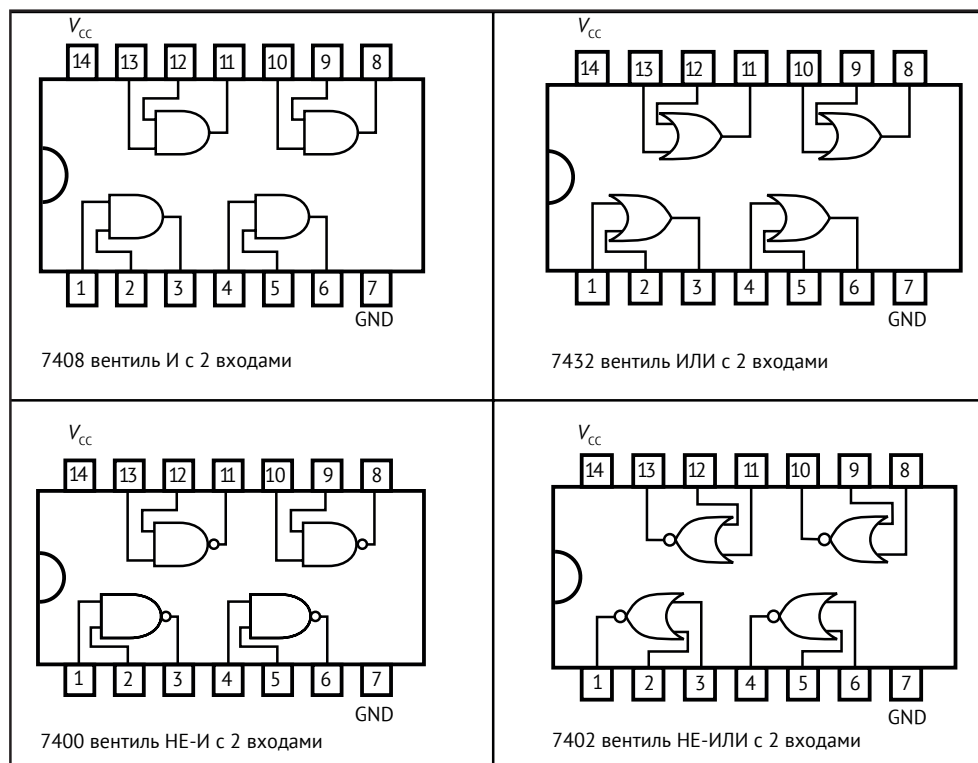


Рис. 4-14. Цоколевка распространенных интегральных схем серии 7400

#### ПРИМЕЧАНИЕ

Обратитесь к проекту №4 на стр. 98, где вы сможете собрать схему, реализующую логическое выражение  $(A \text{ ИЛИ } B) \text{ И } C$ .

## Выводы

В этой главе мы рассмотрели двоичные цифровые схемы – схемы, в которых для представления логической 1 или 0 используются уровни напряжения. Вы узнали, как выключатели могут использоваться для построения логических операторов, таких как И, в физической схеме. Мы рассмотрели ограничения использования механических выключателей для этой цели и представили новый элемент схемы, который может действовать как электрически управляемый выключатель – транзистор. Вы узнали о логических вентилях – элементах схемы, реализующих логические функции. Мы рассмотрели интегральные схемы, включая серию 7400.

В следующей главе мы рассмотрим, как логические вентили могут быть использованы для построения схем, реализующих одну из фундаментальных функций компьютера – математику. Вы увидите, как простые логические вентили, используемые вместе, позволяют выполнять более сложные функции. Мы также рассмотрим представление целых чисел в компьютере, используя знаковые и беззнаковые числа.

## ПРОЕКТ № 3: Построение логических операторов (И, ИЛИ) с помощью транзисторов

В этом проекте вы построите физические схемы для логических операторов И (AND) и ИЛИ (OR) с использованием транзисторов. Для создания этих схем вам понадобятся следующие компоненты:

- макетная плата (либо 400-точечная, либо 830-точечная модель);
- резисторы (набор резисторов);
- 9-вольтовая батарея;
- разъем для 9-вольтовой батареи;
- 5-мм или 3-мм красный светодиод;
- провода-перемычки (для макетной платы);
- два маломощных NPN-транзистора (типа 2N2222 или аналогичные в корпусе TO-92).

Смотрите раздел «Покупка электронных компонентов» на стр. 406, который поможет в приобретении этих деталей.

Прежде чем приступить к подключению компонентов, следует знать, что некоторые транзисторы и интегральные схемы являются устройствами, чувствительными к электростатическому электричеству, т. е. они могут быть повреждены статическим разрядом. Вы когда-нибудь испытывали удар статического электричества, когда прикасались к чему-либо после хождения по ковру? Такой удар электричества может быть фатальным для электронных компонентов. Даже очень маленький статический разряд, который вы и не заметите, может повредить электронный компонент.

Профессионалы в электронной промышленности для предотвращения такой проблемы надевают заземленные браслеты на запястья, работают в антистатических помещениях и носят специальную одежду. Такие меры предосторожности не соблюдаются большинством любителей, но вы должны хотя бы знать о риске повреждения транзисторов или интегральных схем статическим разрядом. Старайтесь избегать накопления статического электричества и прикасайтесь к заземленной поверхности (например, к винту, который удерживает крышку электрической розетки), чтобы снять статическое электричество, прежде чем работать с чувствительными к электростатике устройствами<sup>1</sup>.

<sup>1</sup> Винт, удерживающий крышку в розетке, обычно закручен в пластиковую основу и ни к чему не подключен. Самой доступной заземленной деталью в бытовых помещениях являются подпружиненные латунные контакты заземления в розетках для мощных потребителей (они расположены симметрично по сторонам круглых утопленных гнезд). Однако и эти контакты могут «висеть в воздухе», если в помещении нет настоящего заземления (как часто бывает в квартирах со старой, давно не ремонтировавшейся проводкой). Не следует, однако, и преувеличивать опасности статического электричества для рядовых компонентов, особенно указанных здесь биполярных транзисторов и микросхем классической 74-й серии – в большинстве компонентов приняты меры по защите от повреждения статикой. Хватает обычных в таких случаях рекомендаций – не надевать одежду из синтетических тканей, не вести монтаж в помещениях с синтетическим покрытием полов и на столе с пластиковым покрытием. – *Прим. ред.*

Теперь давайте вернемся к рассматриваемому проекту. Транзистор 2N2222 в корпусе ТО-92 имеет три вывода. Если держать транзистор плоской поверхностью к себе, а выводами вниз, то левый вывод будет эмиттером, средний – базой, а правый – коллектором (см. рис. 4-5). Вы также можете набрать в поиске в интернете «2N2222 TO-92» для получения более подробной информации об этом конкретном транзисторе.

Следуйте рис. 4-15, чтобы построить схему И (AND) с 9-вольтовой батареей, транзисторами, резисторами и светодиодом.

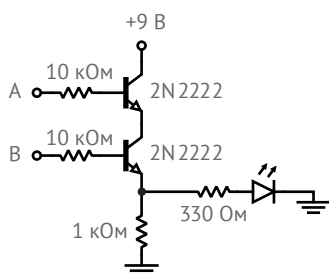


Рис. 4-15. Схема функции И с конкретными резисторами, транзисторами и светодиодом на выходе

Входы А и В должны быть подключены либо к 9 В (для 1), либо к земле (для 0) для проверки входов. Светодиод должен загореться, когда сигнал на выходе (точка соединения эмиттера нижнего транзистора с резисторами 1 кОм и 330 Ом) будет равен логической 1. Смотрите табл. 4-1 для определения ожидаемого состояния выхода при различных комбинациях входов. Помните, что +9 В на схеме означает подключение к положительной клемме батареи, а символ земли на схеме означает подключение к отрицательной клемме батареи. Также помните, что светодиод предназначен для пропускания тока только в одном направлении, так что удостоверьтесь, что вы соединили его короткий вывод с землей. Если схема работает не так, как ожидалось, ознакомьтесь с разделом «Поиск и устранение неисправностей в схемах» на стр. 414.

Собранная схема должна выглядеть примерно так, как показано на рис. 4-16. Конечно, ваше конкретное расположение деталей на макетной плате может отличаться от моего.

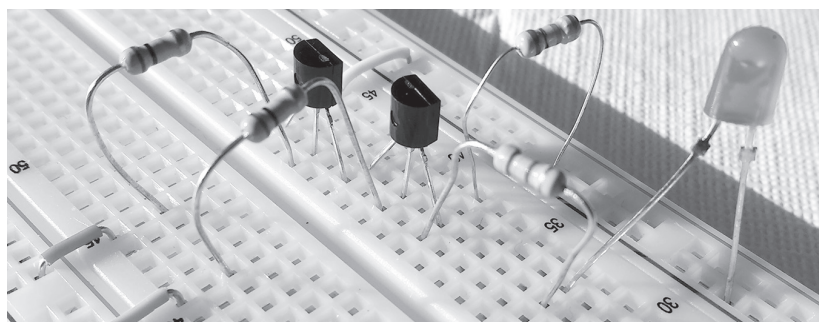


Рис. 4-16. Схема И, представленная на рис. 4-15, собранная на макетной плате

После того как у вас есть работающая схема для функции И, давайте перейдем к аналогичной схеме ИЛИ. Следуя рис. 4-17, постройте схему ИЛИ с 9-вольтовой батареей, транзисторами, резисторами и светодиодом.

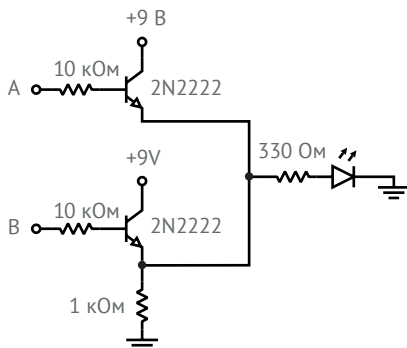


Рис. 4-17. Схема функции ИЛИ с конкретными резисторами, транзисторами и светодиодом на выходе

Как и в предыдущей схеме, для проверки входы А и В должны быть подключены либо к 9 В (для 1), либо к земле (для 0). Светодиод должен загореться, когда состояние выхода (точка объединения эмиттеров транзисторов) будет равно 1. Смотрите табл. 4-2 для определения ожидаемого состояния выхода для различных комбинаций входов.

## ПРОЕКТ № 4: Построение схемы с логическими вентилями

В упражнении 4-2 вы нарисовали схему цепи для функции (А ИЛИ В) И С. Если вы пропустили это упражнение, я советую вам вернуться и выполнить его, прежде чем продолжать далее. Результат должен быть похож на схему, представленную на рис. 4-18.

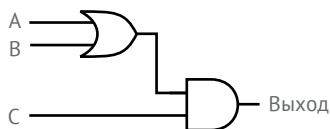


Рис. 4-18. Схема соединения логических вентилях для функции (А ИЛИ В) И С

Теперь давайте физически построим эту схему! Подключите вывод к светодиоду (не забудьте включить резистор!), чтобы вы могли видеть, что получается на выходе – 0 или 1. Ваши три входа (А, В, С) могут быть напрямую подключены к 5 В или земле. Попробуйте подключить различные

комбинации входов, чтобы убедиться, что ваша логика работает так, как ожидается.

Для этого проекта вам понадобятся следующие компоненты<sup>1</sup>:

- макетная плата;
- светодиод;
- токоограничивающий резистор для светодиода, приблизительно 220 Ом;
- провода-перемычки;
- интегральная схема 7408 (содержит четыре вентиля И);
- интегральная схема 7432 (содержит четыре вентиля ИЛИ);
- 5-вольтовый источник питания;
- три кнопки или ползунковых переключателя, которые подойдут для макетной платы (для бонусного проекта);
- три резистора 470 Ом (для бонусного вопроса).

Поскольку для этой схемы требуется 5-вольтовый источник питания, а не 9-вольтовая батарея, посмотрите раздел «Питание цифровых схем» на стр. 410, чтобы узнать, как это можно сделать. Также как и раньше, см. раздел «Покупка электронных компонентов» на стр. 406, который поможет в приобретении этих деталей.

Возможно, вы заметили, что в списке компонентов рекомендуется резистор 220 Ом, а не 330 Ом, который мы использовали ранее. Это связано с тем, что мы снизили напряжение источника с 9 до 5 В. Конкретное значение, необходимое для вашей схемы, будет зависеть от прямого напряжения светодиода, который вы используете, как мы рассматривали в главе 3. При этом значение резистора не обязательно должно быть точным. Вы можете использовать резистор 220 Ом, 200 Ом или 180 Ом<sup>2</sup> – все эти значения легко доступны. Схема подключения на рис. 4-19 показывает детали построения этой схемы.

Помните, что выводы чипа, размещенного на макетной плате, электрически соединены со всей строкой. Не забудьте разместить микросхемы так, чтобы они покрывали промежуток в центре макетной платы, чтобы исключить соединение выводов, расположенных непосредственно напротив друг друга. Готовая схема, если она собрана на макетной плате, будет выглядеть примерно так, как показано на рис. 4-20.

<sup>1</sup> См. также сноску 1 на стр. 93.

<sup>2</sup> Так как светодиод здесь подключен непосредственно к выходу TTL-микросхемы классических серий 74 или 74LS (см. далее), следует учитывать, что логической единице на ее выходе соответствует напряжение, сниженное относительно питания и составляющее около 3,5 В (см. документацию на микросхемы 7432 и 7408). Потому, с учетом прямого падения на светодиоде, ток через вывод при токоограничивающем резисторе 220 Ом составит менее 10 мА, что даст достаточную яркость свечения. Тем не менее не рекомендуется еще больше уменьшать сопротивление токоограничивающего резистора во избежание перегрузки выводов микросхемы. Сопротивление лучше выбирать в пределах 220–300 Ом. – *Прим. ред.*



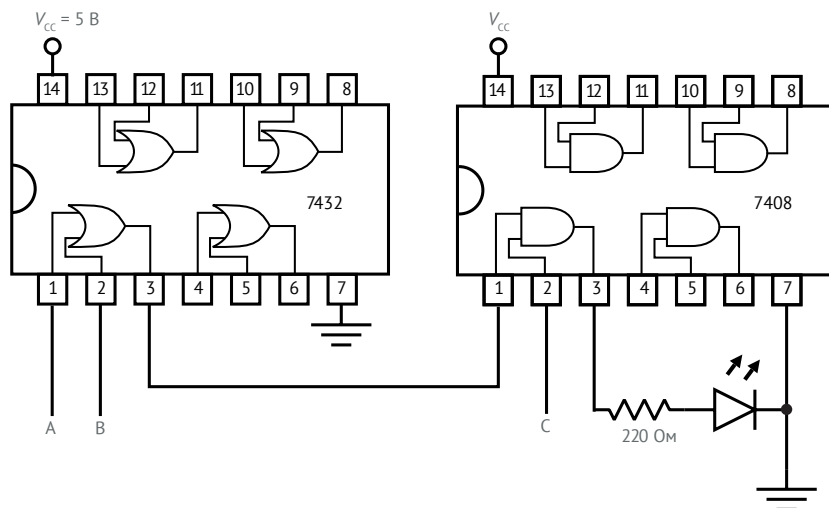


Рис. 4-19. Схема соединений для функции (А ИЛИ В) И С

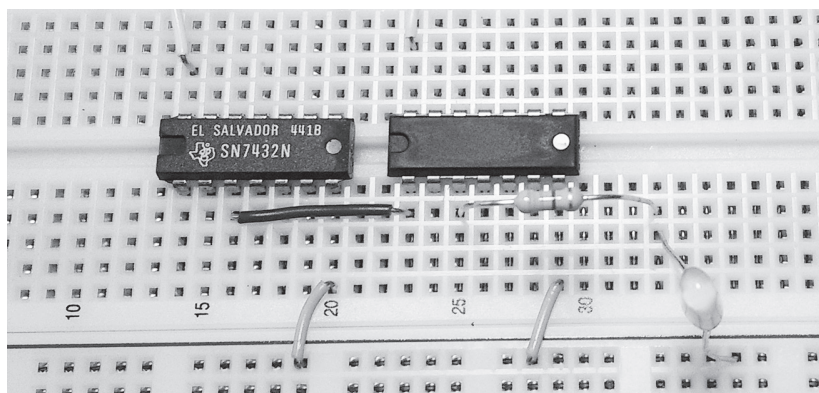


Рис. 4-20. Реализация на макетной плате схемы (А ИЛИ В) И С с входами А, В и С, оставленными отключенными

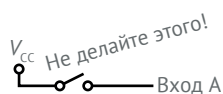
Обратите внимание, что на рис. 4-20 микросхема 7432 находится слева, а 7408 – справа. В этой конкретной схеме шина питания сверху подключена к 5 В, а минусовая шина питания внизу подключена к земле, но на фотографии не показано ни одно из этих соединений.

Также обратите внимание, что входы А, В и С здесь отключены, для тестирования различных состояний их нужно подключить к земле или к 5 В. После того как вы постройте эту схему, можете попробовать подключить различные комбинации входов, чтобы убедиться, что ваша логика работает так, как ожидается. Подключите вход к 5 В, чтобы представить логическую 1, или к земле, чтобы представить логический 0. Проверьте поведение схемы по таблице истинности для функции (А ИЛИ В) И С, представленной в табл. 4-4. Если схема работает не так, как ожидалось, ознакомьтесь с разделом «Поиск и устранение неисправностей в схемах» на стр. 414.

**Таблица 4-4.** Таблица истинности для функции (А ИЛИ В) И С

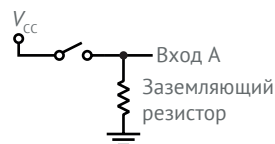
А	В	С	Выход
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Ручное перемещение входных проводов между землей и 5 В не лучшая идея. Лучшим вариантом было бы подключение входов А, В и С к механическим выключателям, чтобы можно было легко менять состояние входов без переделывания схемы. В качестве бонусного проекта давайте добавим несколько кнопочных выключателей для управления нашими входами. Вашим первым побуждением может быть подключение кнопки между входом и  $V_{CC}$ , как показано на рис. 4-21, где замыкание контактов подключает вход к 5 В, что соответствует логической 1.



**Рис. 4-21.** Выключатель между  $V_{CC}$  и входом. Подсказка: не делайте этого!

К сожалению, есть проблема с подходом, показанным на рис. 4-21. Замкнутые контакты работают, как и ожидается, а вот разомкнутые – нет. Вы можете ожидать, что разомкнутые контакты приведут к 0 В на входе А, но это совершенно не так. Когда контакты кнопки разомкнуты, напряжение на входе А «плавает» и имеет непредсказуемое значение. Помните, что вход А на рис. 4-21 представляет собой входной контакт вентиля ИЛИ на чипе 7432. Этот вход предназначен для подключения к высокому или низкому напряжению, если оставить его отключенным, то логический вентиль будет находиться в неопределенном состоянии. Нам нужно подключить наш выключатель таким образом, чтобы при его замыкании на нем оказывалось предсказуемое низкое напряжение. Как показано на рис. 4-22, мы можем сделать это с помощью заземляющего резистора – обычного резистора, используемого для подключения входа к низкому уровню, когда этот вход не подключен к высокому.



**Рис. 4-22.** Использование заземляющего резистора для обеспечения правильности цифровых входов

Рассмотрим, что произойдет, если добавить заземляющий резистор, как показано на рис. 4-22. Чтобы узнать, как будет реагировать вход интегральной схемы 7432 при различных условиях, мы можем посмотреть характеристики напряжения и тока, описанные в спецификации производителя. Я не буду вдаваться в подробности здесь (не стесняйтесь найти в интернете документацию для конкретной микросхемы 7432), но вкратце, когда выключатель разомкнут, небольшой ток течет от входа А через резистор на землю.

Если мы используем резистор с низким значением, ток, протекающий через вход А, приводит к напряжению на входе, достаточно низкому для регистрации логического 0. Когда выключатель замкнут, вход напрямую подключен к  $V_{cc}$  и на входе будет логическая 1. Для компонентов серии 74LS (рассматриваются в приложении В) для входов логических вентилях обычно используются заземляющий резистор в пределах 470 Ом – 1 кОм. Я рекомендую именно эти значения, поскольку они широко доступны и отвечают нашим требованиям. Значения более 1 кОм для компонентов серии 74LS не работают надежно<sup>1</sup>. Если вы используете заземляющие резисторы, можете построить полную схему с выключателями, как показано на рис. 4-23.

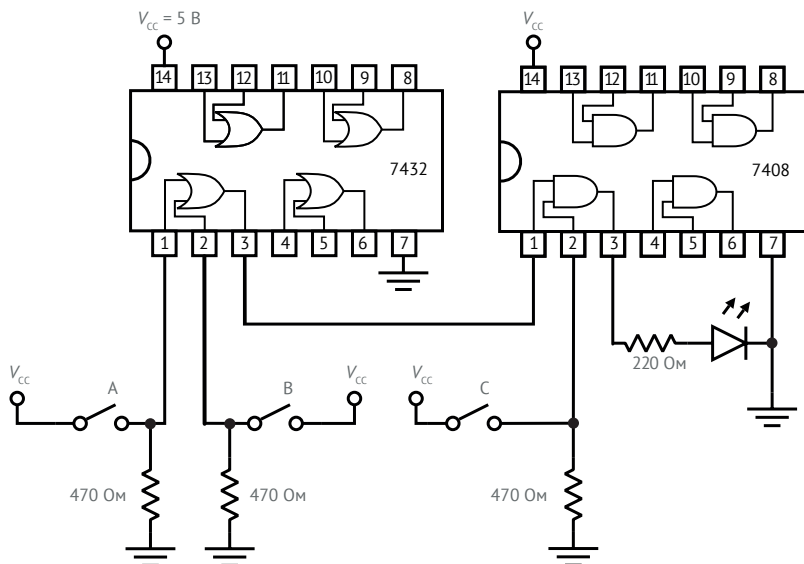


Рис. 4-23. Схема подключения для функции (А ИЛИ В) И С с добавлением выключателей для управления входами

Завершенная схема, если ее собрать на макетной плате, будет выглядеть так, как на фотографии, показанной на рис. 4-24. В моей схеме вместо выключателей я использовал кнопки, как показано в левом нижнем углу. Если вы посмотрите внимательно, то увидите, что заземляющие резисторы

<sup>1</sup> Для «обычной» TTL-серии 7400 рекомендуются такие же значения – от 470 до 1 кОм. Для обычно используемых в любительской электронике КМОП (CMOS)-серий микросхем (например, серии 4000 или серии 74НС) сопротивление заземляющего (как и «подтягивающего») резистора может быть увеличено до единиц и десятков килоом. – Прим. ред.

на этой фотографии имеют номинал 1 кОм, что отличается от предложенного в схеме значения 470 Ом, но и те и другие будут работать.

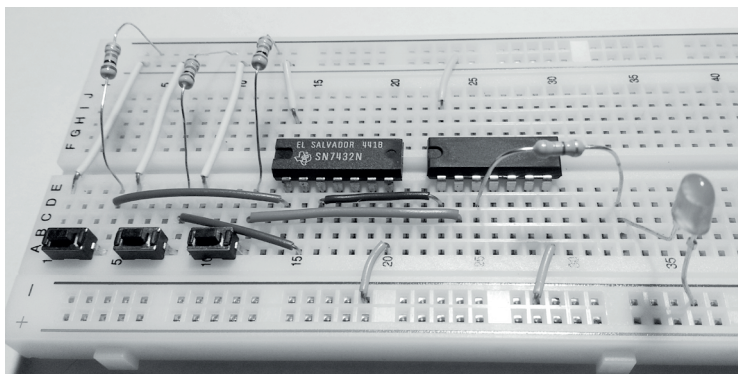
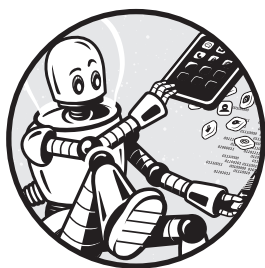


Рис. 4-24. Реализация схемы (А ИЛИ В) И С на макетной плате

Собрав схему, как показано на рис. 4-24, обязательно проверьте различные комбинации входов и убедитесь, что они соответствуют таблице истинности для функции (А ИЛИ В) И С в табл. 4-4. Если ваша схема работает не так, как ожидалось, ознакомьтесь с разделом «Поиск и устранение неисправностей в схемах» на стр. 414.

# 5

## МАТЕМАТИКА В ЦИФРОВЫХ СХЕМАХ



В предыдущей главе мы рассмотрели логические вентили и цифровые схемы, которые позволяют нам реализовать логические выражения аппаратными средствами. Ранее в этой книге мы определили компьютеры как электронные устройства, которые могут быть запрограммированы на выполнение набора инструкций. В этой главе я начинаю объединять эти понятия, показывая, как простые логические вентили прокладывают путь к операциям, выполняемым компьютером. Мы рассмотрим конкретную операцию, которую могут выполнять все компьютеры, – сложение. Сначала рассмотрим основы сложения в двоичной системе счисления. Затем мы используем логические вентили для создания оборудования, которое выполняет сложение, демонстрируя, как простые вентили работают вместе в компьютере для выполнения полезных операций. Наконец, рассмотрим представление целых чисел в виде знаковых и беззнаковых чисел в компьютере.

## Двоичное сложение

Давайте рассмотрим основы сложения в двоичной системе счисления. Принципы сложения одинаковы для всех позиционных систем счисления, поэтому у вас есть преимущество, поскольку вы уже знаете, как складывать в десятичной системе счисления! Вместо того чтобы рассматривать абстрактные понятия, давайте посмотрим на конкретный пример: сложение двух определенных четырехбитных чисел, 0010 и 0011, как показано на рис. 5-1.

$$\begin{array}{r} 0010 \\ + 0011 \\ \hline ???? \end{array}$$

Рис. 5-1. Сложение двух двоичных чисел

Как и в десятичной системе, мы начинаем с крайней правой позиции, называемой младшим значащим битом, и складываем два значения вместе (рис. 5-2). Здесь  $0 + 1$  равно 1.

$$\begin{array}{r} 0010_1 \\ + 0011_1 \\ \hline ???_1 1_1 \end{array}$$

Рис. 5-2. Сложение младшего значащего бита двух двоичных чисел

Теперь давайте сдвинемся на один бит влево и сложим эти значения вместе, как показано на рис. 5-3.

$$\begin{array}{r} 1 \leftarrow \text{Переносим} \\ 0010 \\ + 0011 \\ \hline ?? \underline{0} 1 \end{array}$$

Рис. 5-3. Сложение позиций двоек

Как видно на рис. 5-3, эта позиция требует от нас сложения  $1 + 1$ , что представляет собой интересный поворот. В десятичной системе счисления мы представляем  $1 + 1$  символом 2, но в двоичной системе счисления у нас только два символа – 0 и 1. В двоичной системе счисления  $1 + 1$  равно 10 (объяснение см. в главе 1), что требует двух битов для представления. Мы можем поместить только один бит на эту позицию, поэтому 0 переходит в текущее место, а 1 мы переносим на следующую позицию, как показано на рис. 5-3. Теперь можем перейти на следующую позицию (см. рис. 5-4), и когда мы будем складывать эти биты, то должны включить перенесенный бит с предыдущего места. Это дает нам  $1 + 0 + 0 = 1$ .

$$\begin{array}{r}
 1 \leftarrow \text{Переносим} \\
 \begin{array}{r}
 0010 \\
 + 0011 \\
 \hline
 0101
 \end{array}
 \end{array}$$

Рис. 5-4. Сложение позиций четверок

Наконец, мы добавляем старший значащий бит, как показано на рис. 5-5.

$$\begin{array}{r}
 0010 \\
 + 0011 \\
 \hline
 0101
 \end{array}$$

Рис. 5-5. Сложение позиций восьмерок

Когда мы сложим все позиции, конечный результат в двоичном формате будет равен 0101. Один из способов проверить правильность нашей работы – просто перевести все в десятичную систему счисления, как показано на рис. 5-6.

Двоичная		Десятичная
0010		2
+ 0011		+ 3
<hr/> 0101	→	<hr/> 5

Рис. 5-6. Сложение двух двоичных чисел с последующим переводом в десятичную систему счисления

Как вы видите на рис. 5-6, наш ответ в двоичной системе (0101) совпадает с тем, что мы ожидали бы увидеть в десятичной (5). Достаточно просто!

### УПРАЖНЕНИЕ 5-1: Практика двоичного сложения

Теперь вы можете попрактиковаться в том, чему только что научились. Попробуйте решить следующие задачи на сложение:

0001 + 0010 = \_\_\_\_\_;

0011 + 0001 = \_\_\_\_\_;

0101 + 0011 = \_\_\_\_\_;

0111 + 0011 = \_\_\_\_\_.

Ответы см. в приложении А.

К счастью, сложение работает одинаково независимо от того, с каким основанием вы работаете. Единственное различие между основаниями заключается в количестве символов, которое вам доступно. Двоичное сложение особенно прямолинейно, поскольку в результате сложения на каждой позиции всегда получается ровно два выходных бита, каждый из которых имеет только два возможных значения.

**Выход 1.** Бит суммы ( $S$ ), равный 0 или 1, представляющий собой младший значащий бит результата операции сложения.

**Выход 2.** Выходной бит переноса ( $C_{out}$ ), равный 0 или 1.

## Полусумматоры

Теперь представим, что мы хотим построить цифровую схему, которая складывает одну позицию двух двоичных чисел. Вначале мы сосредоточимся на младшем значащем бите. Для сложения младших значащих битов двух чисел требуется только два двоичных входа (назовем их  $A$  и  $B$ ), а двоичными выходами являются бит суммы ( $S$ ) и выходной бит переноса ( $C_{out}$ ). Мы называем такую схему *полусумматором*. На рис. 5-7 показан символ полусумматора (англ. *half adder*,  $HA$ ).

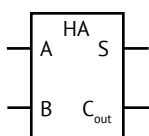


Рис. 5-7. Символ для полусумматора

Чтобы продемонстрировать, как полусумматор сочетается с нашим предыдущим примером сложения двух двоичных чисел, рис. 5-8 связывает эти два понятия.

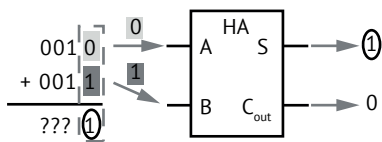


Рис. 5-8. Полусумматор в действии

Как видно на рис. 5-8, младший значащий бит первого числа – это вход  $A$ , а младший значащий бит второго числа – вход  $B$ . Сумма является выходом  $S$ , и перенос также является выходом ( $C_{out}$ ).

Полусумматор может быть реализован как комбинационная логическая схема, поэтому мы также можем описать его с помощью таблицы истинности, как показано в табл. 5-1. Обратите внимание, что  $A$  и  $B$  являются входами, а  $S$  и  $C_{out}$  – выходами.



**Таблица 5-1.** Таблица истинности для полусумматора

Входы		Выходы	
A	B	S	C <sub>out</sub>
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Давайте рассмотрим таблицу истинности, представленную в табл. 5-1. При сложении 0 и 0 получается 0 без переноса. При сложении 0 и 1 (или наоборот) получается 1 без переноса. Сложение 1 и 1 дает 0 с переносом на 1.

Теперь как же реализовать это с помощью цифровых логических вентилях? Решение будет простым, если мы будем рассматривать по одному выходу за раз, как показано на рис. 5-9.

Входы		Выходы	
A	B	S	C <sub>out</sub>
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Исключающее ИЛИ
И

Рис. 5-9. Таблица истинности для полусумматора, выходы соответствуют Исключающему ИЛИ (XOR) (для S), а также И (AND) (для C<sub>out</sub>)

Рассматривая только выход S на рис. 5-9, мы видим, что он точно соответствует таблице истинности для вентиля Исключающего ИЛИ (XOR) (см. главу 4). Если посмотреть только на C<sub>out</sub>, то можно заметить, что он соответствует выходу вентиля И (AND). Таким образом, мы можем реализовать полусумматор, используя только два вентиля: Исключающее ИЛИ (XOR) и И (AND), как показано на рис. 5-10.

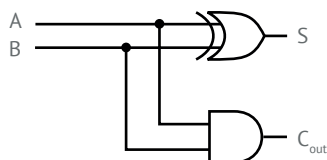


Рис. 5-10. Полусумматор, реализованный с помощью двух логических вентилях, Исключающего ИЛИ (XOR) и И (AND)

Как показано на рис. 5-10, цифровые входы А и В служат входами для вентиля Иключающего ИЛИ и вентиля И. Затем эти вентили производят желаемые выходы, S и  $C_{out}$ .

#### ПРИМЕЧАНИЕ

Обратитесь к проекту № 5 на стр. 120, где вы можете построить схему полусумматора.

## Полные сумматоры

Полусумматор может обрабатывать логику, необходимую для выполнения сложения для младших значащих битов двух двоичных чисел. Однако для каждого последующего бита требуется дополнительный вход для входного бита переноса  $C_{in}$ . Это связано с тем, что каждый бит, за исключением младшего значащего бита, должен обрабатывать ситуацию, когда в результате сложения предыдущего бита получается выходной перенос, который в свою очередь становится входным переносом для текущего бита. Добавление входа  $C_{in}$  к нашему компоненту сумматора требует новой схемы, и мы называем эту схему *полным сумматором* (англ. *full adder*, FA). Обозначение полного сумматора, представленное на рис. 5-11, похоже на обозначение полусумматора и отличается только дополнительным входом  $C_{in}$ .

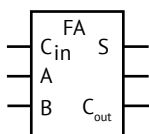


Рис. 5-11. Обозначение полного сумматора

На рис. 5-12 мы видим пример взаимосвязи между двоичным сложением одной позиции и полным сумматором.

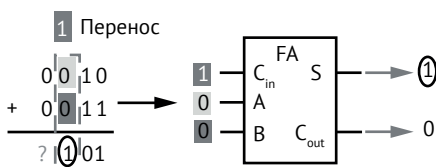


Рис. 5-12. Полный сумматор в действии

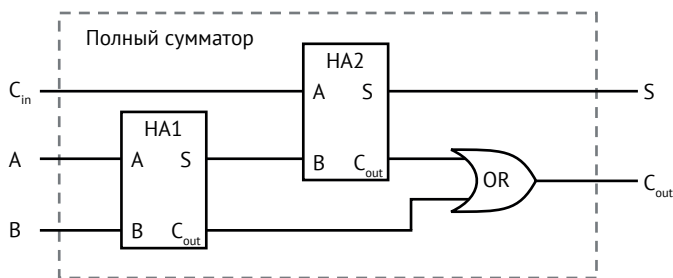
Полный сумматор обрабатывает сложение одной позиции, включая входной бит переноса. В примере, показанном на рис. 5-12, мы складываем биты на позиции четверок. Поскольку биты на предыдущем месте были 1 и 1, входной перенос равен 1. Полный сумматор принимает все три входа ( $A = 0$ ,  $B = 0$  и  $C_{in} = 1$ ) и производит на выходе  $S = 1$  и  $C_{out} = 0$ .

Для полного представления о возможных входах и выходах полного сумматора мы можем использовать таблицу истинности, представленную в табл. 5-2. Эта таблица имеет три входа ( $A$ ,  $B$ ,  $C_{in}$ ) и два выхода ( $S$ ,  $C_{out}$ ). Уделите время рассмотрению выходов при различных комбинациях входов.

**Таблица 5-2.** Таблица истинности для полного сумматора

Входы			Выходы	
A	B	$C_{in}$	S	$C_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Как же нам реализовать полный сумматор? Как следует из названия, полный сумматор может быть реализован путем объединения двух полусумматоров (оис. 5-13).



*Рис. 5-13. Схема полного сумматора, реализованная с помощью двух полусумматоров и вентиля ИЛИ*

Суммарный выход полного сумматора (S) должен быть суммой A и B (которые мы можем вычислить с помощью первого полусумматора- HA1) плюс  $C_{in}$  (который мы можем вычислить с помощью второго полусумматора HA2), как показано на рис. 5-13.

Нам также нужно, чтобы наш полный сумматор выводил выходной бит переноса. Это, оказывается, несложно реализовать, поскольку значение  $C_{out}$  полного сумматора равно 1, если перенос из любого полусумматора равен 1. Поэтому для завершения схемы полного сумматора мы можем использовать вентиль ИЛИ, как показано на рис. 5-13.

Здесь мы видим еще один пример «черного ящика». После построения такой схемы функциональность полного сумматора можно использовать без знания конкретных деталей реализации. В следующем разделе мы рассмотрим, как можно использовать полный сумматор и полусумматор для сложения чисел с несколькими битами.

## Четырехразрядный сумматор

Полный сумматор позволяет нам складывать два однобитных числа плюс входной бит переноса. Это дает нам строительный блок для создания схемы, которая может складывать двоичные числа с более чем одной позицией. Давайте теперь объединим несколько однобитных схем сумматоров, чтобы создать четырехразрядный сумматор. Используем полусумматор для младшего бита (поскольку он не требует входного переноса) и полные сумматоры для остальных битов. Идея заключается в том, чтобы соединить сумматоры вместе так, чтобы выходной перенос каждого сумматора перетекал во входной перенос последующего сумматора, как показано на рис. 5-14.

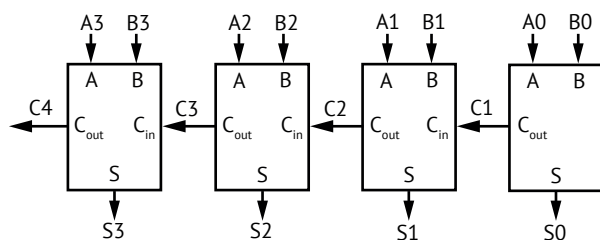


Рис. 5-14. Четырехразрядный сумматор

Для соответствия тому, как люди записывают числа, я расположил рис. 5-14 так, что младший бит находится справа, а ход диаграммы идет справа налево. Это означает, что на блок-схемах наших сумматоров входы и выходы будут расположены иначе, чем на предыдущих рисунках, так что не дайте себя запутать!

На рис. 5-15 я использовал наш предыдущий пример сложения двух (0010) и трех (0011) в четырехразрядный сумматор.

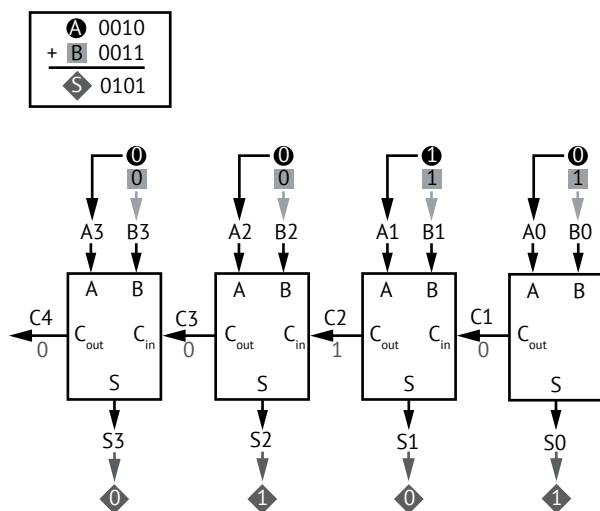


Рис. 5-15. Четырехразрядный сумматор в действии

На рис. 5-15 видно, как каждый бит со входа А (0010) и входа В (0011) поступает в каждый последующий блок сумматора, начиная с младшего бита справа и переходя к старшему биту слева.

Вы можете следить за ходом диаграммы, читая ее справа налево. Сначала складываются крайние правые биты, 0 (А0) и 1 (В0), в результате получается 1 (S0) с переносом 0.

Выходной бит переноса из крайнего правого сумматора поступает в следующий сумматор как C1, где складываются 1 (А1) и 1 (В1) вместе с переносом, равным 0. В результате получается 0 (S1) с переносом 1 (C2). Процесс продолжается до тех пор, пока не завершит действие самый левый сумматор. Конечным результатом является набор выходных битов 0101 (с S3 по S0) и перенос, равный 0 (C4). Если нам нужно обработать большее количество битов, можем расширить конструкцию на рис. 5-15, просто включив больше полных сумматоров.

Этот тип сумматора требует, чтобы биты переноса передавались по схеме. По этой причине мы называем эту схему каскадным сумматором. Каждый бит переноса, который поступает в следующий полный сумматор, вызывает небольшую задержку, поэтому расширение схемы для обработки большого количества битов замедляет схему. Выход схемы будет неточным, пока все биты переноса не успеют распространиться.

Несколько версий четырехразрядных сумматоров доступно в серии ИС 7400. Если вам нужен четырехразрядный сумматор в проекте, вы можете использовать такую микросхему, а не собирать сумматор из отдельных логических вентилях.

Давайте сделаем паузу и рассмотрим более широкое значение того, что мы только что узнали. Да, вы рассмотрели, как построить четырехразрядный сумматор, но как это связано с вычислительной техникой? Напомним, что компьютеры – это электронные устройства, которые можно запрограммировать на выполнение набора логических инструкций. Эти инструкции включают математические операции, и мы только что видели, как логические вентили, собранные из транзисторов, могут быть объединены для выполнения одной из таких операций – сложения. Мы рассмотрели сложение как конкретный пример компьютерной операции, и, хотя мы не будем вдаваться в подробности в этой книге, вы также можете реализовать другие фундаментальные компьютерные операции с помощью логических вентилях. Именно так работают компьютеры – простые логические вентили работают вместе для выполнения сложных задач.

## Знаковые числа

До сих пор в этой главе мы рассматривали только положительные целые числа, но что, если мы захотим иметь возможность работать и с отрицательными целыми числами? Сначала нам нужно рассмотреть, как отрицательное число может быть представлено в цифровой системе, такой как компьютер. Как вы знаете, все данные в компьютере представлены в виде последовательностей нулей и единиц. Отрицательный

знак не является ни 0, ни 1, поэтому нам необходимо принять соглашение для представления отрицательных значений в цифровой системе. В вычислительной технике знаковое число – это последовательность битов, которая может быть использована для представления отрицательного или положительного числа в зависимости от конкретных значений этих битов.

При проектировании цифровой системы необходимо определить, сколько битов используется для представления целого числа. Обычно для представления целых чисел используется 8, 16, 32 или 64 бита. Один из этих битов может быть назначен для представления отрицательного знака. Например, мы можем сказать, что если наибольший значащий бит равен 0, то число положительное, а если наибольший значащий бит равен 1, то число отрицательное. Оставшиеся биты тогда используются для представления абсолютного значения числа. Этот подход известен как знаковое представление величины. Это работает, но требует дополнительной сложности при проектировании системы для учета бита, имеющего особое значение. Например, схемы сумматоров, которые мы построили ранее, должны быть изменены для учета знакового бита.

Лучший способ представления отрицательных чисел в компьютере известен как дополнительный код (или дополнение до двух). В этом контексте дополнительный код числа представляет собой отрицательную часть этого числа. Самый простой способ найти дополнительный код числа – это заменить каждую 1 на 0 и каждый 0 на 1 (другими словами, перевернуть биты), а затем полученному числу прибавить 1. Потерпите немного, сначала это покажется слишком сложным, но если вы будете следить за ходом рассуждений, то все станет понятно.

Давайте рассмотрим 4-битный пример – число 5, или 0101 в двоичном исчислении. На рис. 5-16 показан процесс нахождения дополнительного кода этого числа.

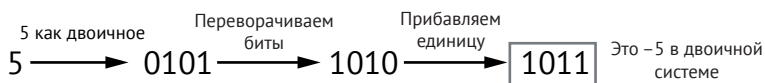


Рис. 5-16. Нахождение дополнительного кода для 0101

Сначала мы переворачиваем биты, затем прибавляем единицу, что дает нам двоичное число 1011. Таким образом, в этой системе 5 представляется как 0101, а -5 как 1011. Следует помнить, что 1011 представляет -5 только в контексте четырехбитного знакового числа. Эта двоичная последовательность может быть интерпретирована по-другому в другом контексте, как мы увидим позже. Что, если мы хотим пойти другим путем, начав с отрицательного значения? Процесс будет таким же, как показано на рис. 5-17.

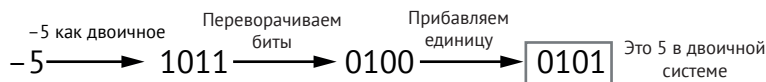


Рис. 5-17. Нахождение дополнительного кода для 1011

Как показано на рис. 5-17, применение дополнительного кода к  $-5$  возвращает нас к исходному значению 5. Это логично, учитывая, что отрицательное число к  $-5$  равно 5.

### УПРАЖНЕНИЕ 5-2: Найдите дополнительный код

Найдите четырехбитный дополнительный код для 6. Ответ см. в приложении А.

Теперь мы знаем, как представить число в виде положительного или отрицательного значения с помощью дополнительного кода, но как это может нам пригодиться? Я думаю, что самый простой способ увидеть преимущества этой системы – просто попробовать на практике. Допустим, мы хотим сложить 7 и  $-3$  (т. е. вычесть 3 из 7). Мы ожидаем, что результат будет положительным (4). Давайте сначала определим, каковы наши входы в двоичном виде, как показано на рис. 5-18.

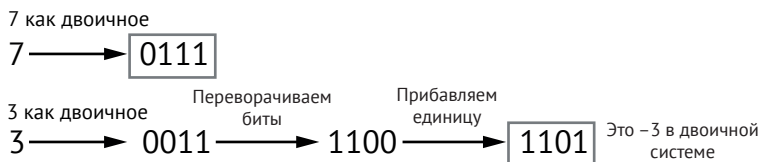


Рис. 5-18. Нахождение четырехбитных дополнительных кодов для 7 и  $-3$

Наши два двоичных входа будут 0111 и 1101. Теперь забудьте на мгновение, что мы имеем дело с положительными и отрицательными значениями. Просто сложите эти два двоичных числа. Не беспокойтесь о том, что представляют собой биты, просто сложите их и приготовьтесь удивиться! Посмотрите на рис. 5-19, когда выполните двоичную математику.

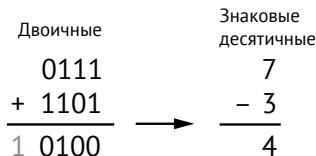


Рис. 5-19. Сложение двух двоичных чисел, интерпретируемое в знаковых десятичных числах

Как видно на рис. 5-19, в результате такого сложения появляется бит переноса, выходящий за рамки того, что может представлять четырехбитное число. Я объясню это более подробно позже, а пока мы можем игнорировать этот бит переноса. Это дает нам четырехбитный результат 0100, который является положительным 4, нашим ожидаемым числом! В этом и заключается прелесть системы с дополнительными

кодами. Нам не нужно делать ничего особенного во время операции сложения или вычитания, это работает и так.

Давайте сделаем паузу и подумаем, что из этого следует. Помните те схемы сумматоров, которые мы построили ранее? Они будут работать и для отрицательных значений! Любая схема, созданная для работы с двоичным сложением, может использовать дополнительный код как способ обращения с отрицательными числами или вычитанием. Подробное математическое объяснение того, почему все это работает, выходит за рамки данной книги, если вам интересно, в интернете можно найти хорошие объяснения.

### Терминология дополнения до двух

Термин «дополнение до двух» на самом деле относится к двум связанным понятиям. Дополнение до двух – это форма обозначения для представления целых положительных и отрицательных чисел. Например, число 5, представленное в четырехбитной системе с дополнением до двух, равно 0101, в то время как  $-5$  представлено как 1011. В то же время дополнение до двух является операцией, используемой для отрицания целого числа, хранящегося в формате дополнения до двух. Например, если взять дополнение до двух числа 0101, то получится 1011.

Вот еще один способ взглянуть на концепцию дополнительного кода (дополнения до двух): старшая позиция имеет вес, равный отрицательному значению этой позиции, а все остальные позиции имеют веса, равные положительным значениям этих позиций. Таким образом, для четырехбитного числа позиции имеют веса, показанные на рис. 5-20.

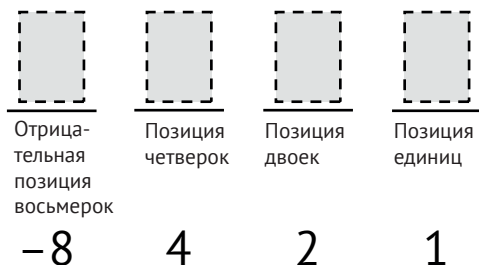


Рис. 5-20. Значения весов знакового четырехбитного числа с использованием системы дополнительных кодов

Если мы применим этот подход к представлению числа  $-3$  (1101) в дополнительном коде, то сможем вычислить десятичное значение, как показано на рис. 5-21.



1	1	0	1
Отрица- тельная позиция восьмерок	Позиция четверок	Позиция двоек	Позиция единиц

$$-8 + 4 + 0 + 1 = -3$$

Рис. 5-21. Нахождение знакового десятичного значения числа 1101 с помощью значений позиций дополнительного кода

При работе с дополнительными кодами я обнаружил, что рассматривать вес старшей позиции, как равный отрицательному значению этой позиции – это удобный короткий путь. Теперь, когда мы рассмотрели веса всех позиций в четырехбитном знаковом числе, мы можем изучить весь диапазон значений, которые могут быть представлены таким числом, как показано в табл. 5-3.

**Таблица 5-3.** Все возможные значения четырехбитного знакового числа

Двоичное	Знаковое десятичное
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Из табл. 5-3 видно, что для знакового четырехбитного числа максимальное значение равно 7, а самое большое отрицательное значение равно -8, и всего существует 16 возможных значений. Обратите внимание, что если старший бит равен 1, то значение будет отрицательным. Для  $n$ -битного знакового числа можно обобщить это следующим образом:

- максимальное значение:  $(2^{n-1}) - 1$ ;
- минимальное значение:  $-(2^{n-1})$ ;
- количество уникальных значений:  $2^n$ .

Итак, например, для восьмибитного знакового числа мы видим, что:

- максимальное значение = 127;
- минимальное значение = -128;
- количество уникальных значений = 256.

## Беззнаковые числа

Знаковые целые числа, использующие дополнительный код для представления отрицательных значений, являются удобным способом работы с отрицательными числами, не требующим специализированной конструкции сумматора. Сумматор, который мы рассмотрели ранее, работает с отрицательными значениями так же хорошо, как и с положительными. Однако в вычислениях бывают ситуации, в которых отрицательные значения просто не нужны, и обработка наших чисел как знаковых лишь тратит понапрасну примерно половину диапазона значений (все отрицательные значения остаются неиспользованными), а также ограничивает максимальное возможное значение примерно наполовину, по сравнению с тем, что могло бы быть в противном случае. Поэтому в таких ситуациях мы хотим рассматривать числа как беззнаковые, т. е. последовательность битов всегда представляет положительное значение или ноль, но никогда отрицательное значение.

Снова рассмотрим четырехбитное число в табл. 5-4, где показано, что представляет собой каждое четырехбитное двоичное значение, если мы интерпретируем его как знаковое или беззнаковое.

**Таблица 5-4.** Все возможные значения четырехбитного числа, знаковые или беззнаковые

Двоичное	Знаковое десятичное	Беззнаковое десятичное
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10

Таблица 5-4. Окончание

Двоичное	Знаковое десятичное	Беззнаковое десятичное
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Для  $n$ -битного беззнакового числа можно обобщить это следующим образом:

- максимальное значение:  $(2^n) - 1$ ;
- минимальное значение: 0;
- количество уникальных значений:  $2^n$ .

Итак, давайте возьмем пример четырехбитного значения, скажем, 1011. Если посмотреть на табл. 5-4, что оно представляет? Представляет ли оно -5 или 11? Ответ: «В зависимости от ситуации!» Оно может представлять либо -5, либо 11, в зависимости от контекста. С точки зрения схемы сумматора это не имеет значения. Для сумматора четырехбитное значение – это просто 1011. Любая операция сложения выполняется одинаково, разница лишь в том, как мы интерпретируем результат. Давайте рассмотрим пример. На рис. 5-22 мы складываем два двоичных числа: 1011 и 0010.

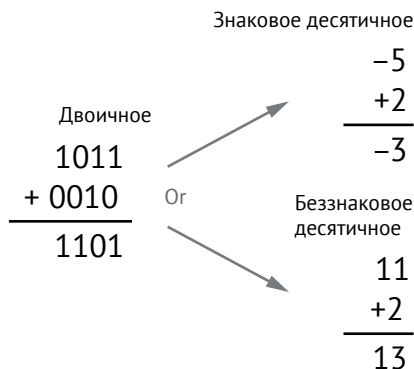


Рис. 5-22. Сложение двух двоичных чисел, интерпретированное как знаковое или беззнаковое

Как показано на рис. 5-22, при сложении этих двух двоичных чисел получается 1101 независимо от того, работаем ли мы со знаковыми или беззнаковыми числами. После завершения вычислений мы должны решить, как интерпретировать этот результат. Либо мы просто сложили -5 и 2 и получили результат -3, либо мы сложили 11 и 2 и получили результат 13. В любом случае математика работает, это просто вопрос интерпретации! В контексте вычислительных технологий за правильную интерпретацию результата операции сложения как знакового или беззнакового отвечает программа, работающая на компьютере.

### **УПРАЖНЕНИЕ 5-3: Сложите два двоичных числа и интерпретируйте их как знаковые и беззнаковые**

Сложите 1000 и 0110. Интерпретируйте вашу работу через знаковые числа. Затем интерпретируйте их как беззнаковые. Имеют ли результаты смысл? Ответ см. в приложении А.

До сих пор мы в основном игнорировали старший выходной бит переноса, но он имеет значение, которое следует понимать. Для беззнаковых чисел перенос, равный 1, означает, что произошло целочисленное переполнение. Другими словами, результат слишком велик, чтобы быть представленным количеством битов, отведенных для представления целого числа. Для знаковых чисел если старший входной бит переноса не равен старшему выходному биту переноса, то произошло переполнение. Также для знаковых чисел если старший входной бит переноса равен старшему выходному биту переноса, то переполнения не произошло, и бит переноса можно игнорировать.

Целочисленные переполнения являются источником ошибок в компьютерных программах. Если программа не проверяет, произошло ли переполнение, то результат операции сложения может быть неверно интерпретирован, что приведет к неожиданному поведению. Известный пример ошибки целочисленного переполнения встречается в аркадной игре Pac-Man. Когда игрок достигает 256-го уровня, правая часть экрана заполняется искаженной графикой. Это происходит потому, что номер уровня хранится как восьмибитное целое число без знака, и прибавление 1 к его максимальному значению 255 приводит к переполнению. Логика игры не учитывает это условие, что и приводит к возникновению сбоя.

## **Выводы**

В этой главе мы использовали сложение в качестве примера того, как компьютеры опираются на логические вентили для выполнения сложных задач. Вы узнали, как выполнять сложение в двоичном формате и как построить аппаратное обеспечение на основе логических вентилей, способное складывать двоичные числа. Вы увидели, как полусумматор может складывать 2 бита и производить сумму и выходной бит переноса, в то время как полный сумматор может складывать 2 бита плюс входной бит переноса. Мы рассмотрели, как однобитовые сумматоры могут быть объединены для выполнения многобитового сложения. Вы также узнали, как целые числа представляются в компьютере с помощью знаковых и беззнаковых чисел.

В следующей главе мы перейдем от схем комбинационной логики к изучению последовательной логики. С помощью последовательной

логики аппаратное обеспечение может иметь память, позволяющую хранить и извлекать данные. Вы увидите, как можно построить схемы памяти. Мы также рассмотрим синхросигналы – метод синхронизации состояния нескольких компонентов компьютерной системы.

## ПРОЕКТ № 5: Построение полусумматора

В этом проекте вы построите полусумматор, используя вентиль Искключающее ИЛИ (XOR) и вентиль И (AND). Входы будут управляться выключателями или кнопками. Выходы должны быть подключены к светодиодам, чтобы легко наблюдать за их состоянием. Для этого проекта вам понадобятся следующие компоненты:

- макетная плата;
- два светодиода;
- два токоограничивающих резистора для использования с вашими светодиодами (приблизительно 220 Ом);
- провода-перемычки;
- Микросхема 7408 (содержит четыре вентили И (AND));
- Микросхема 7486 (содержит четыре вентили Искключающее ИЛИ (XOR));
- две кнопки или выключатели, которые подойдут для макетной платы;
- два резистора 470 Ом;
- 5-вольтовый источник питания.

Напоминаю, если вам нужна помощь по этим темам, обратитесь к разделам «Покупка электронных компонентов» на стр. 406 и «Питание цифровых схем» на стр. 410. Для напоминания о том, как пронумерованы выводы микросхемы 7408, см. рис. 4-14. Интегральная схема 7486 ранее не рассматривалась, поэтому я привожу разводку выводов для нее на рис. 5-23.

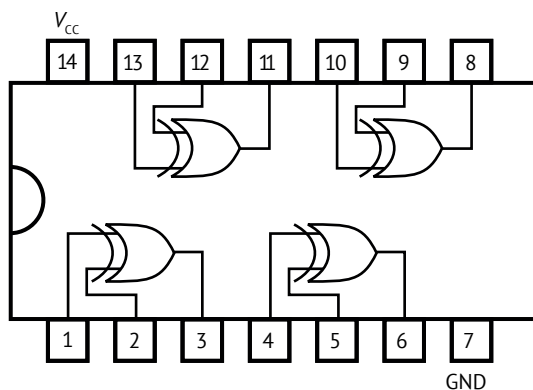


Рис. 5-23. Разводка выводов интегральной схемы 7486 Искключающее ИЛИ

На рис. 5-24 представлена схема подключения полусумматора. Более подробную информацию о том, как собрать эту схему, см. далее.

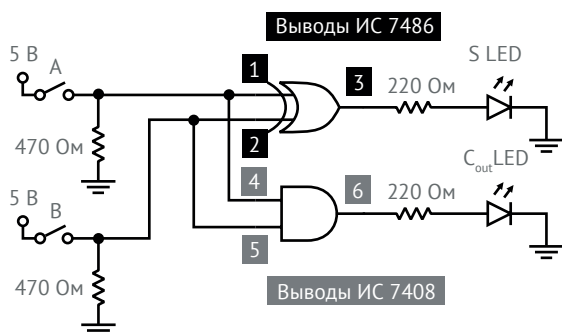


Рис. 5-24. Полусумматор, построенный из вентилей Искключающее ИЛИ и И

На рис. 5-24 показаны соединения для выключателей или кнопок с заземленными резисторами и для светодиодов с токоограничивающими резисторами. Также обратите внимание на номера выводов микросхем 7486 и 7408, показанные в квадратах. Обратите внимание на черные точки на проводах, соединяющие A и B с резисторами и ИС. Точки обозначают места соединения – например, выключатель A, резистор 470 Ом, вывод 1 микросхемы 7486 и вывод 4 микросхемы 7408 – все они соединены. Не забудьте подключить микросхемы 7486 и 7408 к 5 В и земле через контакты 14 и 7 (не показаны на рис. 5-24) соответственно.

На рис. 5-25 показано, как эта схема может выглядеть при реализации на макетной плате.

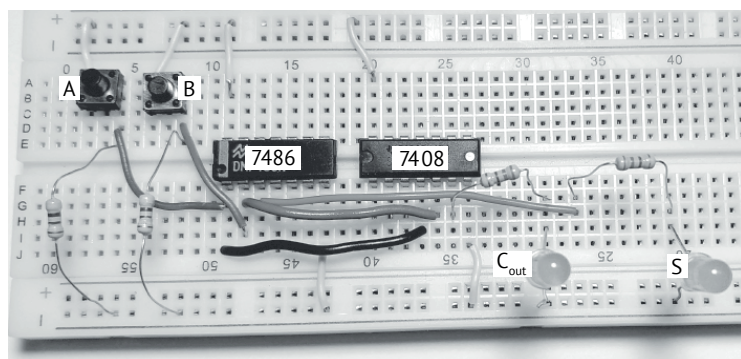
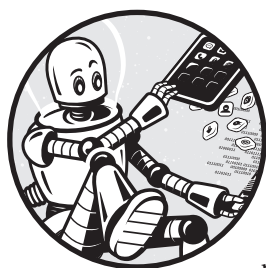


Рис. 5-25. Полусумматор, построенный из вентилей Искключающее ИЛИ и И

После построения этой схемы попробуйте все комбинации входов A и B, чтобы убедиться, что выходы соответствуют ожидаемым значениям, как показано в таблице истинности для полусумматора (табл. 5-1).

# 6

## ПАМЯТЬ И СИНХРОСИГНАЛЫ



В предыдущих главах мы видели, как цифровые логические вентили могут быть объединены для создания полезных комбинационных логических схем, в которых выход является функцией входов. В этой главе мы рассмотрим последовательные логические схемы. Эти схемы обладают памятью – способностью хранить записи о прошлом. Мы рассмотрим некоторые конкретные виды устройств памяти: одноступенчатые триггеры (*защелки*) и некоторые разновидности двухступенчатых триггеров. Мы также узнаем о синхросигналах, которые являются способом синхронизации нескольких компонентов схемы.

### Последовательные логические схемы и память

Теперь рассмотрим тип цифровой схемы, известный как *последовательная логическая схема*. Выход последовательной логической схемы зависит не только от текущего состояния входов, но и от прошлых состояний входов схемы.

Другими словами, последовательная логическая схема обладает некоторыми знаниями о своей предыдущей истории или состоянии. Цифровые устройства хранят запись о прошлом состоянии в так на-

зываемой *памяти* – компоненте, позволяющем хранить и извлекать двоичные данные.

Давайте рассмотрим простой пример последовательной логики: торговый автомат, работающий на монетах. Торговый автомат имеет как минимум два входа: гнездо для монет и кнопку выдачи. Для простоты предположим, что торговый автомат выдает только один вид товара, и этот товар стоит одну монету. Кнопка выдачи ничего не делает, пока не будет опущена монета. Если бы торговый автомат был основан на *комбинационной логике*, где состояние определяется только текущими входами, то монета должна была бы быть опущена в тот же момент, когда нажата кнопка выдачи<sup>1</sup>.

К счастью, торговые автоматы работают не так! Они имеют память, которая отслеживает, была ли опущена монета. Когда мы нажимаем кнопку выдачи, последовательная логика торгового автомата проверяет свою память, чтобы узнать, была ли ранее опущена монета. Если да, то автомат выдает товар. Мы еще будем возвращаться к этому примеру последовательной логики далее в этой главе.

Последовательная логика возможна благодаря памяти. Память хранит двоичные данные, и ее емкость измеряется в битах или байтах. Современные вычислительные устройства, такие как смартфоны, обычно имеют не менее 1 ГБ памяти. Это более 8 млрд бит! Давайте начнем с чего-то более простого: с устройством в 1 бит памяти.

## SR-защелка

*Защелка* – это тип запоминающего устройства, которое запоминает один бит. *SR-защелка* имеет два входа: S (Set, для установки) и R (Reset, для сброса), – и выход под названием Q, единственный бит, который «запоминается». Когда S устанавливается равным 1, выход Q тоже становится 1. Когда S переходит в 0, Q остается равным 1, потому что защелка запоминает предыдущий вход. В этом суть памяти – компонент запоминает предыдущий вход, даже если этот вход изменился. Когда R устанавливается равным 1, это является сигналом сброса/очистки бита памяти, поэтому выход Q становится равным 0. Q останется 0, даже если R вернется к 0.

Поведение SR-защелки представлено в табл. 6-1.

---

<sup>1</sup> Разумеется, читатель легко придумает схему такого автомата с распознаванием опущенной монеты чисто механическим способом, электронная память при этом необязательна, и электронная логика срабатывания будет чисто комбинационной. Но читатель должен учитывать, что и в таком случае память все равно присутствует, пусть и на основе механики, т. е. в целом подобный автомат должен обладать способностью запоминать факт опущенной монеты вне зависимости от конструкции. – *Прим. ред.*



**Таблица 6-1.** Работа SR-защелки

S	R	Q (выход)	Действие
0	0	Поддерживает предыдущее значение	Удержание
0	1	0	Сброс
1	0	1	Установка
1	1	X	Недопустимое состояние

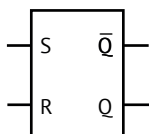


Рис. 6-1. Условное обозначение SR-защелки на схемах

На рис. 6-1 есть дополнительный выход:  $\bar{Q}$ . Читайте это как «дополнение Q», «НЕ Q» или «инверсия Q». Это просто противоположность Q. Когда Q равен 1,  $\bar{Q}$  равен 0, и наоборот. Может быть полезно иметь в наличии и Q, и  $\bar{Q}$ , и, как вы увидите, конструкция такой схемы позволяет включить этот выход без дополнительных усилий.

Мы можем реализовать SR-защелку довольно просто, используя только два вентиля НЕ-ИЛИ (NOR) и несколько проводов. Тем не менее, понимание того, как работает эта схема, требует некоторого размышления. Рассмотрим схему, показанную на рис. 6-2, которая представляет собой реализацию SR-защелки.

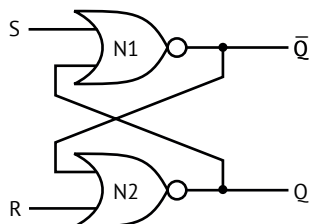


Рис. 6-2. SR-защелка, реализованная с помощью перекрестно соединенных вентилей НЕ-ИЛИ

На рис. 6-2 представлены два вентиля НЕ-ИЛИ в так называемой *конфигурации с перекрестной связью*. Напомним, что вентиль НЕ-ИЛИ выдает 1, только если оба входа равны 0; в противном случае он выдает 0. Выход вентилей N1 подается на вход вентилей N2, а выход вентилей N2 подается на вход вентилей N1. Входами являются S и R. Выходы – Q и  $\bar{Q}$ . Давайте рассмотрим, как работает схема, устанавливая и сбрасывая различные входы и проверяя выходы по ходу дела. Предположим, что первоначально S = 0, а R = 1.

**Исходное состояние ( $S = 0$ ,  $R = 1$ ).**

1.  $R = 1$ , поэтому выход вентиля N2 равен 0.
2. Выход вентиля N2 подается на вентиль N1.
3.  $S = 0$ , поэтому выход вентиля N1 равен 1.
4. Первоначально  $Q = 0$ .

Выводы: когда  $R$  имеет высокий уровень, выход  $Q$  устанавливается в низкий (см. рис. 6-3).

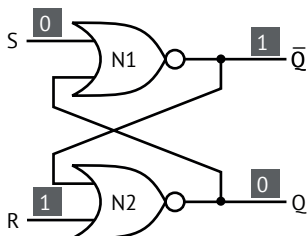


Рис. 6-3. SR-защелка, начальное состояние

**Затем очистим все входы ( $S = 0$ ,  $R = 0$ ).**

1.  $R$  переходит в 0.
2. Другой вход вентиля N2 по-прежнему равен 1, поэтому выход вентиля N2 по-прежнему равен 0.
3. Следовательно,  $Q$  по-прежнему равно 0.

Выводы: схема запомнила предыдущее состояние выхода (см. рис. 6-4).

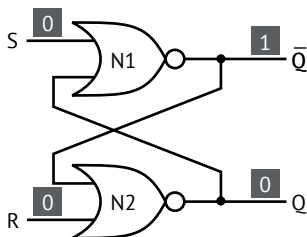


Рис. 6-4. SR-защелка, входы имеют низкие уровни

**Далее активируем вход  $S$  ( $S = 1$ ,  $R = 0$ ).**

1.  $S$  переходит в 1.
2. Это приводит к тому, что выход вентиля N1 становится равным 0.
3. Входы в N2 теперь 0 и 0, поэтому выход вентиля N2 равен 1.
4. Следовательно,  $Q$  теперь равно 1.

Выводы: установка высокого уровня на  $S$  приводит к тому, что выход  $Q$  устанавливается в высокий уровень (см. рис. 6-5).

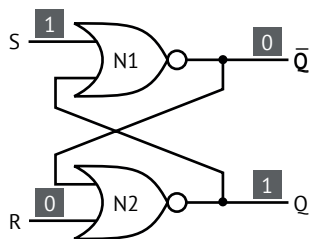


Рис. 6-5. SR-защелка,  $S$  установлен в высокий уровень

**Наконец, снова очистите все входы ( $S = 0$ ,  $R = 0$ ).**

1.  $S$  переходит в 0.
2. Другой вход вентиль N1 по-прежнему равен 1, поэтому на выходе вентиль N1 по-прежнему 0.
3. Входы вентиль N2 не изменяются.
4. Следовательно,  $Q$  по-прежнему равно 1.

Выводы: схема запомнила предыдущее состояние выхода (см. рис. 6-6).

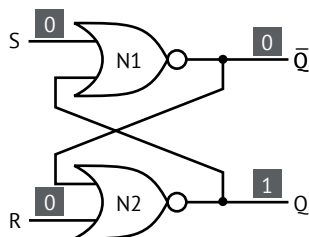


Рис. 6-6. SR-защелка,  $S$  становится низким

Соединив все это вместе, мы только что описали желаемое поведение SR-защелки, как ранее было показано в табл. 6-1. Когда  $S$  (установка) равен 1, выход  $Q$  переходит в 1 и остается в 1, даже когда  $S$  возвращается в 0. Когда  $R$  (сброс) равен 1, выход  $Q$  переходит в 0 и остается в 0, даже когда  $R$  возвращается в 0. Таким образом, схема запоминает либо 1, либо 0, поэтому мы имеем устройство с одним битом памяти! Несмотря на то что имеется два выхода ( $Q$  и  $\bar{Q}$ ), оба являются просто различными представлениями одного и того же сохраненного бита. Помните, что одновременная установка  $S = 1$  и  $R = 1$  является недопустимым состоянием на входах.

Чтобы понять поведение SR-защелки, мы рассмотрели, как схема ведет себя, когда входы удерживаются в состоянии высокого уровня, а затем переводятся в низкое состояние. Однако  $S$  и  $R$ , как правило, просто нуждаются в «импульсах». Когда схема находится в состоянии покоя, и  $S$ , и  $R$  находятся на низком уровне. Когда мы хотим изменить ее состояние, нам незачем долго удерживать  $S$  или  $R$  в высоком состоянии, просто нужно быстро перевести их в высокий уровень, а затем обратно в низкий – создать простой импульс на входе.

## Универсальные логические вентили

Мы только что продемонстрировали, как SR-защелка может быть построена с помощью вентилей НЕ-ИЛИ. На самом деле вентили НЕ-ИЛИ можно использовать для создания любой другой логической схемы, а не только SR-защелки. Вентили НЕ-ИЛИ известны как *универсальные логические вентили*, они могут быть использованы для реализации любой логической функции. То же самое справедливо и для НЕ-И.

Теперь, когда мы изучили внутреннее устройство SR-защелки, можем по желанию вернуться к использованию символа, показанного на рис. 6-1, для представления SR-защелки. Когда мы это делаем, нам больше не нужно думать о внутреннем устройстве защелки. Это еще один пример принципа «черного ящика»! Мы берем конструкцию и помещаем ее в «черный ящик», что облегчает использование этой конструкции, без беспокойства об ее внутреннем устройстве. Я нахожу полезным представить себе SR-защелку в простых терминах: это однобитное устройство памяти, состояние  $Q$  которого равно 1 или 0. Вход  $S$  устанавливает  $Q$  равным 1, а вход  $R$  сбрасывает  $Q$  до 0.

### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 6 на стр. 137, где вы можете собрать SR-защелку.*

## Использование SR-защелки в схеме

Теперь, когда у нас есть базовое устройство памяти – SR-защелка, – давайте используем ее в какой-либо схеме. Вернемся к нашему примеру с торговым автоматом и разработаем схему торгового автомата, в которой используется защелка. Схема имеет следующие требования:

- схема имеет два входа: кнопка **МОНЕТЫ** и кнопка **ПРОДАЖА**. Нажатие кнопки **МОНЕТЫ** означает опускание монеты. Нажатие кнопки **ПРОДАЖА** заставляет автомат продавать товар (схема просто включит светодиод для представления продажи товара);
- схема имеет два светодиодных выхода: **МОНЕТЫ LED** и **ПРОДАЖА LED**. **МОНЕТЫ LED** загорается, когда опускается монета. **ПРОДАЖА LED** загорается, чтобы показать, что товар продается;
- автомат не будет продавать товар, если сначала не будет опущена монета;
- для простоты предположим, что можно опустить только одну монету. Опускание дополнительных монет не изменяет состояние схемы;
- обычно после выполнения операции по продаже товара мы ожидаем, что схема перезагрузится и вернется в состояние «без монеты». Однако сначала для простоты конструкции мы откажемся от автоматического сброса в пользу ручного.

На концептуальном уровне наша торговая схема будет реализована, как показано на рис. 6-7.

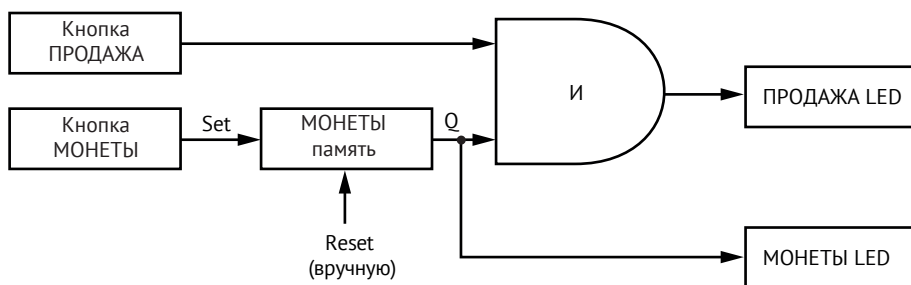


Рис. 6-7. Концептуальная схема торгового автомата с ручным сбросом

Давайте рассмотрим рис. 6-7. Когда вы нажимаете кнопку МОНЕТЫ, устройство памяти МОНЕТЫ (SR-защелка) сохраняет факт того, что была опущена монета. Затем устройство памяти выдает 1, указывая на то, что монета была вставлена, и загорается МОНЕТЫ LED. Когда вы нажимаете кнопку ПРОДАЖА, то, если ранее была вставлена монета, вентиль И выдает 1 и загорается ПРОДАЖА LED. С другой стороны, если нажать кнопку ПРОДАЖА, не опустив предварительно монету, ничего не произойдет. Чтобы выключить МОНЕТЫ LED и перезагрузить устройство, необходимо вручную установить вход Reset в 1.

#### ПРИМЕЧАНИЕ

*Смотрите проект № 7 на стр. 139, где вы сможете собрать только что описанную схему торгового автомата.*

Эта базовая схема торгового автомата демонстрирует практическое использование памяти в схеме. Поскольку наша схема включает элемент памяти, кнопка ПРОДАЖА может вести себя по-разному в зависимости от того, была ли опущена монета в прошлом. Однако, как только бит МОНЕТЫ установлен в памяти, он остается установленным до тех пор, пока схема не будет сброшена вручную. Это нехорошо, поэтому давайте переделаем нашу схему так, чтобы она автоматически сбрасывалась после выполнения операции продажи.

Как только автомат продаст товар, мы ожидаем, что бит МОНЕТЫ снова установится в 0, поскольку операция продажи «использует» монету. Другими словами, продажа товара должна привести к сбросу памяти монет. Чтобы реализовать эту логику, мы можем подключить выход вентиля И к сбросу памяти, как показано на рис. 6-8. Таким образом, когда загорается ПРОДАЖА LED, память МОНЕТЫ сбрасывается.

Система, показанная на рис. 6-8, сбросит значения во время продажи, но в этой конструкции есть проблема. Можете ли вы ее заметить? Проблема может быть неочевидной. Если вы завершили упомянутый проект, можете попробовать этот тип сброса на схеме, которую вы только что построили. Подключите провод от выхода вентиля И ко входу R в SR-защелке, нажмите МОНЕТЫ, затем нажмите ПРОДАЖА. Впереди

спойлер, поэтому не читайте дальше, пока не попробуете это сделать мысленно или на макетной плате!

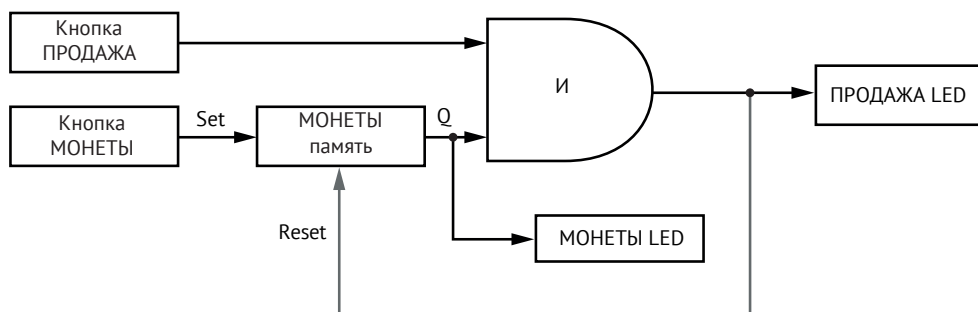


Рис. 6-8. Концептуальная схема торгового автомата с автоматическим сбросом

Проблема в том, что, хотя сброс работает, как и ожидалось, он происходит так быстро, что ПРОДАЖА LED сразу же гаснет или, что более вероятно, ПРОДАЖА LED никогда не загорается. Здесь мы имеем пример конструкции, которая технически работает, но работает так быстро, что пользователь не успевает понять, что произошло. Это довольно распространенная проблема при проектировании пользовательского интерфейса. Устройства и программы, которые мы создаем, часто работают настолько быстро, что нам приходится намеренно замедлять работу, чтобы пользователь мог успевать за ними. В этом случае решением может быть введение задержки на линии сброса, чтобы ПРОДАЖА LED успела загореться на секунду или две до того, как произойдет сброс. Это показано на рис. 6-9.

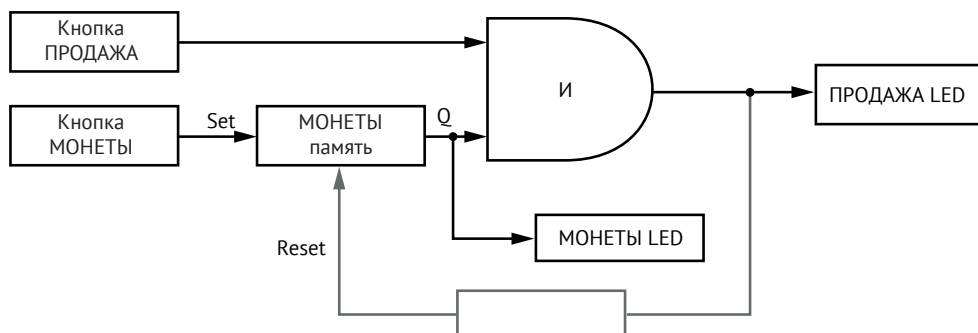


Рис. 6-9. Концептуальная схема торгового автомата с автоматической задержкой сброса

Как мы можем добавить задержку? Один из подходов заключается в использовании конденсатора. *Конденсатор* – это электрический компонент, который накапливает энергию. Он имеет два электрода. Когда ток течет к конденсатору, конденсатор заряжается. Способность конденсатора накапливать электрический заряд называется *емкостью*, которая из-

меряется в *фарадах*. Один фарад – это очень большая величина, поэтому обычно конденсаторы измеряют в *микрофарадах*, сокращенно *мкФ* ( $\mu F$ ).

Когда конденсатор не заряжен, он действует как короткое замыкание. Когда конденсатор заряжен, он действует как разомкнутая цепь. Время, необходимое для зарядки или разрядки конденсатора, зависит от величины емкости конденсатора и сопротивления в цепи. При больших значениях емкости и сопротивления конденсатору требуется больше времени для зарядки. Поэтому мы можем использовать конденсатор и резистор, чтобы ввести в нашу цепь задержку, вызванную временем, которое требуется конденсатору для зарядки.

#### ПРИМЕЧАНИЕ

Обратитесь к проекту № 8 на стр. 140, где вы можете добавить отложенный сброс в схему торгового автомата.

До сих пор в этой главе мы ограничивали наше исследование памяти однобитовыми устройствами. Хотя 1 бит памяти имеет ограниченную применимость, в главе 7 мы увидим, как можно использовать наборы однобитных ячеек памяти вместе для представления больших объемов данных.

## Синхросигналы

Чем сложнее схема, тем чаще в ней требуется обеспечить синхронизацию, т. е. одновременное изменение состояния различных элементов. Например, такое может потребоваться для схем с несколькими устройствами памяти, где мы хотим убедиться, что все хранящиеся биты будут установлены в одно и то же время. Это особенно актуально, когда нам нужно учитывать биты совместно.

Синхронизировать несколько компонентов схемы можно с помощью синхросигналов. *Синхросигнал*, он же *тактовый сигнал*, чередует высокий и низкий уровень напряжения. Как правило, сигнал чередуется с регулярным ритмом, половину времени он имеет высокий уровень, а другую половину времени – низкий. Мы называем этот тип сигнала *меандр*<sup>1</sup>. На рис. 6-10 показан меандр синхросигнала с напряжением 5 В, отображенный во времени.

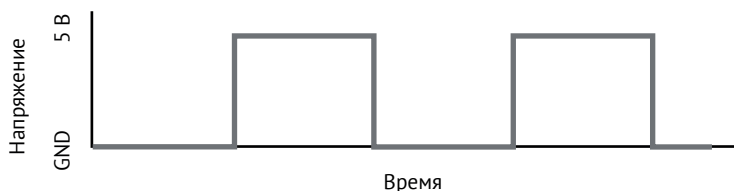


Рис. 6-10. Меандр синхросигнала с напряжением 5 В

<sup>1</sup> Вообще говоря, меандром (от названия извилистой реки Меандр в Малой Азии) называется любой сигнал с чередующимися высоким и низким уровнем. Но чаще всего под меандром без дополнительных пояснений имеют в виду именно симметричный прямоугольный сигнал (см. рис. 6-10 и 6-11), в котором длительности импульсов и пауз равны друг другу. – *Прим. ред.*

Одна итерация повышения и понижения напряжения называется *импульсом*. Полное колебание от низкого уровня к высокому и обратно к низкому (или наоборот) называется *периодом*. Мы измеряем *частоту* синхросигнала в периодах в секунду или герцах (Гц). На рис. 6-11 частота синхросигнала равна 2 Гц, поскольку сигнал совершает два полных колебания за одну секунду.

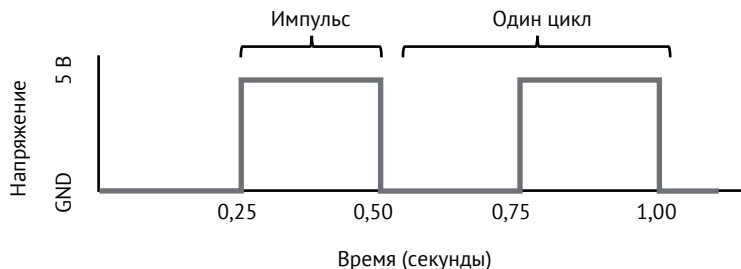


Рис. 6-11. Синхросигнал частотой 2Гц

Когда в схеме используется синхросигнал, все компоненты, которые должны быть синхронизированы, подключаются к нему. Каждый компонент спроектирован таким образом, чтобы изменение состояния происходило только при появлении тактового импульса.

Компоненты, управляемые синхросигналами, обычно вызывают изменение состояния либо на нарастающем, либо на спадающем фронте импульса. Компонент, который изменяет состояние на нарастающем фронте импульса, называется компонентом *со срабатыванием по положительному фронту*. А компонент, который изменяет состояние по спадающему фронту импульса, называется компонентом *со срабатыванием по отрицательному фронту*. На рис. 6-12 приведен пример нарастающего и спадающего фронта импульса.



Рис. 6-12. Иллюстрация фронтов импульса

На графиках в этой книге фронты импульсов изображены в виде вертикальных линий, что подразумевает мгновенное изменение состояния с низкого на высокое или наоборот. На практике, однако, для изменения состояния требуется время, но для целей нашего обсуждения давайте считать, что изменение состояния происходит мгновенно.

#### ПРИМЕЧАНИЕ

Посмотрите проект № 9 на стр. 143, где вы можете использовать SR-защелку в качестве источника ручного синхросигнала.



## JK-триггеры

Однобитовое запоминающее устройство, использующее синхросигнал, называется *динамическим триггером*. Термины «триггер-защелка» и «динамический триггер» частично пересекаются, но термин «защелка» применяется для обозначения устройств памяти без синхросигнала, а просто «триггер» – для обозначения устройств памяти с синхросигналом. В других местах вы можете встретить эти термины, используемые как взаимозаменяемые или с другим значением<sup>1</sup>.

Давайте рассмотрим конкретное устройство памяти с синхросигналом – *JK-триггер*. JK-триггер является результатом усовершенствования SR-защелки, поэтому сравним эти два устройства. SR-защелка имеет вход S для установки и вход R для сброса. Аналогично JK-триггер имеет вход J для установки и вход K для сброса. Однако SR-защелка немедленно изменяет состояние при установке высокого уровня S или R, а JK-триггер изменяет состояние только во время действия тактового импульса. JK-триггер также имеет дополнительную функцию: когда и J и K установлены на высокий уровень, выход переключается один раз с низкого на высокий или с высокого на низкий уровень. Это представлено в табл. 6-2.

**Таблица 6-2.** Сравнение SR-защелки и JK-триггера

	SR-защелка	JK-триггер
<b>Изменение состояния</b>	Немедленно, если S или R становятся высокими	Только во время тактового импульса, когда J или K высокое
<b>Установка</b>	S = 1	J = 1
<b>Сброс</b>	R = 1	K = 1
<b>Переключение</b>	Не применимо	J = 1 и K = 1

Для представления JK-триггера на диаграмме могут использоваться символы, показанные на рис. 6-13.

<sup>1</sup> Триггер-защелка (*latch*) отличается от синхронизирующегося триггера (*flip-flop*) тем, что первый срабатывает при наличии определенного уровня на управляющем входе, в то время как триггер с синхросигналом реагирует на перепад уровня (см. описание компонентов, срабатывающих по перепаду, двумя абзацами выше). В отечественной литературе защелки называют статическими триггерами, а разновидности с синхросигналом – динамическими триггерами. Так как короткого термина (аналогичного английскому *flip-flop*) для динамического триггера в русском техническом языке не сложилось, их чаще всего именуют просто триггерами, а статические триггеры – триггерами-защелками или просто защелками. Заметим, что в реальности классификация триггеров сложнее и разновидностей их гораздо больше (автор отмечает частичную взаимозаменяемость приведенных терминов), но для целей этой книги такого грубого деления достаточно. – *Прим. ред.*

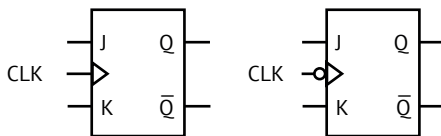


Рис. 6-13. JK-триггер со срабатыванием по положительному фронту (слева) и со срабатыванием по отрицательному фронту (справа)

На рис. 6-13 показаны две версии JK-триггера. Тот, который показан слева, является триггером со срабатыванием по положительному фронту, т. е. он изменяет состояние по нарастающему фронту тактового импульса. Справа показан символ JK-триггера со срабатыванием по отрицательному фронту (обратите внимание на кружок на входе CLK), он изменяет состояние по спадающему фронту тактового импульса. В остальном эти два устройства ведут себя одинаково.

Таким образом, JK-триггер – это однобитное устройство памяти, которое изменяет состояние только при получении тактового импульса. Он очень похож на SR-защелку, за исключением того, что синхросигнал управляет изменением его состояния, и он имеет возможность переключать свое значение. В табл. 6-3 кратко описано поведение JK-триггера.

**Таблица 6-3.** Краткое описание функциональных возможностей JK-триггера

J	K	Clock	Q (выход)	Действие
0	0	Импульс	Поддерживает предыдущее значение	Удержание
0	1	Импульс	0	Сброс (Reset)
1	0	Импульс	1	Установка (Set)
1	1	Импульс	Переворачивает предыдущее значение	Переключение

Мы не будем рассматривать JK-триггер пошагово, как это было с SR-защелкой. Вместо этого лучший способ понять JK-триггер – это поработать с ним непосредственно.

#### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 10 на стр. 143, где вы можете непосредственно поработать с JK-триггером.*

## Т-триггеры

Соединив J и K и рассматривая их как один вход, можно получить триггер, который во время тактового импульса выполняет только одно из двух действий: либо переключается, либо сохраняет свое значение. Чтобы понять, почему это так, просмотрите табл. 6-3 и обратите внимание на поведение, когда оба J и K равны 0 или оба J и K равны 1. Соединение J и K – это широко используемая техника, а триггер, который ведет себя подобным образом, называется *Т-триггером*. На рис. 6-14 справа показан символ для обозначения Т-триггера.

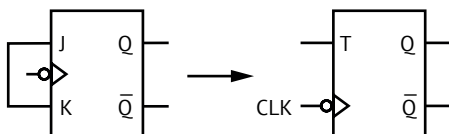


Рис. 6-14. JK-триггер с соединенными J и K, известный как T-триггер

Таким образом, T-триггер просто переключает свое значение при тактовом импульсе, когда T равно 1. В табл. 6-4 представлено поведение T-триггера<sup>1</sup>.

**Таблица 6-4.** Описание функциональных возможностей T-триггера

T	Синхросигнал	Q	Действие
0	Импульс	Сохраняет предыдущее значение	Удержание
1	Импульс	Инвертирует предыдущее значение	Переключение

## Использование синхросигнала в трехбитном счетчике

Чтобы проиллюстрировать использование синхросигнала в схеме, давайте построим трехбитный счетчик – схему, которая считает от 0 до 7 в двоичном формате. Эта схема состоит из трех триггеров, каждый из которых представляет один бит трехразрядного числа. Схема имеет вход синхросигнала (счетный вход), и, когда на него поступает тактовый импульс, трехбитное число увеличивается на 1 (инкрементируется). Поскольку все биты представляют одно число, важно, чтобы изменение их состояния происходило одновременно. Для этого и применяются T-триггеры.

Сначала посмотрите на табл. 6-5, где представлен обзор счета в двоичном формате с использованием трехбитного числа.

Таблица 6-5 представляет наше трехбитное число по одному значению в каждой строке. Давайте теперь назначим каждый из битов триггерам, обозначенным Q0, Q1 и Q2. Q0 – это наименее значимый (младший) бит, а Q2 – наиболее значимый (старший) бит, как показано в табл. 6-6.

<sup>1</sup> Фактически T-триггер представляет собой однобитный счетчик импульсов по входу CLK, в котором вход T разрешает или запрещает счет. Поэтому T-триггер часто называют просто *счетным триггером* (см. также следующий раздел). – *Прим. ред.*

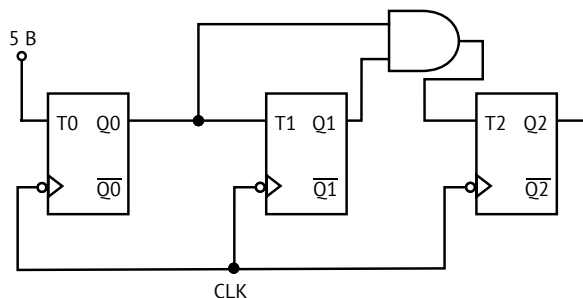
**Таблица 6-5.** Счет в двоичном формате с тремя битами

Двоичный	Десятичный
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

**Таблица 6-6.** Счет в двоичном формате, каждый бит соответствует отдельному триггеру

Все 3 бита	Q2	Q1	Q0	Десятичный
000	0	0	0	0
001	0	0	1	1
010	0	1	0	2
011	0	1	1	3
100	1	0	0	4
101	1	0	1	5
110	1	1	0	6
111	1	1	1	7

Если мы рассмотрим столбцы Q в табл. 6-6 по отдельности, то увидим, что здесь прослеживается определенная закономерность. По мере счета Q0 переключается каждый раз. Q1 переключается, если Q0 ранее был равен 1. Q2 переключается, если и Q1, и Q0 ранее были равны 1. Другими словами, кроме Q0, каждый бит переключается на следующий, если все предыдущие биты равны 1. Т-триггеры идеально подходят для реализации такого счетчика, поскольку переключение – это именно то, что они делают! Схема счетчика показана на рис. 6-15.



*Рис. 6-15. Трехбитный счетчик, построенный из Т-триггеров*

На рис. 6-15 все три Т-триггера используют один и тот же синхросигнал, поэтому они срабатывают одновременно. Разрешающий вход первого триггера Т0 подключен к высокому уровню (5 В), поэтому Q0 переключается при каждом тактовом импульсе. Второй разрешающий вход Т1 подключен к выходу Q0, поэтому тактовый импульс вызывает переключение триггера Q1 только при высоком уровне Q0. Третий разрешающий вход Т2 подключен к выходам Q0 и Q1 (через логический вентиль И), поэтому Q2 переключается при тактовом импульсе только тогда, когда Q0 и Q1 оба имеют высокий уровень.

#### ПРИМЕЧАНИЕ

*Смотрите проект № 11 на стр. 148, где вы можете построить свой собственный трехбитный счетчик.*

Подумайте, как мы могли бы использовать такой счетчик в сочетании со схемой торгового автомата, которую мы разработали ранее. Вместо того чтобы просто отслеживать, опущена монета или нет, мы можем отслеживать количество вставленных монет по крайней мере до семи! Чтобы счетчик был полезен для торгового автомата, он также должен иметь возможность сброса, поскольку при продаже товара количество монет должно обнуляться. Я не буду описывать здесь особенности добавления счетчика в схему торгового автомата, но не стесняйтесь экспериментировать самостоятельно. Схемы счетчиков, которые обнуляются и считают в обоих направлениях, можно найти в интернете, или вы можете использовать готовую микросхему счетчика, например 74191.

Мы построили счетчики из Т-триггеров, построенных из JK-триггеров, которые являются цифровыми логическими схемами на основе транзисторов! Это еще раз демонстрирует, как принцип «черного ящика» позволяет нам создавать сложные системы, скрывая детали по пути.

## Выводы

В этой главе мы рассмотрели последовательные логические схемы и синхросигналы. Вы узнали, что, в отличие от комбинационных логических схем, последовательные схемы обладают памятью, т. е. записью прошлого состояния. Вы узнали об SR-защелке, простом однобитовом устройстве памяти. Мы увидели, как синхронизация нескольких компонентов схемы, включая устройства памяти, может осуществляться с помощью тактового сигнала (синхросигнала) – электрического сигнала, значение напряжения которого постоянно меняется с высокого на низкое и наоборот. Однобитовое устройство памяти с синхросигналом известно как триггер, который позволяет изменять состояние только в синхронизации с тактовым сигналом. Вы узнали, как работают JK-триггеры, как Т-триггеры могут быть построены на основе JK-триггеров и, наконец, как синхросигнал и Т-триггеры могут быть использованы вместе для создания трехбитного счетчика тактовых импульсов.

Память и тактовые сигналы являются ключевыми компонентами современных вычислительных устройств, и в следующей главе мы увидим, какую роль они играют в современных компьютерах. Там вы узнаете об аппаратном обеспечении компьютера – памяти, процессоре и устройствах ввода/вывода.

## **ПРОЕКТ № 6: Построение SR-защелки с использованием вентилей НЕ-ИЛИ**

В этом проекте вы построите SR-защелку на макетной плате. Выход Q будет подключен к светодиоду, чтобы легко наблюдать за состоянием защелки. Вы должны протестировать установку S и R в высокий и низкий уровни и понаблюдать за выходом.

Для этого проекта вам понадобятся следующие компоненты:

- макетная плата;
- светодиод (LED);
- токоограничивающий резистор для светодиода (приблизительно 220 Ом);
- провода-перемычки;
- микросхема 7402 (содержит четыре вентилей НЕ-ИЛИ);
- 5-вольтовый источник питания;
- два резистора по 470 Ом;
- два переключателя или кнопки, подходящие для макетной платы;
- дополнительно: дополнительный резистор 220 Ом и еще один светодиод.

Напоминаю, если вам нужна помощь по этим темам, обратитесь к разделам «Покупка электронных компонентов» на стр. 406 и «Питание цифровых схем» на стр. 410. Также просмотрите проект № 4 на стр. 98, чтобы вспомнить, как использовать кнопки/переключатели с заземляющими резисторами. Подключите ваши компоненты, как показано на рис. 6-16, чтобы собрать SR-защелку. Обратите внимание, что вентили НЕ-ИЛИ расположены у микросхемы 7402 иначе, чем в других ИС, таких как 7408 (вентили И) и 7432 (вентили ИЛИ). Убедитесь, что используете правильные выводы для входов и выходов.

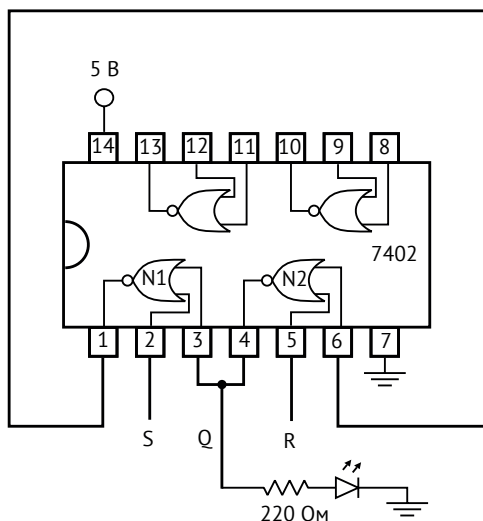


Рис. 6-16. Схема подключения SR-защелки, построенной на базе интегральной схемы 7402

После того как вы собрали схему SR-защелки, как показано на рис. 6-16, подключите S и R к кнопкам (или переключателям) с заземляющими резисторами, как показано на рис. 6-17. Это позволит легко устанавливать значение S или R простым нажатием кнопки.

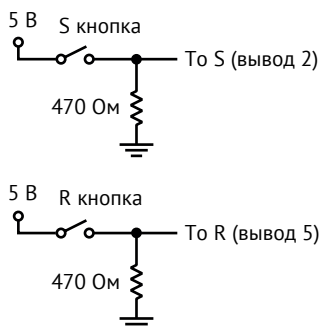


Рис. 6-17. Использование кнопок и заземляющих резисторов для управления входами S и R

После подключения кнопок к SR-защелке попробуйте установить S или R в высокий или низкий уровень, нажимая и отпуская кнопку. Понаблюдайте за результатами. Включается ли Q при нажатии на S, и остается ли он включенным даже после того, как вы отпустите S? Выключается ли Q, когда вы нажимаете R, и остается ли он выключенным даже после того, как вы отпустите R? Если вы хотите также увидеть значение  $\bar{Q}$ , которое всегда должно быть противоположным Q, просто подключите еще один резистор 220 Ом и еще один светодиод к соединенным контактам 1 и 6 микросхемы.

При первоначальной подаче питания выход будет находиться в непредсказуемом состоянии. То есть схема может иметь после запуска либо  $Q = 0$ , либо  $Q = 1$ . Или, может быть, ваша схема выдает после запуска определенное значение  $Q$ . Причина непредсказуемости заключается в том, что такая конструкция приводит к состоянию гонки. Если  $S = 0$  и  $R = 0$  при подаче питания, то оба вентиля N1 и N2 пытаются установить на выходе единицу. Один из них по случайным причинам делает это немного быстрее (поэтому гонка). Если вентиль N1 устанавливается в единицу первым, N2 переходит в низкий уровень, а  $Q$  равен 0. Если устанавливается в единицу первым вентиль N2, то в низкий уровень переходит N1, и  $Q$  равен 1. Эту проблему можно решить, удерживая кнопку R нажатой во время запуска (чтобы принудительно сделать  $Q = 0$ ), а затем отпустив кнопку R после запуска.

Сохраните эту схему, мы будем использовать ее в следующем проекте.

## ПРОЕКТ № 7: Построение базовой схемы торгового автомата

В этом проекте вы построите схему торгового автомата, описанную ранее в этой главе. В качестве блока памяти можно использовать SR-защелку из предыдущего проекта. Обязательно используйте токоограничивающие резисторы для светодиодов и заземляющие резисторы для входов кнопок. Протестируйте схему, чтобы убедиться, что она работает так, как ожидалось. Чтобы сбросить схему, нажмите кнопку R на SR-защелке.

Для этого проекта вам понадобятся следующие компоненты:

- SR-защелка 7402 на макетной плате, которую вы собрали в проекте № 6;
- дополнительный светодиод с токоограничивающим резистором (приблизительно 220 Ом);
- провода-перемычки;
- микросхема 7408 (содержит четыре вентиля И);
- дополнительная кнопка или переключатель, который подойдет для макетной платы;
- дополнительный заземляющий резистор для использования с кнопкой (приблизительно 470 Ом).

Напоминаю, если вам нужна помощь по этим темам, обратитесь к разделам «Покупка электронных компонентов» на стр. 406 и «Питание цифровых схем» на стр. 410.

На схеме, показанной на рис. 6-18, номера выводов микросхемы указаны в квадратах. Хотя это и не показано на схеме, убедитесь, что микросхемы 7402 и 7408 подключены к 5В и к земле (контакты 14 и 7 соответственно).



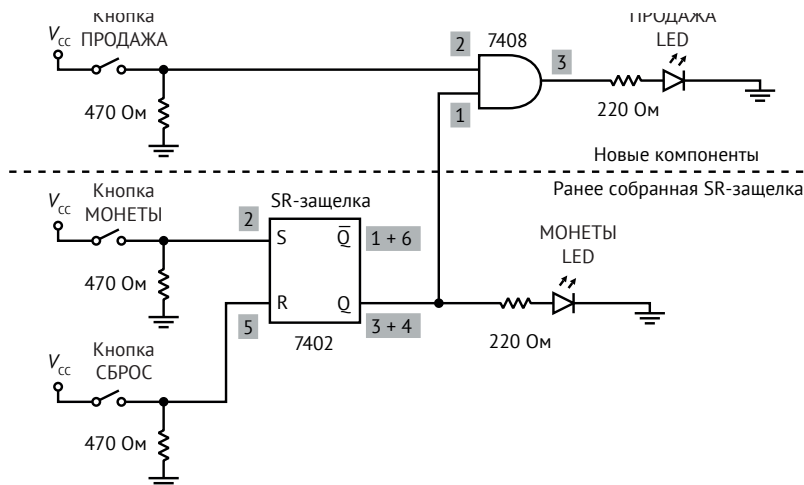


Рис. 6-18. Схема подключения базовой цепи торгового автомата

Нижняя часть рис. 6-18 – это схема, которую вы собрали в предыдущем проекте. Разница лишь в том, что теперь кнопка S представляет собой кнопку МОНЕТЫ, а выходной светодиод Q теперь представляет собой светодиод МОНЕТЫ LED. Чтобы собрать полную схему, вам нужно добавить только верхнюю часть схемы и соединить две части вместе, как показано на рисунке.

Когда схема будет собрана, вы должны увидеть, что при нажатии кнопки МОНЕТЫ загорается светодиод МОНЕТЫ LED. При нажатии кнопки ПРОДАЖА должна загораться ПРОДАЖА LED, но только если светодиод МОНЕТЫ уже горит. Нажмите кнопку СБРОС, чтобы сбросить схему.

Сохраните эту схему, мы будем использовать ее в следующем проекте.

## ПРОЕКТ № 8: Добавление отложенного сброса в схему торгового автомата

В этом проекте вы добавите отложенный сброс в схему торгового автомата из проекта № 7. Вам понадобятся следующие компоненты:

- схема торгового автомата, которую вы собрали в проекте № 7;
- резистор 4,7 кОм;
- электролитический конденсатор 220 мкФ;
- провода-перемычки.

Существует несколько типов конденсаторов, но их обсуждение выходит за рамки данной книги. Для этого проекта вы будете использовать электролитический конденсатор (рис. 6-19). При подключении конденсатора обратите внимание, что электролитические конденсаторы поляризованы,

т. е. один вывод отрицательный, а другой – положительный. Ищите отрицательный знак или стрелку, указывающую на отрицательный вывод. Иногда отрицательный вывод короче. Согласно рис. 6-21 отрицательный вывод должен быть соединен с землей.



Рис. 6-19. Электролитический конденсатор. Более короткий вывод с полоской/стрелкой – это отрицательный вывод

На рис. 6-20 показаны условные обозначения конденсаторов. Слева находится символ неполяризованного конденсатора. В середине и справа находятся символы, используемые для обозначения поляризованных конденсаторов. Оба поляризованных символа позволяют идентифицировать положительные и отрицательные выводы конденсатора.

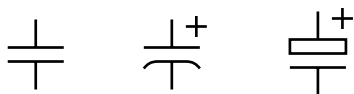


Рис. 6-20. Условные обозначения конденсаторов в электрической схеме

На рис. 6-21 показано, как можно добавить отложенный сброс на основе конденсатора в схему торгового автомата, заменив ручной сброс. Прочитайте текст после рисунка, чтобы получить более четкое представление о том, как собрать эту схему.

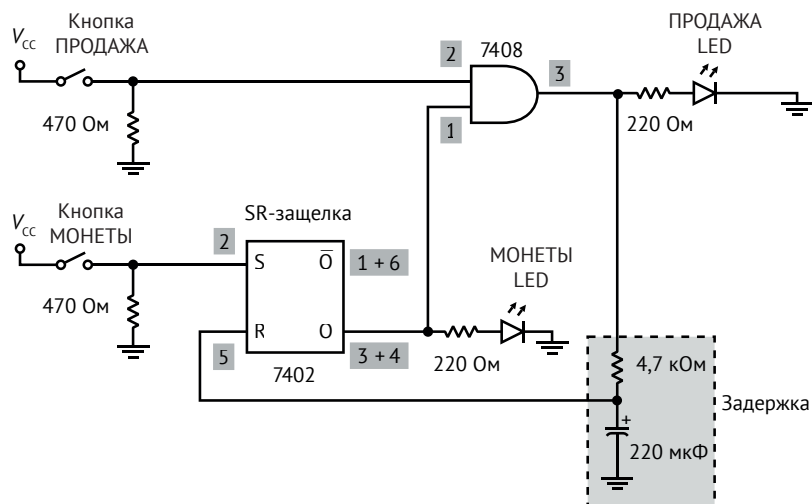


Рис. 6-21. Схема подключения торгового автомата с отложенным сбросом

Если у вас все еще ручной переключатель или кнопка сброса подключены к R (вывод 5 на чипе 7402), обязательно отсоедините их, так как они будут мешать работе отложенного сброса. На рис. 6-21 обратите внимание, что выход ПРОДАЖА нашей схемы (вывод 3 чипа 7408), который становится высоким, когда происходит продажа, подключен ко входу сброса защелки через новый компонент-задержку. Этот новый компонент состоит из резистора и конденсатора, которые вместе создают задержку сброса примерно на 1 с. Давайте рассмотрим, что здесь происходит.

1. Когда происходит операция продажи, на выходе вентиля И чипа 7408 повышается уровень.
2. Незаряженный конденсатор сначала действует как короткое замыкание на землю, и сброс R на защелке остается низким, поэтому сначала сброса не происходит.
3. Поскольку сброса еще не произошло, ПРОДАЖА LED имеет возможность загореться.
4. Если удерживать кнопку ПРОДАЖА нажатой, на выходе И остается высокий уровень, и конденсатор начинает заряжаться.
5. Примерно через 1 с конденсатор достаточно заряжен и действует как разомкнутая цепь, эффективно убирая соединение с землей.
6. Вход сброса R на защелке становится высоким, и происходит сброс.

Несколько моментов, которые следует отметить в этой конструкции:

- кнопку ПРОДАЖА необходимо удерживать нажатой, чтобы дать конденсатору время зарядиться;
- схему сброса можно запустить, только когда светодиод МОНЕТЫ LED уже горит. Для сброса достаточно удерживать кнопку ПРОДАЖА. Эту проблему можно решить с помощью схемы начального сброса, но это выходит за рамки данного проекта;
- если добавление компонента сброса приводит к тому, что вся торговая схема ничего не делает, на входе R, вероятно, остался высокий уровень. Проверьте напряжение на выводе 5 микросхемы 7402, чтобы посмотреть, не является ли оно высоким (любое значение выше 0,8 В), тогда как должно быть низким. Если вы столкнулись с этой проблемой, перепроверьте значения резистора 4,7 кОм и конденсатора 220 мкФ. Также проверьте вашу схему подключения, так как ослабленное соединение или перемычка в неправильном ряду могут все перепутать;
- я выбрал такие значения емкости и сопротивления, потому что они дают задержку около 1 с. Вы можете использовать другие значения. Однако, изменяя эти значения, вы рискуете получить слишком высокое напряжение на входе R, когда оно должно быть низким, как уже отмечалось<sup>1</sup>.

<sup>1</sup> В схеме по рис. 6-22 конденсатор 220 мкФ разряжается при нулевом уровне на выходе элемента И (ПРОДАЖА LED). Разряд идет через тот же резистор 4,7 кОм, т. е. длится столько же времени, сколько протекал заряд. Если постоянно времени выбрать слишком большой (велико сопротивление рези-

Готовая схема должна выглядеть примерно так, как показано на рис. 6-22, хотя ваша конкретная схема, вероятно, будет отличаться.

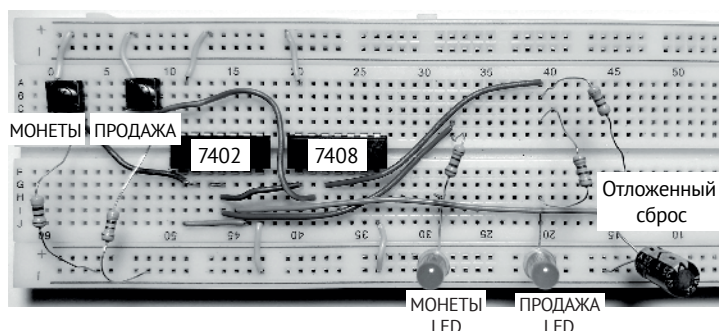


Рис. 6-22. Схема торгового автомата с отложенным сбросом на макетной плате

Я рекомендую вам сохранить часть схемы с SR-защелкой, так как вы будете использовать ее в следующих проектах. Вы можете удалить другие компоненты с платы, но сохраните ту часть, что относится к проекту № 6. Или можете просто собрать еще одну SR-защелку, когда вам понадобится.

## ПРОЕКТ № 9: Использование защелки в качестве ручного синхросигнала

Для последующих проектов в этой главе вам понадобится синхросигнал. В этом проекте вы переделаете ранее созданную SR-защелку в ручной синхросигнал.

Как вы узнали ранее, у синхросигнала должно чередоваться высокое и низкое напряжение. Вы можете попытаться реализовать синхросигнал, перемещая провод между землей и 5 В. Это, конечно, заставит значение напряжения чередоваться, но не так, как нужно. При перемещении провода он некоторое время не будет ни к чему подключен. В эти моменты напряжение на входном выводе для синхросигнала будет «плавать», и вы получите непредсказуемое поведение схемы. Это не лучший вариант.

Или вы можете добавить генератор, который будет автоматически выдавать импульсы с регулярной частотой, скажем, один импульс в секунду. Именно так обычно реализованы тактовые сигналы в реальном мире. Для

стора и/или емкость конденсатора), то он может не успеть разрядиться до уровня логического нуля к моменту, когда вы захотите опять запустить схему в работу. Выход из этой ситуации возможен добавлением ключа для принудительного сброса конденсатора, однако это еще больше усложнит схему. – Прим. ред.

этой цели разработана, например, микросхема 555, называемая таймером. Однако для предстоящих упражнений хорошо иметь возможность внимательно наблюдать за изменениями состояния ваших схем, поэтому вам нужен синхросигнал, управляемый вручную, т. е. синхросигнал, который переходит в высокий или низкий уровень только по вашему указанию. В некотором смысле такой ручной синхросигнал не совсем обычный синхросигнал, потому что он не чередует состояния с регулярной частотой. Тем не менее, является это технически тактовым сигналом или нет, не так уж важно – нам нужно устройство, которое можно использовать для ручного запуска изменений состояния.

У вас может возникнуть соблазн попробовать использовать в качестве синхросигнала обычную кнопку и заземляющий резистор, как показано на рис. 6-23. В конце концов, при нажатии кнопки напряжение повышается, а при отпускании кнопки напряжение понижается.

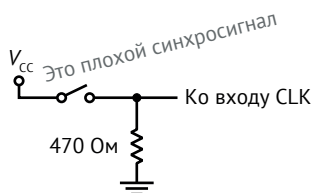


Рис. 6-23. Простой переключатель с заземляющим резистором в качестве входа CLK (будет работать не очень хорошо)

К сожалению, конструкция на рис. 6-23 на самом деле представляет собой очень плохой ручной синхросигнал. Проблема в том, что механические кнопки и переключатели имеют тенденцию «дребезжать». Внутри переключателя есть металлические контакты, которые соединяются, когда переключатель замкнут. При замыкании переключателя происходит первоначальное соединение между контактами, но затем контакты разъединяются и снова соединяются, иногда несколько раз, прежде чем переключатель окончательно перейдет в замкнутое состояние. То же самое происходит, когда переключатель размыкается, только в обратном порядке. Простое нажатие кнопки или щелчок выключателя приводит к тому, что напряжение многократно скачет от высокого к низкому. Это называется *дребезгом переключателя*, как представлено на рис. 6-24.

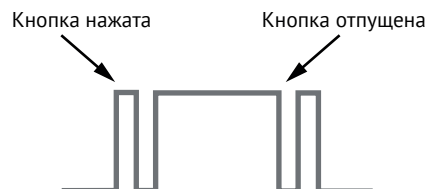


Рис. 6-24. Дребезг переключателя не то, что нам нужно в синхросигнале

Схемы подавления дребезга – это аппаратные средства для устранения дребезга. Одна из таких схем подавления дребезга основана на SR-защелке, которую вы уже собрали! Если подключить S и R к переключателям, то входы защелки все равно будут дребезжать, но выход защелки (Q) будет сохранять свое значение, как показано на рис. 6-25. Это эффективный способ устранения дребезга переключателя.

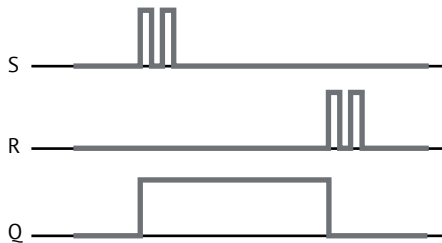


Рис. 6-25. SR-защелка выдает чистый выходной сигнал, даже когда на ее входах есть дребезг

Чтобы использовать SR-защелку в качестве источника синхросигнала, нажмите S, чтобы установить высокий уровень тактового сигнала, а затем – R, чтобы установить низкий уровень тактового сигнала. Только не нажимайте обе кнопки одновременно! В качестве синхросигнала можно использовать SR-защелку, созданную в проекте № 6. Если вы ранее отсоединяли кнопку/переключатель сброса от вывода 5 в рамках проекта № 8, подключите ее снова. Полная SR-защелка в качестве ручного синхросигнала должна быть подключена, как показано на рис. 6-26.

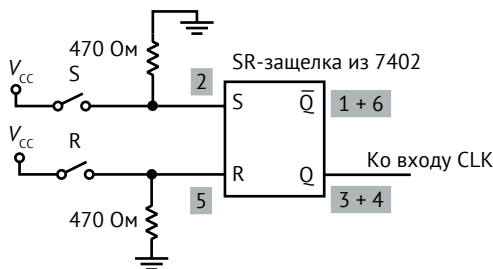


Рис. 6-26. Ручной синхросигнал с подавлением дребезга, созданный из двух кнопок/переключателей и SR-защелки

Нажмите S, чтобы установить высокий уровень тактового импульса, и нажмите R, чтобы установить низкий уровень тактового импульса. Теперь у вас есть ручной синхросигнал, который вы можете использовать в следующих проектах.

## ПРОЕКТ № 10: Тестирование JK-триггера

Хотя можно самостоятельно собрать JK-триггер из простых вентилях, он продается в виде готовой интегральной схемы, поэтому вы можете избавить себя от некоторых проблем. Микросхема 7473 содержит два JK-триггера, срабатывающих по отрицательному фронту импульса. В данном проекте будем использовать эту интегральную схему для проверки функциональности одного JK-триггера. Вы попытаетесь установить J и K на высокий и низкий уровень, а затем пропустите через схему тактовый импульс. Подключите светодиод к выходу Q, чтобы наглядно видеть изменение состояния.

Для этого проекта вам понадобятся следующие компоненты:

- SR-защелка, сконфигурированная как источник синхросигнала (обсуждалось в проекте № 9);
- микросхема 7473 (содержит два JK-триггера);
- провода-перемычки;
- светодиод с токоограничивающим резистором (приблизительно 220 Ом).

Напоминаю, если вам нужна помощь по этим темам, обратитесь к разделам «Покупка электронных компонентов» на стр. 406 и «Питание цифровых схем» на стр. 410.

На рис. 6-27 показана разводка выводов микросхемы 7473.

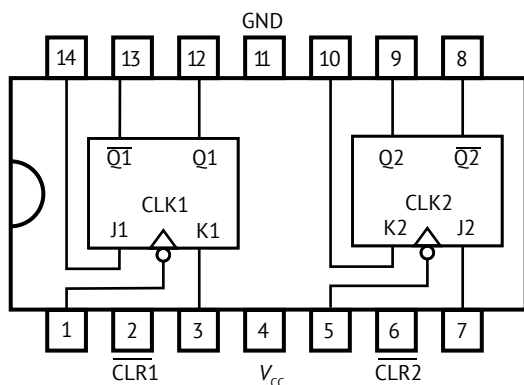


Рис. 6-27. Цоколевка микросхемы 7473

Микросхема 7473 содержит два JK-триггера, как показано на рис. 6-27. Обратите внимание, что подключение напряжения питания и земли производится не в «обычных» местах, а к выводам 4 и 11 соответственно. Также обратите внимание, что входы CLK (синхросигнала) отмечены кружком, что указывает на то, что эта схема срабатывает по отрицательному фронту, т. е. изменение состояния ожидается во время падения тактового импульса. Поскольку вы используете SR-защелку для подачи синхросигнала вручную, это означает, что вы увидите изменение состояния JK-триггера при нажатии на кнопку входа R SR-защелки.

До сих пор не был упомянут дополнительный вход очистки для каждого JK-триггера: CLR. Когда на этом выводе уровень напряжения низкий, триггер очищает сохраненный бит ( $Q = 0$ ). CLR является асинхронным, т. е. он не ждет тактового импульса. Черта над CLR означает, что он активно низкий, т. е. сохраненный бит очищается, когда на входе устанавливается низкий уровень. CLR также иногда называют Сбросом (Reset) или R, не путайте со входом R нашей SR-защелки. Подключите вход CLR (вывод 2) JK-триггера к 5 В, чтобы предотвратить сброс используемого триггера. Для тестирования одного триггера можно подключить микросхему, как показано на рис. 6-28<sup>1</sup>.

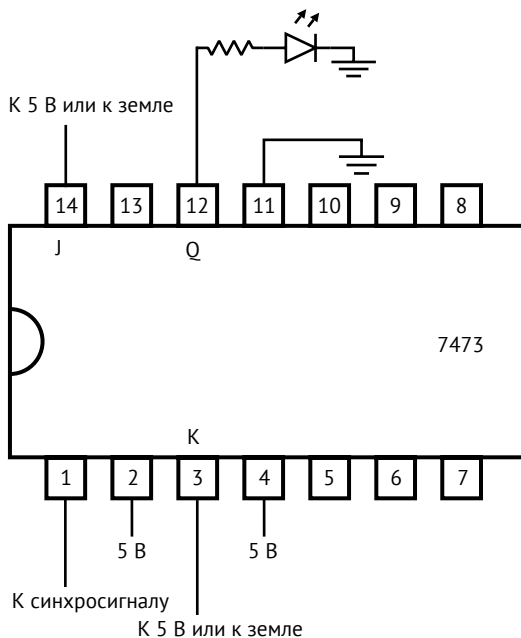


Рисунок 6-28. Простая схема проверки JK

Используя ранее созданную SR-защелку в качестве синхросигнала, подключите выход Q SR-защелки (выводы 3 и 4 на ИС 7402) ко входу синхросигнала

<sup>1</sup> Вход установки в нулевое состояние R в RS-триггере делает ровно то же самое, что рассматриваемый вход обнуления CLR, причем также является асинхронным и никакой путаницы тут нет. Поэтому и принято одно и то же обозначение (R или  $\bar{R}$ , в зависимости от активного уровня, от слова *Reset*) для входа сброса/обнуления у большинства цифровых схем (включая и микропроцессоры). Такое обозначение вы можете встретить в документации на микросхемы. Точно так же входы установки в predetermined состояние в большинстве цифровых схем обозначаются буквой S (или  $\bar{S}$ , от слова *Set*), потому что они делают ту же операцию, что S-вход RS-триггера.

По причине независимости от синхросигнала (см. далее) вывод сброса у счетчиков называют асинхронным входом. Кроме входа сброса Reset, сбрасывающего все разряды в ноль, у многих счетчиков в интегральном исполнении имеется еще ряд асинхронных входов – например, разрешающий или запрещающий счет, он обычно обозначается буквой E (или  $\bar{E}$ , от слова *Enable*). – Прим. ред.



7473 (вывод 1). Теперь попробуйте подключить входы J (вывод 14) и K (вывод 3) на 7473 к 5 В или к земле. Вы должны увидеть, что это не оказывает никакого влияния на выходной светодиод JK-триггера, пока синхросигнал не перейдет с высокого уровня на низкий. Напоминание: подайте импульс на вход синхросигнала JK-триггера, нажав у SR-защелки кнопку S, а затем кнопку R, чтобы установить высокий, а затем низкий уровень тактового сигнала. Вернитесь к табл. 6-3, чтобы увидеть ожидаемую функциональность JK-триггера, и убедитесь, что ваша схема работает в соответствии с ожиданиями.

Сохраните эту схему в таком виде для следующего проекта.

## ПРОЕКТ № 11: Построение трехбитного счетчика

В этом проекте вы соберете трехбитный счетчик, описанный ранее в этой главе. Подключите выходы Q к светодиодам, чтобы легко наблюдать за результатом.

Для этого проекта вам понадобятся следующие компоненты:

- схема, собранная в проекте № 10 (включая синхросигнал из проекта № 9);
- дополнительная микросхема 7473;
- микросхема 7408 (содержит четыре вентиля И);
- резистор 47 кОм;
- электролитический конденсатор 10 мкФ;
- дополнительная кнопка или переключатель;
- провода перемычки;
- два дополнительных светодиода;
- два дополнительных токоограничивающих резистора для использования с вашими светодиодами (примерно 220 Ом каждый).

Подключите все, как показано на рис. 6-29. Номера выводов на микросхемах показаны в квадратах.

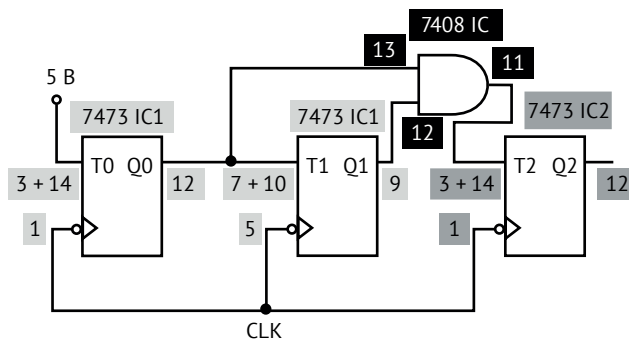


Рис. 6-29. Трехбитный счетчик, собранный из T-триггеров, с номерами выводов

В дополнение к соединениям выводов, показанным на рис. 6-29, обязательно выполните следующие соединения:

- обе микросхемы 7473 должны иметь выводы 4 и 11, подключенные к 5 В и земле соответственно;
- у 7408 вывод 7 должен быть подключен к земле, а вывод 14 – к 5 В;
- выходы Q0, Q1 и Q2 должны быть подключены к светодиодам (через резисторы 220 Ом), чтобы можно было видеть изменение состояния битов;
- выход ручного синхросигнала (выводы 3 и 4 на 7402) должен быть подключен к CLK на всех трех триггерах (выводы 1 и 5 на первых 7473 и вывод 1 на втором 7473).

Эта схема запускается в непредсказуемом состоянии. Вы можете исправить это, сбросив все три триггера вручную, но это утомительно. Вместо этого добавьте схему начального сброса, которая обеспечит запуск всех триггеров с выходом = 0. Каждый триггер в корпусе 7473 имеет вход CLR, который при удержании низкого уровня сбрасывает триггер независимо от состояния синхросигнала. Нам нужно, чтобы при запуске CLR на короткое время переходил в низкий уровень, а затем в высокий и оставался там. Это гарантирует, что при включении счетчик начнет с нуля. Для надежности можно также добавить кнопку СБРОС СЧЕТЧИКА, которая при нажатии сбрасывает счетчик вручную. Эта возможность сброса показана на рис. 6-30.

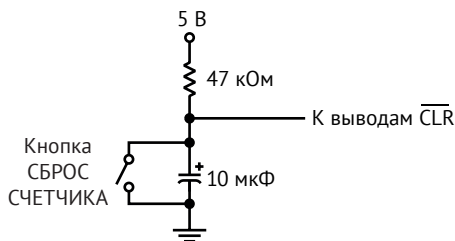


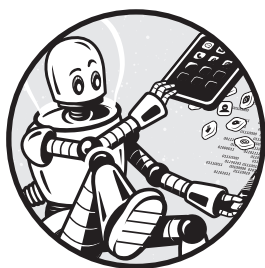
Рис. 6-30. Схема начального сброса для трехбитного двоичного счетчика

При первичной подаче питания на схему, показанную на рис. 6-30, конденсатор действует как короткое замыкание, и CLR удерживается на низком уровне, приводя схему в исходное состояние. После зарядки конденсатора он работает как разомкнутая цепь, CLR переходит на высокий уровень, и схема готова к работе. Кнопка или переключатель СБРОС СЧЕТЧИКА при нажатии также заставляет CLR переходить на низкий уровень и сбрасывать схему. Эта схема должна быть подключена к входам CLR: к выводам 2 и 6 первой микросхемы 7473 и к выводу 2 второй микросхемы 7473. В проекте № 10 вывод 2 первой микросхемы 7473 был подключен к 5 В, потому что перед подключением схемы начального сброса убедитесь, что вы отключили его. Не забудьте правильно сориентировать выводы электролитического конденсатора – отрицательный вывод должен быть соединен с землей.

После начального сброса схема должна запускаться со счетчиком, установленным в 000. Каждая подача тактового импульса на схему должна привести к увеличению счетчика на 1 на спадающем фронте импульса. Напоминание: подайте тактовый импульс на SR-защелку, нажав S, чтобы установить высокий уровень тактового сигнала, и R, чтобы установить низкий уровень тактового сигнала. Протестируйте счет от 000 до 111 и убедитесь, что работа счетчика соответствует ожиданиям.

# 7

## АППАРАТНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРА



В предыдущих главах были рассмотрены основополагающие элементы вычислительной техники – двоичный код, цифровые схемы, память. Теперь давайте рассмотрим, как эти элементы объединяются в компьютере – устройстве, которое представляет собой нечто большее, чем просто сумма его частей. В этой главе я сначала сделаю обзор аппаратного обеспечения компьютера. Затем мы более детально рассмотрим три такие части компьютера, как оперативная память, процессор и устройства ввода/вывода.

### Обзор аппаратного обеспечения компьютера

Давайте начнем с обзора того, чем компьютер отличается от других электронных устройств. Ранее мы уже видели, как можно использовать логические схемы и устройства памяти для создания схем, выполняющих полезные задачи. Схемы, которые мы строили с помощью логических вентилях, имеют набор функций, жестко заложенных в их конструкцию. Если мы хотим добавить или изменить какую-либо функцию, мы должны изменить физическую конструкцию нашей схемы.

На макетной плате это можно сделать, но для устройства, которое уже изготовлено и отправлено заказчику, изменение аппаратного обеспечения обычно не представляется возможным. Определение характеристик устройства только на аппаратном уровне ограничивает возможность быстро вводить новшества и улучшать конструкцию.

Схемы, которые мы собрали до сих пор, дают нам некоторое представление о том, как работают компьютеры, но не хватает одного критически важного элемента: *программируемости*. Это означает, что компьютер должен быть способен выполнять новые задачи *без* изменения аппаратного обеспечения. Для этого компьютер должен уметь принимать набор инструкций (*программу*) и выполнять действия, описанные в этих инструкциях. Поэтому компьютер должен иметь аппаратное обеспечение, способное выполнять различные операции в порядке, указанном в программе. Возможность программирования является ключевым отличием между устройством, которое является компьютером, и тем, которое им не является. В этой главе мы рассмотрим *аппаратное обеспечение* компьютера, т. е. его физические элементы. В то время как *программное обеспечение*, т. е. инструкции, указывающие компьютеру, что делать, мы рассмотрим в следующей главе.

Возможность запускать программное обеспечение отличает компьютер от устройства фиксированного назначения. Тем не менее программное обеспечение все равно нуждается в аппаратных средствах. Так какие же аппаратные средства нам нужны для реализации компьютера общего назначения? Во-первых, память. Мы уже рассматривали однобитные устройства памяти, такие как защелки и триггеры. Тип памяти, используемый в компьютере, является принципиальным расширением этих простых устройств памяти. Основная память, используемая в компьютере, называется *оперативной* (*оперативным запоминающим устройством, ОЗУ*), но часто ее называют просто памятью или *памятью с произвольным доступом* (*Random Access Memory, RAM*). RAM обычно *энергозависима*, т. е. сохраняет данные только при подаче питания. «Произвольный доступ» к оперативной памяти означает, что доступ к любой случайной области памяти может быть получен примерно за такое же время, как и доступ к любой другой области<sup>1</sup>.

Второй ключевой компонент, который нам необходим, это *центральный процессор* (*Central Processing Unit, CPU*). Этот компонент, часто называемый просто *процессором*, выполняет инструкции, заданные в программном обеспечении. Центральный процессор может напрямую обращаться к оперативной памяти. Большинство процессоров сегод-

---

<sup>1</sup> Следует уточнить, что в случае RAM речь идет о произвольном доступе как на чтение, так и на запись информации. Это является ключевым отличием RAM от ROM (*Read-Only Memory*, «памяти только для чтения»), употребляющейся для долговременного хранения данных (вторичные хранилища, см. далее). Строго говоря, все современные разновидности ROM (жесткие диски, флеш-накопители, оптические диски и т. д.) также позволяют как читать, так и записывать данные, но в случае ROM запись требует специальных действий и, как правило, длится существенно дольше, чем чтение.

ня – это *микропроцессоры*, центральные процессоры на одной интегральной схеме. Процессор, построенный на одной интегральной схеме, обладает такими преимуществами, как более низкая стоимость, высокая надежность и повышенная производительность. Центральный процессор является расширением цифровых логических схем, которые мы рассматривали ранее.

Хотя оперативная память и процессор – это минимальные требования к аппаратному обеспечению компьютера, на практике большинству вычислительных устройств необходимо взаимодействовать с внешним миром, и они делают это через устройства ввода/вывода (I/O). В этой главе мы более подробно рассмотрим оперативную память, центральный процессор и устройства ввода/вывода. Эти три элемента показаны на рис. 7-1.



Рис. 7-1. Элементы аппаратного обеспечения компьютера

## Оперативная память

При выполнении программы компьютеру необходимо место для хранения инструкций программы и соответствующих данных. Например, когда компьютер запускает текстовый процессор для редактирования документов, ему нужно место для хранения самой программы, содержимого документа и состояния редактирования, а именно определение того, какая часть документа видна, где находится курсор и т. д. Все эти данные в конечном итоге представляют собой последовательность битов, к которым процессор должен иметь доступ. Оперативная память выполняет задачу хранения этих единиц и нулей.

Давайте рассмотрим, как работает оперативная память в компьютере. Существует два распространенных типа компьютерной памяти: *статическая память с произвольным доступом (SRAM)* и *динамическая память с произвольным доступом (DRAM)*. В обоих типах памяти основной единицей хранения данных является *ячейка памяти* – схема, которая может хранить один бит. В SRAM ячейки памяти представляют собой разновидность триггеров. SRAM является статической, поскольку ее ячейки памяти сохраняют свои битовые значения при подаче питания. С другой стороны, ячейки памяти DRAM реализованы с помощью транзистора и конденсатора. Заряд конденсатора со временем утекает, поэтому данные должны периодически перезаписываться в ячейки. Именно это обновление ячеек памяти делает DRAM динамической. Сегодня DRAM

обычно используется в качестве оперативной памяти благодаря своей относительно низкой цене. SRAM быстрее, но дороже, поэтому она используется там, где скорость критична, например в кеш-памяти, которую мы рассмотрим позже. Пример модуля оперативной памяти DRAM показан на рис. 7-2.

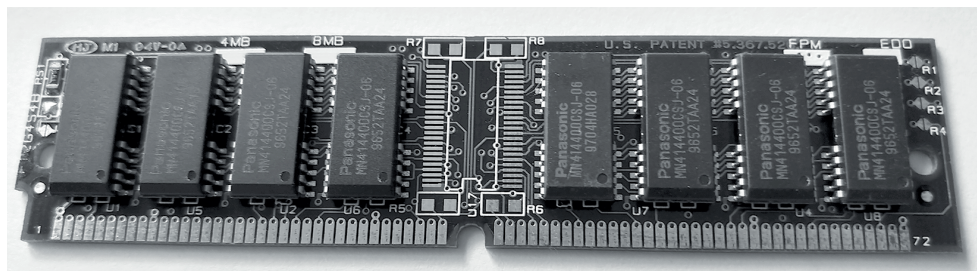


Рис. 7-2. Модуль памяти с произвольным доступом DRAM

Для наглядности можно представить внутреннюю часть оперативной памяти как сетку из ячеек памяти. Каждая однобитовая ячейка в сетке может быть идентифицирована с помощью двумерных координат, обозначающих местоположение этой ячейки в сетке. Обращение к одному биту за раз не очень эффективно, поэтому оперативная память обращается к нескольким сеткам из однобитных ячеек памяти параллельно, что позволяет считывать или записывать несколько битов одновременно, например целый байт. Расположение набора битов в памяти известно как *адрес памяти* – числовое значение, идентифицирующее область памяти. Обычно память имеет *байтовую адресацию*, т. е. один адрес памяти относится к 8 битам данных. Детали внутреннего расположения ячеек памяти не являются обязательным знанием для программиста. Главное – понять, что компьютеры присваивают байтам памяти числовые адреса, а центральный процессор может читать или писать по этим адресам, как показано на рис. 7-3.

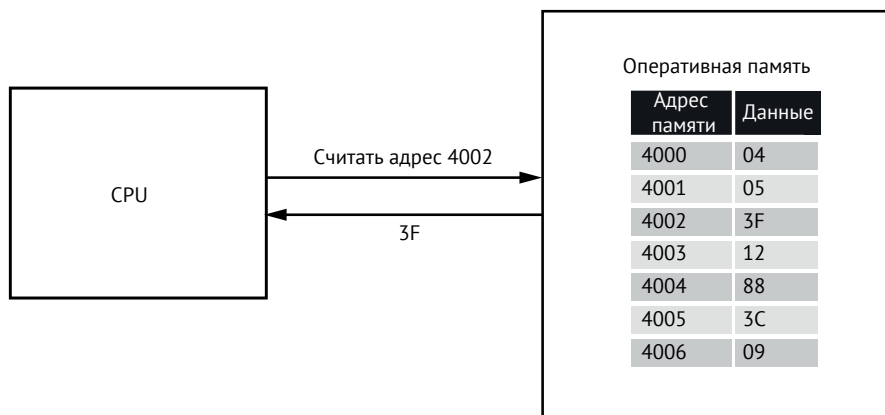


Рис. 7-3. Центральный процессор считывает байт из адреса памяти

Давайте рассмотрим вымышленную компьютерную систему, которая может адресовать до 64 КБ памяти. По сегодняшним меркам это ничтожный объем памяти для компьютера, но для примера он все же полезен. Давайте также представим, что память нашего вымышленного компьютера имеет байтовую адресацию, т. е. каждый адрес памяти представляет собой один байт. Это означает, что нам нужен один уникальный адрес для каждого байта памяти, а поскольку 64 КБ – это  $64 \times 1024 = 65\,536$  байт, то нам нужно 65 536 уникальных адресов. Каждый адрес – это просто число, а адреса памяти обычно начинаются с 0, поэтому наш диапазон адресов будет от 0 до 65 535 (или 0xFFFF).

Поскольку наш фиктивный компьютер с 64 КБ является цифровым устройством, адреса памяти в конечном итоге представлены в двоичном виде. Сколько битов нам нужно для представления адреса памяти в этой системе? Количество уникальных значений, которые могут быть представлены двоичным числом с  $n$  битами, равно  $2^n$ . Поэтому мы хотим узнать значение  $n$  для  $2^n = 65\,536$ . Операцией, обратной возведению 2 в некоторую степень, будет логарифм по основанию 2. Поэтому получаем:  $\log_2(2^n) = n$  и  $\log_2(65\,536) = 16$ . Говоря иначе,  $2^{16} = 65\,536$ . Следовательно, для адресации 65 536 байт необходим 16-битный адрес памяти.

Или проще, поскольку мы уже знаем, что наш адрес памяти с наибольшим номером это 0xFFFF, и знаем, что каждый шестнадцатеричный символ представляет собой 4 бита, то очевидно, что требуется 16 бит (4 шестнадцатеричных символа  $\times$  4 бита на символ). Еще раз, наш вымышленный компьютер способен адресовать 65 536 байт, и каждому байту присвоен 16-битный адрес памяти. В табл. 7-1 представлена 16-битная структура памяти с некоторыми примерами данных.

**Таблица 7-1.** Схема 16-битных адресов памяти с пропуском посередине с примерами данных

Адрес памяти (бинарный)	Адрес памяти (шестнадцатеричный)	Пример данных
0000000000000000	0000	23
0000000000000001	0001	51
0000000000000010	0002	4A
-----	----	--
1111111111111101	FFFD	03
1111111111111110	FFFE	94
1111111111111111	FFFF	82

Почему количество битов имеет значение? Количество битов, используемых для представления адреса памяти, является ключевой частью конструкции компьютерной системы. Оно ограничивает объем памяти, к которой компьютер может получить доступ, и влияет на то, как программы работают с памятью на низком уровне.

Давайте представим, что наш вымышленный компьютер хранит строку ASCII «Hello» по адресу памяти 0x0002. Поскольку каждый символ ASCII занимает 1 байт, для хранения «Hello» требуется 5 байт. При изучении па-



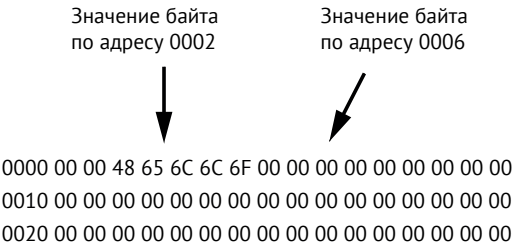
мяти принято использовать шестнадцатеричную систему для представления как адресов памяти, так и содержимого этих адресов. В табл. 7-2 представлен вид слова «Hello», хранящегося в памяти, начиная с адреса 0x0002.

**Таблица 7-2.** «Hello», хранящееся в памяти

Адрес памяти	Байт данных	Данные в ASCII
0000	00	
0001	00	
0002	48	H
0003	65	e
0004	6C	l
0005	6C	l
0006	6F	o
0007	00	
----	--	
FFFF	00	

Использование этого формата дает понять, что каждый адрес хранит только 1 байт, поэтому для хранения всех 5 символов ASCII требуются адреса с 0x0002 по 0x0006. Обратите внимание, что в таблице указано значение 00 для всех других адресов памяти, но на практике небезопасно предполагать, что случайный адрес будет содержать 0, там может быть что угодно. В некоторых языках программирования принято завершать текстовую строку нуль-терминатором (байтом, равным 0), и в этом случае мы бы ожидали увидеть 00 по адресу 0x0007.

Приложения, позволяющие просматривать память компьютера, обычно представляют содержимое памяти в формате, подобном тому, что представлен на рис. 7-4.



**Рис. 7-4.** Типичное представление байтов памяти

Самый левый столбец на рис. 7-4 – это адрес памяти в шестнадцатеричном формате, а следующие 16 значений представляют байты по этому адресу и 15 последующим адресам. Этот подход более компактен, чем табл. 7-2, но в этом случае каждый адрес не имеет уникального названия. На этом рисунке мы снова видим ASCII-строку «Hello», хранящуюся начиная с адреса 0x0002.

Наш гипотетический компьютер с 64 КБ оперативной памяти полезен в качестве примера, но современные вычислительные устройства, как правило, имеют гораздо больший объем памяти<sup>1</sup>. По состоянию на 2020 год смартфоны обычно имеют не менее 1 ГБ памяти, а ноутбуки – не менее 4 ГБ.

### УПРАЖНЕНИЕ 7-1: Вычислите необходимое количество битов

Используя только что описанные методы, определите количество битов, необходимых для адресации 4 ГБ памяти. Для справки о префиксах СИ вам понадобится вернуться к табл. 1-3. Помните, что каждому байту присваивается уникальный адрес, который является просто числом. Ответ вы найдете в приложении А.

## Центральный процессор (CPU)

Память дает компьютеру место для хранения данных и программных инструкций, но именно CPU, или процессор, выполняет эти инструкции. Именно процессор обеспечивает гибкость компьютера, позволяя ему выполнять программы, которые даже не были придуманы на момент разработки процессора. Процессор реализует набор инструкций, которые программисты могут затем использовать для создания программного обеспечения. Каждая такая инструкция проста, но эти базовые инструкции являются строительными блоками для всего программного обеспечения.

Вот некоторые примеры типов инструкций, которые поддерживают процессоры:

**доступ к памяти** – чтение, запись (в память);

**арифметика** – сложение, вычитание, умножение, деление, приращение;

**логика** – операции И, ИЛИ, НЕ;

**ход программы** – переход (к определенной части программы), вызов (подпрограммы).

Мы рассмотрим конкретные инструкции процессора в главе 8, а пока важно понять, что инструкции процессора – это просто операции, которые может выполнять процессор. Они довольно просты (сложить

<sup>1</sup> Следует отметить, что для *микроконтроллеров* – дешевых однокристальных компьютеров, ныне являющихся основным средством управления большинством устройств, от холодильников и стиральных машин до узлов автомобилей и космических аппаратов – объем оперативной памяти в 64 КБ и даже меньше, скорее, типичен. А количество микроконтроллеров в мире значительно превышает численность «настоящих» компьютеров. – *Прим. ред.*

два числа, прочесть адрес из памяти, выполнить логическое И и т. д.). Программы состоят из упорядоченных наборов этих инструкций. Если использовать кулинарную аналогию, то процессор – это повар, программа – это рецепт, а каждая инструкция в программе – это пункт рецепта, который повар знает, как выполнить.

Инструкции программы хранятся в памяти. Центральный процессор считывает эти инструкции, чтобы выполнить программу. На рис. 7-5 показана простая программа, которая считывается из памяти центральным процессором.

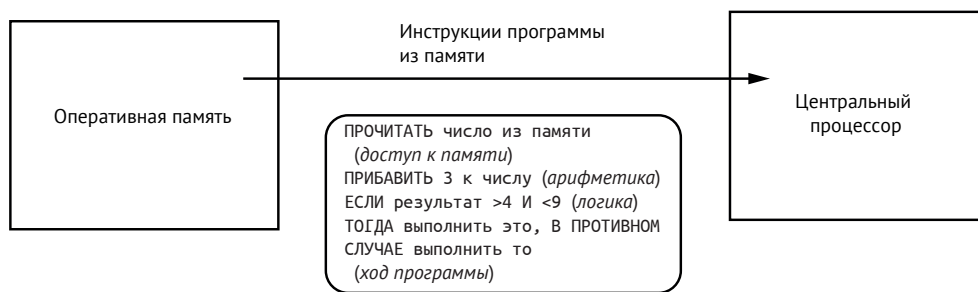


Рис. 7-5. Пример программы, которая считывается из памяти и выполняется на центральном процессоре

Пример программы на рис. 7-5 написан в *псевдокоде* – читабельном описании программы, которое написано на ненастоящем языке программирования. Шаги в программе относятся к только что описанным категориям (доступ к памяти, арифметика, логика и ход программы). На первом шаге программа считывает число, хранящееся по определенному адресу в памяти. Затем программа прибавляет к этому числу 3. После чего она выполняет логическое И из двух условий. Если логический результат истинен, то программа делает «это», в противном случае она делает «то». Хотите верить, хотите нет, но все программы, по сути, являются просто различными комбинациями этих типов фундаментальных операций.

## Архитектура набора команд

Хотя все процессоры реализуют данные типы инструкций, конкретные инструкции, доступные на разных процессорах, отличаются. Некоторые инструкции, существующие для одного типа процессора, просто не существуют для других. Даже те инструкции, которые существуют почти во всех процессорах, реализуются на них по-разному. Например, конкретная двоичная последовательность, используемая для обозначения «сложить два числа», не одинакова для разных типов процессоров. Говорят, что семейство процессоров, использующих одинаковые инструкции, имеет общую *архитектуру набора команд* (*Instruction Set Architecture, ISA*), или просто *архитектуру*, т. е. модель работы процессора. Созданное для определенной архитектуры программное обеспечение работает на любом процессоре, который имеет такую же архитектуру.

Несколько моделей процессоров даже от разных производителей могут реализовывать одну и ту же архитектуру.

Такие процессоры могут работать совершенно по-разному внутри, но, придерживаясь одного и того же набора команд, они могут выполнять одно и то же программное обеспечение. Сегодня существуют две наиболее распространенные архитектуры наборов команд: x86 и ARM<sup>1</sup>.

Большинство настольных компьютеров, ноутбуков и серверов используют процессоры x86. Название происходит от принятой корпорацией Intel схемы наименования своих процессоров (каждый из которых заканчивается на 86), начиная с 8086, выпущенного в 1978 году, и продолжающихся как 80186, 80286, 80386 и 80486. После 80486 (или просто 486) Intel стала называть свои процессоры такими именами, как Pentium и Celeron, но, несмотря на смену названия, эти процессоры по-прежнему являются процессорами x86. Другие компании, помимо Intel, также производят процессоры x86, особенно Advanced Micro Devices, Inc. (AMD).

Термин *x86* относится к набору родственных архитектур. Со временем в архитектуру x86 добавлялись новые инструкции, но каждое поколение старалось сохранить обратную совместимость. Это означает, что программное обеспечение, разработанное для более старого процессора x86, будет работать и на более новом процессоре x86. Но программное обеспечение, созданное для более нового процессора x86 и использующее преимущества новых инструкций x86, не сможет работать на старых процессорах x86, которые «не понимают» новые инструкции.

Архитектура x86 включает в себя три основных поколения процессоров: 16-, 32- и 64-битные. Давайте сделаем паузу и рассмотрим, что мы имеем в виду, когда говорим, что процессор является 16-, 32- или 64-разрядным. Количество битов относительно процессора, также известное как его *разрядность* или *размер слова*, относится к количеству битов, с которыми он может работать одновременно. Так, 32-битный процессор может работать со значениями длиной 32 бита. Более конкретно это означает, что архитектура компьютера имеет 32-битные регистры памяти, 32-битную шину адреса или 32-битную шину данных. Более подробно о регистрах, шинах данных и шинах адреса мы расскажем позже.

Если вернуться к x86 и его поколениям процессоров, то первоначальный процессор 8086, выпущенный в 1978 году, был 16-разрядным. Воодушевленная успехом 8086, компания Intel продолжила выпуск совместимых процессоров. Последующие процессоры Intel x86 также были 16-разрядными, пока в 1985 году не был выпущен процессор

---

<sup>1</sup> Еще раз уточним, что речь идет о «настоящих» компьютерах. Для микроконтроллеров наборы команд могут быть другими, и разнообразие их значительно больше. Впрочем, микроконтроллер может быть создан и на ядре ARM, тогда набор инструкций для него будет совместим (или частично совместим) с инструкциями для «больших» компьютеров. Существовала и ветка микроконтроллеров под названием x51, когда-то развивавшаяся фирмой Intel параллельно с «большой» архитектурой x86 и частично с ней совместимая (в настоящее время потеряла актуальность и применяется исключительно в учебных целях). – Прим. ред.

80386, который принес с собой новую 32-разрядную версию архитектуры x86. Эта 32-разрядная версия x86 иногда называется IA-32 (от *Intel Architecture*). Благодаря обратной совместимости современные процессоры x86 по-прежнему полностью поддерживают IA-32. Пример процессора x86 показан на рис. 7-6.

Интересно, что именно AMD, а не Intel ввела x86 в 64-битную эру. В конце 1990-х годов Intel сосредоточилась на 64-битной архитектуре процессоров под названием IA-64 или Itanium, которая *не* была x86 ISA, и в итоге стала нишевым продуктом для серверов. Поскольку Intel сосредоточилась на Itanium, AMD воспользовалась возможностью расширить архитектуру x86. В 2003 году AMD выпустила процессор Opteron, первый 64-разрядный процессор x86. Архитектура AMD первоначально была известна как *AMD64*, а позже Intel приняла эту архитектуру и назвала свою реализацию *Intel 64*. Эти две реализации в основном функционально идентичны, и сегодня 64-разрядные x86 обычно называют *x64* или *x86-64*.

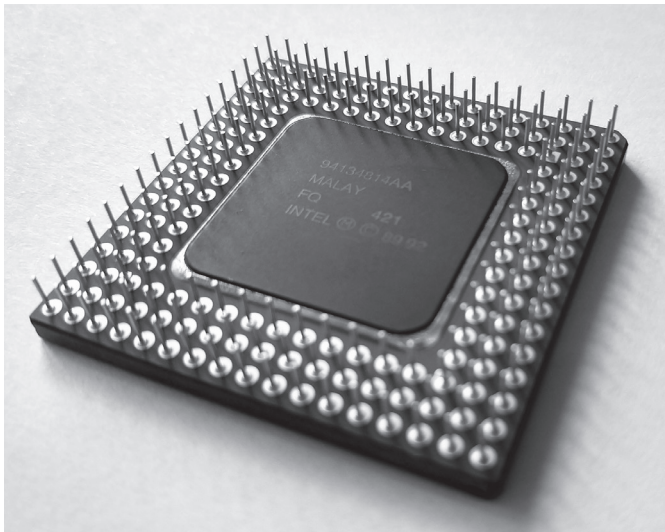


Рис. 7-6. Intel 486 SX, 32-разрядный процессор x86

Несмотря на то что процессоры x86 правят в мире персональных компьютеров и серверов, процессоры ARM занимают ведущее место в сфере мобильных устройств, таких как смартфоны и планшеты. Процессоры ARM производят несколько компаний. Компания под названием ARM Holdings разрабатывает архитектуру ARM и лицензирует свои разработки другим компаниям для реализации. Обычно процессоры ARM используются в *системах на кристалле (SoC)*, где одна интегральная схема содержит не только процессор, но и память и другое оборудование<sup>1</sup>. Ар-

<sup>1</sup> Системы на кристалле (SoC) часто также называют однокристалльными компьютерами. Именно к ним относятся упомянутые микроконтроллеры, что и обуславливает их дешевизну и распространенность. – *Прим. ред.*

хитектура ARM возникла в 1980-х годах как 32-разрядная. 64-разрядная версия архитектуры ARM была представлена в 2011 году. Процессоры ARM предпочитают использовать в мобильных устройствах благодаря снижению энергопотреблению и более низкой стоимости по сравнению с процессорами x86. Процессоры ARM могут использоваться и в персональных компьютерах, но этот рынок в основном остается ориентированным на x86, чтобы сохранить обратную совместимость с существующим программным обеспечением для ПК на x86. Однако в 2020 году Apple объявила о своем намерении перевести компьютеры на основе macOS с процессоров x86 на процессоры ARM.

## Внутреннее устройство процессора

Внутри процессор состоит из множества компонентов, которые работают вместе для выполнения инструкций. Мы сосредоточимся на трех основных компонентах: регистрах процессора, арифметико-логическом устройстве и блоке управления. *Регистры процессора* – это области процессора, в которых хранятся данные во время обработки. *Арифметико-логическое устройство (АЛУ)* выполняет логические и математические операции. *Блок управления* управляет процессором, взаимодействуя с регистрами процессора, АЛУ и оперативной памятью. На рис. 7-7 показан упрощенный вид архитектуры центрального процессора.

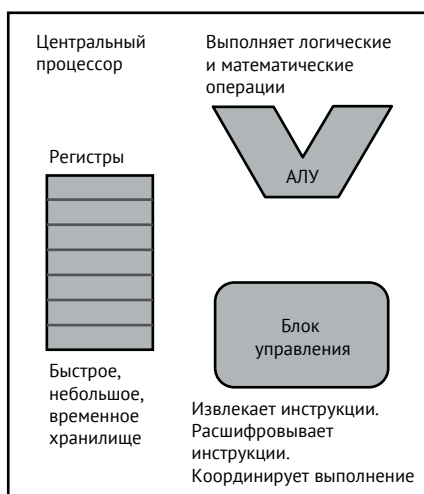


Рис. 7-7. Значительно упрощенный вид архитектуры процессора

Давайте рассмотрим регистры процессора. Оперативная память хранит данные для выполняющейся программы. Однако, когда программе необходимо оперировать частью данных, процессору требуется временное место для хранения данных в аппаратной части процессора. Для этого в процессорах есть небольшие области внутри для хранения данных, известные как регистры процессора, или просто регистры. По сравнению с обращением к оперативной памяти операция обра-

щения к регистрам очень быстрая для процессора, но регистры могут хранить только очень небольшие объемы данных. Регистры так малы, что мы измеряем размер отдельного регистра в битах, а не в байтах. Например, 32-разрядный процессор обычно имеет регистры шириной 32 бита, т. е. каждый регистр может хранить 32 бита данных. Регистры реализованы в компоненте, называемом *регистровый файл* (не путать с файлом данных, таким как документ или фотография). Ячейки памяти, используемые в регистровом файле, обычно представляют собой разновидность статической памяти SRAM.

АЛУ выполняет логические и математические операции в центральном процессоре. Ранее мы рассматривали комбинационные логические схемы, в которых выход является функцией входного сигнала. АЛУ процессора – это просто сложная комбинационная логическая схема. Входами АЛУ являются значения, называемые *операндами*, и код, указывающий, какую операцию нужно выполнить над этими операндами. АЛУ выводит результат операции вместе со статусом, который предоставляет более подробную информацию о выполнении операции.

Блок управления действует как координатор работы центрального процессора. Он работает по повторяющемуся циклу: извлечение инструкции из памяти, ее декодирование и выполнение. Поскольку выполняемая программа хранится в оперативной памяти, блоку управления необходимо знать, какой адрес памяти нужно считать, чтобы получить следующую инструкцию. Блок управления определяет это, просматривая регистр, известный как *счетчик команд* (*Program Counter, PC*) или как *указатель инструкций* в x86. Счетчик команд хранит в памяти адрес следующей инструкции, которую нужно выполнить. Блок управления считывает инструкцию из указанного адреса памяти, сохраняет ее в регистре, называемом *регистром команд*, и обновляет счетчик команд, чтобы указать на следующую инструкцию.

Затем блок управления расшифровывает текущую инструкцию, придавая смысл единицам и нулям, представляющим ее. После декодирования блок управления выполняет инструкцию, что может потребовать координации действий с другими компонентами процессора. Например, операция сложения требует от блока управления указание АЛУ выполнить необходимые математические действия. После завершения выполнения инструкции блок управления повторяет цикл: извлечение, декодирование, выполнение.

## **Синхросигнал, ядра и кеш**

Поскольку процессоры выполняют упорядоченные наборы инструкций, вас может заинтересовать, что же заставляет процессор переходить от одной инструкции к другой. Ранее мы уже демонстрировали, как синхросигнал может использоваться для перевода схемы из одного состояния в другое, например в схеме счетчика. Здесь действует тот же принцип. Центральный процессор принимает входной синхросигнал, как показано на рис. 7-8, и тактовый импульс служит для процессора сигналом для перехода из одного состояния в другое.



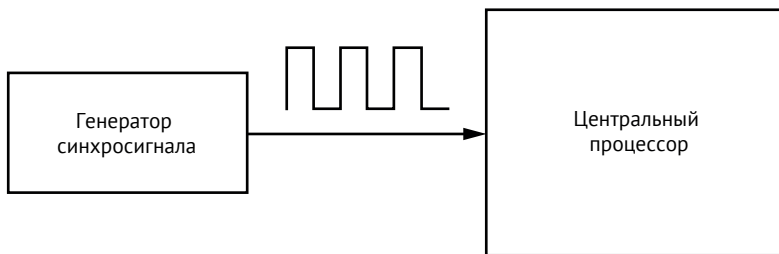


Рис. 7-8. Генератор синхросигнала подает тактовый сигнал на процессор

Будет чрезмерным упрощением считать, что процессор выполняет ровно одну инструкцию за тактовый цикл. Для выполнения некоторых инструкций требуется несколько тактовых циклов. Кроме того, современные процессоры используют подход, называемый *конвейером*, для разделения инструкций на более мелкие шаги, так что части нескольких инструкций могут параллельно выполняться одним процессором. Например, одна инструкция может извлекаться, в то время как другая декодируется, а третья выполняется. Тем не менее полезно рассматривать каждый импульс тактовой частоты как сигнал процессору для дальнейшего выполнения программы. Современные процессоры имеют тактовую частоту, измеряемую в *гигагерцах (ГГц)*. Например, процессор с частотой 2 ГГц имеет синхросигнал, который совершает 2 млрд колебаний в секунду!

Увеличение тактовой частоты позволяет процессору выполнять больше инструкций в секунду. К сожалению, мы не можем просто запустить процессор на произвольно высокой тактовой частоте. У процессоров есть практический верхний предел их входной тактовой частоты, и превышение этого предела приводит к чрезмерному выделению тепла. Кроме того, логические вентили процессора могут не успевать за тактами, что приведет к неожиданным ошибкам и сбоям. В течение многих лет в компьютерной индустрии наблюдалось постоянное увеличение верхнего предела тактовой частоты для процессоров. Это увеличение тактовой частоты в значительной степени было связано с регулярным совершенствованием производственных процессов, которые привели к увеличению плотности транзисторов, что позволило создавать процессоры с более высокой тактовой частотой практически без увеличения энергопотребления. В 1978 году Intel 8086 работал на частоте 5 МГц, а к 1999 году Intel Pentium III имел тактовую частоту 500 МГц, что представляет собой 100-кратное увеличение всего за 20 лет!

Тактовые частоты процессоров продолжали быстро расти, пока в начале 2000-х годов не был преодолен порог в 3 ГГц. С тех пор, несмотря на продолжающийся рост количества транзисторов, физические ограничения, связанные с уменьшением размеров транзисторов, сделали значительное увеличение тактовой частоты нецелесообразным.

В условиях застоя тактовых частот процессорная индустрия обратилась к новому подходу для получения большей скорости работы про-



цессора. Вместо того чтобы сосредоточиться на увеличении тактовой частоты, при разработке процессоров стали уделять больше внимания параллельному выполнению нескольких инструкций. Появилась идея *многоядерного процессора* – процессора с несколькими вычислительными блоками, называемыми *ядрами*. *Ядро процессора* – это фактически независимый процессор, который располагается рядом с другими независимыми процессорами в одном корпусе центрального процессора, как показано на рис. 7-9.

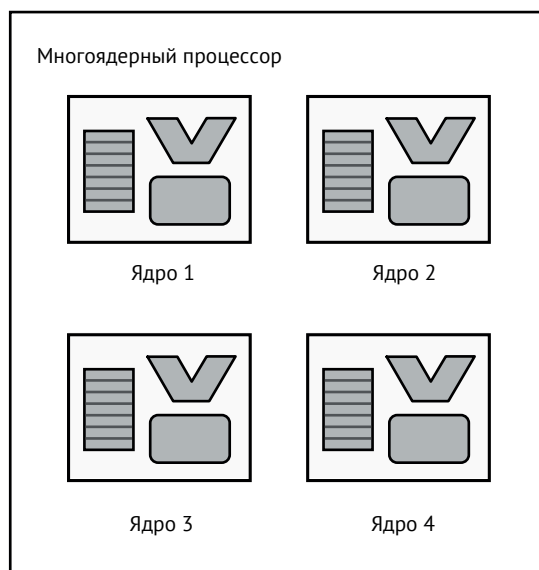


Рис. 7-9. Четырехъядерный процессор – каждое ядро имеет свои собственные регистры, АЛУ и блок управления

Обратите внимание, что параллельная работа нескольких ядер не то же самое, что конвейер. Параллелизм при многоядерности означает, что каждое ядро работает над отдельной задачей, отдельным набором инструкций. В отличие от этого конвейер работает *внутри* каждого ядра, позволяя части некоторых инструкций выполняться параллельно одним ядром.

Каждое ядро, добавленное к процессору, открывает возможность для параллельного выполнения дополнительных инструкций. Однако добавление нескольких ядер в процессор компьютера не означает, что все приложения сразу же получают одинаковые преимущества. Программное обеспечение должно быть написано так, чтобы использовать преимущества параллельной обработки инструкций для получения максимальной пользы от многоядерного оборудования. Однако, даже если отдельные программы не разработаны с учетом параллелизма, компьютерная система в целом может выиграть, поскольку современные операционные системы запускают несколько программ одновременно.

Ранее я уже описывал, как процессоры загружают данные из оперативной памяти в регистры для обработки, а затем сохраняют их обратно из регистров в память для последующего использования.

Оказывается, что программы имеют тенденцию обращаться к одним и тем же участкам памяти снова и снова. Ожидаемо, что многократное возвращение к оперативной памяти для доступа к одним и тем же данным неэффективно! Чтобы избавиться от этой неэффективности, в процессоре имеется небольшой объем быстрой памяти, где хранится копия данных, к которым часто обращаются из оперативной памяти. Эта память называется *кеш-памятью процессора*.

Процессор проверяет кеш, чтобы узнать, есть ли там данные, к которым он хочет получить доступ. Если да, то процессор может ускорить работу, читая или записывая данные в кеш, а не в оперативную память. Если нужные данные отсутствуют в кеше, процессор может переместить их в кеш после того, как они будут считаны из оперативной памяти. Обычно процессоры имеют несколько уровней кеша, часто три. Мы называем эти уровни так: кеш первого уровня (L1), кеш второго уровня (L2) и кеш третьего уровня (L3).

Процессор сначала проверяет L1 на наличие нужных данных, затем L2, затем L3, прежде чем обратиться к оперативной памяти, как показано на рис. 7-10. Кеш первого уровня является самым быстрым для доступа, но он также самый маленький. Кеш второго уровня медленнее и больше, а третьего – еще медленнее и еще больше. Помните, что даже с этими постепенно замедляющимися уровнями кеша доступ к оперативной памяти все равно медленнее, чем к любому уровню кеша.

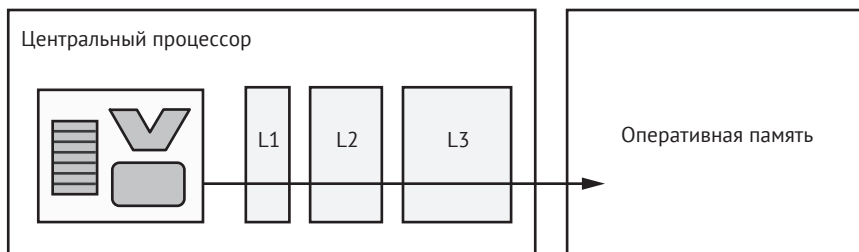


Рис. 7-10. Одноядерный процессор с тремя уровнями кеша

В многоядерных процессорах некоторые кешы являются специфическими для каждого ядра, в то время как другие разделяются между ядрами. Например, каждое ядро может иметь свой собственный кеш первого уровня, в то время как кешы второго и третьего уровней являются общими, как показано на рис. 7-11.

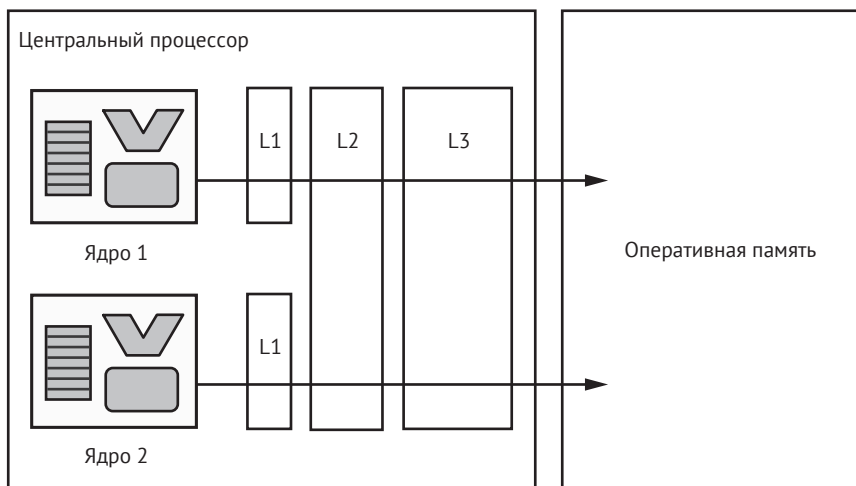


Рис. 7-11. Двухъядерный процессор с кешем. Каждое ядро имеет свой собственный кеш первого уровня, в то время как кеши второго и третьего уровней являются общими

## За пределами памяти и процессора

Я описал два основных компонента, необходимых для компьютера: память и процессор. Однако устройство, состоящее только из памяти и процессора, будет иметь несколько недостатков, которые необходимо устранить, если мы хотим получить полезное устройство. Первый недостаток заключается в том, что и память, и процессор являются энергозависимыми, т. е. они обнуляют состояние при отключении питания. Второй недостаток заключается в том, что компьютер, имеющий только память и процессор, не имеет возможности взаимодействовать с внешним миром. Теперь давайте посмотрим, как вторичное хранилище и устройства ввода/вывода справляются с этими недостатками.

### Вторичное хранилище

Если бы компьютер состоял только из памяти и процессора, то при каждом выключении питания он терял бы все свои данные! Я хочу подчеркнуть, что под *данными* здесь подразумеваются не только файлы и настройки пользователя, но и все установленные приложения, и даже сама операционная система. Такой весьма неудобный компьютер потребовал бы от пользователя загружать ОС и все приложения при каждом включении. Это может отбить у пользователей охоту когда-либо выключать компьютер. Хотите верить, хотите нет, но компьютеры предыдущих поколений работали именно так, но, к счастью, сегодня это изменилось.

Чтобы решить эту проблему, компьютеры оснащены вторичным хранилищем. *Вторичное хранилище* является энергонезависимым и поэтому сохраняет данные даже при выключении питания системы. В отличие

от оперативной памяти центральный процессор не обращается ко вторичному хранилищу напрямую. Такое хранилище обычно намного дешевле в пересчете на байты, чем оперативная память, что позволяет хранить большие объемы данных по сравнению с оперативной памятью. Однако вторичное хранилище также значительно медленнее оперативной памяти, и поэтому оно не является ее полноценной заменой.

В современных вычислительных устройствах наиболее распространенными вторичными хранилищами являются жесткие диски и твердотельные накопители. *Жесткий диск (HDD)* хранит данные с помощью намагниченных областей на быстро вращающейся пластине, в то время как *твердотельный накопитель (SSD)* хранит данные с помощью электрических зарядов в энергонезависимых ячейках памяти. По сравнению с жесткими дисками твердотельные накопители быстрее, тише и более устойчивы к механическим повреждениям, поскольку в них нет движущихся частей<sup>1</sup>. На рис. 7-12 показана фотография пары вторичных хранилищ данных.

При наличии вторичного хранилища компьютер может загружать данные, когда это требуется. Когда компьютер включается, операционная система загружается из вторичного хранилища в операционную память, куда также загружаются все приложения, которые настроены на открытие при запуске. Когда после запуска компьютера открывается приложение, программный код загружается из вторичного хранилища в оперативную память. То же самое касается любых пользовательских данных (документов, музыки, настроек и т. д.), хранящихся локально, перед использованием они должны быть загружены из вторичного хранилища в оперативную память. В обиходе вторичное хранилище часто называют просто хранилищем, а первичное хранилище / оперативную память называют просто памятью или, как мы говорили, оперативным запоминающим устройством, ОЗУ (RAM).



Рис. 7-12. Жесткий диск емкостью 4 ГБ 1997 года рядом с разновидностью твердотельного накопителя – современной картой microSD емкостью 32 ГБ

<sup>1</sup> Однако SSD-накопители в настоящий момент значительно (втрое-вчетверо и более) дороже магнитных жестких дисков в расчете на хранение единицы информации и имеют ограничения по максимальному объему. Поэтому традиционные жесткие диски до сих пор доминируют в области долговременного хранения информации в компьютерных системах. – Прим. ред.

## Устройства ввода/вывода

Даже при наличии вторичного хранилища у нашего гипотетического компьютера все еще есть проблема. Компьютер, состоящий из процессора, памяти и хранилища, не имеет никакого способа взаимодействия с внешним миром! Именно здесь на помощь приходят устройства ввода/вывода. *Устройство ввода/вывода (input/output, I/O)* – это компонент, который позволяет компьютеру получать данные из внешнего мира (клавиатура, мышь), отправлять данные во внешний мир (монитор, принтер) или делать и то и другое (сенсорный экран). Взаимодействие человека с компьютером требует использования устройств ввода/вывода. Взаимодействие компьютера с компьютером также требует посредничества устройств ввода/вывода, часто в форме компьютерной сети, например интернета. Вторичные хранилища фактически являются одним из типов устройств ввода/вывода. Это может быть неочевидным, что доступ к внутреннему хранилищу обеспечивается устройствами ввода/вывода, но с точки зрения центрального процессора чтение или запись в хранилище – это просто еще одна операция ввода/вывода. Чтение с устройства хранения – это ввод, а запись на устройство хранения – это вывод. На рис. 7-13 приведены некоторые примеры ввода и вывода.



Рис. 7-13. Типовые устройства ввода и вывода

Как же процессор взаимодействует с устройствами ввода/вывода? К компьютеру может быть подключено множество устройств ввода/вывода, и центральному процессору необходим стандартный способ связи с любым таким устройством. Чтобы понять это, мы должны сначала обсудить *физическое адресное пространство*, т. е. диапазон адресов аппаратной памяти, доступных компьютеру. Ранее в этой главе, в разделе «Оперативная память» на стр. 153, мы рассмотрели, как байтам памяти присваивается адрес. Все адреса памяти в данной компьютерной системе будут представлены определенным количеством битов. Это количество битов определяет не только размер каждого адреса памяти, но и диапазон адресов, доступных для использования аппаратными средствами компьютера, т. е. физическое адресное пространство. Адресное пространство часто больше, чем объем оперативной памяти, установленной на компьютере, в результате чего некоторые адреса физической памяти остаются неиспользованными.

Для примера, в случае компьютера с 32-битным физическим адресным пространством диапазон физических адресов составляет от 0x00000000 до 0xFFFFFFFF (самый большой адрес, который может быть представлен 32-битным числом). Это приблизительно 4 млрд адресов, каждый из которых представляет один байт, или 4 ГБ, адресного пространства. Допустим, что в этом компьютере 3 ГБ оперативной памяти, тогда 75 % доступных адресов физической памяти отводятся байтам оперативной памяти.

Теперь вернемся к вопросу о том, как процессоры взаимодействуют с устройствами ввода/вывода. Адреса в физическом адресном пространстве не всегда относятся к байтам памяти, они также могут относиться к устройству ввода/вывода. Когда физическое адресное пространство отображается на устройство ввода/вывода, центральный процессор может взаимодействовать с этим устройством, просто читая или записывая в назначенный ему адрес(а) памяти. Это называется *вводом/выводом через память (MMIO)* и проиллюстрировано на рис. 7-14. Когда компьютер относится к памяти устройств ввода/вывода так же, как к оперативной памяти, его центральному процессору не нужны специальные инструкции для операций ввода/вывода.

Однако некоторые семейства процессоров, в частности x86, включают специальные инструкции для доступа к устройствам ввода/вывода. Когда компьютеры используют этот подход вместо того, чтобы сопоставлять устройства ввода/вывода с физическим адресом памяти, устройствам присваивается *порт ввода/вывода*. Порт похож на адрес памяти, но вместо ссылки на место в памяти номер порта относится к устройству ввода/вывода. Вы можете считать набор портов ввода/вывода еще одним адресным пространством, отличным от адресов памяти. Это означает, что порт 0x378 не то же самое, что физический адрес памяти 0x378. Доступ к устройствам ввода/вывода через отдельное адресное пространство портов называется *вводом/выводом через порты (PMIO)*. Современные процессоры x86 поддерживают ввод/вывод как с привязкой к портам, так и с привязкой к памяти.

Порты ввода/вывода и адреса ввода/вывода с отображением в памяти обычно относятся к контроллеру устройства, а не непосредственно к данным, хранящимся на устройстве. Например, в случае жесткого диска байты диска не отображаются непосредственно в адресное пространство. Вместо этого контроллер жесткого диска представляет интерфейс, доступный через порты ввода/вывода или адреса ввода/вывода с отображением в памяти, который позволяет центральному процессору запрашивать операции чтения или записи в области на диске.

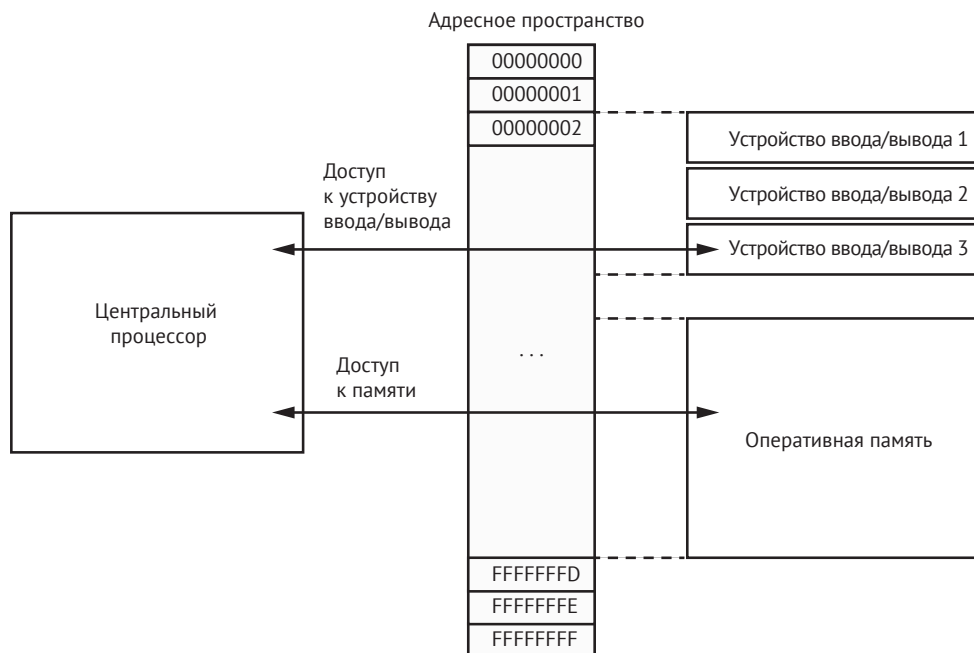


Рис. 7-14. Ввод/вывод через память

## УПРАЖНЕНИЕ 7-2: Познакомьтесь с аппаратными устройствами в вашей жизни

Выберите несколько вычислительных устройств, которыми вы владеете или пользуетесь, – например, ноутбук, смартфон или игровую приставку. Ответьте на следующие вопросы о каждом устройстве. Возможно, вы сможете найти ответы, посмотрев настройки на самом устройстве<sup>1</sup>, а возможно, вам придется провести небольшое исследование в интернете.

- Какой тип процессора установлен в устройстве?
- Является ли процессор 32-битным или 64-битным (или каким-то другим)?
- Какова тактовая частота процессора?
- Имеет ли процессор кеш-память L1, L2 или L3? Если да, то сколько?
- Какую архитектуру набора команд использует процессор?
- Сколько ядер имеет процессор?
- Сколько и какой тип оперативной памяти имеет устройство?
- Сколько и какой тип вторичного хранилища у устройства?
- Какие устройства ввода/вывода есть у устройства?

<sup>1</sup> В системе Windows ответы на важную часть заданных ниже вопросов можно получить, обратившись по адресу: **Панель управления > Система**. – *Прим. ред.*

## Связь по шине

На данном этапе мы рассмотрели роли памяти, центрального процессора и устройств ввода/вывода в компьютере. Мы также коснулись связи центрального процессора с памятью и устройствами ввода/вывода через адресное пространство памяти. Теперь давайте рассмотрим подробнее, как центральный процессор взаимодействует с памятью и устройствами ввода/вывода.

*Шина* – это аппаратная система связи, используемая компьютерными компонентами. Существует множество вариантов реализации шины, но на заре компьютеров шина представляла собой набор параллельных проводов, каждый из которых передавал электрический сигнал. Это позволяло передавать несколько битов данных параллельно, напряжение на каждом проводе представляло один бит. Современные конструкции шин не всегда так просты, но смысл тот же.

Существует три распространенных типа шин, используемых для связи между процессором, памятью и устройствами ввода/вывода. *Шина адреса* действует как селектор адреса памяти, к которому процессор хочет получить доступ. Например, если программа хочет сделать запись в адрес 0x2FE, процессор записывает 0x2FE на шину адреса. *Шина данных* передает значение, считанное из памяти, или значение, которое должно быть записано в память. Так, если центральный процессор хочет записать в память значение 25, то 25 записывается на шину данных. Или если процессор читает данные из памяти, то процессор считывает значение с шины данных. Наконец, *шина управления* управляет операциями, выполняемыми по двум другим шинам. Например, центральный процессор использует шину управления для указания того, что скоро произойдет операция записи, или шина управления может нести сигнал, указывающий на статус операции. На рис. 7-15 показано, как центральный процессор использует шину адреса, шину данных и шину управления для чтения памяти.

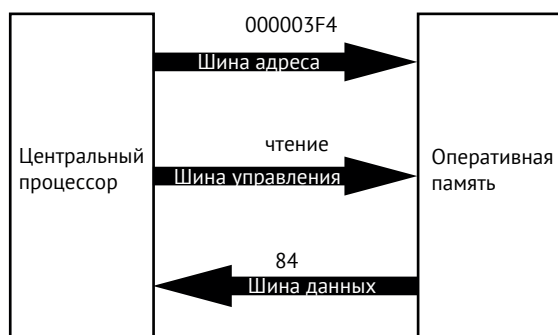


Рис. 7-15. Центральный процессор запрашивает чтение значения по адресу 3F4, и ему возвращается значение 84

В примере, показанном на рис. 7-15, процессору необходимо прочитать значение, хранящееся по адресу памяти 000003F4. Для этого процессор записывает 000003F4 на шину адреса. Процессор также устанавливает



ливают определенное значение на шине управления, указывая, что он хочет выполнить операцию чтения. Эти обновления шин служат входом для контроллера памяти (схемы, управляющей взаимодействием с оперативной памятью), сообщая ему, что процессор хочет прочитать значение, хранящееся по адресу 000003F4 в оперативной памяти. В ответ на это контроллер памяти извлекает значение, хранящееся по адресу 000003F4 (84 в данном примере), и записывает его на шину данных, которую затем может прочитать центральный процессор.

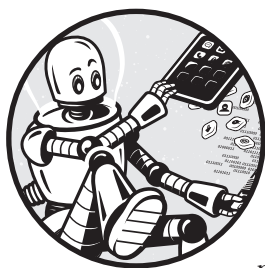
## Выводы

В этой главе мы рассмотрели аппаратное обеспечение компьютера: центральный процессор (CPU) для выполнения инструкций, память с произвольным доступом (RAM), которая хранит инструкции и данные, пока есть питание, и устройства ввода/вывода (I/O), которые взаимодействуют с внешним миром. Вы узнали, что память состоит из однобитных ячеек памяти, реализованных с помощью триггеров в SRAM и с помощью транзистора и конденсатора в DRAM. Мы рассмотрели, как работает адресация памяти, где каждый адрес относится к байту памяти. Вы узнали об архитектурах процессоров, включая x86 и ARM. Мы изучили внутреннюю работу процессоров, рассмотрели регистры, АЛУ и блок управления. Мы рассмотрели вторичное хранилище и другие типы устройств ввода/вывода и, наконец, изучили обмен данными по шине.

В следующей главе мы перейдем от аппаратного обеспечения к тому, что делает компьютеры уникальными среди других устройств, а именно к программному обеспечению. Мы изучим низкоуровневые инструкции, которые выполняют процессоры, и посмотрим, как эти инструкции можно комбинировать для выполнения полезных операций. У вас будет возможность написать программу на языке ассемблера и использовать отладчик для изучения машинного кода.

# 8

## МАШИННЫЙ КОД И ЯЗЫК АССЕМБЛЕРА



Мы рассмотрели физические части компьютера: процессор, оперативную память и устройства ввода/вывода. Понимание аппаратной части компьютера очень важно, но аппаратная часть – это только половина истории. Волшебство компьютеров заключается в программном обеспечении. Именно программы превращают компьютер из устройства с определенным назначением в универсальное устройство, легко приобретающее новые возможности!

В этой главе мы рассмотрим низкоуровневое программное обеспечение – машинный код и язык ассемблера. Я обнаружил, что эти темы лучше всего изучать с помощью интерактивного подхода, поэтому большая часть содержания этой главы представлена в проектах.

### Определение программных терминов

Для обсуждения программного обеспечения (*software*) необходимо ввести несколько терминов. Инструкции, которые указывают компьютеру, что делать, называются *программным обеспечением*, что отличает программное обеспечение от аппаратного обеспечения (*hardware*), из кото-

рого компьютер состоит физически. Упорядоченный набор таких инструкций, выполняющий определенную задачу, называется *программой*, а процесс написания таких программ называется *программированием*.

Термин «приложение» (*application*) иногда используется как синоним программы, хотя *приложение* обычно относится к программе или ее части, которая взаимодействует непосредственно с человеком, а не к программам, которые взаимодействуют с программным или аппаратным обеспечением. Приложение также может состоять из нескольких программ, работающих вместе. Сокращенное слово «app» вошло в обиход примерно в 2008 году и имеет несколько другой смысл, о чем я расскажу в главе 13.

Другое название набора программных инструкций – *компьютерный код*, или просто *код*. Процессоры выполняют *машинный код*, в то время как разработчики программного обеспечения обычно пишут исходный код на языке программирования более высокого уровня. Термин «исходный код» относится к тексту программы, изначально написанному разработчиками. Такой код обычно написан в форме, непонятной непосредственно центральному процессору, поэтому для его запуска на компьютере необходимо выполнить дополнительные действия. Более подробно об исходном коде и высокоуровневых языках программирования я расскажу в главе 9, а сейчас давайте рассмотрим основу программного обеспечения: машинный код.

Машинный код – это программное обеспечение в виде двоичных инструкций *машинного языка*. Как описано в главе 7, архитектура процессора определяет, какие инструкции понимает данный процессор. Подобно тому, как человеческий язык формируется из словарного запаса, машинный язык формируется из списка инструкций, известных семейству процессоров. Слова, объединенные в предложения, передают смысл, и инструкции процессора, объединенные в программы, делают то же самое.

Независимо от того, как программа была изначально написана (а существует множество способов написания программ), в конечном итоге она должна выполняться на центральном процессоре в виде серии инструкций машинного языка. Как вы, возможно, и ожидаете, инструкции процессора сводятся к серии нулей и единиц, как и все остальное, с чем имеет дело компьютер. Стоит повторить: независимо от того, как изначально была написана программа, независимо от того, какой язык программирования использовался, независимо от того, какие технологии были задействованы, в конечном итоге эта программа превращается в серию нулей и единиц, представляющую собой инструкции, которые может выполнить центральный процессор.

Несколько лет назад у меня была работа, связанная с диагностикой сбоев программного обеспечения. Часто проблемы, которые я анализировал, возникали в программном обеспечении, написанном другими компаниями. У меня не было исходного кода этого программного обеспечения, не было достаточно информации о том, как оно должно работать, и все же моя работа заключалась в том, чтобы определить, почему программа дает сбой! У меня был коллега, который воспринимал это спокойно, и он регулярно напоминал мне, что «это всего лишь код». Другими словами, сбоящее программное обеспечение было просто на-

бором из единиц и нулей, которые процессор интерпретировал как инструкции. Если процессор может разобраться в коде, то и вы можете.

## Пример машинной инструкции

Я думаю, что самый простой способ перейти к теме машинного кода – рассмотреть пример. Давайте рассмотрим конкретную машинную инструкцию, которую понимают процессоры семейства ARM. Как вы помните, процессоры ARM используются в большинстве смартфонов, поэтому эта инструкция, вероятно, будет понятна вашему телефону.

Инструкция в нашем примере говорит процессору переместить число 4 в регистр r7, один из нескольких регистров общего назначения в процессорах ARM. Вспомните из нашего обсуждения компьютерного оборудования ранее, что регистр – это небольшое место для хранения данных в процессоре. Инструкция ARM для выполнения этой операции в двоичном виде выглядит следующим образом:

---

```
111000111010000000111000000000100
```

---

Давайте рассмотрим, как процессор ARM будет выполнять эту инструкцию (см. рис. 8-1). Обратите внимание, что мы пропускаем некоторые биты, которые не имеют отношения к нашему обсуждению.



**Детали:**

- условие = 1110 = всегда выполнять (безусловно);
- непосредственно = 1 = значение находится в последних 8 битах инструкции;
- опкод = 1101 = переместить значение, обычно представляется как «mov»;
- регистр назначения = 0111 = r7;
- непосредственное значение = 0000 0100 = 4 десятичных разряда.

Рис. 8-1. Расшифровка инструкции ARM

Секция *условие* определяет условия, при которых инструкция должна быть выполнена. 1110 означает, что инструкция не является условной, поэтому процессор должен выполнять ее всегда. Хотя в данном примере это не так, но некоторые инструкции должны выполняться только при определенных условиях. Следующие два бита, 00 в данном примере, не имеют отношения к нашему обсуждению, поэтому мы их пропустим. Бит *непосредственно* говорит нам, обращаемся ли мы к значению в регистре или к значению, указанному в самой инструкции (известному как

*непосредственное значение*). В данном случае непосредственный бит равен 1, поэтому мы используем число, указанное в инструкции. Если бы непосредственный бит был равен 0, то регистр, к которому следует обратиться, был бы указан в других битах инструкции. *Опкод* (*код операции*) обозначает операцию, которую должен выполнить процессор. В данном случае это операция *mov*, что означает, что процессор должен переместить некоторые данные. *Регистр назначения* 0111 говорит нам, что мы перемещаем значение в регистр r7 (0111 – двоичное число семь).

Наконец, само *непосредственное значение* 00000100 – это 4 в десятичной системе счисления, т. е. число, которое мы хотим переместить в регистр r7. Резюмируем: эта двоичная последовательность говорит процессору ARM переместить число 4 в регистр r7.

Процессор всегда работает в двоичном формате, но большинству людей трудно разобраться со всеми этими нулями и единицами. Давайте представим ту же инструкцию в шестнадцатеричном виде, чтобы ее было легче читать:

---

e3a07004

---

Разве это не лучше? Ну, может быть, и нет. Она компактнее и легче воспринимается, чем двоичная, но ее смысл все равно не очевиден. К счастью для нас, есть еще один способ представить эту инструкцию: язык ассемблера. *Язык ассемблера* – это язык программирования, в котором каждое выражение непосредственно представляет собой инструкцию машинного языка. Каждый тип машинного языка имеет соответствующий язык ассемблера – x86 ассемблер, ARM ассемблер и т. д. Операции на языке ассемблера состоят из *мнемоникодов*, которые представляют собой опкоды процессора, плюс все необходимые операнды (например, регистр или числовое значение). Мнемоникод представляет собой читабельную форму операционного кода, что позволяет программистам на языке ассемблера использовать в своем коде *mov* вместо 1101. Та же самая инструкция ARM, рассмотренная ранее, может быть представлена с помощью следующего выражения языка ассемблера:

---

mov r7, #4

---

По сравнению с соответствующими двоичными и шестнадцатеричными представлениями, это выражение, безусловно, лучший способ сказать: «Переместить 4 в регистр r7!» По крайней мере, его легче прочитать человеку. При этом помните, что выражение на языке ассемблера создано просто для удобства людей. Центральный процессор никогда не выполняет инструкции в текстовом формате, он имеет дело только с двоичной формой инструкции. Если программист пишет программу на языке ассемблера, то перед запуском программы на компьютере инструкции ассемблера должны быть преобразованы в машинный код. Для

этого и используется *ассемблер*<sup>1</sup> – программа, которая переводит команды языка ассемблера в машинный код. Текстовый файл на языке ассемблера подается в программу-ассемблер, и на выходе получается файл двоичных объектов, содержащий машинный код, как показано на рис. 8-2.

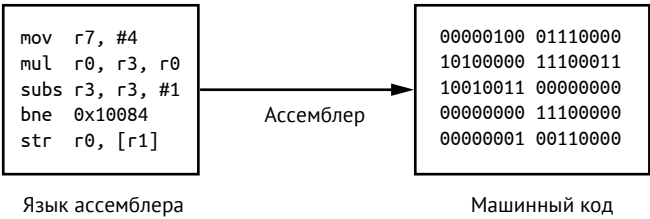


Рис.8-2: Ассемблер превращает язык ассемблера в машинный код

## Вычисление факториала в машинном коде

Теперь, когда мы рассмотрели одну инструкцию ARM, давайте посмотрим, как можно объединить несколько инструкций для выполнения полезной задачи. Давайте рассмотрим машинный код ARM, который вычисляет факториал целого числа. Как вы, возможно, помните из курса математики, факториал числа  $n$  (записывается как  $n!$ ) – это произведение целых положительных чисел, меньших или равных  $n$ . Так, например, факториал числа 4 равен:

$$4! = 4 \times 3 \times 2 \times 1 = 24.$$

Теперь, когда у нас есть определение факториала, давайте посмотрим, как реализовать вычисление факториала в машинном коде ARM. Для простоты мы не будем рассматривать весь программный код, а посмотрим только ту часть, которая выполняет алгоритм вычисления факториала. Мы предполагаем, что изначально значение  $n$  хранится в регистре  $r0$  и что по завершении кода результат вычисления также хранится в  $r0$ .

Машинный код, как и любые другие данные, с которыми имеет дело компьютер, должен быть загружен в память, прежде чем процессор сможет получить к нему доступ. Ниже приведено представление нашего машинного кода в виде 32-битных (4-байтовых) шестнадцатеричных значений, а также адрес памяти каждого значения.

Адрес	Данные
0001007c	e2503001
00010080	da000002
00010084	e0000093
00010088	e2533001
0001008c	1afffffc

<sup>1</sup> *Assembler* переводится, как сборщик, монтажник. – Прим. ред.

Когда наш код загружается в память, логика вычисления факториала начинается с адреса 0001007с. Давайте рассмотрим содержимое памяти, начиная с этого адреса. Обратите внимание, что 0001007с – это не какой-то магический адрес, просто в данном примере код загрузился именно по этому адресу. Также обратите внимание, что значения адресов памяти увеличиваются на 4, потому что каждое значение данных требует 4 байта памяти. Каждая инструкция ARM имеет длину 4 байта, поэтому эти данные представляют собой пять инструкций ARM.

Если смотреть на эти инструкции как на шестнадцатеричные значения, это не даст нам большого понимания их смысла, поэтому давайте расшифруем их, чтобы понять смысл этой программы. В следующем листинге я преобразовал шестнадцатеричные значения данных в соответствующие им мнемокоды языка ассемблера. Если вас интересует этот вопрос, то вам не нужно уметь делать ручной перевод машинного языка на язык ассемблера! Для этого существует программное обеспечение, называемое *дизассемблером*. Пока что для вас в роли дизассемблера выступает эта книга. Вот каждая инструкция в паре с ее ассемблерным выражением.

---

Адрес	Данные	На языке ассемблера
0001007с	e2503001	subs r3, r0, #1
00010080	da000002	ble 0x10090
00010084	e0000093	mul r0, r3, r0
00010088	e2533001	subs r3, r3, #1
0001008с	1affffffc	bne 0x10084
00010090	---	

---

Центральный процессор выполняет эти инструкции последовательно, пока не встретит инструкцию перехода (например, *ble* или *bne*), которая может заставить его перейти к другой части программы. Адрес 00010090 отмечает конец логики вычисления факториала. По достижении этого адреса результат факториала сохраняется в регистре r0. В этот момент центральный процессор выполняет любую инструкцию, находящуюся по адресу 00010090.

Вам может быть интересно, как эти инструкции представляют вычисление факториала. Для большинства людей беглого взгляда на такие инструкции недостаточно, чтобы понять скрытый за ними замысел. Пошаговый подход и отслеживание значений регистров по мере выполнения каждой инструкции может помочь вам понять программу. Я предоставляю вам некоторую необходимую справочную информацию, а затем вы сможете попробовать оценить, как работает эта программа.

Чтобы разобраться в этой программе, сначала вам нужно ознакомиться с каждой используемой инструкцией. В табл. 8-1 я даю вам объяснение каждой инструкции в этой программе. В этой таблице я использовал условные названия регистров, такие как *Rd* и *Rn*. Когда вы будете просматривать ассемблерный код, то увидите, что вместо них используются реальные имена регистров, например r0 или r3. Порядок

следования операндов в коде соответствует порядку следования операндов в табл. 8-1. Например, `subs r3, r0, #1` означает вычитание 1 из значения, хранящегося в `r0`, и сохранение результата в `r3`.

**Таблица 8-1.** Объяснение некоторых инструкций ARM

Инструкция	Пояснение
<code>subs Rd, Rn, #Const</code>	<b>Вычитание</b> Вычитает постоянное значение <code>Const</code> из значения, хранящегося в регистре <code>Rn</code> , и сохраняет результат в регистре <code>Rd</code> . Другими словами, $Rd = Rn - Const$
<code>mul Rd, Rn, Rm</code>	<b>Умножение</b> Перемножает значение, хранящееся в регистре <code>Rn</code> , и значение, хранящееся в регистре <code>Rm</code> , и сохраняет результат в регистре <code>Rd</code> . Другими словами, $Rd = Rn \times Rm$
<code>ble Addr</code>	<b>Переход, если результат меньше или равен</b> Если результат предыдущей операции был меньше или равен 0, то переходит к инструкции по адресу <code>Addr</code> . В противном случае переходит к следующей инструкции
<code>bne Addr</code>	<b>Переход, если не равно</b> Если результат предыдущей операции не равен 0, то переходит к инструкции по адресу <code>Addr</code> . В противном случае переходит к следующей инструкции

## ПЕРЕХОД И РЕГИСТР СОСТОЯНИЯ

Инструкции перехода на самом деле не смотрят на числовой результат предыдущей инструкции. Процессоры ARM, как и большинство центральных процессоров, имеют регистр, предназначенный для отслеживания состояния. Этот регистр состояния имеет 32 бита, и каждый бит соответствует определенному флагу состояния. Например, бит 31 – это флаг `N`, и он устанавливается равным 1, когда в результате выполнения команды получается отрицательное число. Только определенные инструкции влияют на состояние этих флагов. Например, инструкция `subs` изменяет состояние флагов. Если определенная операция вычитания приводит к отрицательному результату, устанавливается флаг `N`, в противном случае он очищается. Другие инструкции, включая инструкции перехода, затем смотрят на флаги состояния, чтобы определить, что делать. Этот путь может показаться запутанным, но на самом деле это упрощает работу с такими инструкциями, как `bne`, когда процессор может совершать переход (или нет) на основе значения одного бита.

Мы подошли к концу объяснения этой темы, оставшаяся часть главы состоит из упражнения и двух проектов. В упражнении 8-1 вы пройдете через пример программы вычисления факториала, используя пояснения из табл. 8-1, чтобы понять, как работает каждая инструкция.



## УПРАЖНЕНИЕ 8-1: Используйте свой мозг в качестве процессора

Попробуйте выполнить следующую программу на языке ассемблера ARM в уме или воспользуйтесь карандашом и бумагой.

---

Адрес	На языке ассемблера
0001007c	subs r3, r0, #1
00010080	ble 0x10090
00010084	mul r0, r3, r0
00010088	subs r3, r3, #1
0001008c	bne 0x10084
00010090	---

---

Предположим, что входное значение  $n = 4$  первоначально хранится в `r0`. Когда программа дойдет до инструкции по адресу 00010090, вы достигнете конца кода, и в `r0` должно быть ожидаемое выходное значение 24. Я рекомендую для каждой инструкции отслеживать значения `r0` и `r3` до и после ее выполнения. Проработайте все инструкции, пока не дойдете до инструкции 00010090, и посмотрите, получили ли вы ожидаемый результат. Если все работало правильно, вы должны были пройти через одни и те же инструкции несколько раз, и это не случайно. Ответ находится в приложении А.

Знакомство с языком ассемблера на бумаге – это отличное начало, но еще лучше попробовать язык ассемблера на компьютере.

### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 12 на стр. 181, где вы сможете создать код для вычисления факториала и протестировать его во время выполнения. Также посмотрите проект № 13 на стр. 194, где вы сможете изучить некоторые дополнительные подходы к тестированию машинного кода.*

## Выводы

В этой главе мы рассмотрели машинный код – серию инструкций для конкретного процессора, представленных в виде байтов в памяти. Вы узнали, как кодируется пример инструкции для процессора ARM, и увидели, как эта инструкция может быть представлена на языке ассемблера. Вы также узнали, что язык ассемблера – это разновидность исходного кода, специальная читабельная форма машинного кода. Мы увидели, как можно комбинировать несколько команд языка ассемблера для выполнения полезных операций.

В следующей главе мы рассмотрим высокоуровневые языки программирования. Такие языки обеспечивают отступление от набора инструкций процессора, позволяя разработчикам писать исходный код, который легче понять и переносить на различные аппаратные платформы компьютеров.

## ПРОЕКТ № 12: Факториал на ассемблере

Необходимые условия: Raspberry Pi, работающий на Raspberry Pi OS. Я рекомендую вам посмотреть приложение В и прочитать весь раздел «Raspberry Pi» на стр. 416. Это поможет вам настроить и научит использовать Raspberry Pi OS, включая работу с файлами, которую вы будете активно выполнять в проектах этой главы.

В этом проекте вы создадите программу для вычисления факториала на языке ассемблера, подобную той, которую мы рассмотрели ранее в этой главе. Затем протестируете сгенерированный машинный код. Программа для вычисления факториала включает в себя некоторый дополнительный код, помимо того, что был рассмотрен выше. В частности, программа также считывает из памяти начальное значение  $n$ , записывает результат обратно в память и в конце передает управление операционной системе.

### ИНСТРУКЦИИ И ДИРЕКТИВЫ АССЕМБЛЕРА

Поскольку вы будете использовать дополнительный код, я привожу табл.8-2, где объясняются используемые в коде инструкции. Некоторые из этих инструкций вы уже видели в табл. 8-1, но я привожу здесь все инструкции для удобства.

**Таблица 8-2.** Инструкции ARM, используемые в проекте № 12

Инструкция	Пояснение
<code>ldr Rd, Addr</code>	<b>Загрузка из памяти в регистр</b> Считывает значение по адресу <code>Addr</code> и помещает его в регистр <code>Rd</code>
<code>str Rd, Addr</code>	<b>Запись регистра в память</b> Записывает значение из регистра <code>Rd</code> на адрес <code>Addr</code>
<code>mov Rd, #Const</code>	<b>Перенос постоянного значения в регистр</b> Переносит постоянное значение <code>Const</code> в регистр <code>Rd</code>
<code>svc</code>	<b>Выполнение системного вызова</b> Делает запрос к операционной системе

Инструкция	Пояснение
<code>subs Rd, Rn, #Const</code>	<b>Вычитание</b> Вычитает постоянное значение <code>Const</code> из значения, хранящегося в регистре <code>Rn</code> , и сохраняет результат в регистре <code>Rd</code> . Другими словами, $Rd = Rn - Const$
<code>mul Rd, Rn, Rm</code>	<b>Умножение</b> Перемножает значение, хранящееся в регистре <code>Rn</code> , и значение, хранящееся в регистре <code>Rm</code> , и сохраняет результат в регистре <code>Rd</code> . Другими словами, $Rd = Rn \times Rm$
<code>ble Addr</code>	<b>Переход, если результат меньше или равен</b> Если результат предыдущей операции был меньше или равен 0, то переходит к инструкции по адресу <code>Addr</code> . В противном случае переходит к следующей инструкции
<code>bne Addr</code>	<b>Переход, если не равно</b> Если результат предыдущей операции не равен 0, то переходит к инструкции по адресу <code>Addr</code> . В противном случае переходит к следующей инструкции

При написании кода на языке ассемблера разработчики также используют директивы ассемблера. Это не инструкции ARM, а команды ассемблеру. Эти директивы начинаются с точки, поэтому их легко отличить от инструкций. В следующем коде вы также увидите текст, за которым следует двоеточие – это метки, т. е. имена, присвоенные адресу памяти. Поскольку при написании кода мы не знаем, где в памяти будут находиться инструкции, мы обозначаем места в памяти метками, а не адресами памяти<sup>1</sup>. Еще один момент: знак @ означает, что следующий за ним текст (на той же строке) является комментарием. Я добавил комментарии, чтобы объяснить действия программы, но вы можете их не вводить.

## Ввод и просмотр кода

Теперь справочной информации достаточно. В конце концов, это же проект! Пора вводить код. Используйте текстовый редактор по вашему выбору, чтобы создать новый файл с именем `fact.s` в корне вашей домашней папки. Подробные шаги по использованию текстовых редакторов на Raspberry Pi OS приведены в разделе «Работа с файлами и папками» документации Raspberry Pi на стр. 421. Введите в текстовый редактор следующий код ассемблера ARM (не обязательно сохранять отступы и пустые строки, но обязательно сохраняйте переносы строк, при этом лишние переносы

<sup>1</sup> В реальные адреса (выраженные в числах) метки преобразуются программой-ассемблером автоматически. Заметьте, что в дизассемблированном коде (т. е. коде, полученном обратным преобразованием из машинных числовых инструкций в мнемонические обозначения, см. далее раздел об отладке), в отличие от исходного кода, написанного человеком, вместо условных названий меток всегда стоят числовые значения реальных адресов в памяти. – Прим. ред.

строк не повредят программу). Не волнуйтесь, если вы еще не понимаете всего этого кода, я объясню то, что вам нужно знать, после ввода.

```
.global _start❶

.text❷
_start:❸
    ldr r1, =n @ установить r1 = адресу переменной n❹
    ldr r0, [r1] @ установить r0 = величине переменной n
    subs r3, r0, #1 @ установить r3 = r0 - 1
    ble end @ перейти к метке end, если r3 <= 0
loop:
    mul r0, r3, r0 @ установить r0 = r3 x r0
    subs r3, r3, #1 @ уменьшить r3 на единицу
    bne loop @ перейти к метке loop, если r3 > 0
end:
    ldr r1, =result @ установить r1 = адресу результата❺
    str r0, [r1] @ сохранить r0 как результат

@ Выход из программы
mov r0, #0❻
mov r7, #1
svc 0

.data❷
n: .word 5❸
result: .word 0
```

После ввода кода сохраните его в текстовом редакторе как *fact.s* в корне домашней папки. Давайте пройдемся по этому коду, начиная с директив и меток.

Как упоминалось ранее, текст, за которым следует двоеточие, например `_start:`, является меткой для ячейки памяти ❸. Метка `_start` отмечает точку, с которой начинается выполнение программы.

Директива `.global` в первой строке делает метку `_start` видимой для компоновщика ❶ (о компоновщике мы поговорим через минуту), чтобы ее можно было установить как точку входа в программу. Директива `.text` сообщает ассемблеру, что следующие за ней строки являются инструкциями ❷.

В конце кода директива `.data` сообщает ассемблеру, что следующие за ней строки – это данные ❷. В секции данных программа сохраняет два 32-битных значения, каждое из которых обозначено директивой `.word` ❸. Первое – это значение *n*, первоначально установленное как 5. Второе – это результат, изначально установленный в значение 0. В данном контексте «word» означает 4 байта, или 32 бита.

Теперь давайте рассмотрим функциональные дополнения к коду, выходящие за рамки того, что было описано в главе. Теперь у нас есть код, который загружает число *n* из памяти, сохраняет результат факториала

в памяти и выходит из программы. Первые две инструкции в `_start` загружают значение `n` из ячейки в памяти ❹. Инструкция `ldr` загружает значение в регистр. Мы ссылаемся на адрес `n` через `=n`. В следующей строке `[r1]` заключено в скобки, потому что программа обращается к значению, хранящемуся по адресу `r1`.

Две инструкции, следующие за меткой `end`, сохраняют результат в ячейке памяти ❺. Первая инструкция перемещает адрес ячейки памяти с именем `result` в регистр `r1`. После этого код сохраняет значение из регистра `r0` (которое оказывается вычисленным факториалом) по адресу памяти результата, на который ссылается `r1`.

Последние три инструкции в секции `.text` используются для аккуратного выхода из программы ❻. Для этого требуется помощь операционной системы, поэтому я пропущу подробности этих инструкций, пока мы не рассмотрим операционные системы в главе 10.

## АССЕМБЛИРОВАНИЕ, КОМПОНОВКА И ЗАПУСК

Теперь у вас есть текстовый файл с инструкциями на языке ассемблера, но это не тот формат, который может выполнить компьютер. Вам нужно превратить инструкции на языке ассемблера в байты машинного кода с помощью двухэтапного процесса. Сначала нужно преобразовать инструкции в байты машинного кода с помощью *ассемблера*. Результатом этого процесса является *объектный файл* – файл, который содержит байты вашей программы, но его формат еще не окончательно готов для выполнения программы. Далее необходимо использовать программу, называемую *компоновщиком*, чтобы превратить объектный файл в исполняемый, который может быть запущен операционной системой<sup>1</sup>. Этот процесс показан на рис. 8-3.

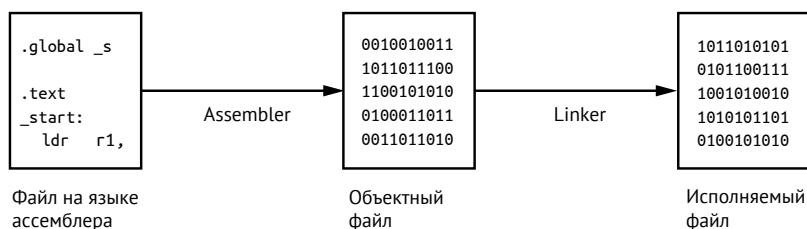


Рис. 8-3. В результате ассемблирования и компоновки получается исполняемый файл

Вы можете задаться вопросом, зачем нужен этот двухэтапный процесс? Если вы собираете несколько исходных файлов, которые работают вместе как одна программа, каждый исходный файл в результате ассемблирования преобразуется в объектный файл. Затем компоновщик объединяет

<sup>1</sup> Компоновка по-английски обозначается словом *link*, потому что процесс компоновки (*linking*) в среде русскоязычных программистов часто называют *линкованием*, а программу (*linker*) – *линковщиком*. – Прим. ред.

различные объектные файлы в один исполняемый. Это позволяет компоновать созданные ранее объектные файлы так, как нужно. В данном случае у вас получается только один объектный файл, и компоновщик просто преобразует его в формат, готовый к выполнению.

Давайте ассемблируем наш код:

---

```
$ as -o fact.o fact.s
```

---

Инструмент `as` – это GNU Assembler, который превращает ваши выражения на языке ассемблера в машинный код. Эта команда записывает сгенерированный машинный код в файл `fact.o`, являющийся объектным файлом. Ассемблер может выдать предупреждение, если ваш файл `fact.s` не заканчивается переводом строки, но вы можете смело игнорировать это предупреждение.

После сборки исходного кода в объектный файл необходимо использовать компоновщик GNU (`ld`) для преобразования объектного файла в исполняемый:

---

```
$ ld -o fact fact.o
```

---

Эта команда принимает `fact.o` в качестве входного файла и выдает исполняемый файл с именем `fact`. На этом этапе вы можете запустить свою программу с помощью следующей команды:

---

```
$ ./fact
```

---

Результатом выполнения этой команды будет немедленный переход на следующую строку без какого-либо вывода. Это потому, что ваша программа на самом деле не выводит никакого текста на экран. Она просто вычисляет факториал, сохраняет результат в памяти, а затем завершает работу. Чтобы взаимодействовать с пользователем, программа должна запросить помощь у операционной системы. Однако, поскольку при составлении этой программы мы стремились к минимализму, вам не нужно этого делать.

## **Запуск программы с отладчиком**

Если ваша программа ничего не выводит, то как вы можете узнать, что она делает? Вы можете использовать отладчик – программу, которая может исследовать процесс во время его выполнения<sup>1</sup>. Отладчик может подключиться к запущенной программе, а затем остановить ее выполнение. Пока программа остановлена, отладчик может проверить регистры и память

---

<sup>1</sup> Отладчик по-английски *debugger*, поэтому в среде русскоязычных программистов его часто так и называют дебагером. – Прим. ред.

объекта. Здесь в качестве отладчика используется GNU Debugger, `gdb`, а объектом является программа `fact`.

Чтобы начать, просто выполните следующую команду:

---

```
$ gdb fact
```

---

Когда вы выполняете эту команду, `gdb` загружает файл `fact`, но никакие инструкции еще не выполняются. В командной строке (`gdb`) введите следующее, чтобы просмотреть начальный адрес программы:

---

```
(gdb) info files
```

---

Вы должны увидеть строку, подобную этой, хотя конкретный адрес может отличаться:

---

```
Entry point: 0x10074
```

---

Это говорит о том, что точкой входа программы является адрес `0x10074`. Помните, когда вы писали программу, вы не знали, какие адреса памяти будут использоваться, поэтому вместо них вы использовали метки.

Теперь, когда программа создана и загружена в память, у вас есть реальные адреса памяти для проверки. Адрес точки входа соответствует метке `_start`, поскольку именно с нее начинается программа. Теперь вы можете использовать `gdb`, чтобы дизассемблировать машинный код, начиная с точки входа программы. *Дизассемблирование* – это процесс просмотра байтов машинного кода как инструкций языка ассемблера. Следующая команда использует `0x10074` в качестве начального адреса; если ваша точка входа другая, используйте ваш адрес.

---

```
(gdb) disas 0x10074
```

```
Dump of assembler code for function _start:
```

```
0x00010074 <+0>:    ldr     r1, [pc, #40]    ; 0x100a4 <end+20>
0x00010078 <+4>:    ldr     r0, [r1]
0x0001007c <+8>:    subs   r3, r0, #1
0x00010080 <+12>:   ble     0x10090 <end>
```

---

После выполнения этой команды вы должны увидеть первые четыре инструкции в дизассемблированном виде, как показано здесь. В начале выведенного фрагмента отладчик сообщает, что это «фрагмент (дамп<sup>1</sup>) ассемблерного кода для функции `_start`». По умолчанию `gdb` разбира-

---

<sup>1</sup> Дамп (англ. *dump* – мусорная куча, свалка; выбрасывать, вываливать) – мгновенный «снимок» информации, содержащее рабочей памяти в определенный момент времени. – Прим. ред.

ет только несколько инструкций. Это хорошее начало, но лучше увидеть всю программу целиком. Для этого вам нужно указать `gdb` конечный адрес кода, который вы хотите увидеть. Если посмотрите на предыдущий код, который вы ввели в *fact.s*, то увидите, что всего в вашей программе 12 инструкций. Каждая инструкция занимает 4 байта, поэтому длина программы должна составлять 48 байт. Это означает, что ваша программа должна заканчиваться через 48 байт после начального адреса, поэтому конечный адрес должен быть `0x00010074 + 48`. Вы можете выполнить это сложение вручную или в программе-калькуляторе, но, поскольку вы находитесь в `gdb`, можете попросить его выполнить эти математические вычисления за вас и найти конечный адрес вашей программы (опять же, замените `0x10074` на адрес точки входа, который вам нужен):

---

```
(gdb) print/x 0x00010074 + 48
$1 = 0x100a4
```

---

Вывод команды `print` может показаться немного странным поначалу. Символ `/x` в команде означает «вывести результат в шестнадцатеричном виде». Если вы посмотрите на вывод, то левое значение (`$1`) – это *вспомогательная переменная*, временное место хранения в `gdb`. Сохранение значения во вспомогательной переменной – это способ `gdb` сделать так, чтобы вам было легко вернуться к этому результату позже. Значение после знака равенства – это выведенное значение, результат вычисления, в данном случае `0x100a4`.

Итак, теперь вы знаете конечный адрес (`0x100a4`) и можете попросить `gdb` дизассемблировать всю программу. Обратите внимание, что, если ваш начальный адрес отличается от моего, нужно заменить два адреса в следующей команде.

---

```
(gdb) disas 0x10074,0x100a4
Dump of assembler code from 0x10074 to 0x100a4:
0x00010074 <_start+0>:    ldr     r1, [pc, #40] ; 0x100a4 <end+20>❶
0x00010078 <_start+4>:    ldr     r0, [r1]
0x0001007c <_start+8>:    subs    r3, r0, #1
0x00010080 <_start+12>:   ble     0x10090 <end>
0x00010084 <loop+0>:     mul     r0, r3, r0
0x00010088 <loop+4>:     subs    r3, r3, #1
0x0001008c <loop+8>:     bne     0x10084 <loop>
0x00010090 <end+0>:     ldr     r1, [pc, #16] ; 0x100a8 <end+24>❷
0x00010094 <end+4>:     str     r0, [r1]
0x00010098 <end+8>:     mov     r0, #0
0x0001009c <end+12>:    mov     r7, #1
0x000100a0 <end+16>:    svc     0x00000000
```

---

Это очень похоже на то, что вы первоначально ввели в *fact.s* и собрали, только теперь каждой инструкции присвоен адрес, а ссылки на `n` и `result` заменены смещениями в памяти относительно регистра счетчика команд (например, `[pc, #40]` ❶). Регистр счетчика команд, или указатель инструкции,



хранит в памяти адрес текущей инструкции. Для простоты изложения я не буду вдаваться в подробности того, почему здесь используются смещения счетчика команд, просто знайте, что инструкции по адресам 0x10074 ❶ и 0x10090 ❷ загружают в r1 адреса памяти n и result соответственно.

## **ЗАПУСК И ТЕСТИРОВАНИЕ ПРОГРАММЫ С ПОМОЩЬЮ ТОЧЕК ОСТАНОВА ОТЛАДЧИКА**

Теперь, когда вы видите, что программа загружена в память, давайте проверим, работает ли она так, как ожидается. Для этого следует расставить точки останова на определенных инструкциях, что позволит вам исследовать состояние программы в этот конкретный момент. *Точка останова* указывает отладчику прекратить выполнение программы при достижении определенного адреса. Точка останова на определенном адресе останавливает выполнение программы непосредственно перед выполнением соответствующей инструкции. В следующих примерах команд я использую адреса, указанные в моей системе, но, если ваши адреса памяти отличаются, убедитесь, что вы используете правильные адреса.

Ваши точки останова будут следующими:

**0x10074** – начало программы;

**0x1007c** – начало логики факториала, 8 байт после первой инструкции. Когда программа дойдет до этой инструкции, в регистре r0 должно быть входное значение n, которое в программе было жестко установлено равным 5;

**0x10090** – конец логики факториала, 0x1C байт после первой инструкции. Когда программа доходит до этой инструкции, в регистре r0 должно храниться значение факториала, которое было вычислено;

**0x100a0** – последняя инструкция программы. Когда программа дойдет до этой инструкции, в ячейке памяти с меткой result должен храниться результат вычисления факториала.

Расставьте точки останова следующим образом (опять же, скорректируйте адреса, если ваш начальный адрес не 0x10074):

---

```
(gdb) break *0x10074
(gdb) break *0x1007c
(gdb) break *0x10090
(gdb) break *0x100a0
```

---

Теперь начните выполнение программы:

---

```
(gdb) run
Starting program: /home/pi/fact

Breakpoint 1, 0x00010074 in _start ()
```

---

Вы должны увидеть вот такой вывод, указывающий на то, что выполнение остановилось на первой точке останова. В этот момент программа готова к выполнению первой инструкции, и вы можете посмотреть на состояние дел. Сначала изучите регистры. На самом деле единственный, который нас интересует в данный момент, – это программный счетчик (pc), потому что мы хотим подтвердить, что текущая инструкция имеет начальный адрес 0x10074. Теперь попросите отладчик показать значение регистра pc:

---

```
(gdb) info register pc
pc 0x10074 0x10074 <_start>
```

---

Это говорит о том, что счетчик программы указывает на начальный адрес и первую точку останова, как и ожидалось. Другой способ подтвердить текущую инструкцию – это просто разобрать текущий код следующим образом:

---

```
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010074 <+0>: ldr r1, [pc, #40] ; 0x100a4 <end+20>
    0x00010078 <+4>: ldr r0, [r1]
    0x0001007c <+8>: subs r3, r0, #1
    0x00010080 <+12>: ble 0x10090 <end>
```

---

Обратите внимание на символ =>, указывающий на текущую инструкцию. Теперь, когда вы убедились, что программа готова выполнить свою первую инструкцию, можете проверить текущие значения двух помеченных адресов памяти: `n` и `result`. Они должны быть равны 5 и 0 соответственно, поскольку именно так вы определили их начальные значения в исходном коде `fact.s`. Вы можете снова использовать команду `print`, чтобы увидеть эти значения. При этом необходимо указать тип данных `int` (32-битное целое число), чтобы команда `print` знала, как отобразить эти значения.

---

```
(gdb) print (int)n
$2 = 5
(gdb) p (int)result
$3 = 0
```

---

Обратите внимание, как `p` заменяет `print` во второй команде. Сокращенные версии команд поддерживаются `gdb`, что может сэкономить вам время на наборе текста. Как видите, команда `print` позволяет легко выводить значения помеченных областей памяти!

Хотя печать значения помеченной области памяти весьма удобна, возникает вопрос: откуда команда `print` в `gdb` знает о метках, которые вы присвоили

этим ячейкам памяти в исходном файле *fact.s*? Процессор не использует эти метки, он использует только адреса памяти. Машинный код также не ссылается на эти области памяти по названиям. Отладчик способен делать это, потому что файл, в котором хранится машинный код, *fact*, также содержит символьную информацию. Эти *отладочные символы* сообщают отладчику об определенных именованных областях памяти, таких как *n* и *result*. Обычно символьная информация удаляется из исполняемых файлов перед их распространением среди конечных пользователей, но символьная информация все еще присутствует в вашем исполняемом файле *fact*.

Помня, что *n* и *result* – это просто метки для мест в памяти, как найти фактические адреса этих переменных в памяти? Один из способов заключается в выведении адреса с помощью оператора *&*, который в *gdb* означает «адрес». Таким образом, *&n* означает «адрес *n*». Теперь выведите адрес *n* и адрес *result*.

---

```
(gdb) p &n
$4 = (<data variable, no debug info> *) 0x200ac
(gdb) p &result
$5 = (<data variable, no debug info> *) 0x200b0
```

---

Это говорит о том, что значение *n* хранится по адресу *0x200ac*, а значение *result* – по адресу *0x200b0*. Обратите внимание, что это последовательные значения в памяти, поскольку и *n* и *result* имеют длину 4 байта. Вы можете просмотреть содержание в этой памяти с помощью команды *x*:

---

```
(gdb) x/2xw 0x200ac
0x200ac:      0x00000005      0x00000000
```

---

Команда *x/2xw* означает просмотр двух последовательных значений, отображаемых в шестнадцатеричном формате, каждое размера «word» (4 байта), начиная с адреса *0x200ac*. Здесь вы снова видите, что *n* равно 5, а *result* равен 0. Это просто другой способ просмотра памяти, на этот раз без использования именованных меток.

Вернемся к программе – теперь вы убедились, что начальные значения памяти установлены так, как ожидалось. Продолжите выполнение до следующей точки останова, где вы сможете убедиться, что *g0* установлено на начальное значение *n*.

---

```
(gdb) continue
Continuing.
```

```
Breakpoint 2, 0x0001007c in _start ()
```

```
(gdb) disas
Dump of assembler code for function _start:
   0x00010074 <+0>:    ldr     r1, [pc, #40]    ; 0x100a4 <end+20>
   0x00010078 <+4>:    ldr     r0, [r1]
=>  0x0001007c <+8>:    subs    r3, r0, #1
   0x00010080 <+12>:   ble     0x10090 <end>
End of assembler dump.
```

```
(gdb) info registers r0
r0                0x5        5
```

---

Из предыдущего вывода видно, что программа перешла к инструкции 0x1007c, как и ожидалось, и r0 имеет ожидаемое значение 5 (значение n). Пока все хорошо. Теперь перейдите к следующей точке останова, где r0 теперь должно быть равно рассчитанному значению факториала 5, т. е. должно составить 120. Вы можете сократить команду continue до просто c, а команду info registers до просто i r.

---

```
(gdb) c
Continuing.
```

```
Breakpoint 3, 0x00010090 in end ()
```

```
(gdb) disas
Dump of assembler code for function end:
=>  0x00010090 <+0>:    ldr     r1, [pc, #16]    ; 0x100a8 <end+24>
   0x00010094 <+4>:    str     r0, [r1]
   0x00010098 <+8>:    mov     r0, #0
   0x0001009c <+12>:   mov     r7, #1
   0x000100a0 <+16>:   svc     0x00000000
   0x000100a4 <+20>:   andeq   r0, r2, r12, lsr #1
   0x000100a8 <+24>:   strheq  r0, [r2], -r0    ; <UNPREDICTABLE>
End of assembler dump.
```

```
(gdb) i r r0
r0                0x78       120
```

---

Все выглядит хорошо. Вспомните, что на данный момент результат факториала не был сохранен в адрес памяти result. Теперь проверьте, что result не изменился:

---

```
(gdb) p (int)result
$6 = 0
```

---

Хотя результат факториала временно хранится в `g0`, он еще не был записан в память. Дойдите до конца программы (последняя точка останова) и посмотрите, обновилась ли область памяти `result`.

---

```
(gdb) c
Continuing.
```

```
Breakpoint 4, 0x000100a0 in end ()
```

```
(gdb) p (int)result
$7 = 120
```

---

Вы должны увидеть значение `result`, равное 120. Если это так, то вы молодец, ваша программа работала как надо!

## **ВЗЛОМАЙТЕ ПРОГРАММУ ДЛЯ ВЫЧИСЛЕНИЯ ДРУГОГО ФАКТОРИАЛА**

Эта программа жестко закодирована для вычисления факториала числа 5. Что, если вы захотите, чтобы она вычисляла факториал какого-то другого числа? Вы можете изменить жестко закодированное значение в исходном коде программы, перестроить код и запустить программу снова. Или вы можете написать код, который позволит пользователю ввести желаемое значение `n` во время выполнения программы. Но представьте, что у вас больше нет доступа к исходному коду и просто нужен быстрый способ изменить поведение программы во время ее работы, заменив жестко закодированное значение `n` на какое-либо значение, отличное от 5.

---

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi/fac
```

```
Breakpoint 1, 0x00010074 in _start ()
```

---

В начале после запуска (по команде `(gdb) run`) отладчик сообщает вам, что «отлаживаемая программа уже запущена», и спрашивает, запустить ли ее с начала? Для ответа предлагается ввести `y` (от *yes*, «да») или `n` (от *no*, «нет»). Как мы видим, пользователь ответил утвердительно (введя символ `y`), после чего программа *fac* запустилась в отладочном режиме с начального адреса. Теперь вы вернулись в начало программы, в точку останова 1. Вы можете изменить значение `n` в памяти, установив его равным 7, а не 5. Сначала узнайте адрес памяти, содержащий `n`, затем установите значение по этому адресу равным 7. Затем вы можете напечатать `n`, чтобы убедиться, что изменение работало.

---

```
(gdb) p &n
$8 = (<data variable, no debug info> *) 0x200ac
```

```
(gdb) set {int}0x200ac = 7
```

```
(gdb) p (int)n
$9 = 7
```

---

Теперь перейдите в конец программы и посмотрите, обновится ли `result` до ожидаемого значения факториала семи, которое равно 5040. Вы можете избавиться от двух средних точек останова (номер 2 и 3), поскольку вы хотите перейти сразу к концу:

---

```
(gdb) disable 2
(gdb) disable 3
```

```
(gdb) c
Continuing.
```

```
Breakpoint 4, 0x000100a0 in end ()
```

```
(gdb) p (int)result
$10 = 5040
```

---

Вы должны увидеть значение 5040 в результате. Если это так, то вы только что успешно взломали программу, заставив ее выполнять ваши пожелания. И все это, не касаясь исходного кода!

На данном этапе вы можете попробовать установить для `n` другие значения и посмотреть, получите ли вы ожидаемые результаты. Для этого перезапустите программу с помощью команды `run`, измените значение `n` в памяти, перейдите к последней точке останова и проверьте значение `result`. Однако если вы используете значение `n` больше 12, то получите неверный результат. Причину этого см. в ответе на упражнение 8-1 в приложении А.

Если вы позволите программе дойти до конца, процесс завершится, и вы получите сообщение типа `Inferior 1 (process 946) exited normally` («Подчиненный 1 (процесс 946) закрыт нормально»). «Подчиненный» здесь означает только то, каким образом `gdb` ссылается на отлаживаемую программу. Вы можете выйти из отладчика в любое время, введя в `gdb` команду `quit`.

## ПРОЕКТ № 13: Исследование машинного кода

Необходимое условие: проект № 12.

Предположим, что вам дали исполняемый файл *fact*, но не предоставили исходный файл на языке ассемблера. Вы хотите узнать, что делает программа, но у вас нет исходного кода. Как вы видели в проекте № 12, для тестирования исполняемого файла *fact* можно использовать отладчик *gdb*. В этом проекте я покажу вам другой набор инструментов для исследования машинного кода.

Откройте терминал<sup>1</sup> на вашем Raspberry Pi. По умолчанию терминал должен открыться в домашней папке, обозначенной символом `~`. В этой папке у вас должны уже быть три файла, связанных с факториалом, из последнего проекта. Проверьте это с помощью следующей команды:

---

```
$ ls fact*
```

---

Вы должны увидеть:

*fact* – исполняемый файл;

*fact.o* – объектный файл, созданный во время сборки;

*fact.s* – исходный код на языке ассемблера.

В нашем вымышленном сценарии у вас есть только исполняемый файл *fact*, и вы хотите узнать, что можно понять о программе из содержимого этого файла. Сначала посмотрите на байты, содержащиеся в файле, в шестнадцатеричном виде, используя инструмент *hexdump*:

---

```
$ hexdump -C fact
```

---

Начало вывода по команде *hexdump* должно выглядеть примерно, как показано на рис. 8-4 (за исключением примечаний), что отвечает содержимому исполняемого файла *fact*.

То, что вы видите, – это просто последовательный список значений байт из файла *fact*, отображаемых в виде шестнадцатеричного значения из двух символов каждый. Если количество выводимых данных слишком велико, чтобы поместиться в окне терминала, прокрутите его вверх, чтобы увидеть начальные байты. Восьмисимвольные шестнадцатеричные числа в левой колонке представляют собой смещение относительно начала файла первого байта в соответствующей строке. В каждой строке 16 байт, поэтому номер смещения в каждой строке (слева) увеличивается на `0x10`.

---

<sup>1</sup> Терминал – устройство ввода/вывода (обычно монитор с клавиатурой). ТТУ (от *teletype*) – подсистема связи с терминалами, точнее, с их программными оболочками, эмуляторами терминала – программами, устанавливающими единый интерфейс для связи с терминалами различного типа, например с консолью (см. сноску 1 на стр.221). – *Прим ред.*

В правой части вывода находятся те же байты, интерпретированные как символы ASCII. Байты, не соответствующие какому-либо коду печатаемого символа ASCII, обозначаются точкой.

```

00000000  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 |.ELF.....|
00000010  02 00 28 00 01 00 00 00 74 00 01 00 34 00 00 00 |..(.....t...4...|
00000020  94 02 00 00 00 02 00 05 34 00 20 00 02 00 28 00 |.....4. ....(|
00000030  07 00 06 00 00 00 00 00 00 00 00 00 00 01 00 00 |.....|
00000040  00 00 01 00 00 00 00 00 00 00 00 00 05 00 00 00 |.....|
00000050  00 00 01 00 01 00 00 00 00 00 00 00 00 02 00 00 |.....|
00000060  ac 00 02 00 08 00 00 00 08 00 00 00 00 00 00 00 |.....|
00000070  00 00 01 00 28 10 9f e5 00 00 91 e5 01 30 50 e2 |....(.....0P.|
00000080  02 00 00 da 93 00 00 e0 01 30 53 e2 fc ff ff 1a |.....0S.....|
00000090  10 10 9f e5 00 00 81 e5 00 00 a0 e3 01 70 a0 e3 |.....p...|
000000a0  00 00 00 ef ac 00 02 00 b0 00 02 00 05 00 00 00 |.....|
000000b0  00 00 00 00 41 13 00 00 00 61 65 61 62 69 00 01 |....A....aeabi..|
  
```

Секция данных начинается с 00ac

Рис. 8-4. Hexdump исполняемого файла Linux

По смещению 00000000 в самом начале файла вы должны увидеть 7F, за которым следует 45 4c 46, что в ASCII соответствует ELF. Это индикатор того, что это файл в формате, пригодном для исполнения и компоновки (*Executable and Linkable Format, ELF*). Файлы ELF – это стандартный формат на Linux для исполняемых программ. Эти 4 байта отмечают начало заголовка файла ELF, содержащего набор свойств, которые описывают содержимое файла. За заголовком ELF следует заголовок программы, в котором содержатся сведения, необходимые операционной системе для выполнения программы.

Теперь пропустите заголовки и найдите текстовый раздел, содержащий машинные инструкции программы. В моей системе смещение 00000074 является началом текстового раздела, и он начинается с байтов 28 10 9f e5. Если вы переставите эти байты в обратном порядке, то получите e59f1028, что является инструкцией машинного кода для `ldg r1, [pc, #40]`. Каждый набор из 4 байтов в этом разделе представляет машинную инструкцию. Такой взгляд на программу является хорошим напоминанием о том, что код программы *fact* просто представлен в виде последовательности байтов. Обратитесь к рис.8-1, чтобы вспомнить, как машинный код представляется в двоичном виде.

Далее, по смещению 000000ac в моей системе вы должны увидеть секцию данных файла, содержащую два начальных 4-байтовых значения, определенных программой. Здесь не видно меток `n` и `result`, но должны быть видны 05 00 00 00 и 00 00 00 00. Смещение этих байтов в вашей системе может отличаться от моего.

В качестве примечания: последовательность, в которой компьютеры хранят байты данных для больших числовых значений, называется *порядок байтов (endianness)*. Когда компьютер хранит младший байт первым (по



наименьшему адресу), это называется порядком от младшего к старшему (*little-endian*). Хранение старшего байта первым называется порядком от старшего к младшему (*big-endian*). В выводе `hexdump` вы видите порядок от младшего к старшему, поскольку 32-битная машинная инструкция `e59f1028` хранилась в виде байтов в таком порядке: `28 10 9f e5`. Младший байт был сохранен первым. То же самое можно сказать о значениях `n` и `result`. Значение `n` хранится как `05 00 00 00`, что означает `00000005`, если рассматривать его как 32-битное целое число.

Если вы хотите просмотреть часть этих шестнадцатеричных данных, но сгруппированных по разделам, можете использовать инструмент `objdump`:

---

```
$ objdump -s fact
```

---

Это выгрузит некоторые из тех же байтов, что и раньше, но сгруппированные в секции, как показано ниже:

---

```
Contents of section .text:
10074 28109fe5 000091e5 013050e2 020000da  (.....0P.....
10084 930000e0 013053e2 fcffff1a 10109fe5  ....0S.....
10094 000081e5 0000a0e3 0170a0e3 000000ef  ....p.....
100a4 ac000200 b0000200  ....
Contents of section .data:
200ac 05000000 00000000  ....
Contents of section .ARM.attributes:
0000 41130000 00616561 62690001 09000000  A....aeabi.....
0010 06010801  ....
```

---

Обратите внимание, как изменились номера вдоль левой стороны. Вместо того чтобы начинаться с `0074`, раздел `.text` (т. е. код) начинается с `10074`. Вместо начала с `00ac` секция `.data`, содержащая значения `n` и `result`, начинается с `200ac`. Инструмент `hexdump` просто показывает смещение байта в файле, в то время как вывод `objdump` относится к адресу, по которому байты загружаются в память при выполнении программы. Другим способом просмотра адресов различных разделов исполняемого файла ELF является команда `readelf -e fact`. Этот инструмент отображает заголовки в файле.

Теперь вы можете попробовать другую функцию `objdump` – дизассемблирование кода, чтобы увидеть инструкции языка ассемблера рядом со значениями байтов машинного кода.

```

$ objdump -d fac

fac:      file format elf32-littlearm

Disassembly of section .text:

00010074 <_start>:
   10074:      e59f1028      ldr     r1, [pc, #40] ; 100a4 <end+0x14>❶
   10078:      e5910000      ldr     r0, [r1]
   1007c:      e2503001      subs   r3, r0, #1
   10080:      da000002      ble    10090 <end>

00010084 <loop>:
   10084:      e0000093      mul    r0, r3, r0
   10088:      e2533001      subs   r3, r3, #1
   1008c:      1affffff      bne    10084 <loop>

00010090 <end>:
   10090:      e59f1010      ldr     r1, [pc, #16] ; 100a8 <end+0x18>
   10094:      e5810000      str     r0, [r1]
   10098:      e3a00000      mov     r0, #0
   1009c:      e3a07001      mov     r7, #1
   100a0:      ef000000      svc     0x00000000
   100a4:      000200ac      .word   0x000200ac
   100a8:      000200b0      .word   0x000200b0

```

Вы должны увидеть что-то подобное тому, что показано здесь. Обратите внимание, что инструкция по адресу 10074 ❶ – это та же последовательность байтов, которая была выделена на рис. 8-4, первые 4 байта машинного кода. Этот вывод очень похож на вывод `gdb` в предыдущем проекте. Подумайте, что это значит: используя такие инструменты, как `gdb` или `objdump`, вы можете легко просмотреть машинный код и соответствующий ему код на языке ассемблера для любого исполняемого файла!

Используя приемы, описанные на предыдущих страницах, вы можете получить представление о содержимом исполняемого файла ELF. Это относится к любому стандартному ELF-файлу в системе Linux, а не только к написанному вами коду. Попробуйте рассмотреть машинный код любого ELF-файла на вашем компьютере. Например, предположим, что вы хотите посмотреть машинный код `ls` – инструмента, который вы использовали ранее для просмотра содержимого директории. Сначала вам нужно найти расположение ELF-файла `ls` в файловой системе, как показано ниже:

```

$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz

```

Результат выполнения команды `whereis` (что можно перевести, как «где-находится») говорит нам, что двоичный исполняемый файл для `ls` находится по адресу `/bin/ls` (вы можете игнорировать любые дополнительные

выведенные данные). Теперь можете запустить `objdump` (или любой другой инструмент, о котором уже говорилось), чтобы посмотреть машинный код для `ls`:

---

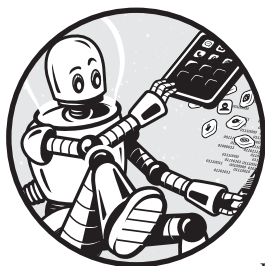
```
$ objdump -d /bin/ls > ls.txt
```

---

Вывод этой команды довольно громоздкий, поэтому он перенаправляется в файл с именем *ls.txt*. Вы не увидите разобранный код в окне терминала, вместо этого он записывается в файл *ls.txt*, который можно просмотреть с помощью текстового редактора по вашему выбору. Конечно, поскольку Linux является открытым исходным кодом, вы можете просто посмотреть исходный код инструмента `ls` в интернете. Однако не все коды открыты, и этот проект должен дать вам представление о том, как можно просмотреть разобранный код любой исполняемой программы Linux.

# 9

## ПРОГРАММИРОВАНИЕ ВЫСОКОГО УРОВНЯ



В прошлой главе мы рассмотрели основы программного обеспечения: машинный код, который выполняется на процессорах, и язык ассемблера – читабельное представление машинного кода. Хотя в конечном итоге все программное обеспечение должно иметь вид машинного кода, большинство разработчиков программного обеспечения работает на более высоком, абстрактном уровне. В этой главе вы узнаете о высокоуровневом программировании. Мы сделаем обзор программирования высокого уровня, обсудим общие элементы, встречающиеся в различных языках программирования, и рассмотрим примеры программ.

### Обзор программирования высокого уровня

Хотя можно писать программы на языке ассемблера (или даже в машинном коде!), это занимает много времени и приводит к ошибкам, а также ведет к созданию программ, которые трудно поддерживать. Кроме того, язык ассемблера специфичен для архитектуры процессора, поэтому, если разработчик ассемблера хочет запустить свою программу на другом типе процессора, код необходимо переписать. Для

устранения этих недостатков были разработаны *высокоуровневые языки программирования*, которые позволяют писать программы на языке, не зависящем от конкретного процессора и синтаксически более близком к человеческому языку. Для многих из этих языков требуется *компилятор* – программа, которая преобразует высокоуровневые программные выражения в машинный код для конкретного процессора. Используя высокоуровневый язык, разработчик программного обеспечения может написать программу один раз, а затем скомпилировать ее для нескольких типов процессоров, иногда практически без изменений исходного кода.

Результатом работы компилятора является объектный файл, содержащий машинный код для конкретного процессора. Как мы уже говорили в проекте № 12, объектные файлы не имеют правильного формата для выполнения компьютером. Другая программа, называемая *компоновщик*, используется для преобразования одного или нескольких объектных файлов в исполняемый файл, который затем может быть запущен операционной системой. При необходимости компоновщик может подключить другие библиотеки скомпилированного кода. Процесс компиляции и компоновки показан на рис. 9-1.

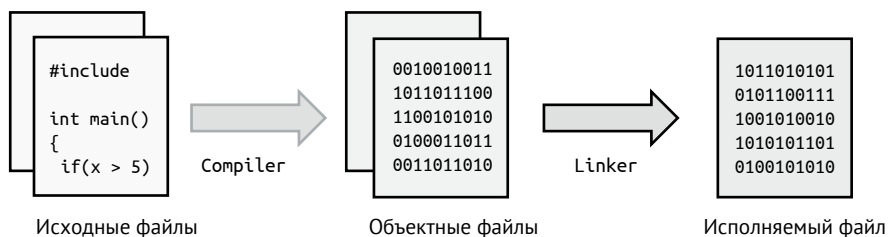


Рис. 9-1. Создание исполняемого файла из исходного кода

Процесс компиляции и компоновки вместе называется *сборкой (building)* программного обеспечения. Однако в обиходе разработчики программ иногда говорят о *компиляции* кода, подразумевая под этим весь процесс компиляции, компоновки и любых других шагов, необходимых для приведения кода в окончательную форму. Компиляторы часто вызывают этап компоновки автоматически, что делает его менее заметным для разработчика программного обеспечения<sup>1</sup>.

<sup>1</sup> Следует отметить, что англоязычные термины *assembler* и *compiler* формально переводятся одинаково, как «сборщик». Да и делают они одно и то же, переводя текстовые инструкции в машинные коды (при этом ассемблеры можно рассматривать просто как разновидность компиляторов), только исходный код составляется для них по разным правилам. Компилятор чаще всего устроен намного сложнее ассемблера – исходный код высокоуровневых языков программирования из-за их универсальности привести к набору машинных инструкций существенно труднее, чем ассемблерный код. Еще сложнее осуществить процесс *декомпиляции*, который из-за сложности задачи часто сводят к более простому *дисассемблированию*. – Прим. ред.

# Введение в С и Python

Лучший способ узнать о высокоуровневом программировании – рассмотреть языки программирования и написать несколько программ на этих языках. Для этой главы я выбрал два высокоуровневых языка: С и Python. Оба они мощные и удобные, а также могут служить иллюстрацией, что языки программирования обычно предоставляют схожие функциональные возможности, но разными способами. Давайте начнем с краткого введения в каждый из них.

Язык программирования С берет свое начало в начале 1970-х годов, когда он был использован для написания версии операционной системы Unix. Несмотря на то что это высокоуровневый язык, С не так уж далек от машинного кода, что делает его отличным выбором для разработки операционной системы или другого программного обеспечения (ПО), которое напрямую взаимодействует с аппаратным обеспечением<sup>1</sup>. Обновленная версия языка С, известная как С++, появилась в 1980-х годах. С и С++ – это мощные языки, которые могут быть использованы практически для любой задачи. Однако эти языки сложны и не очень хорошо обеспечивают защиту от ошибок программистов. Они остаются популярным выбором для программ, которым необходимо взаимодействовать с аппаратным обеспечением, и программ, требующих высокой производительности, таких как игры. Язык С также полезен для образовательных целей, так как он обеспечивает простой переход между низкоуровневыми и высокоуровневыми концепциями, поэтому я и выбрал его для этой главы.

По сравнению с С язык программирования Python более удален от оборудования. Первоначально выпущенный в 1990-х годах, Python с годами приобрел большую популярность. Он известен тем, что легко читается и прост для начинающих, но при этом предоставляет все необходимое для поддержки сложных программных проектов. Python придерживается философии «батарейки в комплекте», т. е. стандартный дистрибутив Python включает библиотеку полезных возможностей, которые разработчики могут легко использовать в своих проектах. Прямолинейность Python делает его хорошим выбором для обучения концепциям программирования.

Теперь давайте рассмотрим элементы, встречающиеся в большинстве высокоуровневых языков программирования. Цель не в том, чтобы научить вас быть программистом на каком-то конкретном языке, а в том, чтобы ознакомить вас с идеями, часто встречающимися в языках программирования. Помните, что возможности высокоуровневых языков программирования являются абстракциями инструкций процессора. Как вы знаете, процессоры предоставляют инструкции для доступа к памяти, математическим и логическим операциям, а также для управления ходом программ. Давайте рассмотрим, как высокоуровневые языки раскрывают эти базовые возможности.

---

<sup>1</sup> Обычно такое ПО называют «системным программным обеспечением». Распространенный пример системного ПО, с которым приходилось сталкиваться практически любому пользователю, – драйверы оборудования. – *Прим. ред.*

## Комментарии

Давайте начнем с такой опции языков программирования, которая на самом деле ничего не предписывает процессору! Почти все языки программирования предоставляют возможность включать в код комментарии. *Комментарий* – это текст в исходном коде, который дает некоторое представление о выполняемой операции. Комментарии предназначены для чтения другими разработчиками и обычно игнорируются компилятором, они не оказывают никакого влияния на скомпилированную программу. В языке программирования C комментарии задаются следующим образом:

---

```
/*  
    Это комментарий в стиле C.  
    Он может занимать несколько строк.  
*/  
  
// Это однострочный комментарий, первоначально введенный в C++.
```

---

В Python для комментариев используется символ решетки, как, например, в этом примере:

---

```
# Это комментарий в Python.
```

---

Python не предоставляет специальной поддержки для многострочных комментариев, программист может просто использовать несколько однострочных комментариев, один за другим.

## Переменные

Доступ к памяти – фундаментальная возможность процессоров, поэтому она должна быть свойственна и высокоуровневым языкам. Основным способом, которым языки программирования предоставляют доступ к памяти, – переменные. *Переменная* (*variable*) есть именованное место хранения в памяти. Переменные позволяют программистам дать *имя* (*name*) адресу памяти (или диапазону адресов памяти) и затем получить доступ к данным по этому адресу. В большинстве языков программирования переменные имеют *тип* (*type*), указывающий, какого рода данные они хранят. Например, переменная может быть целочисленного типа или типа текстовой строки. Переменные также имеют *значение* (*value*), которое представляет собой данные, хранящиеся в памяти. Хотя это часто скрыто от программиста, переменные также имеют *адрес* (*address*), т. е. место в памяти, где хранится значение переменной. Наконец, переменные имеют *область видимости* (*scope*), т. е. доступ к ним возможен только из определенных частей программы, тех частей, где они находятся «в области видимости».

## Переменные в С

Давайте рассмотрим пример переменной в языке программирования С.

---

```
// Объявление переменной и присвоение ей значения в языке С.  
int points = 27;
```

---

Этот код объявляет переменную с именем `points` и типом `int`, что в языке С означает, что переменная содержит целое число<sup>1</sup>. Затем переменной *присваивается* значение 27. Когда этот код выполняется, значение 27 в десятичном формате сохраняется по адресу памяти, но разработчику не нужно беспокоиться о конкретном адресе, по которому хранится переменная. Большинство компиляторов языка С сегодня рассматривает `int` как 32-битное число, поэтому во время *выполнения программы* для этой переменной выделяется 4 байта (4 байта × 8 бит на байт = 32 бита), а адрес переменной в памяти относится к первому байту.

Теперь объявим вторую переменную и присвоим ей значение, и затем мы сможем посмотреть, как эти две переменные располагаются в памяти.

---

```
// Две переменные в С  
int points = 27;  
int year = 2020;
```

---

Теперь у нас есть две переменные, `points` и `year`, объявленные друг за другом. Обе они являются целыми числами, поэтому для хранения каждой из них требуется 4 байта. Переменные могут храниться в памяти, как показано в табл. 9-1.

**Таблица 9-1.** Переменные, хранящиеся в памяти

Адрес	Имя переменной	Значение переменной
0x7efff1cc	?	?
0x7efff1d0	year	2020
0x7efff1d4	points	27
0x7efff1d8	?	?

Адреса памяти, используемые в табл. 9-1, являются лишь примерами. Фактические адреса зависят от аппаратного обеспечения, операционной системы, компилятора и т. д. Обратите внимание, что адреса увеличиваются на четыре, поскольку мы храним 4-байтовые целые числа. Адреса до и после известных переменных имеют вопросительный знак для имени и значения переменной, поскольку, исходя из предыдущего кода, мы не знаем, что там может храниться.

---

<sup>1</sup> От слова *integer*, что и переводится как «целочисленный». – Прим. ред.



Как следует из названия *переменной*, ее значение может меняться. Если бы в нашей предыдущей программе на языке C нужно было бы поменять значение `points` на другое значение, мы могли бы просто сделать это позже в программе:

---

```
// Установка нового значения points в C
points = 31;
```

---

Обратите внимание, что, в отличие от нашего предыдущего фрагмента кода на C, в этом коде перед именем переменной не указывается `int` или любой другой тип. Нам нужно указывать тип только при первоначальном объявлении переменной. В данном случае переменная была объявлена ранее, поэтому здесь мы просто присваиваем ей значение. Однако язык C требует, чтобы тип переменной оставался неизменным, поэтому, как только `points` объявлена как `int`, этой переменной можно присваивать только значения целых чисел. Попытка присвоить переменной другой тип, например текстовую строку, приведет к ошибке при компиляции кода.

## Переменные в Python

Не все языки требуют объявления типа. Например, Python позволяет объявлять и присваивать переменные следующим образом:

---

```
# Python позволяет создавать новые переменные без указания типа
age = 22
```

---

В данном случае Python распознает, что тип данных является целым числом, но программисту не обязательно это указывать. В отличие от языка C тип переменной может меняться со временем, поэтому в Python допустимо следующее:

---

```
# Присвоение переменной значения другого типа допустимо в Python
age = 22
age = 'twenty-two '
```

---

Давайте рассмотрим подробнее, что на самом деле происходит в этом примере. Переменная в Python не имеет типа, но значение, на которое она ссылается, имеет тип. Это важное различие: тип связан со значением, а не с переменной. Переменная Python может ссылаться на значение любого типа. Поэтому, когда переменной присваивается новое значение, тип переменной не меняется, а скорее переменная привязывается к значению другого типа. Это отличает

Python от языка C, где переменная сама имеет тип и может хранить только значения этого типа. Это различие объясняет, почему переменной в Python можно присваивать значения разных типов, а переменной в C – нет.

#### ПРИМЕЧАНИЕ

*Посмотрите проект № 15 на стр. 232, где вы сможете изменить тип значения, на которое ссылается переменная в Python.*

## Стек и куча

Когда программист использует высокоуровневый язык для доступа к памяти, детали того, как эта память управляется внутри системы, остаются в той или иной мере скрытыми, в зависимости от используемого языка программирования. Такой язык программирования, как Python, делает детали распределения памяти практически невидимыми для программиста, в то время как такой язык, как C, раскрывает некоторые базовые механизмы управления памятью. Независимо от того, открыты ли такие детали программисту или нет, программы обычно используют два типа памяти: стек и кучу.

### Стек

*Стек* – область памяти, которая работает по принципу «последним пришел – первым вышел» (*last in – first out, LIFO*). То есть последний элемент, помещенный в стек, является первым элементом, который выходит со стека. Вы можете представить себе стек памяти как стопку тарелок. Когда вы добавляете в стопку тарелку, то кладете новую тарелку на самый верх. Когда приходит время взять тарелку из стопки, вы сначала берете верхнюю тарелку. Это не означает, что к элементам в стеке можно *обращаться* (читать или изменять) только в порядке LIFO. На самом деле любой элемент, находящийся в данный момент в стеке, может быть прочитан или изменен в любое время. Однако, когда приходит время удалить ненужные элементы из стека, элементы отбрасываются сверху вниз, т. е. последний элемент, помещенный в стек, отбрасывается первым.

Адрес памяти значения на вершине стека хранится в регистре процессора, называемом *указатель стека*. Когда значение добавляется в голову стека, значение указателя стека изменяется, чтобы увеличить размер стека и освободить место для нового значения. Когда значение удаляется из головы стека, указатель стека корректируется, чтобы уменьшить размер стека.

Компилятор генерирует код, который использует стек для отслеживания состояния выполнения программы и как место для хранения локальных переменных. Механика этого процесса прозрачна для программиста на высокоуровневом языке. На рис. 9-2 показано, как про-

грамма на языке C использует стек для хранения двух локальных переменных, которые мы рассматривали ранее в табл. 9-1<sup>1</sup>.

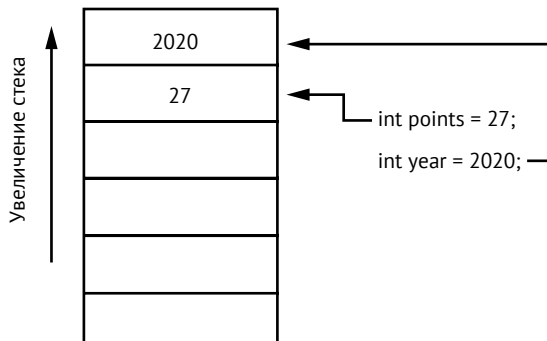


Рис. 9-2. Стекковая память используется для хранения значений переменных в программе, написанной на языке C

На рис. 9-2 переменная `points` объявляется первой, и ей присваивается значение 27, которое хранится в стеке. Затем объявляется переменная `year`, которой присваивается значение 2020. Это второе значение помещается «над» предыдущим значением в стеке. Дополнительные значения будут добавляться в голову стека до тех пор, пока в них не отпадет необходимость, после чего они будут удалены из стека. Помните, что каждая ячейка на диаграмме – это просто место в памяти с присвоенным адресом, хотя адреса на рисунке и не показаны. Это может вас удивить, но во многих архитектурах адреса памяти, назначенные стеку, *уменьшаются* по мере роста стека. В данном примере это означает, что переменная `year` имеет меньший адрес памяти, чем переменная `points`.

Стекковая память является быстрой и хорошо подходит для размещения небольших объемов данных в памяти. Для каждого потока выполнения программы выделяется отдельный стек. Более подробно мы рассмотрим потоки в главе 10, а пока вы можете думать о потоках как о параллельных задачах в программе. Стек является ограниченным ресурсом, это значит, что существует предел того, сколько памяти выделяется под стек. Помещение слишком большого количества значений в стек приводит к сбою, известному как *переполнение стека*.

<sup>1</sup> Понятие *глобальных* и *локальных* переменных связано с упомянутой автором *областью видимости*. Глобальные переменные (обычно объявляются в начале текста программы) «видны» в любом месте программы, т. е. ими можно оперировать внутри любого блока программы или вне его. Локальные переменные «видны» только внутри того блока (например, внутри функции), в котором были объявлены. Язык C и некоторые другие допускают использовать для локальных переменных те же имена, что и для глобальных, при этом до конца области видимости локальной переменной все операции с этим именем действуют только на нее.

## Куча

Стек предназначен для хранения небольших значений, которые должны быть доступны только временно. Для распределения в памяти данных большого объема или хранящихся в течение долгого времени лучше использовать кучу. Куча – это весь объем памяти, доступный программе. В отличие от стека память кучи не работает по модели LIFO. Не существует стандартной модели того, как выделяется память кучи. В то время как память стека является специфичной для потока, доступ к выделенной памяти из кучи может получить любой поток программы.

Программы выделяют память из кучи, и эта память используется до тех пор, пока она не будет освобождена программой или программа не завершит свою работу. Освободить выделенную память означает просто вернуть ее в объем доступной памяти. Некоторые языки программирования автоматически освобождают память кучи, когда распределение больше не востребовано. Один из распространенных подходов для этого называется *сборка мусора*. Другие языки программирования, такие как С, требуют от программиста написать код для освобождения памяти кучи. *Утечка памяти* происходит, когда неиспользуемая память не освобождается.

В языке программирования С для отслеживания распределения памяти используется специальный тип переменной, называемый *указателем*. Указатель – это просто переменная, в которой хранится адрес памяти. Значение указателя (адрес памяти) может храниться в локальной переменной на стеке, и это значение может ссылаться на место в куче, как показано на рис. 9-3.

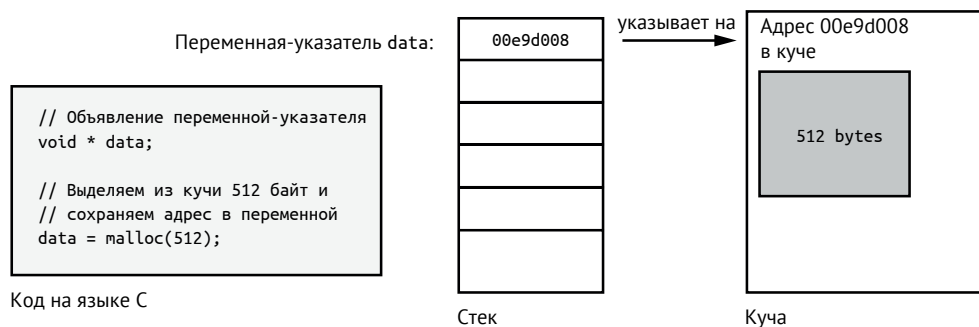


Рис. 9-3. Переменная-указатель с именем *data*, находящаяся в стеке и указывающая на адрес в куче

На рис. 9-3 представлен фрагмент кода, в котором объявлена переменная с именем *data*. Эта переменная имеет тип *void \**, что означает, что это указатель (обозначенный \*), указывающий на адрес памяти, который может содержать любой тип данных (*void* означает, что тип не определен). Поскольку *data* является локальной переменной, она располагается в стеке. Следующая строка кода вызывает *malloc*, функцию в С, которая выделяет память из кучи. Программа запраши-

вает 512 байт памяти, и функция `malloc` возвращает адрес первого байта только что выделенной памяти. Этот адрес хранится в переменной `data` на стеке. В итоге мы получаем локальную переменную с адресом в стеке, которая хранит адрес расположения в куче.

#### ПРИМЕЧАНИЕ

*Посмотрите проект № 16 на стр. 233, где вы сможете увидеть, как располагаются переменные в работающей программе.*

## Математика

Поскольку процессоры предоставляют инструкции для выполнения математических операций, высокоуровневые языки тоже это делают. В отличие от программирования на языке ассемблера, где для выполнения математических операций требуются специальные именованные инструкции (например, инструкция `subs` для вычитания в процессорах ARM), высокоуровневые языки обычно используют символы, представляющие общие математические операции, что упрощает выполнение математических операций в коде.

Большое количество языков программирования, включая C и Python, используют одни и те же операторы для сложения, вычитания, умножения и деления, как показано в табл. 9-2.

**Таблица 9-2.** Общие математические операторы

Операция	Оператор
Сложение	+
Вычитание	-
Умножение	*
Деление	/

Другой распространенной практикой в различных языках программирования является использование знака равенства (=) для обозначения присваивания, а не равенства. То есть утверждение типа `x = 5` означает установку значения `x` равным 5. Присвоение результата математической операции представляется естественным образом, как показано ниже:

---

```
// Сложение в языке C осуществляется просто
cost = price + tax;
```

---

---

```
# В Python сложение тоже простое
cost = price + tax
```

---

До сих пор мы имели дело с целочисленной математикой, которая часто встречается в компьютерных вычислениях. Однако компьютеры и высокоуровневые языки также поддерживают *числа с плавающей точкой*<sup>1</sup>. В отличие от целых чисел значения с плавающей точкой могут представлять дроби. Некоторые языки программирования скрывают детали этого процесса, но процессоры внутри используют разные инструкции для математики с плавающей точкой и для целочисленной математики. В языке C переменные с плавающей точкой объявляются с использованием типа вроде `float` или `double`, как показано далее:

---

```
// Объявление переменной с плавающей точкой в C
double price = 1.99;
```

---

С другой стороны, Python подразумевает автоматическое присвоение типа для переменных, поэтому и целые числа, и значения с плавающей точкой объявляются одинаково:

---

```
# Объявление целых переменных и переменных с плавающей точкой в Python
year = 2020 # year - целое число
price = 1.99 # price - переменная с плавающей точкой
```

---

Различия между целыми числами и числами с плавающей точкой иногда приводят к неожиданным результатам. Например, допустим, у вас есть следующий код на языке C:

---

```
// Деление целых чисел в C
int x = 5;
int y = 2;
int z = x / y;
```

---

Каким, как вы думаете, должно быть значение `z`? Оказывается, поскольку все числа являются целыми, `z` в итоге будет равно 2. Не 2,5, а 2. Будучи целым числом, `z` не может иметь дробных значений.

А что, если мы немного изменим код, например так:

---

<sup>1</sup> В русскоязычной литературе (а также в большинстве европейских языков) для отделения дробной части используется не точка, а запятая, потому реальные числа именуются «с плавающей запятой». Так как в большинстве языков программирования используются американские стандарты, в дальнейшем в этой книге в программных текстах и иллюстрациях используется терминология и нотация автора с использованием «точки». Однако в некоторых средах программирования стандарты считываются из настроек операционной системы, поэтому во избежание недоразумений, например при программировании в русской версии Windows, всегда следует сначала проверять, каким именно знаком отделять дробную часть – точкой или запятой. – *Прим. ред.*

---

```
// Деление целых чисел в C, результат хранится в типе float
int x = 5;
int y = 2;
float z = x / y;
```

---

Обратите внимание, что `z` теперь имеет тип `float`. Каким теперь, как вы думаете, будет значение `z`? Что интересно, `z` теперь равно 2,0, но это все еще не 2,5! Это произошло потому, что операция деления производилась с двумя целыми числами, поэтому результат тоже был целым числом. Результат деления был равен 2, и когда он был присвоен переменной `z` с плавающей точкой, она получилась равной 2,0. Язык C очень буквален, он состоит из инструкций, которые точно повторяют то, что указал программист. Это хорошо для программистов, которым нужен тонкий контроль над процессом, но не всегда хорошо для программистов, которые ожидают более интуитивного поведения от языка программирования.

Python пытается быть более услужливым, автоматически назначая тип, который позволяет получать дробные результаты в подобной ситуации. Если мы напишем эквивалентную версию этого кода на Python, результат, хранящийся в `z`, будет равен 2,5.

---

```
# Деление целых чисел в Python
# z будет равен 2,5, а его предполагаемый тип будет float
x=5
y=2
z=x/y
```

---

Некоторые языки предоставляют математические операторы, которые являются сокращенными способами описания операции. Например, язык C предоставляет операторы инкремента (прибавление 1) и декремента (вычитание 1), как показано здесь:

---

```
// В языке C мы можем добавить единицу к переменной длинным путем,
x = x + 1;
// или мы можем использовать это сокращение для инкрементирования x.
x++;
// С другой стороны, это выражение декрементирует x.
x--;1
```

---

<sup>1</sup> Внимательный читатель может озадачиться тем, что в Python инкремент записывается всегда традиционно, как `n = n + 1`, и сокращенная запись (`n++`), как в языке C, в нем отсутствует (то же относится и к операции декрементирования). Два равнозначных способа в языке C возникли исторически – формально на уровне инструкций процессора они выполнялись по-разному. Конструкция с прибавлением единицы (`n = n + 1`) выполнялась «в лоб», как сложение с константой, в то время как во всех процессорах имеется отдельная инструкция инкрементирования, и именно она выполняется при указании операции типа `n++`. Это экономит время выпол-

В программных текстах на С далее читатель может встретить так называемую префиксную запись операций инкремента/декремента: `++x` и `--x` вместо `x++` и `x--` (см., например, упражнение 9-2). В заголовке циклов они приводят к идентичным результатам, но могут различаться по действию в конструкциях типа `y = x++` (эквивалентно последовательности операций `y = x, x = x + 1`) и `y = ++x` (эквивалентно последовательности `x = x + 1, y = x`). Подобные неоднозначности (эти примеры далеко не единственные) – типичный пример того, почему автор назвал языки С и С++ сложными для изучения.

#### ПРИМЕЧАНИЕ

*Забавный факт: название языка программирования С++ призвано передать идею о том, что он является улучшением или инкрементом языка программирования С.*

Python также предоставляет некоторые операторы сокращения для математики. Операторы `+=` и `-=` позволяют программистам прибавлять к переменной или вычитать из нее. Например:

---

```
# В Python мы можем добавить 3 к переменной следующим образом...
cats = cats + 3

# Или мы можем сделать то же самое с помощью этого сокращения...
cats += 3
```

---

Операторы `+=` и `-=` работают также и в языке С.

## Логика

Как мы уже говорили ранее, процессоры очень хорошо справляются с логическими операциями, поскольку логика является основой цифровых схем. Ожидаемо, что и языки программирования также предоставляют возможность работы с логикой. Большинство высокоуровневых языков предоставляет два вида операторов, которые работают с логикой: побитовых операторов, которые работают с битами целых чисел, и булевых (*Boolean*) операторов, которые работают с булевыми значениями (истина/ложь). Терминология здесь может вас запутать, поскольку разные языки программирования используют разные термины. В Python используются термины «побитовый» и «булев», в то время как в С используются термины «побитовый» и «логический», а в других языках используются еще другие термины. Давайте здесь будем придерживаться терминов «побитовый» и «булев».

---

нения программы. Современные оптимизирующие компиляторы С не делают разницы между этими способами, во всех случаях замещая инкрементирование одной инструкцией процессора, а более современный язык Python и вовсе обходится единым способом, более понятным при чтении кода. – *Прим. ред.*



# Побитовые операторы

*Побитовые операторы* воздействуют на отдельные биты целочисленных значений и в результате дают целочисленное значение. Побитовый оператор похож на математический оператор, но вместо сложения или вычитания он выполняет И, ИЛИ или другую логическую операцию над битами целых чисел. Эти операторы работают в соответствии с таблицами истинности, рассмотренными в главе 2, выполняя операцию параллельно над всеми битами целого числа.

Многие языки программирования, включая C и Python, используют для побитовых операций набор операторов, приведенный в табл. 9-3.

**Таблица 9-3.** Побитовые операции, распространенные в языках программирования

Побитовая операция	Побитовый оператор
И (AND)	&
ИЛИ (OR)	
Исключающее ИЛИ (XOR)	^
НЕ (дополнение)	~

Давайте рассмотрим пример побитовой логики в Python.

```
# Python осуществляет побитовую логику.  
x=5  
y=3  
a= x & y  
b = x | y
```

В результате выполнения приведенного выше кода **a** будет равно 1, **b** будет равно 7. Давайте рассмотрим эти операции в двоичном виде (рис. 9-4), чтобы было понятно, почему это так.

x = 5 = 0101		x = 5 = 0101
y = 3 = 0011		y = 3 = 0011
<hr/>	AND	<hr/>
0001		0111

**Рис. 9-4.** Побитовые операции И (AND), ИЛИ (OR) для значений 5 и 3

Сначала посмотрите на операцию И (AND) на рис. 9-4 и вспомните из главы 2, что И означает, что результат равен 1, когда оба входа равны 1. Здесь мы рассматриваем биты по одному столбцу за раз, и, как вы можете видеть, только самый правый бит равен 1 на обоих входах. Поэтому результат И равен 0001 в двоичном исчислении или 1 в десятичном. Поэтому в предыдущем коде **a** присваивается значение 1.

С другой стороны, ИЛИ (OR) означает, что результат равен 1, если один из входов (или оба входа) равен 1. В данном примере крайние три бита равны 1 на одном или другом входе, поэтому результат равен 0111 в двоичном виде, или 7 в десятичном. Поэтому в предыдущем коде `b` присваивается значение 7.

### УПРАЖНЕНИЕ 9-1: Побитовые операторы

Рассмотрите следующие выражения на языке Python. Какими будут значения `a`, `b` и `c` после выполнения этого кода?

```
x = 11
y = 5
a = x & y
b = x | y
c = x ^ y
```

Ответы можно найти в приложении А.

## Булевы операторы

Другим видом логических операторов в высокоуровневых языках программирования являются *булевы операторы*. Эти операторы работают с булевыми значениями и приводят к булеву значению.

Давайте немного поговорим о булевых значениях. *Булево значение* – это либо истина, либо ложь. Разные языки программирования представляют истину (`true`) или ложь (`false`) по-разному.

*Булева переменная* – это именованный адрес памяти, который хранит булево значение «истина» (`true`) или «ложь» (`false`). Например, в Python мы можем иметь переменную, которая отслеживает, продается ли товар: `item_on_sale = True`.

Выражение может иметь значение «истина» или «ложь» без сохранения результата в переменной. Например, выражение `item_cost > 5` во время выполнения программы оценивается как истинное или ложное в зависимости от значения переменной `item_cost`.

Булевы операторы позволяют нам выполнять логические операции типа И (AND), ИЛИ (OR) или НЕ (NOT) над булевыми значениями. Например, мы можем проверить, истинны ли оба условия, используя оператор булево И (Boolean AND) в Python: `item_on_sale and item_cost > 5`. Выражения слева и справа от AND оцениваются как булевы значения, и в свою очередь все выражение оценивается как булево значение. В данном случае в языках C и Python используются разные операторы, как показано в табл. 9-4.

**Таблица 9-4.** Булевы операторы в языках программирования C и Python

Булева операция	Оператор в C	Оператор в Python
И (AND)	&&	and
ИЛИ (OR)		or
НЕ (NO)	!	not

Раз уж мы заговорили об операторах, возвращающих булевы значения, *оператор сравнения* сравнивает два значения и в результате сравнения выдает значение «истина» или «ложь». Например, *оператор «больше»* позволяет сравнить два числа и определить, больше ли одно из них, чем другое. В табл. 9-5 показаны операторы сравнения, используемые как в C, так и в Python.

**Таблица 9-5.** Операторы сравнения в языках программирования C и Python

Операция сравнения	Оператор сравнения
РАВНО	= =
НЕ РАВНО	!=
БОЛЬШЕ	>
МЕНЬШЕ	<
БОЛЬШЕ ИЛИ РАВНО	>=
МЕНЬШЕ ИЛИ РАВНО	<=

Вы уже видели один из них в использовании в нашем предыдущем примере: `item_cost > 5`. Обратите внимание на оператор равенства. И в C, и в Python используется двойной знак равенства для представления сравнения на равенство, а для представления присваивания используется одинарный знак равенства. Это означает, что `x == 5` – это сравнение, которое возвращает значение «истина» или «ложь» (равен ли `x` 5?), тогда как `x = 5` – присваивание, которое устанавливает значение `x` равным 5.

## Порядок выполнения программы

Булевы операторы и операторы сравнения позволяют нам оценить истинность выражения, но это само по себе не очень полезно. Нам нужен способ сделать что-то в ответ! *Порядок выполнения программы*, или *поток управления*, позволяет нам делать именно это, изменять поведение программы в ответ на некоторое условие. Давайте рассмотрим некоторые распространенные конструкции для управления порядком выполнения программ, встречающиеся в разных языках программирования.

## Операторы if

Оператор `if` (если) позволяет программисту сделать что-то, если некоторое условие истинно. В свою очередь оператор `else` (иначе, в противном случае), часто применяющийся в сочетании с оператором `if`, позволяет программе сделать что-то другое, если условие ложно. Вот пример на языке Python:

---

```
# Age check in Python
❶ if age < 18:
    ❷ print('You are a youngster!')
❸ else:
    ❹ print('You are an adult.')
```

---

В этом примере первый оператор `if` ❶ проверяет, ссылается ли переменная `age` (возраст) на значение, которое меньше 18. Если да, то печатается сообщение, указывающее на то, что пользователь молод ❷. Оператор `else` ❸ указывает программе напечатать, что пользователь взрослый, если возраст равен 18 или больше ❹.

Вот та же логика «проверки возраста», на этот раз написанная на языке C:

---

```
// Age check in C
if (age < 18)
❶ {
    printf("You are a youngster!");
❷ }
else
{
    printf("You are an adult.");
}
}
```

---

В примере на языке C обратите внимание на фигурные скобки, используемые после оператора `if` ❶ ❷. Они выделяют блок кода, который должен быть выполнен в ответ на `if`. В языке C блок кода может состоять из нескольких строк кода, хотя скобки можно опустить, если блок состоит из одной строки. В Python скобки не используются для разделения блоков кода, вместо них используется отступ. В Python смежные строки с одинаковым уровнем отступа (например, четыре пробела) считаются частью одного блока.

В Python также есть оператор `elif`, что означает «else if» (в противном случае, если...). Оператор `elif` оценивается только в том случае, если предыдущий оператор `if` или `elif` был ложным.

---

```
# Лучшая проверка возраста в Python
if age < 13:
    print('You are a youngster!')
elif age < 20:
    print('You are a teenager.')
else:
    print('You are older than a teen.')
```

---

Того же самого можно добиться в языке C, используя `else` в сочетании с `if`:

---

```
// Лучшая проверка возраста в C
if (age < 13)
    printf("You are a youngster!");
else if (age < 20)
    printf("You are a teenager!");
else
    printf("You are older than a teen.");
```

---

Обратите внимание, что в C-коде я опустил фигурные скобки, поскольку все мои блоки кода состоят из одной строки.

## Циклы

Иногда программа должна выполнять определенное действие снова и снова. Цикл `while` (*до тех пор пока...*) позволяет повторять код до тех пор, пока не будет выполнено некоторое условие. В следующем примере на языке Python цикл `while` используется для печати чисел от 1 до 20.

---

```
# Счет до 20 в Python.
n = 1
while n <= 20:
    print(n)
    n = n + 1
```

---

Изначально переменная `n` установлена равной 1. Начинается цикл `while`, указывающий, что цикл должен выполняться до тех пор, пока `n` меньше или равно 20. Поскольку `n` равно 1, оно удовлетворяет этому требованию, поэтому тело цикла `while` выполняется, печатая значение `n` и прибавляя к нему 1. Теперь `n` равно 2, и код возвращается в начало цикла `while`. Этот процесс продолжается до тех пор, пока `n` не станет равным 21, после чего оно больше не удовлетворяет требованиям цикла `while`, и цикл завершается.

То же самое, реализованное на языке C:

---

```
// Считаем до 20 на C.
int n = 1;
while (n <= 20)
{
    printf("%d\n", n);
    n++;
}
```

---

В обоих примерах в теле цикла `while` инкрементируется значение `n`. На самом деле есть более аккуратный способ сделать это.

Цикл `for` (для) позволяет выполнять итерации по диапазону чисел или группе значений, чтобы программист мог выполнить некоторую операцию над каждым из них. Здесь приведен пример на языке C, который печатает от 1 до 10.

---

```
// C использует цикл for для итерации по числовому диапазону.  
// Этот код будет печатать от 1 до 10.  
for(❶int x = 1; ❷x <= 10; ❸x++)  
{  
❹ printf("%d\n", x);  
}
```

---

В заголовке цикла `for` объявляется переменная-счетчик `x` и устанавливается ее начальное значение, равное 1 ❶, указывается, что цикл будет продолжаться, пока `x` меньше или равно 10 ❷, и, наконец, объявляется, что после выполнения тела цикла переменная `x` должна быть инкрементирована ❸. Поместив всю эту информацию в заголовок оператора `for` на одной строке, мы можем легче видеть условия, при которых будет выполняться цикл. Тело цикла `for` просто печатает значение `x` ❹.

Python использует другой подход к циклам `for`, позволяя программе многократно выполнять действие для каждого элемента в диапазоне значений. Следующий пример на Python выводит названия животных из списка.

---

```
# Python использует цикл for для итерации по диапазону.  
# Этот код будет выводить каждое имя животного в списке animal_list.  
animal_list = ['cat', 'dog', 'mouse']  
for animal in animal_list:  
    print(animal)
```

---

Сначала объявляется список имен животных и присваивается переменной с именем `animal_list`. В Python список (`list`) – это упорядоченная группа значений. Далее в цикле `for` блок кода выполняется один раз для каждого элемента в списке `animal_list`, и каждый раз, когда код выполняется, текущее значение в списке присваивается переменной `animal`. Поэтому при первом запуске цикла `animal` равно `cat`, и программа печатает `cat`. В следующий раз будет напечатана `dog`, а в последний раз `mouse`.

## Функции

Циклы позволяют выполнять набор инструкций несколько раз подряд. Однако часто бывает, что программа выполняет определенный набор инструкций несколько раз, но не обязательно в цикле. Вместо этого такие инструкции могут быть вызваны из разных частей программы, в разное время и с разными входами и выходами. Когда программист

понимает, что один и тот же код нужен в нескольких местах, он может написать этот код в виде функции.

*Функция* – это набор программных инструкций, которые могут быть вызваны другим кодом. Функции могут принимать входные данные (называемые *параметры*) и возвращать выходные данные (называемые *возвращаемое значение*). В различных высокоуровневых языках для обозначения функции используются разные термины, включая *подпрограмму*, *процедуру* или *метод*. В некоторых случаях эти различные названия передают немного разный смысл, но для наших целей давайте ограничимся термином «функция»<sup>1</sup>.

Преобразование буквенных символов в строчные, печать текста на экране и загрузка файла из интернета – все это примеры того, что можно сделать с помощью многократно используемого кода в виде функции. Программисты стремятся избегать многократного ввода одного и того же кода. Потому что это приводит к необходимости поддерживать несколько копий одного и того же кода и увеличивать общий размер программы. Это нарушает принцип программной инженерии, известный как «не повторяйся» (*don't repeat yourself – DRY*), который поощряет сокращение дублирующего кода.

Функции – это еще один пример применения принципа «черного ящика». Ранее мы рассматривали «черные ящики» в аппаратном обеспечении, а здесь мы видим их снова уже в программах. Функции инкапсулируют в себе внутренние детали блока кода, предоставляя при этом интерфейс для использования этого кода. Разработчику, который хочет использовать функцию, достаточно понимать только ее входы и выходы, полное понимание внутренней работы функции не требуется.

## Определение функций

Функция должна быть определена, прежде чем ее можно будет использовать. После того как функция единожды определена, ее можно использовать посредством вызова. *Определение функции* включает в себя объявление имени функции и входных параметров, программный блок функции (называемый *телом функции*), а в некоторых языках также

<sup>1</sup> *Подпрограмма (subroutine)* – обобщенное название поименованного блока операторов, т. е. фрагмент кода, который может использоваться неоднократно. Сейчас название «подпрограмма» обычно применяется лишь в низкоуровневых языках вроде ассемблера. В Pascal и других классических алгоритмических языках подпрограммы делятся на *процедуры (procedure)* и *функции (function)*. *Процедура* – любая подпрограмма, делающая что-то с данными внутри нее. В отличие от процедур *функция* обязательно возвращает значение заданного типа (что близко обычному математическому смыслу термина «функция»). В C от этого деления решили избавиться – в нем есть только функции, причем то, что называлось процедурами (функции, не возвращающие значение), обозначается добавлением перед названием функции слова *void (пустота)* вместо указания типа возвращаемого значения. Термин *метод (method)* относится к объектно-ориентированным языкам (например, C++) и представляет собой дальнейшее обобщение понятия функций применительно к *классам и объектам* (см. далее). – *Прим. ред.*

объявление типа возвращаемого значения. Далее приведен пример функции на языке C, которая вычисляет площадь круга на основании его радиуса.

---

```
// Функция C для вычисления площади круга
❶ double ❷areaOfCircle(❸double radius)
{
    double area = 3.14 * radius * radius;
    ❹ return area;
}
```

---

Тип `double` в начале ❶ указывает на то, что функция возвращает число с плавающей запятой (`double` – один из типов с плавающей запятой в C). У функции есть имя, `areaOfCircle` ❷, которое должно передавать, что делает функция – в данном случае вычисляет площадь круга. Функция принимает один входной параметр с именем `radius` ❸, также имеющий тип `double`.

Между открывающими и закрывающими фигурными скобками находится тело функции, которое определяет, как именно работает функция. Мы объявляем локальную переменную с именем `area`. Она также имеет тип `double`. Площадь вычисляется как  $\pi \times \text{radius}^2$  и присваивается переменной `area`. Наконец, функция возвращает значение переменной `area` ❹. Обратите внимание, что область видимости переменной `area` ограничена, к ней нельзя получить доступ вне этой функции. Когда функция возвращает значение, локальная переменная `area` отбрасывается (вероятно, она была сохранена в стеке), но ее значение возвращается вызывающей стороне, вероятно, через регистр процессора.

Ниже приведена аналогичная функция вычисления площади, на этот раз написанная на языке Python.

---

```
# Функция Python для вычисления площади круга
def area_of_circle(radius)
    area = 3.14 * radius * radius
    return area
```

---

Давайте сравним два примера функций. Обе вычисляют площадь как  $\pi \times \text{radius}^2$  и затем возвращают это значение. Обе принимают один входной параметр с именем `radius`. В версии на языке C тип возвращаемого значения явно определен как `double`, тип радиуса – тоже, как `double`, в то время как в версии на языке Python эти типы объявлять не требуется, они присваиваются автоматически. Python указывает на начало определения функции с помощью ключевого слова `def`.



## Вызов функций

Определения функции в программе недостаточно, чтобы гарантировать ее выполнение. Определение функции просто делает ее доступной для вызова из другого места, когда это необходимо. Такое обращение к функции называется *вызовом функции*. Вызывающий код устанавливает все необходимые параметры и передает управление функции. Затем функция выполняет свой код и возвращает управление (и выходное значение) обратно вызывающему коду. Ниже показан вызов функции из нашего примера на языке C:

---

```
// Вызов функции дважды в C, каждый раз с разными входными данными
double area1 = areaOfCircle(2.0);
double area2 = areaOfCircle(38.6);
```

---

и в Python:

---

```
# Вызов функции дважды в Python
area1 = area_of_circle(2.0)
area2 = area_of_circle(38.6)
```

---

Когда функция возвращается, вызывающий код должен где-то хранить возвращаемое значение. В обоих примерах объявлены переменные `area1` и `area2` для хранения возвращаемых значений после вызова функции. В обоих языках `area1` равна 12,56, `area2` равна 4678,4744. На самом деле вызывающий код может просто игнорировать возвращаемое значение и не присваивать его переменной, но это не очень удобно, учитывая назначение этой функции. На рис. 9-5 показано, как вызов функции временно передает управление этой функции.

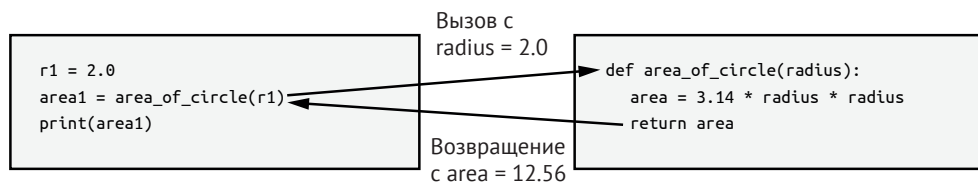


Рис. 9-5. Вызов функции

На рис. 9-5 код Python слева вызывает функцию `area_of_circle`, передавая ей значение входного параметра `radius` равное 2,0. После чего код слева ждет, пока функция справа завершит свою работу. После того как функция возвращает управление, код слева сохраняет возвращаемое значение в переменной `area1`, а затем продолжает выполнение.

## Использование библиотек

Хотя программисты определяют функции для собственного использования, важной частью программирования является знание того, как лучше использовать функции, которые уже написаны другими людьми. Языки программирования обычно включают в себя большой набор функций, известный как *стандартная библиотека* для данного языка. В данном контексте *библиотека* – это коллекция фрагментов кода, предназначенная для использования другими программами. И С, и Python включают стандартные библиотеки, которые предоставляют функции для таких задач, как вывод на консоль<sup>1</sup>, работа с файлами и обработка текста. Стандартная библиотека Python особенно обширна и хорошо известна. Хотя это не всегда так, но большинство реализаций языка включает стандартную библиотеку этого языка, поэтому программисты могут полагаться на эти функции.

### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 17 на стр. 236, где вы сможете использовать полученные знания для написания простой игры в угадку на Python. Написание будет включать в себя использование стандартной библиотеки Python.*

Помимо стандартной библиотеки, для многих языков программирования существуют дополнительные библиотеки функций. Разработчики пишут библиотеки для использования другими пользователями и предоставляют их в виде исходного кода или скомпилированных файлов. Иногда эти библиотеки распространяются неформально, а некоторые языки программирования имеют хорошо известный, общепринятый механизм публикации библиотек. Совместно используемый набор библиотек называется *пакетом*, а система для совместного использования таких пакетов называется *менеджером пакетов*. Для языка С доступно несколько менеджеров пакетов, но ни один из них не является общепризнанным стандартом для программистов на С. Менеджер пакетов Python называется *pip*. Он позволяет легко устанавливать разработанные сообществом программистов программные библиотеки для Python, и широко используется разработчиками Python.

<sup>1</sup> Консоль в программировании – элемент компьютерной архитектуры, представляющий собой устройство взаимодействия с пользователем через текстовый интерфейс (например, экран в совокупности с клавиатурой). Иногда используется как синоним *терминала* (см. сноску 1 на стр. 194). Несколько иной смысл термина «консоль» в главе 10 («игровая консоль» – приставка к телевизору для запуска видеоигр). – *Прим. ред.*

# Объектно-ориентированное программирование

Языки программирования разработаны для поддержки определенных *парадигм*, или подходов к программированию. Примерами являются процедурное программирование, функциональное программирование и объектно-ориентированное программирование. Язык может быть разработан для поддержки одной или нескольких парадигм, и от разработчика программного обеспечения зависит, как использовать язык в соответствии с определенной парадигмой. Рассмотрим одну из популярных парадигм: *объектно-ориентированное программирование* – подход к программированию, при котором код и данные группируются вместе в конструкцию, известную как *объект*. Объекты предназначены для представления логической группировки данных и функциональности таким образом, чтобы моделировать концепции реального мира.

В объектно-ориентированных языках программирования обычно используется подход, основанный на классах. *Класс* – это схема объекта. Объект, созданный на основе класса, считается *экземпляром* этого класса. Функции, определенные в классе, называются *методами*, а переменные, объявленные в классе, называются *полями*.

В Python поля, имеющие разные значения для каждого экземпляра класса, называются *переменными экземпляра*, а поля, имеющие одно и то же значение для всех экземпляров класса, называются *переменными класса*.

Например, можно написать класс, описывающий банковский счет. Класс банковского счета может иметь поле для баланса, поле для имени владельца, а также методы для снятия и внесения денег. Класс описывает общий банковский счет, но конкретного экземпляра банковского счета не существует, пока не будет создан объект банковского счета из этого класса. Это показано на рис. 9-6.

Как видно на рис. 9-6, класс `BankAccount` описывает поля и методы банковского счета, давая нам представление о том, что такое банковский счет. Было создано два объекта, экземпляра класса `BankAccount`. Эти объекты представляют собой конкретные банковские счета, которым присвоены имена и балансы. Мы можем использовать методы `withdraw` или `deposit` каждого объекта для изменения поля `balance`. На языке Python пополнение банковского счета с именем `myAccount` будет выглядеть следующим образом, что приведет к увеличению поля баланса на 25:

---

```
myAccount.deposit(25)
```

---

## ПРИМЕЧАНИЕ

Обратитесь к проекту № 18 на стр. 237, где вы сможете попробовать реализовать на языке Python только что описанный класс банковского счета.

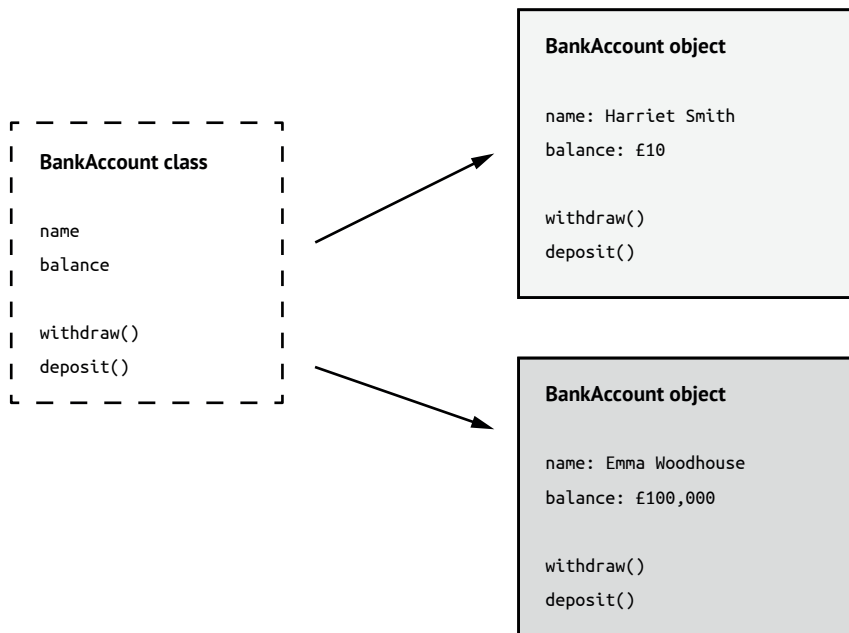


Рис. 9-6. Объекты банковского счета создаются из класса банковского счета

## Компилируемый или интерпретируемый

Как уже упоминалось ранее, исходный код является текстом программы, изначально написанным разработчиками и обычно созданным на языке программирования, который процессоры не понимают напрямую. Процессоры понимают только машинный язык, поэтому требуются дополнительные шаги: исходный код должен быть либо скомпилирован в машинный код, либо интерпретирован другим кодом во время выполнения.

В *компилируемом языке*, таком как C, исходный код преобразуется в машинные инструкции, которые могут быть непосредственно выполнены процессором. Этот процесс был описан ранее в этой главе в разделе «Обзор программирования высокого уровня» на стр. 199. Исходный код компилируется в процессе разработки, и скомпилированные исполняемые файлы (иногда называемые *двоичными файлами*) передаются конечным пользователям. Когда конечные пользователи запускают двоичные файлы, им не нужен доступ к исходному коду. Скомпилированный код, как правило, работает быстро, но он выполняется только на той архитектуре, для которой он был скомпилирован. На рис. 9-7 показан пример того, как разработчик компилирует и запускает программу на языке C из командной строки с помощью GNU C Compiler (*gcc*).

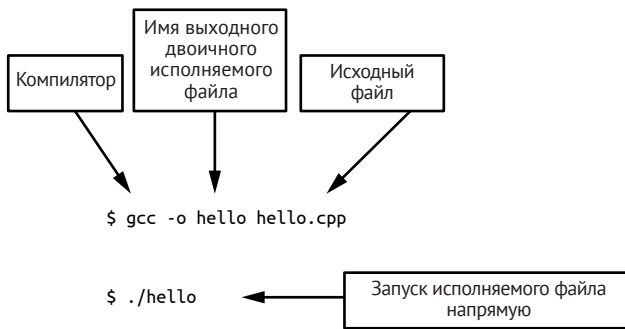


Рис. 9-7. Компиляция исходного файла на языке C в исполняемый файл, который может быть запущен сам по себе

В *интерпретируемом языке*, таком как Python, исходный код не компилируется заранее. Вместо этого он считывается программой, называемой *интерпретатором*, которая читает и выполняет инструкции программы. Именно машинный код интерпретатора фактически выполняется на процессоре. Разработчики кода на интерпретируемых языках могут распространять свой исходный код, а конечные пользователи могут запускать его непосредственно, без довольно сложного этапа компиляции. В этом сценарии разработчикам не нужно беспокоиться о компиляции кода для множества различных платформ, так как пока пользователь имеет соответствующий интерпретатор на своей системе, он может запускать код. Таким образом, распространяемый код является независимым от платформы.

Интерпретированный код, как правило, работает медленнее, чем скомпилированный, из-за времени, требуемого на интерпретацию кода в процессе его выполнения. Распространение интерпретируемого кода лучше всего работает, если у пользователя уже установлен необходимый интерпретатор или пользователь достаточно технически грамотен, чтобы установка интерпретатора не стала для него препятствием. В противном случае разработчику необходимо либо поставлять интерпретатор в комплекте со своим программным обеспечением, либо помочь пользователю установить интерпретатор.

На рис. 9-8 показан пример запуска программы Python из командной строки, предполагающий, что интерпретатор Python версии 3 уже установлен. Обратите внимание, как исходный код Python в *hello.py* передается непосредственно интерпретатору, никаких промежуточных шагов не требуется.

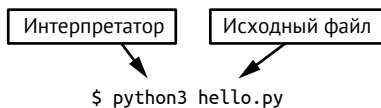


Рис. 9-8. Интерпретатор Python выполняет исходный код на Python

Некоторые языки используют систему, представляющую собой гибрид этих двух подходов. Такие языки компилируются в *промежуточный язык*, или *байт-код*. Байт-код похож на машинный код, но вместо того, чтобы ориентироваться на конкретную аппаратную архитектуру, байт-код предназначен для запуска на виртуальной машине, как показано на рис. 9-9.

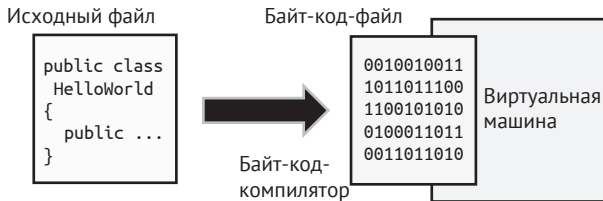


Рис. 9-9. Компилятор байт-кода превращает исходный код в байт-код, который запускается внутри виртуальной машины

В данном контексте *виртуальная машина* – это программная платформа, предназначенная для запуска другого программного обеспечения. Виртуальная машина предоставляет виртуальный процессор и среду выполнения, абстрагируясь от деталей реального базового оборудования и операционной системы. Например, исходный код Java компилируется в байт-код Java, который затем запускается в виртуальной машине Java. Аналогично исходный код C# компилируется в Common Intermediate Language и запускается в виртуальной машине .NET Common Language Runtime (CLR). CPython, оригинальная реализация Python, фактически преобразует исходный код Python в байт-код перед его запуском, хотя это деталь реализации интерпретатора CPython и в основном скрыта от разработчиков на языке Python. Языки программирования, использующие байт-код, сохраняют независимость от платформы интерпретируемых языков, имея при этом некоторые преимущества в производительности скомпилированного кода.

## Вычисление факториала в C

В завершение нашего знакомства с высокоуровневым программированием давайте рассмотрим реализацию алгоритма факториала на этот раз на языке C. Мы уже делали это раньше на ассемблере ARM, поэтому рассмотрение той же логики на языке C должно послужить хорошим сравнением между языком ассемблера и высокоуровневым языком. Этот код на языке C использует несколько концепций, которые мы только что рассмотрели.

Я решил использовать C, а не Python, потому что это компилируемый язык, и мы можем изучить скомпилированный машинный код. Вот простая функция на языке C, которая вычисляет факториал числа:

---

```
// Вычисление факториала n
int factorial(int n)
{
    int result = n;

    while(--n > 0)
    {
        result = result * n;
    }

    return result;
}
```

---

Другой код может вызывать эту функцию, передавая параметр *n* как значение, факториал которого должен быть вычислен. Затем функция производит внутренний расчет факториала, сохраняя его в локальной переменной *result* и возвращает вычисленное значение вызывающей стороне. Как и в случае с ассемблерным кодом в главе 8, давайте снова воспользуемся упражнением для углубленного изучения этого кода.

### УПРАЖНЕНИЕ 9-2: Выполните программу на языке C в уме

Попробуйте выполнить предыдущую функцию факториала в уме или с помощью карандаша и бумаги. Предположим, что входное значение  $n = 4$ . Когда функция вернет результат, возвращаемое значение должно быть равно 24. Я рекомендую для каждой строки отслеживать значения *n* и *result* до и после завершения оператора. Проработайте код до конца цикла *while* и посмотрите, получите ли вы ожидаемый результат. Ответ находится в приложении А. Обратите внимание, что в условии цикла *while* ( $--n > 0$ ) оператор декремента ( $--$ ) помещен перед переменной *n*. Это означает, что *n* уменьшается на 1, прежде чем его значение сравнивается с 0. Это происходит каждый раз, когда оценивается условие цикла *while*.

Я надеюсь, что версия нашего алгоритма на языке C кажется вам более читабельной, чем версия на ассемблере ARM! Другим важным преимуществом этой версии нашего кода для вычисления факториала является то, что он не привязан к определенному типу процессора. Он может быть скомпилирован для любого процессора при наличии соответствующего компилятора. Если вы скомпилируете предыдущий C-код для процессора ARM, то увидите сгенерированный машинный код, похожий на ассемблер ARM, который мы рассмотрели ранее. У вас будет возможность сделать это в проекте № 19, а пока я скомпилировал и разобрал код для вас:

---

Адрес	На ассемблере
0001051c	sub r3, r0, #1
00010520	cmp r3, #0
00010524	bxle lr
00010528	mul r0, r3, r0
0001052c	subs r3, r3, #1
00010530	bne 00010528
00010534	bx lr

---

Как вы можете видеть, код, сгенерированный из исходного текста на языке C, очень похож на пример вычисления факториала на ассемблере, который мы рассматривали в главе 8. Есть некоторые отличия, но они не имеют отношения к нашему обсуждению. Главное – нужно отметить, что программа может быть написана на высокоуровневом языке, таком как C, а компилятор может выполнить всю тяжелую работу по переводу высокоуровневых выражений в машинный код. Как вы можете видеть, программирование на высокоуровневом языке может упростить работу разработчика, но в конечном итоге мы все равно получаем байты машинного кода, потому что это то, что нужно процессору.

#### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 19 на стр. 239, где вы сможете попробовать скомпилировать и затем разобрать программу для вычисления факториала на языке C.*

Здесь произошло кое-что интересное, и я хочу убедиться, что вы это не пропустили. Мы начали с исходного кода, написанного на языке программирования C, скомпилировали его в машинный код, а затем разобрали его на языке ассемблера. Из этого следует, что, если у вас на компьютере есть скомпилированная программа или библиотека, можете изучить ее код как язык ассемблера! У вас может не быть доступа к исходному коду, но ассемблерная версия программы всегда находится в вашем распоряжении.

Мы рассматривали машинный код и язык ассемблера для процессора ARM, но, как уже упоминалось ранее, одним из преимуществ разработки на языке высокого уровня, таком как C, является то, что тот же код может быть скомпилирован для другого процессора. Фактически тот же самый код может быть даже скомпилирован для другой операционной системы при условии, что данный код не использует функционал, специфичный для конкретной операционной системы. Чтобы проиллюстрировать этот момент, я скомпилировал тот же самый код на C для вычисления факториала для 32-битного x86-процессора, на этот раз под Windows, а не под Linux. Вот сгенерированный машинный код, представленный в виде языка ассемблера:



---

Адрес	На ассемблере
00406c35	mov ecx,dword ptr [esp+4]
00406c39	mov eax,ecx
00406c3b	jmp 00406c40
00406c3d	imul eax,ecx
00406c40	dec ecx
00406c41	test ecx,ecx
00406c43	jg 00406c3d
00406c45	ret

---

Я не буду подробно останавливаться на деталях этого кода, но не стесняйтесь изучить набор инструкций x86 и интерпретировать код самостоятельно. Главное, что, я надеюсь, вы вынесете из этого примера, что высокоуровневые языки, такие как С, позволяют разработчикам писать код, который легче понять, чем ассемблер, и который может быть легко скомпилирован для различных процессоров.

## Выводы

В этой главе мы рассмотрели высокоуровневые языки программирования. Такие языки не зависят от конкретного процессора и синтаксически ближе к человеческому языку. Вы узнали об общих элементах, встречающихся во всех языках программирования, таких как комментарии, переменные, функции и возможности циклов. Вы увидели, как эти элементы используются в двух языках программирования: С и Python. Наконец, мы рассмотрели пример программы на языке С, и вы увидели разобранный машинный код, полученный в результате компиляции кода высокого уровня.

В следующей главе мы рассмотрим операционные системы. Начнем с обзора возможностей, предоставляемых операционными системами, узнаем о различных семействах операционных систем и углубимся в то, как работают операционные системы. Попутно у вас будет возможность изучить Raspberry Pi OS, версию Linux, предназначенную для Raspberry Pi.

## ПРОЕКТ № 14: Изучение переменных

Необходимые условия: Raspberry Pi, работающий на Raspberry Pi OS. Я рекомендую вам обратиться к приложению В и прочитать весь раздел «Raspberry Pi» на стр. 416, если вы этого еще не сделали.

В этом проекте вы напишете высокоуровневый код, использующий переменные, и изучите, как это работает в памяти. С помощью выбранного вами текстового редактора создайте новый файл `vars.c` в корне вашей домашней папки. Введите в текстовый редактор следующий код на языке C (не обязательно сохранять отступы и пустые строки, но обязательно сохраняйте переносы строк).

---

```
#include <stdio.h>❶
#include <signal.h>

int main()❷
{
    int points = 27;❸
    int year = 2020;❹

    printf("points is %d and is stored at 0x%08x\n", points, &points);❺
    printf("year is %d and is stored at 0x%08x\n", year, &year);

    raise(SIGINT);❻

    return 0;
}
```

---

Прежде чем продолжить, давайте рассмотрим исходный код. Он начинается с включения пары заголовочных файлов ❶. Эти файлы содержат сведения, необходимые компилятору языка C, о функциях `printf` и `raise`, которые используются далее в программе. Далее определяется функция `main` ❷, это точка входа программы, с которой начинается выполнение. Затем программа объявляет две целочисленные переменные, `points` ❸ и `year` ❹, и присваивает им значения. Затем программа выводит значения переменных и их адреса в памяти (в шестнадцатеричном формате) ❺. Оператор `raise(SIGINT)` заставляет программу прекратить выполнение ❻. Это не является обычным кодом, выполняемым конечным пользователем, это просто техника, которую мы используем здесь для помощи в отладке.

После сохранения файла используйте GNU C Compiler (`gcc`) для компиляции кода в исполняемый файл. Откройте терминал на вашем Raspberry Pi и введите следующую команду, чтобы вызвать компилятор. Эта команда принимает файл `vars.c` в качестве входных данных, компилирует, компоует код и выводит исполняемый файл с именем `vars`.

---

```
$ gcc. -o. vars. vars.c
```

---

Теперь попробуйте запустить скомпилированный код с помощью следующей команды. Программа должна вывести значения и адреса двух переменных программы.

---

```
$ ./vars
```

---

Убедившись, что программа работает, запустите ее под GNU Debugger (gdb) и просмотрите переменные в памяти.

---

```
$ gdb vars
```

---

В этот момент gdb загрузил файл, но ни одна инструкция еще не запущена. В окне (gdb) введите следующее, чтобы запустить программу, которая будет выполняться до тех пор, пока не будет выполнен оператор `raise(SIGINT)`.

---

```
(gdb) run
```

---

Когда программа вернется к окну (gdb), вы должны увидеть несколько строк, где будут напечатаны значения и адреса памяти переменных. После этих строк вы также можете увидеть тревожное сообщение об «отсутствии такого файла или каталога» («no such file or directory»), но его можно проигнорировать. Это просто отладчик пытается найти исходный код, которого нет в вашей системе. Вывод, на который вам стоит обратить внимание, должен выглядеть примерно так:

---

```
Starting program: /home/pi/vars
points is 27 and is stored at 0x7efff1d4
year is 2020 and is stored at 0x7efff1d0
```

---

Теперь вы знаете адреса памяти, и, поскольку вы находитесь в отладчике, можно проверить, что хранится по этим адресам. В этом выводе видно, что `year` хранится по меньшему адресу, а `points` хранятся на 4 байта дальше, поэтому вы посмотрите память, начиная с адреса переменной `year`, `0x7efff1d0` в моем случае. Ваш адрес может быть другим. Следующая команда выгрузит в память три 32-битных значения в шестнадцатеричном исчислении, начиная с адреса `0x7efff1d0`. Замените `0x7efff1d0` на адрес `year` в вашей системе, если он отличается.

---

```
(gdb) x/3xw 0x7efff1d0
0x7efff1d0:      0x000007e4      0x0000001b      0x0000000
```

---

Здесь видно, что значение, хранящееся по адресу 0x7efff1d0, равно 0x000007e4. Это 2020 в десятичной системе, ожидаемое значение `ueag`. А значение, сохраненное 4 байта спустя, равно 0x0000001b, или 27 в десятичной системе, т. е. ожидаемое значение `points`. Следующее значение в памяти равно 0 и не является одной из наших переменных. Память обычно представлена в шестнадцатеричном формате, но если вы хотите увидеть эти значения в десятичном виде, то вместо этого можно использовать следующую команду:

---

```
(gdb) x/3dw 0x7efff1d0
0x7efff1d0:  2020  27  0
```

---

Вы смотрите на память в 32-битных (4-байтовых) отрезках, поскольку таков размер переменных, используемых в этой программе. Но на самом деле память имеет байтовую адресацию, т. е. каждый байт имеет свой собственный адрес. Вот почему у `points` адрес на 4 байта больше, чем адрес `ueag`. Давайте рассмотрим тот же диапазон памяти как серию байтов:

---

```
(gdb) x/12xb 0x7efff1d0
0x7efff1d0: 0xe4  0x07  0x00  0x00  0x00  0x1b  0x00  0x00  0x00  0x00
0x7efff1d8: 0x00  0x00  0x00  0x00
```

---

Посмотрите на выделенное здесь значение для `ueag`. Обратите внимание, что младший байт (0xe4) идет первым. Это связано с хранением данных в порядке от младшего к старшему, как обсуждалось на стр. 195 в проекте № 13. Вы можете выйти из `gdb` с помощью `q` (программа спросит вас, хотите ли вы выйти, даже если сеанс отладки активен, ответьте `y`).

## ПРОЕКТ № 15: Изменение типа значения, на которое ссылается переменная в PYTHON

Необходимые условия: Raspberry Pi, работающий на Raspberry Pi OS. Я рекомендую вам обратиться к приложению В и прочитать весь раздел «Raspberry Pi» на стр. 416, если вы этого еще не сделали.

В этом проекте вам предстоит написать код, который устанавливает переменную Python в значение определенного типа, а затем обновляет эту переменную для ссылки на значение другого типа. С помощью выбранного вами текстового редактора создайте новый файл `vartype.py` в корне вашей домашней папки. Введите следующий код Python в текстовый редактор:

```
age = 22
print('What is the type?')
print(type(age))

age = 'twenty-two'
print('Now what is the type?')
print(type(age))
```

Этот код устанавливает переменную с именем `age` как целочисленное значение, а затем печатает тип этого значения. Затем он устанавливает `age` как строковое значение и снова печатает тип.

После сохранения файла его можно запустить из окна терминала с помощью интерпретатора Python, как показано ниже:

```
$ python3 vartype.py
```

Вы должны увидеть вывод, подобный следующему:

```
What is the type?
<class 'int'>
Now what is the type?
<class 'str'>
```

Вы можете увидеть, как меняется тип с целого числа на строку, просто посредством присвоения переменной нового значения. Пусть вас не смущает термин *класс* (*class*), в Python 3 встроенные типы, такие как `int` и `str`, считаются классами (что рассматривалось в разделе «Объектно-ориентированное программирование» на стр. 222). Установить переменную в значение другого типа можно легко в Python, но совершенно невозможно в C.

### ВЕРСИИ PYTHON

Сегодня используются две основные версии Python: Python 2 и Python 3. Начиная с 1 января 2020 года, Python 2 больше не поддерживается, т. е.

для него не будет новых исправлений ошибок. Разработчикам Python рекомендуется перенести старые проекты на Python 3, а новые проекты должны быть ориентированы на Python 3. Соответственно, проекты в этой книге используют Python 3. В Raspberry Pi OS и некоторых других дистрибутивах Linux запуск `python` из командной строки вызовет интерпретатор Python 2, а запуск `python3` вызовет интерпретатор Python 3. Поэтому в проектах, описанных в этой книге, нужно специально запускать `python3`, а не `python`. Однако на других платформах или даже на будущих версиях Raspberry Pi OS это может быть не так, и ввод `python` может вызвать Python 3. Вы можете проверить версию вызываемого Python следующим образом:

---

```
$ python --version
```

---

или:

---

```
$ python3 --version
```

---

## ПРОЕКТ № 16: Стек или куча

Необходимое условие: проект № 14.

В этом проекте вы рассмотрите, где в запущенной программе располагаются переменные, в памяти стека или в памяти кучи. Откройте терминал на вашем Raspberry Pi и начните с отладки программы `vars`, которую вы ранее скомпилировали в проекте № 14:

---

```
$ gdb vars
```

---

На данный момент `gdb` загрузил файл, но ни одна инструкция еще не запущена. В окне `gdb` введите следующее выражение для запуска программы, выполнение которой будет продолжаться до тех пор, пока не будет выполнен оператор `SIGINT`.

---

```
(gdb) run
```

---

Снова посмотрите на адреса памяти переменных `points` и `year`. В моем случае эти переменные были найдены по адресам `0x7efffd4` и `0x7efffd0`, но ваши адреса могут отличаться. Теперь используйте следующую команду, чтобы увидеть все отображенные области памяти для вашей запущенной программы:

---

```
(gdb) info proc mappings
```

---

В результате вы увидите начальные и конечные адреса различных диапазонов памяти, используемых этой программой. Найдите тот, который включает адреса ваших переменных. Адреса обеих переменных должны попадать в один диапазон. У меня получилось следующее:

---

```
0x7efdf000 0x7f000000 0x21000 0x0 [stack]
```

---

Как видите, `gdb` показывает, что этот диапазон памяти выделен для стека, т. е. именно там, где должны находиться локальные переменные. Вы можете выйти из `gdb` с помощью `q` (программа спросит вас, хотите ли вы выйти, даже если сеанс отладки активен, ответьте `y`).

Давайте теперь посмотрим на память, выделенную для кучи. Вам нужно изменить файл `vars.c` и перестроить его так, чтобы программа выделяла часть памяти в куче. С помощью любого текстового редактора откройте существующий файл `vars.c`. Добавьте следующую строку кода в качестве самой первой:

---

```
#include <stdlib.h>
```

---

Затем добавьте эти две строки непосредственно перед строкой `SIGINT`:

---

```
void * data = malloc(512);  
printf("data is 0x%08x and is stored at 0x%08x\n", data, &data);
```

---

Давайте разберемся, что означают эти изменения. Мы вызываем функцию выделения памяти `malloc` для выделения 512 байт памяти из кучи. Функция `malloc` возвращает адрес вновь выделенной памяти. Этот адрес хранится в новой локальной переменной `data`. Затем программа печатает два адреса памяти: адрес нового расположения в куче и адрес самой переменной `data`, которая должна находиться в стеке.

После сохранения файла используйте `gcc` для компиляции кода:

---

```
$ gcc -o vars vars.c
```

---

Теперь запустите программу снова:

---

```
$ gdb vars  
(gdb) run
```

---

Проверьте новые выведенные значения. У меня значения следующие:

---

```
data is 0x00022410 and is stored at 0x7efff1ac
```

---

Мы ожидаем, что первый адрес, вернувшийся из `malloc`, будет находиться в куче. Второе значение, адрес локальной переменной `data`, должно находиться в стеке. Снова выполните следующую команду, чтобы увидеть диапазоны памяти этой программы и посмотреть, куда попадают эти два адреса.

---

```
(gdb) info proc mappings
```

```
...
          0x22000    0x43000    0x21000    0x0 [heap]
...
    0x7efdf000 0x7f000000    0x21000    0x0 [stack]
```

---

Найдите соответствующие диапазоны адресов в вашей системе и убедитесь, что адреса попадают в ожидаемые диапазоны кучи и стека. Вы можете выйти из `gdb` с помощью `q`.



## ПРОЕКТ № 17: Напишите игру-угадайку

В этом проекте вы напишете игру-угадайку на Python, основываясь на том, что изучили в этой главе. С помощью любого текстового редактора создайте новый файл с именем `guess.py` в корне вашей домашней папки. Введите в текстовый редактор следующий код на языке Python. В Python отступы имеют значение, поэтому убедитесь, что вы правильно расставили их.

```
from random import randint❶

secret = randint(1, 10)❷
guess = 0❸
count = 0❹

print('Guess the secret number between 1 and 10')

while guess != secret:❺
    guess = int(input())❻
    count += 1

    if guess == secret:❼
        print('You got it! Nice job.')
    elif guess < secret:
        print('Too low. Try again.')
    else:
        print('Too high. Try again.')

print('You guessed {0} times.'.format(count))❽
```

Давайте рассмотрим, как работает эта программа. Этот код начинается с импорта функции `randint`, которая генерирует случайные целые числа ❶. Это пример использования функции, которая была написана кем-то другим. Функция `randint` является частью стандартной библиотеки Python. Ее вызов возвращает случайное целое число в диапазоне от 1 до 10, которое мы сохранили в переменной с именем `secret` ❷. Затем код устанавливает переменную под названием `guess` равной 0 ❸. Эта переменная хранит догадку игрока, и ей присваивается начальное значение 0, которое, как мы можем быть уверены, не совпадет со значением `secret`. Третья переменная с именем `count` ❹ отслеживает количество попыток, которое потребовалось игроку.

Цикл `while` выполняется до тех пор, пока введенное игроком значение не совпадет со значением `secret` ❺. Код внутри цикла вызывает встроенную функцию `input`, чтобы получить ответ пользователя с консоли ❻. Результат преобразуется в целое число и сохраняется в переменной `guess`. Каждый раз при вводе угадываемого числа оно сверяется с переменной `secret`, чтобы определить, совпадает ли оно, слишком мало или слишком велико ❼. Как только угаданная игроком цифра совпадает с `secret`, цикл завершается, и программа выводит количество попыток, которое потребовалось игроку ❽.

После сохранения файла его можно запустить с помощью интерпретатора Python следующим образом:

```
$ python3 guess.py
```

Попробуйте выполнить программу несколько раз. Загаданное число должно меняться каждый раз, когда вы ее запускаете. Вы можете попробовать изменить программу так, чтобы диапазон допустимых целых чисел был больше, или, возможно, захотите вывести свои собственные сообщения. В качестве эксперимента попробуйте модифицировать программу так, чтобы при очень близком попадании программа выводила другое сообщение.

## ПРОЕКТ № 18: Использование класса банковского счета в PYTHON

В этом проекте вы создадите класс банковского счета в Python, а затем создадите объект на основе этого класса. С помощью выбранного вами текстового редактора создайте новый файл с именем *bank.py* в корне вашей домашней папки. Введите в текстовый редактор следующий код на языке Python. Вы можете не вводить комментарии (строки, начинающиеся с #), если не хотите. Обратите внимание, что `__init__` имеет два символа подчеркивания в начале и в конце.

```
# Определите класс банковского счета в Python.
class BankAccount: ❶
    def __init__(self, balance, name): ❷
        self.balance = balance ❸
        self.name = name ❹

    def withdraw(self, amount): ❺
        self.balance = self.balance - amount

    def deposit(self, amount): ❻
        self.balance = self.balance + amount

# Создайте объект банковского счета на основе класса.
smithAccount = BankAccount(10.0, 'Harriet Smith') ❼

# Положите на счет дополнительные деньги.
smithAccount.deposit(5.25) ❸

# Выведите баланс счета.
print(smithAccount.balance) ❹
```

Этот код определяет новый класс под названием `BankAccount` ❶. Его функция `__init__` ❷ автоматически вызывается при создании экземпляра класса. Эта функция устанавливает переменные экземпляра `balance` ❸ и `name` ❹ в значениях, переданных в функцию при вызове. Эти переменные уникальны для каждого создаваемого экземпляра класса. Определение класса также включает два метода: `withdraw` ❺ и `deposit` ❻, которые просто изменяют баланс. После определения класса код переходит к созданию экземпляра класса ❼. Теперь этот объект банковского счета можно использовать, обращаясь к его переменным и методам. Здесь производится пополнение счета ❸, затем извлекается новый баланс, который выводится на печать ❾.

После сохранения файла его можно запустить с помощью интерпретатора Python следующим образом:

---

```
$ python3 bank.py
```

---

Вы увидите, что в окне терминала будет выведен баланс счета 15,25. На самом деле это был слишком сложный способ вычисления банковского баланса! Все числа были жестко закодированы в программе, и нам на самом деле не нужно было использовать объектно-ориентированный подход для решения этой задачи. Однако я надеюсь, что этот пример поможет вам понять, как работают классы и объекты.

## ПРОЕКТ № 19: Факториал на C

Необходимые условия: проекты № 12 и № 13.

В этом проекте вы создадите программу для вычисления факториала на языке программирования C, подобную той, которую мы рассматривали ранее в этой главе. Затем вы изучите машинный код, сгенерированный при компиляции программы. С помощью выбранного вами текстового редактора создайте новый файл с именем *fact2.c* в корне вашей домашней папки. Введите следующий код на языке C:

```
#include <stdio.h>

// Calculate the factorial of n.
int factorial(int n)❶
{
    int result = n;

    while(--n > 0)
    {
        result = result * n;
    }

    return result;
}

int main()❷
{
    int answer = factorial(4);❸
    printf('%d\n', answer);❹
}
```

Вы можете видеть, что функция `factorial` <sup>❶</sup> точно такая же, как и в приведенном ранее в этой главе примере на языке C. Это основной код для вычисления факториала. Однако, чтобы сделать эту программу пригодной для использования, необходимо привлечь функцию `main` <sup>❷</sup>, которая служит точкой входа, именно с нее начинается выполнение программы. Из `main` программа вызывает функцию `factorial` со значением 4, сохраняя результат в локальной переменной с именем `answer` <sup>❸</sup>. Затем программа выводит значение `answer` на терминал <sup>❹</sup>.

После сохранения файла используйте `gcc` для компиляции кода в исполняемый файл. Следующая команда принимает *fact2.c* в качестве входного файла и выдает исполняемый файл с именем *fact2*. Отдельного шага компоновки тут не требуется. Также обратите внимание на опцию командной строки `-O` (это заглавная буква O): она означает включение режима оптимизации компилятора<sup>1</sup>. Я добавил эту опцию здесь, потому что в данном

<sup>1</sup> Как вы могли заметить, обычно применяющаяся опция с той же буквой «-o», но строчной (см. пример командной строки) задает имя создаваемого исполняемого файла (в данном случае *fact2*). – Прим. ред.

случае она производит код, более похожий на код ассемблера из проекта № 12.

---

```
$ gcc -O -o fact2 fact2.c
```

---

Теперь попробуйте запустить код с помощью следующей команды. Если все работает как надо, программа должна вывести вычисленный результат 24 в следующей строке.

---

```
$ ./fact2
```

---

Теперь, когда у вас есть исполняемый файл *fact2*, используйте методы из проектов № 12 и 13, чтобы проверить скомпилированный файл. Я не буду повторять все детали, но те же подходы, которые вы использовали ранее, будут работать и здесь. Вот вам несколько команд для начала работы:

---

```
$ hexdump -C fac2
$ objdump -s fac2
$ objdump -d fac2
$ gdb fac2
```

---

Вы должны сразу заметить, что в файле *fact2* очень много данных! Скомпилированный двоичный файл ELF несет некоторые дополнительные данные, необходимые для программы, написанной на C. На моем компьютере оригинальный файл *fact* ELF имел размер 940 байт, тогда как файл *fact2* ELF имеет размер 8 364 байт, что в 9 раз больше! Конечно, версия на языке C включает дополнительный функционал для вывода значения на печать, поэтому некоторое увеличение размера ожидаемо.

При рассмотрении разобранного кода мы хотим прежде всего исследовать функцию *factorial*. Сравните ее с кодом факториала, который вы написали на языке ассемблера в главе 8. Вы можете заметить, что *gdb* показывает точку входа, отличную от *main*. Это потому, что в программах на языке C есть код инициализации, который вызывается до вызова основной точки входа. Если вы хотите пропустить этот код и сразу перейти к функции *factorial*, вы можете установить точку останова (*break factorial*), затем ввести *run*, а затем *disassemble*.

Сгенерированные на вашем компьютере машинные инструкции могут несколько отличаться, но вот машинный код функции *factorial* и соответствующий ему на языке ассемблера, сгенерированный на моем компьютере. Это вывод из *objdump -d fact2*:

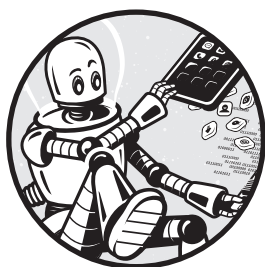
00010408 <factorial>:

10408:	e2403001	sub	r3, r0, #1 <sup>❶</sup>
1040c:	e3530000	cmp	r3, #0 <sup>❷</sup>
10410:	d12fff1e	bxle	lr <sup>❸</sup>
10414:	e0000093	mul	r0, r3, r0 <sup>❹</sup>
10418:	e2533001	subs	r3, r3, #1 <sup>❺</sup>
1041c:	1affffff	bne	10414 <factorial+0xc> <sup>❻</sup>
10420:	e12fff1e	bx	lr <sup>❼</sup>

До вызова этой функции значение *n* было сохранено в *r0*. Когда функция запускается, она сразу же декрементирует *n* и сохраняет результат в *r3* <sup>❶</sup>. Затем программа сравнивает *r3* (т. е. *n*) с нулем <sup>❷</sup>. Если *n* меньше или равно нулю <sup>❸</sup>, то программа выходит из функции. В противном случае *result*, хранящийся в *r0*, вычисляется как *result* × *n* <sup>❹</sup>. Далее *n* декрементируется <sup>❺</sup>, и если *n* не равно нулю <sup>❻</sup>, то программа снова проходит через цикл, возвращаясь по адресу 10414 <sup>❼</sup>. Как только *n* достигает нуля, цикл завершается, и следует выход из функции *factorial* <sup>❼</sup>.

# 10

## ОПЕРАЦИОННЫЕ СИСТЕМЫ



Мы уже рассмотрели аппаратное и программное обеспечение компьютера. В этой главе разберем особый вид программного обеспечения: операционные системы (ОС<sup>1</sup>). Сначала мы рассмотрим проблемы программирования без операционной системы. Затем сделаем обзор операционных систем. Большую часть главы посвятим подробному описанию некоторых основных возможностей операционных систем. В проектах у вас будет возможность изучить работу Raspberry Pi OS.

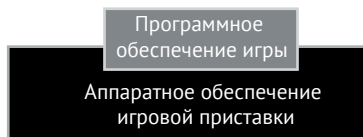
### Программирование без операционной системы

Давайте начнем с рассмотрения того, каково это – использовать и программировать устройство без операционной системы. Как вы увидите через минуту, операционные системы обеспечивают интерфейс между оборудованием и другим программным обеспечением. Однако на устройстве без ОС программное обеспечение имеет прямой доступ к оборудованию.

---

<sup>1</sup> В отечественной литературе сложилась практика именования операционных систем русскоязычным сокращением (ОС), однако при воспроизведении оригинальных названий принято пользоваться англоязычным сокращением от *operating system* (iOS, macOS, Raspberry Pi OS и т. п.). – *Прим. ред.*

Существует множество примеров компьютеров, работающих подобным образом, но давайте остановимся на одном типе: ранних игровых приставках. Если мы вспомним такие игровые приставки, как Atari 2600, Nintendo Entertainment System или Sega Genesis, то увидим аппаратное обеспечение, которое запускает код с картриджа, не задействуя операционную систему. Рисунок 10-1 иллюстрирует идею о том, как программное обеспечение игры запускается непосредственно на аппаратуре консоли, без какого-либо промежуточного программного обеспечения.



*Рис. 10-1. Ранние видеоигры запускались непосредственно на аппаратуре игровой консоли, без операционной системы*

Чтобы воспользоваться такой системой, достаточно было вставить картридж и включить систему, чтобы запустить игру. Игровая консоль могла запускать одновременно только одну программу – игру, находящуюся в слоте картриджа. В большинстве подобных систем включение системы без вставленного картриджа ничего не делало, так как у процессора не было никаких инструкций для выполнения. Чтобы переключиться на другую игру, нужно было выключить систему, поменять картридж и снова включить ее. Не существовало возможности переключения между программами во время работы системы. Не было и программ, работающих в фоновом режиме. Аппаратура была полностью занята только одной программой, игрой.

Для программиста создание игры для такой системы означало принятие на себя ответственности за непосредственное управление аппаратным обеспечением с помощью кода. Как только система включалась, процессор начинал выполнять код на картридже. Разработчику игры нужно было не только написать программу, поддерживающую логику игровых действий, но и инициализировать систему, управлять видеоаппаратурой, считывать состояние входов контроллера и т. д. Различные аппаратные средства консолей имели радикально отличающийся дизайн, поэтому разработчику необходимо было разбираться в тонкостях устройства оборудования, на которое ему приходилось ориентироваться.

К счастью для разработчиков игр старой школы, игровые консоли более или менее сохраняли один и тот же дизайн аппаратного обеспечения в течение всех лет их производства. Например, все консоли Nintendo Entertainment System имели один и тот же тип процессора, оперативной памяти, блока обработки изображений (PPU) и блока обработки звука (APU). Чтобы стать успешным разработчиком для Nintendo, нужно было хорошо разбираться во всем этом оборудовании, но, по крайней мере, оно было одним и тем же в каждой консоли Nintendo, продаваемой геймерам. Разработчики точно знали, какое



оборудование будет стоять в системе, поэтому они могли нацелить свой код на это конкретное оборудование, что позволяло им выжимать все из производительности системы. Однако для переноса игры на другой тип игровой консоли часто приходилось переписывать значительную часть кода. Кроме того, каждый игровой картридж должен был содержать один и тот же код для выполнения обязательных рутинных задач, таких как инициализация оборудования. Хотя разработчики могли повторно использовать код, написанный ими ранее для других игр, это все равно означало, что разные разработчики были вынуждены решать одни и те же задачи заново и с разной степенью успеха.

## Обзор операционных систем

Операционные системы предоставляют другую модель программирования и при этом решают многие проблемы, связанные с написанием кода, ориентированного непосредственно на конкретное оборудование. *Операционная система* – это программное обеспечение, которое взаимодействует с компьютерным оборудованием и обеспечивает среду для выполнения программ. Операционные системы позволяют программам запрашивать системные услуги, такие как чтение из памяти или обмен данными по сети. Операционные системы выполняют инициализацию компьютерной системы и управляют выполнением программ. Это включает в себя параллельное выполнение нескольких программ, или *многозадачность*, обеспечивающую совместное использование процессора и системных ресурсов несколькими программами. ОС устанавливает границы, чтобы обеспечить изоляцию программ друг от друга и от самой ОС, а также гарантировать пользователям, работающим в одной системе, предоставление соответствующего разграничения доступа к оборудованию. Операционную систему можно рассматривать как слой кода между оборудованием (аппаратным обеспечением) и приложениями, как показано на рис. 10-2.

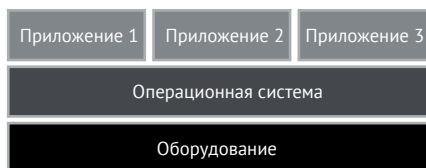


Рис. 10-2. Операционная система действует как промежуточный слой между оборудованием и приложениями

Этот слой предоставляет набор возможностей, которые позволяют абстрагироваться от деталей базового оборудования, позволяя разработчикам программного обеспечения сосредоточиться на логике своих программ, а не на взаимодействии с конкретной аппаратурой. Это очень полезное свойство, учитывая разнообразие современных вычислительных устройств. Принимая во внимание огромное множество

разнообразных типов аппаратного обеспечения для смартфонов и ПК, писать код для каждого типа устройства непрактично. Операционные системы скрывают детали аппаратного обеспечения и предоставляют сервисы, на которые могут опираться приложения.

Компоненты, входящие в состав операционной системы, можно грубо разделить на два основных блока:

- ядро;
- все остальное.

*Ядро* операционной системы отвечает за управление памятью, облегчение работы устройств ввода/вывода и предоставление набора системных услуг для приложений. Ядро позволяет нескольким программам работать параллельно и совместно использовать аппаратные ресурсы. Оно является основной частью операционной системы, но само по себе не дает конечным пользователям возможности взаимодействовать с системой.

Операционные системы также включают в себя компоненты, не относящиеся к ядру, но необходимые для использования системы. К ним относится *оболочка* – пользовательский интерфейс для работы с ядром. Термины «*оболочка*» и «*ядро*» являются частью метафоры операционных систем, в которой ОС рассматривается как орех или семя. Ядро находится в сердцевине, оболочка окружает его.

Оболочка может представлять собой интерфейс командной строки (*command line interface* – CLI) или графический интерфейс пользователя (*graphical user interface* – GUI). Примерами таких оболочек являются оболочка GUI Windows (включая рабочий стол, меню **Пуск**, панель задач и проводник) и оболочка Bash CLI, используемая в системах Linux и Unix.

Некоторые возможности операционных систем обеспечиваются программами, работающими в фоновом режиме отдельно от ядра, называемыми как *демонами* (*daemons*) или *службами* (*services*). Их не следует путать с системными службами ядра, упомянутыми ранее. Примерами таких служб являются Task Scheduler в Windows или cron в Unix и Linux, которые позволяют пользователю планировать график выполнения программ в определенное время.

Операционные системы также обычно включают *библиотеки программного обеспечения* для разработчиков. Такие библиотеки содержат общий код, который может использоваться многими приложениями. Кроме того, компоненты самой операционной системы, такие как оболочка и службы, используют функционал, предоставляемый такими библиотеками.

При взаимодействии с оборудованием ядро работает в партнерстве с драйверами устройств. *Драйвер устройства*, или просто *драйвер*, – это программа, предназначенная для взаимодействия с конкретным оборудованием. Ядро операционной системы должно работать с широким спектром аппаратных средств, поэтому вместо того, чтобы разрабатывать ядро, умеющее взаимодействовать с каждым аппаратным устройством в мире, разработчики программного обеспечения реализуют код для конкретных устройств в драйверах этих устройств. Операционные

системы обычно включают набор драйверов для распространенного оборудования, а также предоставляют механизм для установки дополнительных драйверов.

Большинство операционных систем включает в себя набор базовых приложений, таких как текстовый редактор или калькулятор, которые часто называют *утилитами*. Веб-браузер также является стандартной частью многих операционных систем. Такие утилиты и программы не являются на самом деле частью операционной системы, они представляют собой скорее просто обычные приложения, но на практике большинство ОС включает в себя подобные вещи. На рис. 10-3 представлен обобщенный вид компонентов, входящих в состав операционной системы.

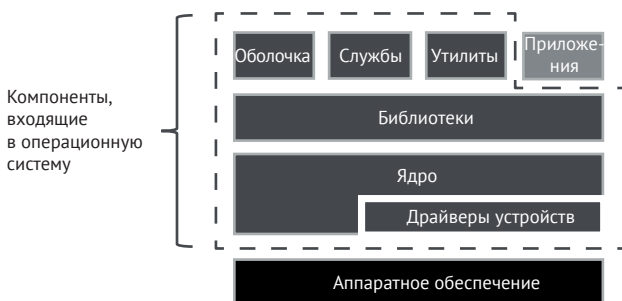


Рис. 10-3. Операционная система включает в себя множество компонентов

Как видно на рис. 10-3, в основе программного стека прямо над аппаратным обеспечением находятся ядро и драйверы устройств. Библиотеки обеспечивают функционал, на котором строятся приложения, поэтому библиотеки показаны как слой между ядром и приложениями. Оболочка, службы и утилиты также основаны на библиотеках.

## Семейства операционных систем

Сегодня существует два доминирующих семейства операционных систем: Unix-подобные операционные системы и Microsoft Windows. Как следует из названия, *Unix-подобные* операционные системы ведут себя как операционная система Unix. Linux, macOS, iOS и Android – все это примеры Unix-подобных операционных систем. *Unix* была впервые разработана в Bell Labs и ведет свою историю с 1960-х годов. Первоначально Unix работала на миникомпьютере PDP-7, но затем была перенесена на многие виды компьютеров. Изначально написанная на языке ассемблера, Unix была позже переписана на языке C, что позволило компилировать ее для различных процессоров. Сегодня она используется на серверах, а также широко представлена на персональных компьютерах и смартфонах благодаря macOS и iOS от Apple, которые основаны на Unix. Unix поддерживает работу нескольких

пользователей, многозадачность и единую иерархическую структуру каталогов. Она имеет традиционную оболочку командной строки, поддерживаемую четко определенными стандартными инструментами командной строки, которые можно использовать вместе для выполнения сложных задач.

Ядро *Linux* было первоначально разработано Линусом Торвальдсом, который задался целью создать операционную систему, похожую на Unix. Linux – это не Unix, но она определенно Unix-подобная. Она ведет себя так же, как Unix, но при этом не содержит исходный код Unix. *Дистрибутив Linux* – это операционная система, представляющая собой ядро Linux в комплекте с другим программным обеспечением. Ядро Linux является *открытым*, т. е. его исходный код находится в свободном доступе. Многие дистрибутивы Linux доступны бесплатно. Типичный дистрибутив Linux включает в себя ядро Linux и набор Unix-подобных компонентов из проекта GNU (произносится как «гну»).

GNU (рекурсивный акроним<sup>1</sup>, означающий *GNU's Not Unix*) – это проект программного обеспечения, начатый в 1980-х годах с целью создания Unix-подобной операционной системы в качестве свободного программного обеспечения. Проект GNU и Linux – это отдельные проекты, но они оказались тесно связаны между собой. Выпуск ядра Linux в 1991 году послужил толчком к попытке перенести программное обеспечение GNU в Linux. В то время у GNU не было полного ядра, а у Linux не было оболочки, библиотек и т. д. Linux предоставил ядро для запуска кода GNU, а проект GNU предоставил оболочку, библиотеки и утилиты для Linux. Таким образом, эти два проекта дополняют друг друга и вместе образуют полноценную операционную систему.

Сегодня люди обычно используют термин *Linux* для обозначения операционных систем, которые представляют собой комбинацию ядра Linux и программного обеспечения GNU. Это несколько неправильно, поскольку название всей операционной системы Linux не учитывает ту большую роль, которую программное обеспечение GNU играет во многих дистрибутивах Linux. Тем не менее в этой книге я следую общепринятой традиции называть всю ОС Linux, а не GNU/Linux или как-нибудь еще.

Сегодня Linux часто встречается на серверах и встраиваемых системах и пользуется популярностью у разработчиков программного обеспечения. Операционная система Android основана на ядре Linux, поэтому Linux широко распространен на рынке смартфонов. Raspberry Pi OS (ранее называлась Raspbian) также является дистрибу-

---

<sup>1</sup> *Рекурсия* – определение какого-либо объекта внутри этого объекта. Так, в программировании рекурсия – вызов функции (процедуры) из нее самой. *Акроним* – разновидность сокращенных наименований (аббревиатур), в которой начальные буквы образуют самостоятельное слово, читающееся слитно: например, *вуз* (произносится, как единое *вуз*) в отличие от *ФСБ* (произносится по буквам *эф-эс-бэ*). То есть *рекурсивный акроним* – аббревиатура, расшифровка которой включает и саму аббревиатуру. – *Прим. ред.*

тивом Linux, включающим программное обеспечение GNU, и мы будем использовать Raspberry Pi OS для дальнейшего изучения Linux. В целом в этой книге я буду опираться на Linux, а не на Unix, когда буду приводить примеры Unix-подобного поведения.

Microsoft Windows является доминирующей операционной системой на персональных компьютерах, включая настольные компьютеры и ноутбуки. Она также хорошо представлена в серверном пространстве (Windows Server). Уникальность Windows заключается в том, что ее происхождение не имеет отношения к Unix.

Ранние версии Windows были основаны на MS-DOS (Microsoft Disk Operating System). Несмотря на популярность на рынке домашних компьютеров, эти ранние версии Windows не были достаточно надежными, чтобы конкурировать с Unix-подобными операционными системами на рынке серверов или передовых рабочих станций.

Параллельно с разработкой Windows в 1980-х годах Microsoft сотрудничала с IBM для создания операционной системы OS/2, предполагаемого преемника MS-DOS на IBM PC. Microsoft и IBM разошлись во мнениях относительно направления развития проекта OS/2, и в 1990 году IBM взяла на себя разработку OS/2, в то время как Microsoft переключила свои усилия на другую операционную систему, которая уже находилась в разработке, Windows NT. В отличие от версий Windows на базе MS-DOS, Windows NT была основана на новом ядре. Windows NT была разработана с учетом переносимости на различное оборудование, совместимости с различными типами программного обеспечения, поддержки нескольких пользователей, а также обеспечения высокого уровня безопасности и надежности. Microsoft наняла Дейва Катлера из корпорации Digital Equipment Corporation (DEC) для руководства работой над Windows NT. Он привел с собой несколько бывших инженеров DEC, и элементы дизайна ядра NT берут свое начало в работе Дейва Катлера над операционной системой VMS в DEC.

В своих ранних выпусках Windows NT позиционировалась как версия Windows, ориентированная на бизнес, которая будет сосуществовать с версией Windows, ориентированной на потребителя. Эти две версии Windows были совершенно разными в своей основе, но они имели схожий пользовательский интерфейс и интерфейс программирования. Сходство пользовательского интерфейса означало, что пользователи, знакомые с Windows, могли легко работать в системе Windows NT. Общий интерфейс программирования позволял программам, разработанным для Windows на базе DOS, работать на Windows NT иногда без каких-либо изменений. С выпуском Windows XP в 2001 году компания Microsoft перенесла ядро NT в версию Windows, ориентированную на потребителя. С момента выпуска Windows XP все версии Windows для настольных компьютеров и серверов были построены на ядре NT.

В табл. 10-1 перечислены некоторые операционные системы и устройства, широко используемые сегодня, а также семейства операционных систем для каждой из них.

**Таблица 10-1.** Распространенные операционные системы

ОС или устройство	Семейство	Примечание
Android	Unix-подобные	Android использует ядро Linux, хотя в остальном он не очень похож на Unix. Его пользовательский интерфейс и интерфейсы прикладного программирования сильно отличаются от типичной системы Unix
iOS	Unix-подобные	iOS основана на Unix-подобной операционной системе с открытым исходным кодом Darwin. Как и у Android, пользовательский интерфейс и программный интерфейс iOS отличаются от типичной Unix-системы
macOS	Unix-подобные	macOS основана на Unix-подобной операционной системе с открытым исходным кодом Darwin
PlayStation 4	Unix-подобные	PlayStation 4 OS основана на Unix-подобном ядре FreeBSD
Raspberry Pi OS	Unix-подобные	Raspberry Pi OS – это дистрибутив Linux
Ubuntu	Unix-подобные	Ubuntu – это дистрибутив Linux
Windows 10	Windows	В Windows 10 используется ядро Windows NT
Xbox One	Windows	Xbox One имеет OS, использующую ядро Windows NT

### УПРАЖНЕНИЕ 10-1: Познакомьтесь

#### с операционными системами в вашей жизни

Выберите несколько вычислительных устройств, которыми вы владеете или пользуетесь, например ноутбук, смартфон или игровую приставку. Какая операционная система работает на каждом устройстве? К какому семейству операционных систем (Windows, Unix-подобные, др.) принадлежит каждая из них?

## Режим ядра и режим пользователя

Операционная система отвечает за то, чтобы программы, работающие на ней, вели себя как надо. Что это означает на практике? Давайте рассмотрим несколько примеров. Каждая программа не должна вмешиваться в работу других программ или ядра. Пользователи не должны иметь возможности изменять системные файлы. Приложениям не должно быть позволено напрямую обращаться к аппаратному обеспечению, все такие запросы должны проходить через ядро. Но как операционная си-

система может гарантировать, что не относящийся к ней код соответствует этим требованиям ОС? Этот вопрос решается с помощью возможностей процессора, которые предоставляют операционной системе особые права и накладывают ограничения на другой код. Это называется *уровнем привилегий* кода. Процессор может предоставлять более двух уровней привилегий, но большинство операционных систем использует только два уровня. Уровень более высоких привилегий называется *режимом ядра*, а уровень более низких привилегий – *режимом пользователя*. Режим ядра также называют *режимом супервизора*. Код, запущенный в режиме ядра, имеет полный доступ к системе, включая доступ ко всей памяти, устройствам ввода/вывода и специальным инструкциям процессора. Код, работающий в режиме пользователя, имеет ограниченный доступ. В целом ядро и многие драйверы устройств работают в режиме ядра, тогда как все остальное работает в режиме пользователя, как показано на рис. 10-4.

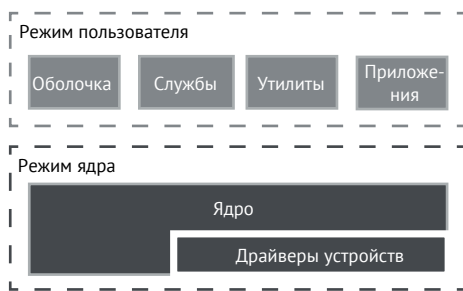


Рис. 10-4. Разделение кода, выполняемого в режиме пользователя и режиме ядра

Код, которому разрешено работать в режиме ядра, является *доверенным*, в то время как код пользовательского режима является *недоверенным*. Код, выполняемый в режиме ядра, получает полный доступ ко всему в системе, поэтому ему лучше быть доверенным! Разрешая только доверенному коду работать в режиме ядра, операционная система может гарантировать, что код пользовательского режима будет работать как надо.

## КОМПОНЕНТЫ РЕЖИМА ЯДРА В WINDOWS

Стоит отметить, что в Microsoft Windows есть еще несколько основных компонентов, которые работают в режиме ядра. В Windows основные возможности режима ядра фактически разделены между двумя компонентами: ядром и исполняющей системой (*the executive system*). Это раз-



личие уместно только при обсуждении внутренней архитектуры Windows, для большинства разработчиков и пользователей программного обеспечения это разделение не имеет значения. Фактически скомпилированный машинный код для ядра и исполняющей системы содержится в одном и том же файле (*ntoskrnl.exe*). В оставшейся части этой книги я не буду делать различий между ядром и исполняющей системой Windows NT. Помимо ядра, исполняющей системы и драйверов устройств в Windows есть и другие основные компоненты, работающие в режиме ядра. Слой аппаратных абстракций (*Hardware Abstraction Layer, HAL*) освобождает ядро, исполняющую систему и драйверы устройств от различий в низкоуровневом оборудовании, например от различий в материнских платах. Оконный интерфейс и графическая система (*win32k*) предоставляют возможности для рисования графики и программного взаимодействия с элементами пользовательского интерфейса.

## Процессы

Одной из основных функций операционной системы является предоставление платформы для выполнения программ. Как мы видели в предыдущей главе, программы – это последовательности машинных инструкций, обычно хранящиеся в исполняемом файле. Однако набор инструкций, хранящийся в файле, не может сам по себе выполнять какую-либо работу. Что-то должно загрузить инструкции файла в память и направить центральный процессор на выполнение программы, при этом необходимо обеспечить правильное поведение программы. Это и есть работа операционной системы. Когда операционная система запускает программу, она создает *процесс*, работающий экземпляр этой программы. Ранее мы рассмотрели объекты, которые работают в режиме пользователя (такие как оболочка, службы и утилиты), каждый из них выполняется в рамках процесса. Если код выполняется в пользовательском режиме, он выполняется в рамках процесса, как показано на рис. 10-5.

Процесс – это контейнер, в котором выполняется программа. Этот контейнер включает частное адресное пространство виртуальной памяти (подробнее об этом позже), копию кода программы, загруженную в память, и другую информацию о состоянии процесса. Программа может быть запущена несколько раз, и каждый запуск приводит к созданию операционной системой нового процесса. Каждый процесс имеет уникальный идентификатор (число), называемый *идентификатором процесса*, *ID процесса* или просто *PID*.



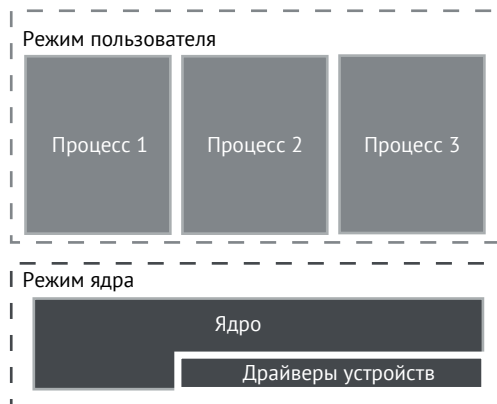


Рис. 10-5. Процессы, запущенные в пользовательском режиме

За исключением начальных процессов, запускаемых ядром, каждый процесс имеет родителя, т. е. процесс, который его запустил. Эти отношения родителя и ребенка создают дерево процессов. Если родительский процесс завершается раньше дочернего процесса, то дочерний процесс становится *процессом-сиротой*, что означает, как и следует из названия, что у него нет родителя. В Windows осиротевший дочерний процесс просто остается процессом без родителя. В Linux осиротевший процесс обычно подхватывается *начальным процессом* (*init*), первым процессом пользовательского режима, который запускается в системе Linux.

На рис. 10-6 показано дерево процессов в Raspberry Pi OS. Это представление было создано с помощью утилиты `pstree`.

На рис. 10-6 мы видим, что начальным процессом был `systemd`, он был первым запущенным процессом, и он же в свою очередь запускал другие процессы. Дочерние потоки выполнения показаны фигурными скобками (подробнее о потоках мы поговорим позже). Чтобы сгенерировать этот вывод, я выполнил команду `pstree` из командной строки, и в выводе видно, что сам `pstree` запущен, как и следовало ожидать. Он является дочерним компонентом `bash` (оболочки), который в свою очередь является дочерним компонентом `sshd`. Другими словами, из этого вывода можно понять, что я запустил `pstree` из оболочки Bash, которая была открыта в удаленном сеансе Secure Shell (SSH).

Чтобы увидеть дерево процессов на компьютере под управлением Windows, я рекомендую использовать инструмент Process Explorer, который можно загрузить с сайта Microsoft. Это приложение с графическим интерфейсом, которое даст вам богатый обзор процессов, запущенных на вашем компьютере.

#### ПРИМЕЧАНИЕ

Смотрите проект № 20 на стр. 252, где вы сможете просмотреть запущенные процессы на вашем устройстве.

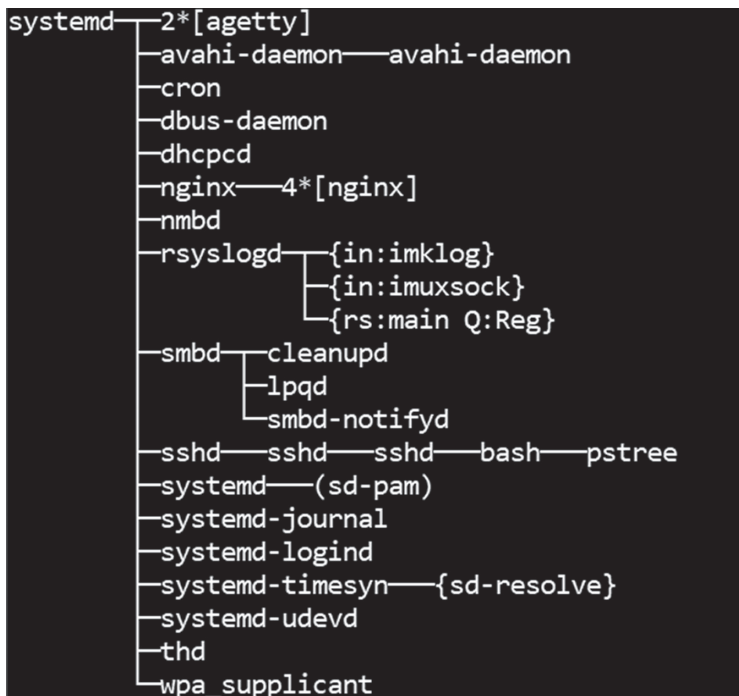


Рис. 10-6. Пример дерева процессов Linux, представленный pstree

## Потоки

По умолчанию программа выполняет инструкции последовательно, решая одну задачу за раз. Но что, если программе необходимо выполнить две или более задач параллельно? Например, допустим, программе нужно выполнить какой-то длительный расчет и одновременно обновить пользовательский интерфейс, возможно, чтобы показать индикатор выполнения. Если программа полностью последовательная, то, как только она начинает вычисления, пользовательский интерфейс игнорируется, поскольку время процессора, выделенное на программу, должно быть использовано в другом месте. Но нам хотелось бы, чтобы пользовательский интерфейс обновлялся, пока выполняется расчет, т. е. имеются две отдельные задачи, которые должны выполняться параллельно. Операционные системы предоставляют такую возможность с помощью *потоков выполнения*, или просто *потоков (threads)*. Поток – это планируемая единица выполнения программы в рамках процесса. Поток выполняется внутри процесса и может выполнять любой программный код, загруженный в этот процесс.

Выполняемый потоком код обычно решает определенную задачу, которую программа должна выполнить своевременно. Поскольку потоки принадлежат процессу, они разделяют адресное пространство, код и другие ресурсы со всеми остальными потоками в этом процессе. Процесс

начинается с одного потока и может создавать другие потоки по мере необходимости, когда работа должна выполняться параллельно. Каждый поток имеет идентификатор, называемый *идентификатором потока*, или *TID*. Ядро также создает потоки для управления своей работой. На рис. 10-7 показаны отношения между потоками, процессами и ядром.

В Windows потоки и процессы – это разные типы объектов. Объект процесса является контейнером, а потоки принадлежат процессу. В Linux это различие более тонкое. Ядро Linux представляет процессы и потоки с помощью одного типа данных, который служит и процессом, и потоком. В Linux группа потоков, разделяющих адресное пространство и имеющих общий идентификатор процесса, считается процессом. Отдельного типа процесса не существует.

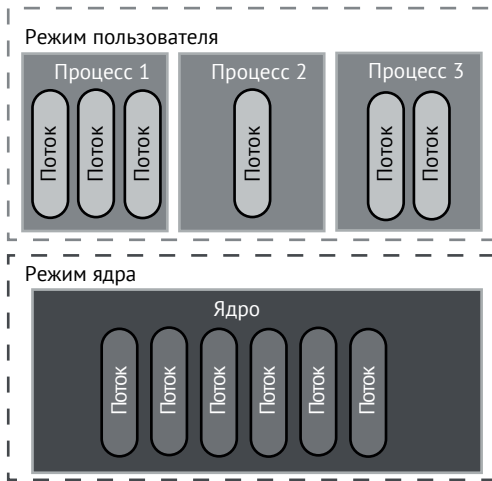


Рис. 10-7. Потоки принадлежат процессам пользовательского режима или ядру

Терминология Linux, используемая для обозначения идентификаторов процессов и потоков, может показаться немного путанной. В пользовательском режиме процесс имеет идентификатор процесса (PID), а поток – идентификатор потока (TID). Это так же, как и в Windows. Однако ядро Linux называет идентификатор потока PID, а идентификатор процесса – *идентификатором группы потоков (TGID)*!

#### ПРИМЕЧАНИЕ

Обратитесь к проекту № 21 на стр. 275, где вы сможете создать свой собственный поток.

Что на самом деле означает параллельная работа нескольких потоков? Допустим, на вашем компьютере запущено 10 процессов, и каждый процесс имеет 4 потока. Это 40 потоков только в пользовательском режиме! Мы говорим, что потоки работают параллельно, но действительно ли все 40 потоков могут выполняться одновременно? Нет, если только ваш компьютер не имеет 40 процессорных ядер, а это, скорее всего, не так. Каждое ядро процессора может выполнять только один

поток за раз, поэтому количество ядер в устройстве определяет, сколько потоков может выполняться одновременно.

## ФИЗИЧЕСКИЕ И ЛОГИЧЕСКИЕ ЯДРА

Не все ядра одинаково способны к параллелизму. *Физическое ядро* – это аппаратная реализация ядра в процессоре. *Логические ядра* представляют собой способность одного физического ядра выполнять несколько потоков одновременно (один поток на одно логическое ядро). Intel называет эту способность *гиперпоточностью*. Например, компьютер, на котором я пишу эту книгу, имеет два физических ядра, каждое из которых имеет два логических ядра, в общей сложности четыре логических ядра. Это означает, что мой компьютер может выполнять четыре потока одновременно, хотя логические ядра не могут достичь полного параллелизма физических ядер.

Итак, что произойдет, когда у нас есть 40 потоков, которые необходимо запустить, но только 4 ядра? Операционная система задействует *планировщик* – программный компонент, который отвечает за то, чтобы каждый поток получил свою очередь на выполнение. В разных операционных системах используются разные подходы к реализации планирования, но основная цель одна: дать потокам время на выполнение. Поток получает короткий период времени для выполнения (называемый *квант*), затем его выполнение приостанавливается, чтобы дать возможность выполнить другой поток. Позже выполнение первого потока запланировано снова, и он продолжит работу с того места, на котором остановился. Это в основном скрыто от кода потока и от разработчика, написавшего приложение. С точки зрения кода потока он работает непрерывно, и разработчики пишут свои многопоточные приложения так, как будто все их потоки работают непрерывно и параллельно<sup>1</sup>.

<sup>1</sup> Читатель должен понимать, что многоядерные процессоры возникли значительно позже многозадачных ОС (например, настольные ПК начали получать двух- и четырехъядерные процессоры только во второй половине 2000-х, после четвертьвекового победного шествия персональных компьютеров по миру). До этого все многозадачные ОС на ПК выполняли все необходимое множество процессов и потоков на единственном ядре с разделением времени, как описывает здесь автор. Причем в современных ОС одновременно обычно бывает запущено очень много процессов (как свидетельствует Диспетчер задач, одновременно с набором этого текста Windows выполняет более 20 процессов). И, хотя многоядерность заметно ускоряет их выполнение, с практической точки зрения уже количество ядер более 4–6 создает такие трудности в программировании с корректным разделением задач между ядрами, что далее повышать количество физических ядер целесообразно только в специальных случаях (например, в высокопроизводительных многопроцессорных системах, программы к которым вручную подгоняются под конкретную конфигурацию). По этой причине многозадачные ОС и сегодня можно рассматривать, как ОС, в которых между задачами в первую очередь делится время работы процессора, и лишь во вторую – физические и логические ядра.

## Виртуальная память

Операционные системы поддерживают несколько запущенных процессов, каждый из которых должен использовать память. Чаще всего одному процессу не нужно взаимодействовать с памятью другого процесса, и, вообще-то, это нежелательно. Мы не хотим, чтобы неправильно ведущий себя процесс крал данные или перезаписывал данные другого процесса или, что еще хуже, ядра. Кроме того, разработчики не хотят, чтобы адресное пространство их процесса становилось фрагментированным из-за использования памяти другими процессами. По этим причинам операционные системы не предоставляют процессам пользовательского режима доступ к физической памяти, и вместо этого каждому процессу предоставляется *виртуальная память* – абстракция, которая дает каждому процессу свое собственное большое частное адресное пространство.

В главе 7 мы рассмотрели адресацию памяти, при которой каждому физическому байту в аппаратном обеспечении присваивается адрес. Такие адреса аппаратной памяти называются *физическими адресами*. Эти адреса обычно скрыты от процессов режима пользователя. Вместо этого операционные системы предоставляют процессам *виртуальную память*, где каждый адрес является *виртуальным адресом*. Каждому процессу предоставляется собственное пространство виртуальной памяти для работы. Для отдельного процесса память представляется как большой диапазон адресов. Когда процесс записывает в определенный виртуальный адрес, этот адрес не относится непосредственно к области аппаратной памяти. При необходимости виртуальный адрес преобразуется в физический, как показано на рис. 10-8, но детали этого преобразования скрыты от процесса.

Преимущество этого подхода заключается в том, что каждому процессу предоставляется большой частный диапазон адресов виртуальной памяти, с которым он может работать. В общем случае каждому процессу в системе предоставляется один и тот же диапазон адресов памяти. Например, каждому процессу может быть предоставлено 2 ГБ виртуального адресного пространства, от адреса 0x00000000 до 0x7FFFFFFF. Это может показаться проблематичным. Что произойдет, если две программы попытаются использовать один и тот же адрес памяти? Может ли одна программа перезаписать или прочесть данные другой программы? Благодаря виртуальной адресации это не проблема.

Один и тот же виртуальный адрес для нескольких программ соотносится с разными физическими адресами, поэтому вероятность случайного доступа одной программы к данным другой в памяти исключена.

Это означает, что данные, хранящиеся по определенному виртуальному адресу, различаются в разных процессах. Виртуальные адреса могут быть одинаковыми, но хранящиеся там данные – разные. Тем не менее существуют механизмы, позволяющие программам совместно использовать память, если это необходимо. В более старых операционных системах пространство памяти не было разделено так явно, что давало программам широкие возможности для повреждения памя-

ти в других программах или даже в операционной системе. К счастью, все современные операционные системы обеспечивают разделение памяти между процессами.

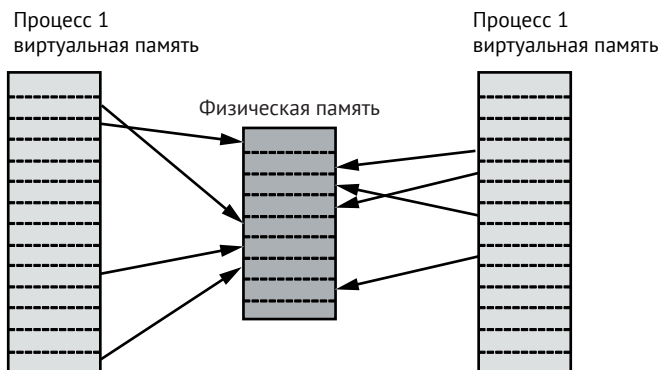


Рис. 10-8. Виртуальное адресное пространство для каждого процесса отображается в физическую память

Важно понимать, что, хотя диапазон адресов процесса может иметь размер 2 Гб (например), это не означает, что все 2 Гб виртуальной памяти немедленно доступны для использования процессом. Только некое подмножество этих адресов обеспечено физической памятью. Вспомните проекты, которые вы выполняли в главах 8 и 9, на самом деле вы рассматривали тогда адреса виртуальной памяти, а не физической.

Ядро имеет отдельное виртуальное адресное пространство для работы, его диапазон адресов отличается от диапазона адресов, назначенных процессам пользовательского режима. В отличие от адресного пространства пользовательского режима адресное пространство ядра является общим для всего кода, работающего в режиме ядра. Это означает, что любой код, запущенный в режиме ядра, имеет доступ ко всему в адресном пространстве ядра. Это также дает такому коду возможность изменять содержимое любой ячейки памяти ядра. Что подкрепляет идею о том, что код, работающий в режиме ядра, должен быть доверенным!

Как же разделено виртуальное адресное пространство между пользовательским режимом и режимом ядра? Давайте рассмотрим 32-битные операционные системы. Как обсуждалось в главе 7, для 32-битной системы адреса памяти представлены в виде 32-битных чисел, что означает 4 Гб адресного пространства в целом. Диапазон адресов этого адресного пространства должен быть разделен между режимом ядра и пользовательским режимом. Для адресного пространства в 4 Гб как Windows, так и Linux позволяют разделить его либо пополам, т. е. по 2 Гб пользовательскому режиму и ядру, либо отдать 3 Гб пользовательскому режиму и 1 Гб ядру, в зависимости от конфигурационных настроек. На рис. 10-9 показано равномерное разделение виртуальной памяти по 2 Гб.

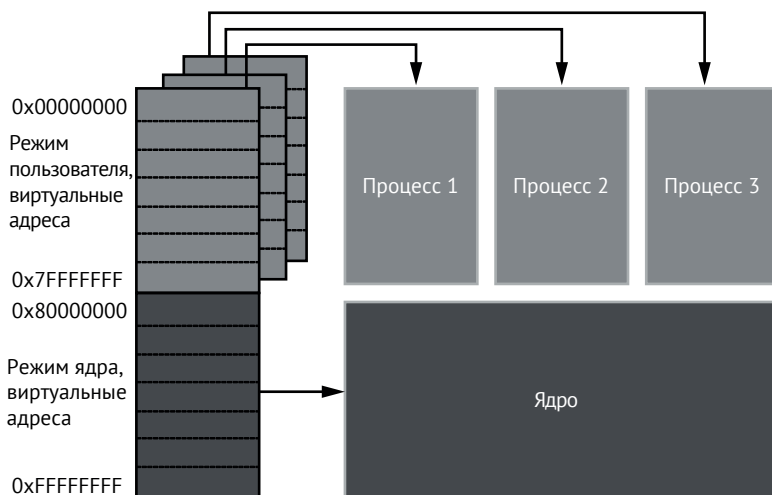


Рис. 10-9. Виртуальное адресное пространство в 32-битной системе с равномерным разделением 2/2 ГБ

Помните, что здесь мы рассматриваем только *виртуальные* адреса. 32-разрядная система имеет 4ГБ виртуального адресного пространства независимо от того, сколько у нее *физической* памяти. Допустим, компьютер имеет только 1 ГБ оперативной памяти, он все равно будет иметь 4 ГБ виртуального адресного пространства под 32-битной ОС. Напомним, что диапазон виртуальных адресов не является отображением физической памяти, он только представляет собой диапазон, в который *может быть* отображена физическая память. Это означает, что ядро и все запущенные процессы могут запросить больше байт виртуальной памяти, чем общий объем оперативной памяти. В такой ситуации операционная система может переместить байты памяти во вторичное хранилище, чтобы освободить место в оперативной памяти для вновь запрашиваемой памяти, этот процесс известен как *подкачка страниц (paging)*. Как правило, сначала перемещается наименее используемая память, чтобы активно используемая память оставалась в оперативной. Когда сохраненная память понадобится, ОС должна загрузить ее обратно в оперативную память. Подкачка позволяет увеличить объем виртуальной памяти за счет снижения производительности при перемещении байтов во вторичное хранилище и обратно. Имейте в виду, что вторичное хранилище<sup>1</sup> работает значительно медленнее, чем оперативная память.

#### ПРИМЕЧАНИЕ

Обратитесь к проекту №22 на стр. 277, где вы сможете изучить виртуальную память.

С появлением 64-битных процессоров и операционных систем появилась возможность создания гораздо больших адресных пространств.

<sup>1</sup> Обычно это жесткий диск. – Прим. ред.

Если бы мы представляли адреса памяти полными 64 битами, виртуальное адресное пространство было бы примерно в 4 млрд раз больше 32-битного адресного пространства! Однако сегодня такое большое адресное пространство не нужно, поэтому 64-битные операционные системы используют меньшее количество битов для представления адресов. Различные 64-битные операционные системы на разных процессорах используют разное количество битов для представления адреса. И 64-битная ОС Linux, и 64-битная ОС Windows поддерживают 48-битные адреса, что соответствует 256 ТБ виртуального адресного пространства. Это примерно в 65 000 раз больше 32-битного адресного пространства. Такой объем предоставляет более чем достаточно места для типичных современных приложений.

## Интерфейс прикладного программирования (API)

Когда большинство людей думают об операционной системе, они думают о пользовательском интерфейсе, оболочке (*shell*). Оболочка – это то, что люди видят, и она определяет восприятие системы. Например, пользователь Windows обычно думает о Windows как о панели задач, меню **Пуск**, рабочем столе и т. д. Однако на самом деле пользовательский интерфейс – это лишь небольшая часть кода операционной системы, и это всего лишь интерфейс, место встречи системы и пользователя. С точки зрения приложения (или разработчика программного обеспечения) взаимодействие с операционной системой определяется не пользовательским интерфейсом, а *интерфейсом прикладного программирования (API)* операционной системы. API предназначены не только для операционных систем, любое программное обеспечение, которое хочет предоставить программные средства взаимодействия, может предоставить API, но наше внимание здесь сосредоточено именно на API ОС.

API ОС – это спецификация, определенная в исходном коде и описанная в документации, подробно поясняющей, как программа должна взаимодействовать с ОС. Типичный API ОС включает список функций (включая их имена, входы и выходы) и структур данных, необходимых для взаимодействия с операционной системой. Программные библиотеки, входящие в состав операционной системы, обеспечивают реализацию спецификации API. Разработчики программного обеспечения говорят о «вызове» или «использовании» API, когда хотят коротко сказать, что их код обращается к одной из функций, указанных в API (и реализованных в программной библиотеке).

Точно так же, как пользовательский интерфейс определяет «индивидуальность» ОС для пользователей, API определяет индивидуальность ОС для приложений. На рис. 10-10 показано, как пользователи и приложения взаимодействуют с операционной системой.



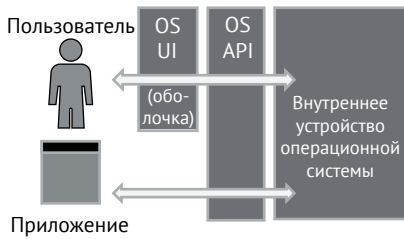


Рис. 10-10. Интерфейсы операционной системы: пользовательский интерфейс для пользователей, API для приложений

Как показано на рис. 10-10, пользователи взаимодействуют с пользовательским интерфейсом операционной системы, также известным как оболочка. Оболочка переводит команды пользователя в вызовы API. Затем API вызывает внутренний код операционной системы для выполнения запрошенного действия. Приложениям не нужно проходить через пользовательский интерфейс, они просто вызывают API напрямую. С этой точки зрения оболочка взаимодействует с API операционной системы так же, как и любое другое приложение.

Давайте рассмотрим пример взаимодействия с операционной системой через API. Создание файла является обычной возможностью операционных систем, это необходимо делать как пользователям, так и приложениям. Графические оболочки и оболочки командной строки предоставляют пользователям простые способы создания файлов. Однако приложению не обязательно обращаться к графическому интерфейсу или к командной строке, чтобы создать файл. Давайте рассмотрим, как приложение может программно создать файл.

В системах Unix или Linux для создания файла можно использовать функцию API под названием `open`. Следующий пример на языке C использует функцию `open` для создания нового файла `hello.txt`. Флаг `O_WRONLY` указывает на операцию только для записи, а `O_CREAT` указывает на то, что файл должен быть создан.

---

```
open("hello.txt", O_WRONLY|O_CREAT);
```

---

То же самое можно сделать в Windows с помощью API-функции `CreateFileA`:

---

```
CreateFileA("hello.txt", GENERIC_WRITE, 0, NULL,  
    CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
```

---

В обоих этих примерах используется язык программирования C. Операционные системы обычно пишутся на языке C, поэтому их API, как правило, естественным образом подходят для использования в программах на языке C. Для программ, написанных на других языках, API ОС все равно должен вызываться при выполнении программы, но язык

программирования оборачивает этот вызов API в свой собственный синтаксис, скрывая детали API от разработчика. Это позволяет создавать код, переносимый между операционными системами. Даже язык C делает это, предоставляя стандартную библиотеку функций, которые работают в любой операционной системе. Эти функции в свою очередь при выполнении должны делать вызов API для конкретной ОС. Рассмотрим еще раз пример создания файла. На языке C мы можем вместо указанной ранее функции API использовать функцию `fopen`, как показано в следующем коде. Эта функция является частью стандартной библиотеки языка C и работает в любой операционной системе.

---

```
fopen("hello.txt", "w");
```

---

В качестве другого примера можем использовать следующий код Python для создания нового файла. Этот код работает в любой ОС, где установлен интерпретатор Python. Интерпретатор Python позаботится о вызове соответствующего API ОС от имени приложения.

---

```
open('hello.txt', 'w')
```

---

Для Unix-подобных операционных систем API несколько различается в зависимости от конкретной разновидности Unix или Linux и версии ядра. Однако большинство Unix-подобных операционных систем полностью или частично соответствует стандартной спецификации. Этот стандарт известен как *переносимый интерфейс операционных систем (Portable Operating System Interface, POSIX)*, и он обеспечивает стандарт не только для API ОС, но и для поведения оболочки и входящих в нее утилит. POSIX обеспечивает базовый уровень для Unix-подобных операционных систем, но современные Unix-подобные ОС часто имеют свой собственный API. *Cocoa* – это API Apple для macOS, и существует аналогичный API для iOS, известный как *Cocoa Touch*. Android также имеет свой собственный набор интерфейсов программирования, известный как *API платформы Android*.

Другое важное семейство операционных систем, Windows, имеет свой собственный API. Со временем *API Windows* вырос и расширился. Первоначальная версия Windows API была 16-битной, известной сейчас как *Win16*.

Когда в 1990-х годах Windows была обновлена до 32-разрядной операционной системы, была выпущена 32-разрядная версия API, *Win32*. Теперь, когда Windows стала 64-разрядной операционной системой, появился соответствующий API *Win64*. Microsoft также представила новый API в Windows 10, *Universal Windows Platform (UWP)*, с целью сделать разработку приложений единообразной для различных типов устройств, работающих на Windows.

#### **ПРИМЕЧАНИЕ**

Обратитесь к проекту № 23 на стр. 280, где вы сможете попробовать взаимодействовать с API операционной системы Linux.

## Пользовательский режим и системные вызовы

Как упоминалось ранее, код, выполняемый в режиме пользователя, имеет ограниченный доступ к системе. Итак, что же *может* делать код пользовательского режима? Он может читать и записывать в свою собственную виртуальную память, выполнять математические и логические операции. Он может управлять программным потоком своего собственного кода. С другой стороны, код, работающий в пользовательском режиме, *не может* обращаться к адресам физической памяти, включая адреса, используемые для ввода/вывода с привязкой к памяти. Это означает, что он не может сам по себе печатать текст в консольном окне, получать ввод с клавиатуры, рисовать графику на экране, воспроизводить звук, получать ввод с сенсорного экрана, общаться по сети или читать файл с жесткого диска! Мне нравится говорить, что «код пользовательского режима работает в «пузыре»<sup>1</sup>» (рис. 10-11). Он не может взаимодействовать с внешним миром, по крайней мере, без посторонней помощи. Это означает также, что код пользовательского режима не может напрямую выполнять операции ввода/вывода. Практический эффект этого заключается в том, что код, работающий в режиме пользователя, может выполнять полезную работу, но он не может поделиться результатами этой работы без посторонней помощи.



Рис. 10-11. Процесс работает в «пузыре» пользовательского режима. Он может выполнять математические вычисления, логические операции, обращаться к виртуальной памяти и управлять потоком программы, но не может напрямую взаимодействовать с внешним миром

Вы можете задаться вопросом, как же тогда приложения пользовательского режима взаимодействуют с пользователями? Конечно, приложения каким-то образом могут взаимодействовать с внешним миром,

<sup>1</sup> Придуманый автором метафорический термин «пузырь» (т. е. помещение пользовательского приложения в операционную среду, изолирующую его от системного кода и аппаратуры) имеет официальное название *кольца защиты* или, проще, *уровни безопасности*. В современных ОС таких «колец» может быть много, но обычно, как поступает и автор, все их многообразие сводят к двум уровням: уровню ядра и уровню пользовательских приложений. С уровнями безопасности тесно связаны понятия «прав» и «привилегий» — у системного кода их намного больше, чем у приложений, что и составляет суть устройства авторского «пузыря». — Прим. ред.

но как это происходит? Ответ заключается в том, что у кода пользовательского режима есть еще одна важная возможность: он может попросить код режима ядра выполнить работу от его имени.

Когда код режима пользователя просит код режима ядра выполнить привилегированную операцию от его имени, это называется *системным вызовом*, что показано на рис. 10-12.

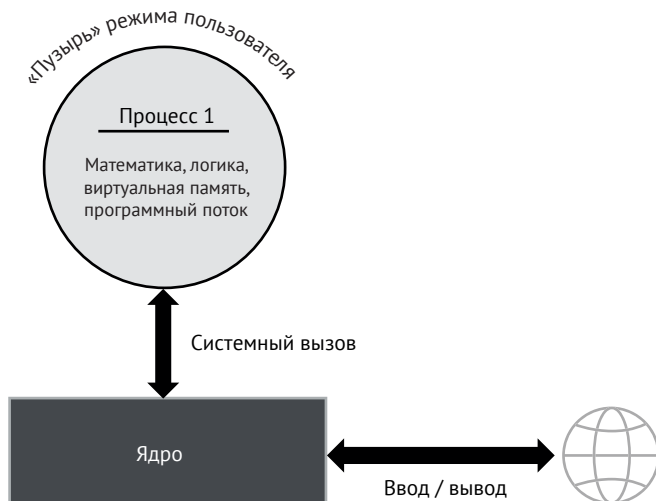


Рис. 10-12. Процесс пользовательского режима может взаимодействовать с внешним миром с помощью ядра, выполнив системный вызов

Например, если код пользовательского режима нуждается в чтении из файла, он выполняет системный вызов, чтобы запросить ядро прочитать определенные байты из определенного файла. Ядро, работая совместно с драйвером устройства хранения, выполняет необходимые операции ввода/вывода для чтения файла, а затем предоставляет запрошенные данные обратно процессу пользовательского режима. Это проиллюстрировано на рис. 10-13.



Рис. 10-13. Ядро выступает в качестве посредника для кода пользовательского режима, которому необходим доступ к аппаратным ресурсам, таким как вторичное хранилище

Коду пользовательского режима не нужно знать ничего о физическом устройстве хранения или связанных с ним драйверах устройств. Ядро предоставляет абстракцию, скрывая детали и позволяя коду пользовательского режима просто выполнять действия. Примеры функций API `open` и `CreateFileA`, которые мы рассматривали ранее, работают именно таким образом, используя системные вызовы для запроса привилегированных операций.

Конечно, существуют ограничения на то, что ядро позволяет делать. Например, процесс в режиме пользователя не может прочитать файл, к которому у него нет доступа.

В процессорах предусмотрены инструкции, специально предназначенные для осуществления системных вызовов. В процессорах ARM используется инструкция `SVC` (ранее `SWI`), которая называется *вызовом супервизора*. В процессорах x86 для этой цели предусмотрены инструкции `SYSCALL` и `SYSENTER`. И Linux, и Windows реализуют большое количество системных вызовов, и каждый вызов идентифицируется уникальным номером. Например, в Linux для ARM системный вызов `write` (запись в файл) имеет номер 4. Чтобы выполнить системный вызов, программа должна загрузить определенный регистр процессора нужным номером системного вызова, поместить дополнительные параметры в другие определенные регистры, а затем выполнить инструкцию системного вызова.

Хотя разработчики программного обеспечения могут выполнять системные вызовы непосредственно в машинном коде или на языке ассемблера, к счастью, в большинстве случаев это не требуется. Операционные системы и высокоуровневые языки программирования предоставляют возможности для выполнения системных вызовов естественным для программистов способом, обычно через API ОС или стандартную библиотеку языка. Программисты просто пишут код для выполнения действия и могут даже не подозревать, что за кулисами выполняется системный вызов.

#### ПРИМЕЧАНИЕ

*Смотрите проект № 24 на стр. 283, где вы сможете наблюдать системные вызовы, выполняемые программами.*

## API и системные вызовы

Ранее мы рассмотрели тему API операционной системы и только что рассмотрели системные вызовы. Чем же API ОС отличается от системного вызова? Эти два понятия связаны, но они не эквивалентны. Системные вызовы определяют механизм, с помощью которого код пользовательского режима запрашивает услуги режима ядра. API описывает способ взаимодействия приложений с операционной системой независимо от того, вызывается ли код режима ядра. Некоторые функции API выполняют системные вызовы, в то время как другие функции API не требуют системного вызова. Специфика этого зависит от операционной системы.

Давайте сначала рассмотрим Linux. Если мы ограничим наше определение Linux ядром, то можем сказать, что Linux API – это фактически спецификация для использования системных вызовов Linux, поскольку системные вызовы – это программный интерфейс к ядру. Однако операционные системы на базе Linux – это нечто большее, чем ядро. Например, рассмотрим Android, который использует ядро Linux. Android имеет свой собственный набор программных интерфейсов, Android Platform APIs.

В случае Microsoft Windows ядро Windows NT предоставляет набор системных вызовов, доступных через интерфейс, известный как Native API. Разработчики приложений редко используют Native API напрямую, он предназначен для использования компонентами операционной системы. Вместо этого разработчики используют Windows API, который действует как обертка вокруг Native API. Однако не все функции Windows API требуют системного вызова. Давайте рассмотрим несколько примеров из Windows API.

Функция Windows API `CreateFileW` создает или открывает файл. Она представляет собой обертку для Native API `NtCreateFile`, которая выполняет системный вызов ядра. В отличие от этого функция Windows API `PathFindFileNameW` (которая находит имя файла в каталогах) не взаимодействует с Native API и не выполняет никаких системных вызовов. Для создания файла требуется помощь ядра, в то время как для поиска имени файла требуется только доступ к виртуальной памяти, что может происходить в пользовательском режиме.

Итак, API операционной системы описывает программный интерфейс ОС. Системные вызовы обеспечивают механизм, с помощью которого код пользовательского режима запрашивает привилегированные операции режима ядра. Некоторые функции API зависят от системных вызовов, в то время как другие – нет.

## Программные библиотеки операционной системы

Как упоминалось ранее, API операционной системы описывает программный интерфейс операционной системы. Хотя техническое описание интерфейса полезно для программиста, но, когда программа выполняется, ей необходим конкретный метод вызова API. Это осуществляется с помощью программных библиотек. *Программная библиотека операционной системы* представляет собой набор кода, входящего в состав ОС, который обеспечивает реализацию API ОС. То есть библиотека содержит код, выполняющий операции, описанные в спецификации API. В главе 9 мы говорили о библиотеках, доступных для языков программирования: о стандартной библиотеке языка и о дополнительных библиотеках, поддерживаемых сообществом разработчиков, работающих на этом языке. Программные библиотеки, которые мы обсуждаем здесь, похожи на перечисленные. Разница лишь в том, что эти библиотеки являются частью операционных систем.

Библиотека ОС похожа на исполняемую программу, так как это файл, содержащий байты машинного кода. Однако она обычно не имеет точки входа и поэтому не может запускаться самостоятельно. Вместо этого библиотека *экспортирует* (делает доступным) набор функций, которые могут быть использованы программами. Программа, использующая программную библиотеку, *импортирует* функции из этой библиотеки и, как говорят, *связывается* с ней.

Операционные системы включают набор библиотечных файлов, которые экспортируют различные функции, определенные API. Некоторые из этих функций являются просто обертками, которые сразу же выполняют системный вызов ядра. Другие функции полностью реализуются в коде пользовательского режима, содержащемся в самом файле библиотеки. Другие же занимают промежуточное положение, реализуя некоторую логику в пользовательском режиме и одновременно выполняя один или несколько системных вызовов, как показано на рис. 10-14.

В типичном дистрибутиве Linux многие системные вызовы ядра Linux доступны через библиотеку *GNU C Library* (или *glibc*). Эта библиотека также включает стандартную библиотеку языка программирования C, в том числе функции, не требующие системных вызовов. Основным файлом *glibc* обычно имеет название *libc.so.6*, где *so* означает *shared object* (*общий объект*), а *6* указывает на версию. Используя эту библиотеку, разработчик программного обеспечения, работающий на C или C++, может легко использовать возможности, предоставляемые ядром Linux и библиотекой среды выполнения C.

Учитывая повсеместное распространение этой библиотеки в большинстве дистрибутивов Linux, разумно рассматривать функции в *glibc* как часть стандартного API Linux.

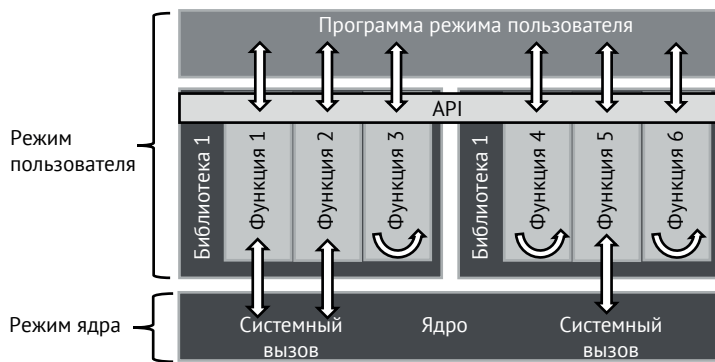


Рис. 10-14. API операционной системы реализован в виде набора библиотек. Некоторые функции в этих библиотеках выполняют системные вызовы ядра, другие – нет. Программы пользовательского режима взаимодействуют с API

#### ПРИМЕЧАНИЕ

Обратитесь к проекту № 25 на стр. 284, где вы сможете протестировать библиотеку *GNU C Library*.



API Microsoft Windows довольно обширен, за прошедшие годы в него вошло множество библиотек. Три основных библиотечных файла Windows API – это *kernel32.dll*, *user32.dll* и *gdi32.dll*. Системные вызовы, экспортируемые из ядра NT, доступны программам пользовательского режима через *kernel32.dll*. Системные вызовы, экспортируемые из *win32k* (оконной и графической системы), доступны программам пользовательского режима через *user32.dll* и *gdi32.dll*.

Расширение *dll* в этих файлах указывает на то, что это *динамически подключаемые библиотеки* (*Dynamic Link Library*), что соответствует файлам общих объектов (*.so*) в Linux. То есть расширение *dll* указывает на то, что файл содержит код общей библиотеки, который процесс может загрузить и запустить. Суффикс *32* в имени файла был добавлен при переходе от 16-битной к 32-битной версии Windows. Сегодня 64-разрядные версии Windows все еще сохраняют суффикс *32* для этих файлов по причине совместимости. Фактически 64-битные версии Windows включают две версии этих файлов (одно и то же имя, разные каталоги), одну для 32-битных приложений и одну – для 64-битных.

#### ПРИМЕЧАНИЕ

*Программа может вызывать системные вызовы, не обращаясь к программной библиотеке. Установив значения в регистрах процессора и выдав специфическую для процессора инструкцию, такую как SVC на ARM или SYSCALL на x86, программа может напрямую выполнить системный вызов. Однако это требует программирования на языке ассемблера, что приводит к исходному коду, который не будет работать в разных архитектурах процессоров. Кроме того, API операционной системы может включать функции, которые реализуются без помощи системного вызова, поэтому прямые системные вызовы не заменяют программные библиотеки операционной системы.*

## ПОДСИСТЕМА WINDOWS ДЛЯ LINUX

Ядро Linux и ядро Windows NT используют разные системные вызовы, а их исполняемые файлы хранятся в разных форматах, что делает программы, скомпилированные для одной ОС, несовместимыми с другой ОС. Однако в 2016 году Microsoft анонсировала *Windows Subsystem for Linux (WSL)*, слой в Windows 10, который позволяет многим 64-битным программам Linux работать без модификации в Windows. В первой версии WSL это достигалось путем перехвата системных вызовов, выполняемых исполняемыми файлами Linux, и их обработки в ядре NT. Вторая версия WSL полагается на настоящее ядро Linux для обработки системных вызовов. Это ядро Linux работает в виртуальной машине вместе с ядром NT. Подробнее о виртуальных машинах мы поговорим в главе 13.



## Двоичный интерфейс приложений

Теперь, когда мы рассмотрели понятие интерфейса прикладного программирования (API) и то, как он связан с системными вызовами и библиотеками, давайте рассмотрим родственное понятие, ABI. *Двоичный интерфейс приложения* (*application binary interface, ABI*) определяет интерфейс машинного кода к программной библиотеке в отличие от API, который определяет интерфейс исходного кода. В целом API является единым для различных семейств процессоров, в то время как ABI различается по семействам процессоров. Разработчик может написать код, использующий API операционной системы, а затем скомпилировать его для нескольких типов процессоров. Исходный код ориентирован на общий API, в то время как скомпилированный код ориентирован на ABI, специфичный для конкретной архитектуры.

После компиляции полученный машинный код соответствует ABI для целевой архитектуры. Это означает, что во время выполнения именно ABI, а не API, определяет взаимодействие между скомпилированными программами и программными библиотеками. Важно, чтобы ABI, предоставляемый библиотеками ОС, оставался неизменным с течением времени. Эта неизменность позволяет старым программам продолжать работать на новых версиях операционной системы без необходимости перекompиляции.

## Драйверы устройств

Современные компьютеры поддерживают широкий спектр аппаратных устройств, таких как дисплеи, клавиатуры, камеры и т. д. Каждое из этих устройств реализует интерфейс ввода/вывода, позволяющий устройству взаимодействовать с остальной частью системы. Различные типы устройств используют разные подходы для ввода/вывода, так адаптеру Wi-Fi требуется совсем не то, что игровому контроллеру. Даже устройства одного общего типа могут использовать различные подходы к вводу/выводу. Например, две разные модели видеокарт могут совершенно по-разному взаимодействовать с остальной системой. Прямое взаимодействие с аппаратным обеспечением ограничено кодом, выполняемым в режиме ядра, но не стоит ожидать, что ядро операционной системы будет знать, как взаимодействовать с каждым устройством.

В этом случае на помощь приходят драйверы устройств. *Драйвером устройства* называют программу, которая взаимодействует с аппаратной частью этого устройства и обеспечивает программный интерфейс с ним.

Обычно драйвер устройства реализуется как *модуль ядра*, т. е. файл, содержащий код, который ядро может загрузить и выполнить в режиме ядра. Это необходимо для предоставления драйверам доступа к аппаратному обеспечению. Из-за этого драйверы устройств имеют высокий уровень доступа, такой же, как и у самого ядра, поэтому устанавливать следует только доверенные драйверы. Ядро работает совместно с драйверами

устройств для взаимодействия с оборудованием от имени программ, работающих в пользовательском режиме. Это позволяет использовать любое оборудование без знания операционной системой или приложениями подробностей работы с конкретной аппаратурой. Это одна из форм применения принципа «черного ящика». В некоторых случаях драйверы могут работать в режиме пользователя (например, драйверы, использующие Microsoft User-Mode Driver Framework), но такой подход все равно требует наличия компонента в режиме ядра для обработки аппаратных взаимодействий, обычно предоставляемого операционной системой.

#### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 26 на стр. 287, где представлены загруженные модули ядра, включая драйверы устройств, в Raspberry Pi OS.*

## Файловые системы

Почти в каждом компьютере есть некое вторичное устройство хранения данных, обычно жесткий диск (HDD) или твердотельный накопитель (SSD). Такие устройства фактически являются хранилищами битов, которые можно читать и записывать и где данные сохраняются даже при выключении питания системы. Устройства хранения данных разделены на области, называемые *разделами*. Операционные системы используют *файловые системы* для организации данных на устройствах хранения в файлы и каталоги. Для того чтобы раздел мог использоваться операционной системой, он должен быть *отформатирован* в соответствии с определенной файловой системой. В разных операционных системах используются разные файловые системы. В Linux обычно используется семейство файловых систем ext (extended) (ext2, ext3, ext4), а в Windows – FAT (File Allocation Table) и NTFS (NT File System). Некоторые операционные системы представляют хранилище в виде *тома (volume)* – логической абстракции, построенной на одном или нескольких разделах. В такой системе файловые системы располагаются в томе, а не в разделе.

*Файл* – это хранилище данных, а *каталог* (также известный как *папка*) – это хранилище файлов или других каталогов. Содержимое файла может быть любым. Структура данных, хранящихся в файле, определяется программой, которая записала файл в хранилище. Unix-подобные системы организуют свою структуру каталогов в виде единой иерархии каталогов. Иерархия начинается с корня, обозначаемого одной прямой косой чертой (/), а все остальные каталоги являются потомками корня. Например, библиотечные файлы хранятся в */usr/lib*, где *usr* – подкаталог корня, а *lib* – подкаталог *usr*. Эта единая иерархия применяется даже в том случае, если в системе имеется более одного устройства хранения данных. Дополнительные устройства хранения сопоставляются с местоположением в структуре каталогов, это называется *монтированием устройства*. Например, USB-накопитель может быть смонтирован в каталог */mnt/usb1*.

В отличие от этого Microsoft Windows присваивает букву диска (A–Z) каждому тому. Поэтому вместо единой структуры каталогов каждый

диск имеет свой корень и иерархию каталогов. Windows использует обратную косую черту (\) в путях к каталогам и двоеточие (:) после буквы диска. Например, системные файлы Windows, хранящиеся на диске С, обычно располагаются в каталоге `C:\windows\system32`. Это правило восходит к DOS (и более ранним версиям), когда буквы А и В были зарезервированы для дискет, а диск С представлял собой внутренний жесткий диск. По сей день диск С обычно используется в качестве буквы диска для «системного» тома, того, на котором установлена Windows<sup>1</sup>.

#### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 27 на стр. 288, где вы сможете ознакомиться с деталями хранения данных и файлов в Raspberry Pi OS.*

## Службы и демоны

Операционные системы предоставляют возможность автоматической работы процессов в фоновом режиме без участия пользователя. Такие процессы называются *службами* в Windows и *демонами* в Unix-подобных системах. Типичная операционная система включает в себя несколько таких служб, которые запускаются по умолчанию, например службу для настройки сетевых параметров или службу, выполняющую задачи по расписанию. Службы используются для предоставления возможностей, которые не привязаны к конкретному пользователю, не должны работать в режиме ядра, но должны быть доступны по требованию.

Операционные системы обычно включают компонент, отвечающий за управление службами. Некоторые службы должны запускаться при загрузке ОС, другие – в ответ на определенное событие. Часто службы должны перезапускаться в случае неожиданного сбоя. В Windows такие функции выполняет *диспетчер управления службами* (*Service Control Manager, SCM*). Исполняемый файл SCM, *services.exe*, запускается в самом начале процесса загрузки Windows и продолжает работать до тех пор, пока работает сама Windows. Многие современные дистрибутивы Linux приняли *systemd* в качестве стандартного компонента для управления демонами, хотя в Linux могут использоваться и другие механизмы для запуска и управления демонами. Как обсуждалось ранее, *systemd* также действует как начальный процесс, поэтому он запускается очень рано в процессе загрузки Linux и продолжает функционировать, пока система находится в рабочем состоянии.

Термин «демон» в Unix и Linux происходит от «демона Максвелла» – гипотетического существа, описанного в физическом эксперименте. Это существо работало в фоновом режиме, подобно компьютерному

<sup>1</sup> Отметим, что в современных версиях Windows присвоение буквы С именному тому диску, на котором установлена Windows (системному), совершенно необязательно, и буквы всех дисков могут быть в любой момент изменены пользователем с правами администратора. Правда, делать такое (особенно менять букву системного диска) без нужды не рекомендуется, так как некоторые пользовательские программы могут при этом потерять свои данные. – *Прим. ред.*

демону. Вне компьютерных систем слово «демон» обычно произносится как пишется, но когда речь идет о фоновых процессах, также приемлемо произношение «дэймон» (*DAY-mon*). Исторически термин «служба» был специфическим для Windows, но теперь он используется и в Linux, часто для обозначения демонов, запускаемых *systemd*.

#### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 28 на стр. 289, где вы сможете протестировать службы на Raspberry Pi OS.*

## Безопасность

Операционная система обеспечивает модель безопасности для кода, который работает на этой ОС. В данном контексте *безопасность* означает, что программное обеспечение и пользователи этого программного обеспечения должны иметь доступ только к соответствующим частям системы. Это может показаться не очень важным для персонального устройства, такого как ноутбук или смартфон. Если только один пользователь входит в систему, разве он не должен иметь доступ ко всему? На самом деле нет. По крайней мере, не по умолчанию. Пользователи совершают ошибки, в том числе запускают код, который не заслуживает доверия. Если пользователь случайно запустил вредоносное программное обеспечение на своем устройстве, ОС может снизить ущерб, ограничив доступ этого пользователя. В общей системе, в которую может войти несколько пользователей, пользователь не должен иметь возможности читать или изменять данные другого пользователя, по крайней мере, по умолчанию.

Операционные системы используют множество методов для обеспечения безопасности. Давайте рассмотрим лишь некоторые из них. Простой перевод приложений на уровень пользовательского режима в значительной степени гарантирует, что программы не будут намеренно или случайно вмешиваться в работу других приложений или ядра. Операционные системы также обеспечивают безопасность файловой системы, гарантируя, что доступ к данным, хранящимся в файлах, могут получить только соответствующие пользователи и процессы. Виртуальная память сама по себе может быть защищена, области памяти могут быть помечены как доступные только для чтения или как исполняемые, что помогает ограничить нецелевое использование памяти. Обеспечение системы входа для пользователей позволяет операционной системе управлять безопасностью на основе личности пользователя. Все это базовые требования к современной операционной системе. К сожалению, в операционных системах регулярно обнаруживаются слабые места, позволяющие злоумышленникам обходить защиту ОС.

Регулярное обновление современных операционных систем, подключенных к интернету, имеет решающее значение для поддержания безопасности.

## Выводы

В этой главе мы рассмотрели операционные системы – программное обеспечение, которое взаимодействует с компьютерным оборудованием и обеспечивает среду для выполнения программ. Вы узнали о ядре операционной системы, компонентах, не относящихся к ядру, и разделении режима ядра и пользовательского режима. Мы рассмотрели два доминирующих семейства операционных систем: Unix-подобные операционные системы и Microsoft Windows. Вы узнали, что программа выполняется в контейнере, называемом процессом, и несколько потоков могут выполняться параллельно в рамках этого процесса. Мы рассмотрели различные аспекты программного взаимодействия с операционной системой: API, системные вызовы, программные библиотеки и ABI. В следующей главе мы выйдем за рамки вычислительных операций на одном устройстве и исследуем интернет, рассмотрев различные уровни и протоколы, которые делают интернет возможным.

### ПРОЕКТ № 20: Исследование запущенных процессов

Необходимые условия: Raspberry Pi, работающий под Raspberry Pi OS. Я рекомендую вам обратиться к приложению В и прочитать весь раздел «Raspberry Pi», если вы этого еще не сделали.

В этом проекте вы посмотрите на процессы, запущенные на Raspberry Pi. Инструмент `ps` предоставляет различные виды запущенных процессов. Давайте начнем со следующей команды, которая предоставляет древовидное представление процессов.

---

```
$ ps -eH
```

---

Результат должен выглядеть примерно так, как показано ниже. Здесь я воспроизвел только его часть.

---

1 ?	00:00:10	systemd
93 ?	00:00:09	systemd-journal
133 ?	00:00:01	systemd-udev
233 ?	00:00:01	systemd-timesyn
274 ?	00:00:02	thd
275 ?	00:00:01	cron
276 ?	00:00:00	dbus-daemon
286 ?	00:00:03	rsyslogd
287 ?	00:00:01	systemd-logind
291 ?	00:00:08	avahi-daemon
296 ?	00:00:00	avahi-daemon
297 ?	00:00:01	dhcpcd

---

---

351	tty1	00:00:00	agetty
352	?	00:00:00	agetty
358	?	00:00:00	sshd
5016	?	00:00:00	sshd
5033	?	00:00:00	sshd
5036	pts/0	00:00:00	bash
5178	pts/0	00:00:00	ps

---

Уровень отступа указывает на отношения родитель/потомок. Например, в этом выводе мы видим, что `systemd` является родителем `systemd-journal`, `systemd-udevd` и т. д. Или, наоборот, мы видим, что `ps` (команда, запущенная в данный момент) является дочерней для `bash`, которая является дочерней для `sshd` и т. д.

Отображаемые столбцы означают следующее:

**PID** – идентификатор процесса;

**TTY** – связанный терминал;

**TIME** – суммарное процессорное время;

**CMD** – имя исполняемого файла.

Количество запущенных процессов может удивить вас, когда вы запустите `ps` таким образом! Операционная система обрабатывает множество вещей, и поэтому естественно, что в любой момент времени запущено большое количество процессов. Обычно вы видите, что первый процесс в списке – это PID 2, `kthreadd`. Это родитель потоков ядра, а дочерние процессы, которые вы видите в списке под `kthreadd`, – это потоки, запущенные в режиме ядра. Другой процесс, на который следует обратить внимание, – это PID 1, процесс `init`, первый запускаемый процесс пользовательского режима. В предыдущем выводе процесс `init` – это `systemd`. Ядро Linux запускает процесс `init` и `kthreadd` в таком порядке, чтобы обеспечить им присвоение PID 1 и 2 соответственно.

Давайте посмотрим на `init`-процесс. Это первый процесс, запускаемый в пользовательском режиме, и конкретный исполняемый файл, который запускается, может отличаться в разных версиях Linux. Вы можете использовать `ps`, чтобы найти команду, используемую для запуска PID 1:

---

```
$ ps 1
```

---

Вы должны увидеть вывод, похожий на следующий:

---

PID	TTY	STAT	TIME	COMMAND
1	?	Ss	0:03	/sbin/init

---

Это говорит о том, что команда, использованная для запуска процесса `init`, была `/sbin/init`. Так как же запуск `/sbin/init` приводит к выполнению `systemd`, как вы видели в предыдущем выводе `ps`? Это происходит потому, что `/sbin/init` на самом деле является символической ссылкой на `systemd`.

Символическая ссылка ссылается на другой файл или каталог. Вы можете увидеть это с помощью следующей команды:

```
$ stat /sbin/init
```

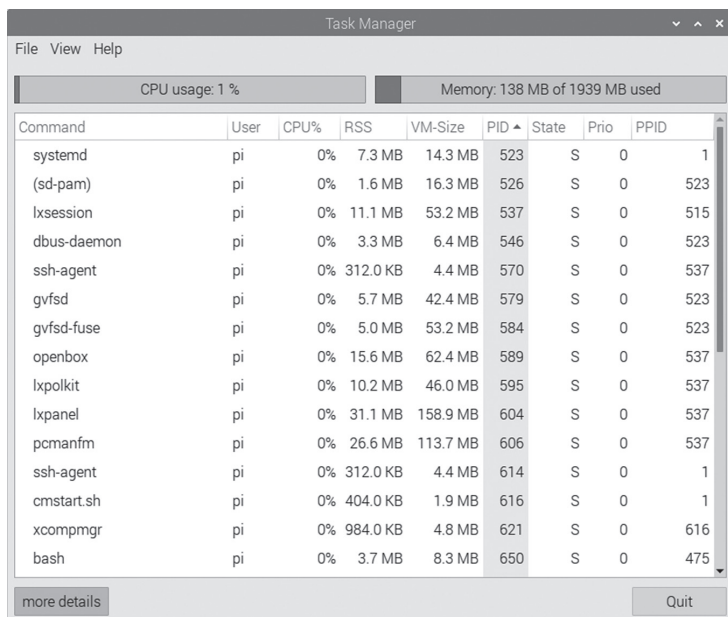
```
File: /sbin/init -> /lib/systemd/systemd
Size: 20          Blocks: 0          IO Block: 4096   symbolic link
```

В этом выводе видно, что `/sbin/init` является символической ссылкой на `/lib/systemd/systemd`.

Другой удобный вид дерева процессов можно получить с помощью инструмента `pstree`, о котором говорилось ранее в этой главе. Запуск `pstree` представляет дерево процессов в режиме пользователя в хорошем формате, начиная с процесса `init`. Попробуйте:

```
$ pstree
```

Также, если ваш Raspberry Pi настроен для загрузки в среду рабочего стола, вы можете попробовать приложение Диспетчер задач (Task Manager), которое входит в Raspberry Pi OS. Оно предоставляет графическое представление запущенных процессов, как показано на рис. 10-15.



Command	User	CPU%	RSS	VM-Size	PID	State	Prio	PPID
systemd	pi	0%	7.3 MB	14.3 MB	523	S	0	1
(sd-pam)	pi	0%	1.6 MB	16.3 MB	526	S	0	523
lxsession	pi	0%	11.1 MB	53.2 MB	537	S	0	515
dbus-daemon	pi	0%	3.3 MB	6.4 MB	546	S	0	523
ssh-agent	pi	0%	312.0 KB	4.4 MB	570	S	0	537
gvfsd	pi	0%	5.7 MB	42.4 MB	579	S	0	523
gvfsd-fuse	pi	0%	5.0 MB	53.2 MB	584	S	0	523
openbox	pi	0%	15.6 MB	62.4 MB	589	S	0	537
lxpolkit	pi	0%	10.2 MB	46.0 MB	595	S	0	537
lxpanel	pi	0%	31.1 MB	158.9 MB	604	S	0	537
pcmanfm	pi	0%	26.6 MB	113.7 MB	606	S	0	537
ssh-agent	pi	0%	312.0 KB	4.4 MB	614	S	0	1
cmstart.sh	pi	0%	404.0 KB	1.9 MB	616	S	0	1
xcompmgr	pi	0%	984.0 KB	4.8 MB	621	S	0	616
bash	pi	0%	3.7 MB	8.3 MB	650	S	0	475

Рис. 10-15. Диспетчер задач в ОС Raspberry Pi

## ПРОЕКТ № 21: Создание потока выполнения и наблюдение за ним

Необходимые условия: Raspberry Pi под Raspberry Pi OS. Смотрите раздел «Raspberry Pi» на стр. 416.

В этом проекте вы напишете программу, которая создает поток. Затем будете наблюдать за работой потока. С помощью выбранного вами текстового редактора создайте новый файл *threader.c* в корне домашней папки. Введите в текстовый редактор следующий код на языке C (не обязательно сохранять отступы и пустые строки, но обязательно сохраняйте переносы строк).

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>

void * mythread(void* arg)❶
{
    while(1)❷
    {
        printf("mythread PID: %d\n", (int) getpid());❸
        printf("mythread TID: %d\n", (int) syscall(SYS_gettid));
        sleep(5);❹
    }
}

int main()❺
{
    pthread_t thread;

    pthread_create(&thread, NULL, &mythread, NULL);❻

    while(1)❼
    {
        printf("main PID: %d\n", (int) getpid());❸
        printf("main TID: %d\n", (int) syscall(SYS_gettid));
        sleep(10);❹
    }

    return 0;
}
```

Прежде чем продолжить, давайте рассмотрим исходный код. Я не буду вдаваться во все подробности, но вкратце – программа начинается с функции `main` ❺, которая создает поток ❻, запускающий функцию `mythread` ❶. Это означает, что существует два потока, `main` и `mythread`. Оба потока работают в бесконечном цикле ❷ ❷, где время от времени выводят PID и TID текущего потока ❸ ❸. Для разнообразия `mythread` выводит данные примерно каждые 5 с ❹, а `main` – примерно каждые 10 с ❹. Это помога-



ет проиллюстрировать, что потоки на самом деле работают параллельно и выполняют работу по собственному расписанию. Давайте попробуем это сделать.

После сохранения файла используйте GNU C Compiler (*gcc*) для компиляции кода в исполняемый файл. Следующая команда принимает файл *threader.c* в качестве входного и выдает исполняемый файл с именем *threader*.

---

```
$ gcc -pthread -o threader threader.c
```

---

Теперь попробуйте запустить код с помощью следующей команды:

---

```
$ ./threader
```

---

Запущенная программа должна вывести что-то вроде этого, хотя номера PID и TID будут другими:

---

```
main      PID: 2300
main      TID: 2300
mythread  PID: 2300
mythread  TID: 2301
```

---

По мере выполнения программы ожидайте, что эти два потока будут продолжать выводить информацию о своих PID и TID. Номера TID и PID не будут меняться для данного примера программы, так как это один и тот же процесс и потоки, работающие все время. Вы должны увидеть, что *mythread* осуществляет вывод в два раза чаще, чем *main* – каждые 5 с против каждых 10 с.

Оставьте эту программу запущенной и посмотрите на список запущенных процессов и потоков. Для этого вам нужно открыть второе окно терминала и выполнить следующую команду (символ | можно ввести с помощью **SHIFT** и клавиши «обратный слеш» в верхнем ряду клавиатуры справа, над клавишей **ENTER**).

---

```
$ ps -e -T | grep threader
2300 2300 pts/0    00:00:00 threader
2300 2301 pts/0    00:00:00 threader
```

---

Добавление опции **-T** к команде *ps* показывает потоки наряду с процессами. Утилита *grep* фильтрует вывод, чтобы вывести только информацию о процессе, порожденном нашей программой *threader*. В этом выводе первый столбец – это PID, а второй – TID. Таким образом, вы можете видеть, что вывод *ps* совпадает с выводом вашей программы. У двух потоков один PID, но разные TID. Также обратите внимание, что TID потока *main* совпадает с его PID. Это ожидаемо для первого потока в процессе.

Чтобы остановить выполнение программы *threder*, вы можете нажать **CTRL-C** в окне терминала, где она запущена. Или во втором окне терминала вы можете использовать утилиту *kill*, указав PID потока *main*, как показано ниже:

```
$ kill 2300
```

## ПРОЕКТ № 22: Исследование виртуальной памяти

Необходимые условия: Raspberry Pi под Raspberry Pi OS. Смотрите раздел «Raspberry Pi» на стр. 416.

В этом проекте вы изучите использование виртуальной памяти в Raspberry Pi OS. Начнем с того, как распределяется адресное пространство между режимом ядра и режимом пользователя. В этом проекте предполагается, что вы используете 32-битную версию Raspberry Pi OS, что означает наличие 4 ГБ виртуального адресного пространства. Linux позволяет разделить эти 4 ГБ либо на 2 ГБ пользователю и 2 ГБ ядру, либо на 3 ГБ пользователю и 1 ГБ ядру. Меньшие адреса используются для пользовательского режима, а большие – для режима ядра. Это означает, что при разделении 2:2 адреса режима ядра начинаются с 0x80000000, а при разделении 3:1 адреса режима ядра начинаются с 0xC0000000. Вы можете увидеть начало адресного пространства режима ядра с помощью этой команды:

```
$ dmesg | grep lowmem
```

Если команда *dmesg* не выдает никаких результатов, просто перезагрузите Raspberry Pi, а затем снова запустите команду *dmesg*. Команда должна выдать результат, аналогичный следующему.

```
lowmem : 0x80000000 - 0xbb400000 (948 MB)
```

Если вам интересно, почему нужно перезагрузить Raspberry Pi, если эта команда выдает пустой результат, вот некоторая справочная информация. Ядро Linux записывает диагностические сообщения в так называемый кольцевой буфер ядра, который отображается инструментом *dmesg*. Сообщения в буфере предназначены для того, чтобы дать пользователям некоторое представление о работе ядра. Здесь хранится только ограниченное количе-

ство сообщений, по мере добавления новых сообщений старые удаляются. Конкретное сообщение, которое мы хотим увидеть (относительно `lowmem`), записывается при запуске системы, поэтому, если ваша система работала некоторое время, оно могло быть перезаписано. Перезапуск системы гарантирует, что сообщение будет записано снова.

Как вы видите, в моей системе `lowmem` ядра начинается с `0x80000000`, что указывает на разделение 2:2. Это означает, что процессы пользовательского режима могут использовать адреса от `0x000000` до `0x7fffffff`. Этот диапазон адресов может ссылаться на 2 ГБ памяти, и, хотя все адресное пространство доступно каждому процессу, типичному процессу фактически нужна только часть этого диапазона. Некоторые адреса отображаются на физическую память, но другие остаются неотображенными.

Если ваша система возвращает значение `0xc0000000` для начала `lowmem`, значит, ваша система работает с разделением 3:1. Это дает процессам пользовательского режима 3 ГБ виртуального адресного пространства, от `0x000000` до `0xbfffffff`.

Давайте выберем процесс и рассмотрим, как он использует виртуальную память. Raspberry Pi OS использует Bash в качестве оболочки процесса по умолчанию, поэтому, если вы работаете с командной строкой в Raspberry Pi OS, должен быть запущен хотя бы один экземпляр `bash`. Давайте найдем PID экземпляра `bash`:

---

```
$ ps | grep bash
```

---

Это должно вывести текст, подобный следующему:

---

```
2670 pts/0      00:00:00 bash
```

---

В моем случае PID `bash` был 2670. Теперь выполните следующую команду, чтобы увидеть отображение виртуальной памяти в процессе `bash`. При вводе команды не забудьте заменить `<pid>` на PID, полученный в вашей системе.

---

```
$ rmap <pid>
```

---

Вывод будет похож на следующий, где каждая строка представляет область виртуальной памяти в адресном пространстве процесса.

```

2670:  -bash
00010000    872K r-x-- bash
000f9000     4K r---- bash
000fa000    20K rw--- bash
000ff000    36K rw--- [ anon ]
00ee7000   1044K rw--- [ anon ]
76b30000    36K r-x-- libnss_files-2.24.so
76b39000    60K ----- libnss_files-2.24.so
76b48000     4K r---- libnss_files-2.24.so
76b49000     4K rw--- libnss_files-2.24.so
...
7ec2c000    132K rw--- [ stack ]
7ec74000     4K r-x-- [ anon ]
7ec75000     4K r---- [ anon ]
7ec76000     4K r-x-- [ anon ]
ffff0000     4K r-x-- [ anon ]
total      6052K

```

Первый столбец показывает начальный адрес диапазона, второй – размер диапазона, третий – разрешения для данного диапазона (г = чтение, w = запись, x = выполнение, r = частный, s = общий) и последний – имя диапазона. Имя диапазона – это либо имя файла, либо имя, идентифицирующее диапазон памяти, если он не сопоставлен с файлом.

Вы можете видеть, что почти каждый диапазон в представленном выводе находится в ожидаемом диапазоне пользовательского режима от 0x000000 до 0x7fffffff. Исключением является последняя запись, которая соответствует векторной странице процессора ARM и представляет собой особый случай, так как находится вне стандартного диапазона адресов пользовательского режима. Как видно из предыдущего вывода, данный конкретный экземпляр bash занимает всего 6052 КБ (около 6 МБ) виртуальной памяти из возможных 2 Гб, т. е. около 0,3 %.

## ПРОЕКТ № 23: Исследование API операционной системы

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы попытаетесь вызвать API операционной системы различными способами. В особенности вы сосредоточитесь на создании файла и записи в него некоторого текста. С помощью текстового редактора по вашему выбору создайте файл *newfile.c* в корне домашней папки. Введите в текстовый редактор следующий код на языке C.

```
#include <fcntl.h>
#include <unistd.h>

#define msg "Hello, file!\n"❶

int main()❷
{
    int fd;❸
    fd = open("file1.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);❹
    write(fd, msg, sizeof(msg) - 1);❺
    close(fd);❻
    return 0;❼
}
```

Прежде чем продолжить, давайте изучим исходный код, чтобы понять, что именно он делает. Вкратце программа использует три функции API (*open*, *write* и *close*), чтобы создать новый файл, записать в него текст и, наконец, закрыть файл. Наша цель здесь увидеть, как API операционной системы позволяет программе взаимодействовать с аппаратным обеспечением компьютера, в частности с устройством хранения данных. Давайте рассмотрим программу более подробно.

После необходимых операторов *include*<sup>1</sup> следующая строка определяет *msg* как текстовую строку ❶, которая позже будет записана во вновь созданный файл. Затем код определяет *main*, точку входа программы ❷. Внутри *main* объявляется целочисленная переменная с именем *fd* ❸. Далее вызывается функция OS API *open* для создания нового файла с именем *file1.txt* ❹. Другие аргументы, передаваемые функции *open*, определяют детали того, как файл должен быть открыт. Чтобы не усложнять, я не буду рассматривать эти детали здесь, но не стесняйтесь исследовать значения этих аргументов. Функция *open* возвращает дескриптор файла<sup>2</sup>, который сохраняется в переменной *fd*.

<sup>1</sup> Операторы *#include* подключают к программе системные библиотеки (в данном случае *fcntl.h* и *unistd.h*), в которых находятся используемые далее функции. Если их не указывать, компилятор не поймет, что означает, например, слово *open* и остальные названия функций, используемых в программе. – Прим. ред.

<sup>2</sup> Дескриптор (*descriptor* – описатель, идентификатор) файла – специальное число, идентифицирующее файл, открытый в системе. В системе Windows дескрипторы именуют словом *handle* (приводная ручка). Подробнее о дескрипторах см. в проекте № 24. – Прим. ред.

Затем функция `write` используется для записи текста `msg` в файл `file1.txt` (идентифицируемый дескриптором файла, хранящимся в `fd`) ⑤. Функция `write` требует ввода данных для записи (`msg`) и количества байт для записи, вычисляемого как `sizeof(msg) - 1`<sup>1</sup>. Вы вычитаете 1, потому что язык C завершает строки нулевым символом, и вам не нужно записывать этот байт в выходной файл. Теперь программа закончила работу с файлом и вызывает функцию `close` для дескриптора файла, чтобы указать, что файл больше не используется ⑥. Наконец, программа завершает работу с кодом возврата 0 ⑦, что означает успех.

После сохранения файла используйте GNU C Compiler (`gcc`) для компиляции кода в исполняемый файл. Команда ниже принимает `newfile.c` в качестве входных данных и генерирует исполняемый файл с именем `newfile`.

---

```
$ gcc -o newfile newfile.c
```

---

Теперь попробуйте запустить код с помощью следующей команды. Вы не увидите никакого вывода, поскольку текст записывается в файл, а не выводится на терминал.

---

```
$ ./newfile
```

---

Чтобы определить, успешно ли выполнялась программа, нужно посмотреть, был ли создан файл. Файл должен иметь имя `file1.txt` и находиться в вашем текущем каталоге. Вы можете использовать команду `ls` для просмотра содержимого текущего каталога и поиска файла. Если файл `file1.txt` существует, можете просмотреть его содержимое с помощью команды `cat`.

---

```
$ ls
$ cat file1.txt
```

---

Эта команда должна вывести на терминал сообщение *Hello, file!*, поскольку именно такой текст программа записала в файл. Или вы можете просмотреть свойства файла в приложении File Manager на рабочем столе Raspberry Pi OS, а также открыть файл `file1.txt` в выбранном вами текстовом редакторе.

Когда вы используете язык программирования C, вы получаете представление о специфике функций ОС API, поскольку функции `open`, `write` и `close` определены как функции C. Однако при взаимодействии с ОС вы не ограничены языком C. Другие языки предоставляют свой собственный слой поверх API, скрывая часть от разработчиков программного обеспечения. Чтобы проиллюстрировать это, давайте напишем эквивалентную программу на языке Python.

---

<sup>1</sup> Название функции `sizeof` переводится как «размер» (того, что указано в качестве аргумента, в данном случае строки `msg`). – Прим. ред.

С помощью выбранного вами текстового редактора создайте файл *new-file.py* в корне домашней папки. Введите в текстовый редактор следующий код на языке Python.

---

```
f = open('file2.txt', 'w') ❶  
f.write('Привет из Python!\n'); ❷  
f.close()
```

---

Прежде чем продолжить, давайте рассмотрим исходный код. Эта программа фактически делает то же самое, что и предыдущая, только имя выходного файла другое (*file2.txt*) ❶ и текст, записываемый в этот файл, тоже другой ❷. В данном случае Python использует те же названия, что и ОС API (*open*, *write*, *close*), но здесь это не прямые вызовы операционной системы, а вызовы стандартной библиотеки Python.

После того как вы сохранили этот код, его можно запустить. Помните, что Python является интерпретируемым языком, поэтому вместо компиляции кода Python просто запустите его с помощью интерпретатора Python, как показано ниже:

---

```
$ python3 newfile.py
```

---

Чтобы определить, что программа выполнена успешно, нужно посмотреть, был ли создан файл *file2.txt* с ожидаемым содержимым. Вы можете снова использовать *ls* и *cat*, чтобы проверить это, или посмотреть файл в File Manager на рабочем столе.

---

```
$ ls  
$ cat file2.txt
```

---

Хотя может показаться, что вы просто используете возможности Python для работы с файлом, имейте в виду, что Python не может сделать это сам по себе. Интерпретатор Python выполняет вызовы системного API от вашего имени, когда запускает код. Вы сможете понаблюдать за этим в следующем проекте.

## ПРОЕКТ № 24: Наблюдение за системными вызовами

Необходимые условия: завершить проект № 23.

В этом проекте вы будете наблюдать за системными вызовами, выполняемыми программами, которые написали в проекте № 23. Для этого вы будете использовать инструмент под названием *strace*, который отслеживает системные вызовы и выводит информацию о них на терминал.

Откройте терминал на вашем Raspberry Pi и используйте *strace* для запуска программы *newfile*, которую вы ранее написали на C и скомпилировали:

---

```
$ strace ./newfile
```

---

Инструмент *strace* запускает программу (в данном случае *newfile*) и показывает все системные вызовы, которые выполняются во время работы этой программы. В начале вывода можно увидеть ряд системных вызовов, которые представляют собой действия, необходимые для загрузки исполняемого файла и указанных библиотек. Это работа, которая происходит до выполнения написанного вами кода, и вы можете пропустить этот текст. Ближе к концу вывода вы должны увидеть текст, похожий на следующий:

---

```
openat(AT_FDCWD, "file1.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) = 3
write(3, "Hello, file!\n", 13)      = 13
close(3)                      = 0
```

---

Это должно показаться знакомым, ведь это почти те же три функции API, которые вы использовали для создания файла *file1.txt* и записи в него текста. Функции C, которые вы вызывали из своей программы, – это просто тонкие обертки вокруг одноименных системных вызовов, за исключением функции *open*, которая вызывает системный вызов *openat*. Значения после знаков равенства – это возвращаемые значения трех системных вызовов. В моей системе функция *openat* вернула 3, число, известное как *файловый дескриптор*, который ссылается на открытый файл.

Вы можете видеть значение файлового дескриптора, используемое в качестве параметра для последующих вызовов *write* и *close*. Функция *write* вернула 13 – количество записанных байтов. Функция *close* вернула 0 – индикатор успеха.

Теперь используйте тот же подход, чтобы проверить системные вызовы, сделанные из программы Python, которую вы написали в проекте № 23.

---

```
$ strace python3 newfile.py
```

---



Ожидайте увидеть здесь еще больший объем вывода, поскольку `strace` на самом деле следит за интерпретатором Python, который в свою очередь должен загрузить `newfile.py` и запустить его. Если вы посмотрите в конец вывода, то увидите вызовы `openat`, `write` и `close`, точно так же, как и в программе на языке C. Это показывает, что, несмотря на различия в исходном коде между C и Python, в конечном итоге для взаимодействия с файлами осуществляются одни и те же системные вызовы.

Инструмент `strace` можно использовать, чтобы получить быстрое представление о том, как программа взаимодействует с операционной системой. Например, ранее в этой главе мы использовали утилиту `ps` для получения списка процессов. Если вы хотите понять, как работает `ps`, можете запустить `ps` под `strace`, например, так:

---

```
$ strace ps
```

---

Посмотрите на результат выполнения этой команды, чтобы увидеть, какие системные вызовы выполняет `ps`.

## ПРОЕКТ № 25: Использование GLIBC

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы напишете код для использования библиотеки C и детально рассмотрите, как это работает. Используйте текстовый редактор по своему выбору, чтобы создать новый файл `random.c` в корне домашней папки. Введите в текстовый редактор следующий код на языке C.

---

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    srand(time(0));❶
    printf("%d\n", rand());❷
    return 0;
}
```

---

Эта небольшая программа просто выводит на терминал случайное целочисленное значение. Первое, что делает программа, – вызывает функцию `srand` для активации генератора случайных чисел ❶, выдающего уникальную последовательность чисел. В качестве начального значения используется текущее время, возвращаемое функцией `time`. В следующей строке выводится случайное значение, возвращаемое функцией `rand` ❷.

Для выполнения этого алгоритма программа использует четыре функции из библиотеки C (`time`, `srand`, `rand` и `printf`).

После сохранения файла вы можете использовать GNU C Compiler (`gcc`) для компиляции кода в исполняемый файл. Следующая команда принимает `random.c` в качестве входного файла и выводит исполняемый файл с именем `random`.

---

```
$ gcc -o random random.c
```

---

Теперь попробуйте запустить код с помощью следующей команды. Программа должна вывести на экран случайное число. Запустите ее несколько раз, чтобы убедиться, что она выводит разные числа. Однако быстрый запуск программы дважды может дать один и тот же результат, поскольку начальное значение, возвращаемое функцией `time`, увеличивается только раз в секунду.

---

```
$ ./random
```

---

После того как вы убедились, что программа работает, посмотрите на библиотеки, которые импортирует программа. Один из способов сделать это – запустить утилиту `readelf`, например, так:

---

```
$ readelf -d random | grep NEEDED
```

---

Найдите в результатах секцию `NEEDED`, как показано ниже:

---

```
0x00000001 (NEEDED)           Shared library: [libc.so.6]
```

---

Это говорит о том, что для работы этой программы требуется библиотека `libc.so.6`. Это ожидаемо, так как это библиотека `GNU C` (также известная как `glibc`). Другими словами, поскольку программа полагается на функции стандартной библиотеки C, операционная система должна загрузить файл библиотеки `libc.so.6`, чтобы код библиотеки был доступен. Это хорошее начало, но что, если вы хотите увидеть конкретный список функций, которые программа `random` использует из этой библиотеки? Это можно проверить следующим образом:

---

```
$ objdump -TC random
```

---

Это даст вам результат, подобный следующему:

---

```
random:      file format elf32-littlearm
```

DYNAMIC SYMBOL TABLE:

00000000	w	D	*UND*	00000000	__gmon_start__
00000000		DF	*UND*	00000000	GLIBC_2.4
00000000		DF	*UND*	00000000	GLIBC_2.4
00000000		DF	*UND*	00000000	GLIBC_2.4
00000000		DF	*UND*	00000000	GLIBC_2.4
00000000		DF	*UND*	00000000	GLIBC_2.4
00000000		DF	*UND*	00000000	GLIBC_2.4
00000000		DF	*UND*	00000000	GLIBC_2.4

---

В этом выводе, в крайнем правом столбце, вы можете увидеть ожидаемые четыре функции (`srand`, `rand`, `printf` и `time`), а также некоторые дополнительные функции.

Теперь, когда вы выяснили, какие функции `glibc` были импортированы вашей программой `random`, можете посмотреть список всех функций, экспортируемых `glibc`. Это функции, которые эта библиотека делает доступными для использования программами. Вы можете получить эту информацию с помощью следующей команды:

---

```
$ objdump -TC /lib/arm-linux-gnueabi/libc.so.6
```

---

Иногда полезно посмотреть информацию о загруженных библиотеках во время отладки запущенного процесса. Давайте попробуем это сделать, отлаживая программу `random`. Для начала введите следующую команду:

---

```
$ gdb random
```

---

На данный момент `gdb` загрузил файл, но ни одна инструкция еще не запущена. В окне (`gdb`) введите следующее, чтобы начать выполнение программы. Отладчик останавливает выполнение, как только достигает начала функции `main`.

---

```
(gdb) start
```

---

Посмотрите на загруженные общие библиотеки:

---

```
(gdb) info sharedlibrary
```

From	To	Syms Read	Shared Object Library
0x76fcea30	0x76fea150	Yes	/lib/ld-linux-armhf.so.3❶
0x76fb93ac	0x76fbc300	Yes (*)	/usr/lib/arm-linux-gnueabi/libarmmem-v71.so❷
0x76e6e050	0x76f702b4	Yes	/lib/arm-linux-gnueabi/libc.so.6❸

(\*): Shared library is missing debugging information.

---

Первая библиотека, `ld-linux-armhf.so.3` ❶, является динамически подключаемой библиотекой Linux. Она отвечает за загрузку других библиотек. Двоичные файлы Linux ELF компилируются для использования определенной

библиотеки, эта информация содержится в заголовке ELF скомпилированной программы. Вы можете найти подключаемую библиотеку для программы `gandom` с помощью следующей команды из окна терминала (не в `gdb`):

---

```
$ readelf -l random | grep interpreter
Requesting program interpreter: /lib/ld-linux-armhf.so.3
```

---

Как видно из предыдущего вывода, подключаемая библиотека, указанная для программы `gandom`, – это `ld-linux-armhf.so.3`, та самая динамически подключаемая библиотека, которую мы только что обсуждали.

Посмотрите еще раз на вывод `info sharedlibrary` в `gdb`, вы увидите, что второй библиотекой в списке является `libarmmem-v71.so` ❷. Эта библиотека указывается в файле `/etc/ld.so.preload`, текстовом файле, в котором перечислены библиотеки, загружаемые для каждой выполняемой в системе программы.

Теперь перейдем к третьей библиотеке, которая нас интересует, `libc.so.6` ❸, библиотека GNU C (`glibc`). В выводах `readelf` и `objdump` ранее вы видели, что эта библиотека была импортирована исполняемым файлом, и здесь вы можете убедиться, что она действительно успешно загрузилась во время работы. Вы также можете увидеть конкретный диапазон адресов, по которым она загрузилась (`0x76e6e050 - 0x76f702b4`), и конкретный путь к каталогу, из которого она загрузилась.

Вы можете выйти из отладчика в любое время, набрав `quit` в `gdb`.

## ПРОЕКТ № 26: Просмотр загруженных модулей ядра

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы посмотрите на загруженные модули ядра на Raspberry Pi OS, включая драйверы устройств. Драйверы устройств обычно реализуются в Linux как модули ядра, хотя не все модули ядра являются драйверами устройств. Чтобы получить список загруженных модулей, можете просмотреть содержимое файла `/proc/modules` или использовать инструмент `lsmod`, как показано ниже:

---

```
$ lsmod
```

---

Чтобы просмотреть более подробную информацию о конкретном модуле, используйте утилиту `modinfo` следующим образом (на примере модуля `snd`):

---

```
$ modinfo snd
```

---

## ПРОЕКТ № 27: Исследование устройств хранения данных и файловых систем

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы рассмотрите устройства хранения данных (то, что мы называем *вторичными хранилищами*) и файловые системы. Начнем с перечисления блочных устройств, именно так Linux характеризует устройства хранения данных.

```
$ lsblk
NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
mmcblk0       179:0    0  29.8G  0 disk ❶
├─mmcblk0p1   179:1    0   256M  0 part /boot❷
└─mmcblk0p2   179:2    0   29.6G  0 part /❸
```

Здесь мы видим один «диск» под названием `mmcblk0` ❶, который является картой microSD в Raspberry Pi. Видно, что он разделен на два раздела разного размера. Раздел 1 отображается в каталог `/boot` в единой структуре каталогов ❷, а раздел 2 отображается в корень `/` ❸.

Теперь посмотрим на общее использование устройства хранения данных с помощью команды `df`:

```
$ df -h -T
Filesystem      Type      Size  Used Avail Use% Mounted on
/dev/root       ext4       30G   3.0G   25G  11% /❶
devtmpfs        devtmpfs   459M    0   459M   0% /dev
tmpfs            tmpfs      464M    0   464M   0% /dev/shm
tmpfs            tmpfs      464M  6.3M   457M   2% /run
tmpfs            tmpfs      5.0M  4.0K   5.0M   1% /run/lock
tmpfs            tmpfs      464M    0   464M   0% /sys/fs/cgroup
/dev/mmcblk0p1  vfat       253M   52M   202M  21% /boot❷
tmpfs            tmpfs      93M    0    93M   0% /run/user/1000
```

Эта команда позволяет просмотреть различные смонтированные файловые системы, их размер и степень заполнения. Только каталоги `root` ❶ и `/boot` ❷ отображены на устройства хранения данных. Остальные – это временные файловые системы, которые находятся в памяти, а не на постоянном устройстве хранения.

Вы можете получить представление о каталогах в вашей системе, выполнив команду `tree`. Используемые в приведенном примере параметры этой команды ограничивают вывод только каталогами и только на три уровня вглубь иерархии.

```
$ tree -d -L 3 /
```

Вы также можете получить аналогичный вид из среды рабочего стола, используя приложение File Manager.

## ПРОЕКТ № 28: Просмотр служб

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы рассмотрите службы (демоны). Raspberry Pi OS использует систему инициализации `systemd` `init`, и она включает утилиту `systemctl`, которую можно использовать для просмотра состояния служб:

```
$ systemctl list-units --type=service --state=running
```

Полученный результат должен быть похож на следующее:

UNIT	LOAD	ACTIVE	SUB	DESCRIPTION
avahi-daemon.service	loaded	active	running	Avahi mDNS/DNS-SD Stack
bluealsa.service	loaded	active	running	BlueZALSA proxy
bluetooth.service	loaded	active	running	Bluetooth service
cron.service	loaded	active	running	Regular background ...
dbus.service	loaded	active	running	D-Bus System Message Bus
dhcpcd.service	loaded	active	running	dhcpcd on all interfaces
getty@tty1.service	loaded	active	running	Getty on tty1
hciuart.service	loaded	active	running	Configure Bluetooth Modems ...
rsyslog.service	loaded	active	running	System Logging Service
ssh.service	loaded	active	running	OpenBSD Secure Shell server
systemd-journald.service	loaded	active	running	Journal Service
systemd-logind.service	loaded	active	running	Login Service
systemd-timesyncd.service	loaded	active	running	Network Time Synchronization
systemd-udevd.service	loaded	active	running	udev Kernel Device Manager
triggerhappy.service	loaded	active	running	triggerhappy global hotkey daemon
user@1000.service	loaded	active	running	User Manager for UID 1000

Если вы автоматически не вернулись в окно терминала, нажмите `Q` в терминале, чтобы выйти из просмотра служб. Чтобы увидеть подробную информацию о конкретной службе, попробуйте выполнить эту команду, используя в качестве примера `cron.service`:

```
$ systemctl status cron.service
```

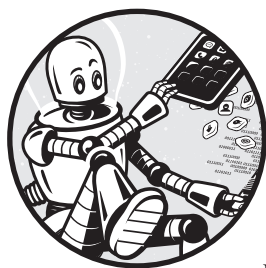
Вывод этой команды включает путь и PID процесса, связанного со службой. В случае с `cron.service` путь в моей системе – `/usr/sbin/cron`, а PID оказался 367.

Другой подход к просмотру процессов служб (демонов) заключается в просмотре всех процессов, которые являются дочерними для `systemd`, т. е. PID 1. Это вполне уместно, поскольку службы запускаются `systemd` и отображаются как дочерние процессы PID 1. Обратите внимание, что этот вывод может включать не только службы, так как процессы-сироты усыновляются PID 1.

```
$ ps --ppid 1
```

# 11

## ИНТЕРНЕТ



До сих пор мы рассматривали вычисления, выполняемые на одном устройстве. В этой и следующей главах мы рассмотрим вычислительные операции, которые выполняются на нескольких устройствах. Мы разберем две значительные инновации в области вычислительных технологий – интернет и Всемирную паутину (World Wide Web, WWW), и это не одно и то же! Эта глава посвящена интернету, и мы начнем с определения ключевых терминов. Затем рассмотрим многоуровневую модель сетей и углубимся в некоторые основополагающие протоколы, используемые в Интернете.

### Определение сетевых терминов

Чтобы обсуждать интернет и сети в целом, сначала необходимо ознакомиться с некоторыми понятиями и терминами, которые мы здесь и рассмотрим.

*Компьютерная сеть* – это система, которая позволяет вычислительным устройствам общаться друг с другом, как показано на рис. 11-1. Устройства в сети могут быть соединены без проводов с помощью таких технологий, как Wi-Fi, которая передает данные с помощью радиоволн. Они также могут быть соединены кабелями на основе медных прово-

дников или оптоволокну. Вычислительные устройства в сети должны использовать общий *протокол передачи данных*, т. е. набор правил, которые описывают, как должен происходить обмен информацией.

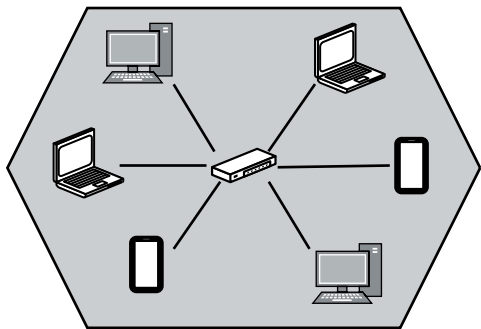


Рис. 11-1. Компьютерная сеть

*Интернет* – это глобальная совокупность компьютерных сетей, связанных между собой и использующих набор общих протоколов. Интернет представляет собой *сеть сетей*, соединяющую сети различных организаций по всему миру, как показано на рис. 11-2.

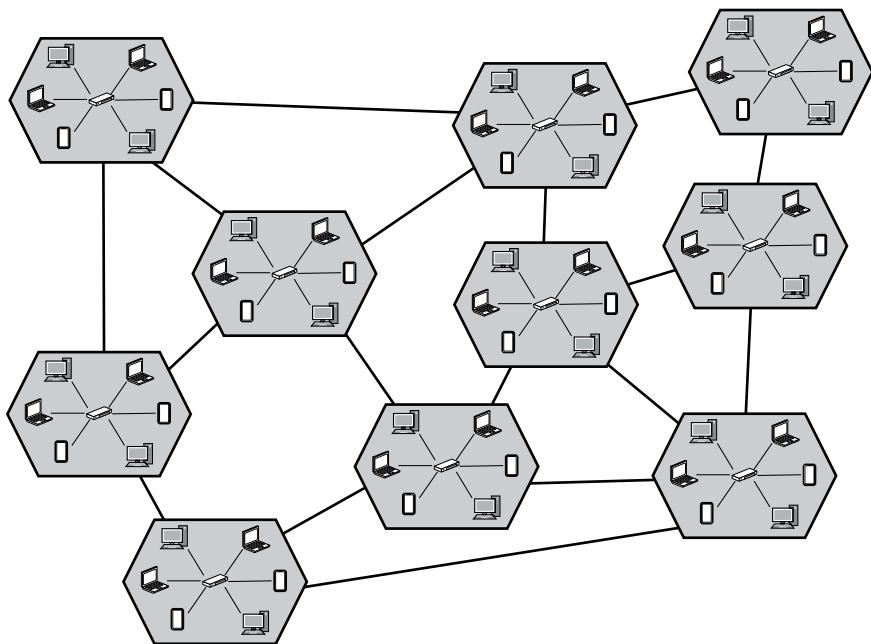


Рис. 11-2. Интернет: сеть сетей

*Хост* или *узел* – это одно вычислительное устройство, подключенное к сети. Хост может выступать в качестве сервера или клиента сети, а иногда может являться и тем и другим. Сетевой *сервер* – это хост, который прослушивает входящие сетевые соединения и предоставляет



услуги другим хостам. Примерами являются веб-сервер и сервер электронной почты. *Клиент* сети – это хост, который с помощью исходящих соединений запрашивает услуги у серверов в сети. Примерами клиентов являются смартфоны или ноутбуки с веб-браузерами или почтовыми приложениями. Клиент делает *запрос* на сервер, а сервер выдает на него *ответ*, как показано на рис.11-3.



Рис. 11-3. Клиент делает запрос на сервер, а сервер отвечает

Термин «сервер», как только что было сказано, относится к любому устройству, которое принимает входящие запросы и предоставляет услуги клиентам. Однако название «сервер» может также относиться к классу компьютерного оборудования, специально предназначенного для работы в качестве сетевого сервера. Эти специализированные компьютеры сконструированы для установки в компьютерные стойки в центрах обработки данных и часто включают аппаратное резервирование и возможности управления, которых нет в обычных ПК. Однако любое компьютерное устройство с соответствующим программным обеспечением может работать в качестве сервера в сети.

## Набор интернет-протоколов

Физического соединения сетей всего мира недостаточно для того, чтобы устройства в этих сетях могли взаимодействовать друг с другом. Все участвующие компьютеры должны взаимодействовать одинаково. *Набор интернет-протоколов* стандартизирует метод связи в интернете, гарантируя, что все устройства в сети говорят на одном языке. Два основных протокола в наборе интернет-протоколов – это *Transmission Control Protocol (TCP)* и *Internet Protocol (IP)*, известные под общим названием *TCP/IP*.

Сетевые протоколы работают в многоуровневой модели, и реализация такой модели называется *сетевым стеком* (не путать со стеком в памяти, о котором говорилось в главе 9). Протоколы на самом нижнем уровне взаимодействуют с базовым сетевым оборудованием, в то время как приложения взаимодействуют с протоколами на верхних уровнях. Протоколы промежуточных уровней предоставляют такие услуги, как адресация и надежная доставка данных. Протоколу определенного уровня не нужно иметь дело со всем сетевым стеком, а только с теми уровнями, с которыми он взаимодействует, что упрощает общую конструкцию. Это еще один пример применения принципа «черного ящика».

Набор интернет-протоколов построен на основе четырехслойной модели. Иногда ее называют *моделью TCP/IP*. Четыре уровня модели TCP/IP, начиная снизу вверх, – это канальный, межсетевой, транспортный и прикладной уровень, как показано на рис. 11-4.

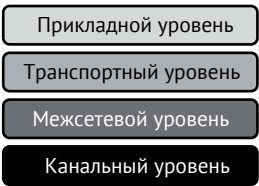


Рис. 11-4. Модель сети с набором интернет-протоколов

### OSI – еще одна сетевая модель

Другой широко используемой моделью сетевых протоколов является модель *Open Systems Interconnection (OSI)*. Модель OSI делит протоколы на семь уровней, а не на четыре. На эту модель часто ссылаются в технической литературе, но интернет основан на наборе интернет-протоколов, поэтому в этой книге основное внимание уделяется модели TCP/IP.

Эти сетевые уровни представляют собой абстракцию, модель, которую мы можем использовать при обсуждении работы интернета. На практике каждый уровень реализуется с помощью конкретных сетевых протоколов. Каждый сетевой уровень представляет собой область определенных обязанностей, и протоколы должны выполнять обязательства назначенного им уровня. В табл. 11-1 представлено описание каждого уровня.

Таблица 11-1. Описание четырех уровней набора интернет-протоколов

Уровень	Описание	Примеры протоколов
	Протоколы, работающие на прикладном уровне, обеспечивают функциональность, специфичную для конкретного приложения, например отправку электронной почты или поиск веб-страницы. Эти протоколы выполняют задачи, которые хотят решить конечные пользователи (или внутренние службы). Протоколы прикладного уровня структурируют данные, используемые при взаимодействии между процессами в сети. Все протоколы нижнего уровня являются как бы «водопроводом» для поддержки прикладного уровня	HTTP, SSH

Уровень	Описание	Примеры протоколов
Транспортный	Протоколы транспортного уровня обеспечивают канал связи для приложений для отправки и получения данных между узлами (хостами). Приложение структурирует данные в соответствии с протоколом прикладного уровня, а затем передает эти данные протоколу транспортного уровня для доставки на удаленный узел	TCP, UDP
Межсетевой	Протоколы межсетевого уровня обеспечивают механизм взаимодействия между сетями. Этот уровень отвечает за идентификацию узлов с помощью адресов и обеспечивает маршрутизацию данных из сети в сеть через интернет. Транспортный уровень опирается на межсетевой уровень для адресации и маршрутизации	IP
Канальный	Протоколы канального уровня обеспечивают взаимодействие в локальной сети. Протоколы этого уровня тесно связаны с типом сетевого оборудования в локальной сети, например с Wi-Fi. Протоколы межсетевого уровня опираются на протоколы канального уровня для взаимодействия в локальной сети	Wi-Fi, Ethernet

Протоколы на каждом уровне взаимодействуют с протоколами на соседних уровнях. Исходящая передача от узла проходит вниз через сетевые уровни, от протокола прикладного уровня к протоколу транспортного уровня, к протоколу межсетевого уровня и, наконец, к протоколу канального уровня. Входящая передача на узел проходит вверх по сетевым уровням в обратном порядке.

Хотя сетевые узлы (такие как клиент или сервер) используют протоколы всех четырех уровней, другие типы сетевого оборудования (такие как коммутаторы и маршрутизаторы) используют только протоколы, относящиеся к более низким уровням. Такие устройства могут выполнять свою работу, не утруждая себя изучением данных протокола более высокого уровня, содержащихся в сетевой передаче.

Исходящий запрос от клиента к серверу и его связь с сетевыми уровнями показана на рис. 11-5.

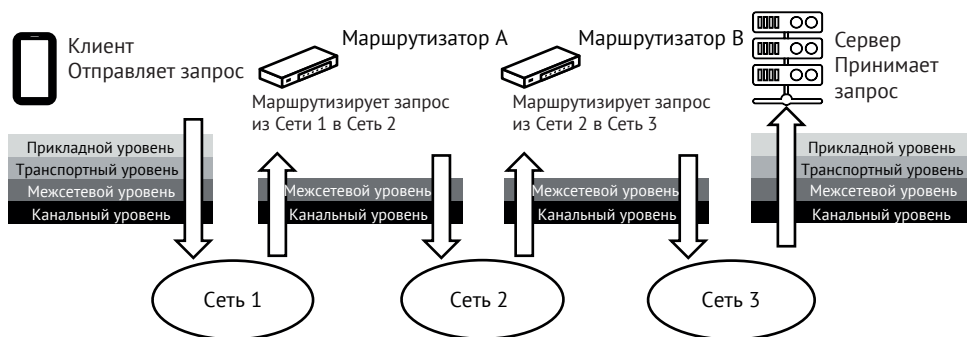


Рис. 11-5. Сетевой запрос проходит через различные уровни сети

Давайте пройдемся по потоку, показанному на рис. 11-5. Приложение на клиентском устройстве формирует запрос, используя протокол прикладного уровня. Этот запрос передается протоколу транспортного уровня, затем протоколу межсетевого уровня и, наконец, протоколу канального уровня. Все это происходит на клиентском устройстве. На этом этапе запрос передается в локальную сеть, обозначенную на схеме как Сеть 1. Запрос проходит свой путь через интернет, переходя от сети к сети. В нашем примере маршрутизатор А направляет запрос из Сети 1 в Сеть 2, а маршрутизатор В направляет запрос из Сети 2 в Сеть 3. Когда запрос достигает сервера назначения, он проходит свой путь через сетевые протоколы, начиная с протокола канального уровня и заканчивая протоколом прикладного уровня. Процесс, запущенный на сервере, получает запрос, который отформатирован в соответствии с протоколом прикладного уровня, первоначально использованным клиентом. Серверный процесс интерпретирует запрос и отвечает соответствующим образом.

Давайте теперь рассмотрим каждый уровень, начиная с самого нижнего.

## Канальный уровень

Самый нижний уровень набора интернет-протоколов – *канальный уровень*. Физические и логические соединения между узлами называются сетевыми *каналами*. Протоколы канального уровня используются устройствами в одной сети для связи друг с другом. Каждое устройство на канале имеет сетевой адрес, который однозначно идентифицирует его. Для многих протоколов канального уровня этот адрес известен как *media access control address* (или *MAC-адрес*). Данные канального уровня разделены на небольшие блоки, известные как *кадры*, каждый из которых включает заголовок, описывающий кадр, полезную нагрузку из данных и, наконец, хвостовую часть кадра, используемую для обнаружения ошибок. Это показано на рис. 11-6.



Рис. 11-6. Кадр канального уровня

Заголовок кадра содержит MAC-адреса источника и назначения. Заголовок также включает описание типа данных, передаваемых в секции данных кадра.

Если в вашем доме есть сеть Wi-Fi, то Wi-Fi является каналом между узлами этой сети. Протокол Wi-Fi, определенный спецификацией IEEE 802.11, не знает и не заботится о том, какой тип данных передается по беспроводной сети, он просто обеспечивает связь между устройствами. Каждое устройство, подключенное к сети Wi-Fi, имеет MAC-адрес и получает кадры, отправленные на его адрес. MAC-адреса можно использовать только в локальной сети, компьютер в удаленной сети не может напрямую отправлять данные на MAC-адрес в вашей локальной сети.

Другой важной технологией канального уровня является *Ethernet*, используемый для проводных физических соединений. Ethernet определен стандартом IEEE 802.3. В Ethernet обычно используется кабель с парами медных проводов внутри, который заканчивается разъемом, известным как *RJ45*, показанным на рис. 11-7.

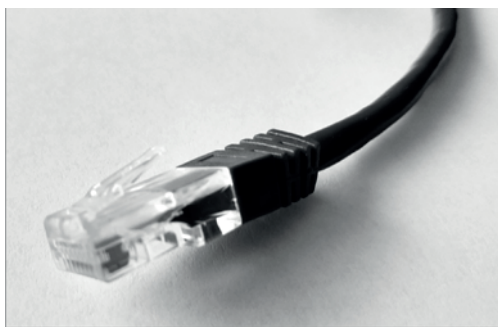


Рис. 11-7. Кабель с разъемом RJ45, обычно используемый для Ethernet

Все устройства, подключенные к интернету, участвуют в работе канального уровня. Это необходимо, поскольку именно канальный уровень обеспечивает подключение (проводное или беспроводное) к локальной сети. Узел, например ноутбук или смартфон, участвует во всех уровнях, но некоторые сетевые устройства работают только на канальном уровне. Самый простой пример – концентратор (он же хаб). *Сетевой концентратор* – это сетевое устройство, которое соединяет несколько устройств в локальной сети без какой-либо информации, касающейся отправляемых кадров. Простой концентратор может предоставлять несколько портов Ethernet для подключения устройств. Концентратор просто ретранслирует каждый кадр, полученный на одном физическом порту, на все остальные порты. Более умным устройством канального уровня является *сетевой коммутатор*, который изучает MAC-адреса в получаемых им кадрах и отправляет эти кадры на физический порт, к которому подключено устройство с MAC-адресом назначения.

**ПРИМЕЧАНИЕ**

Обратитесь к проекту № 29 на стр. 313, где рассматриваются устройства канального уровня и MAC-адреса.

## Межсетевой уровень

*Межсетевой уровень* позволяет данным выходить за пределы локальной сети. Основной протокол, используемый на этом уровне, называется Internet Protocol (IP). Он обеспечивает *маршрутизацию* – процесс определения пути для данных, передаваемых между сетями. Каждому узлу в интернете присваивается IP-адрес, номер, который однозначно идентифицирует узел в глобальной сети интернет. Можно также иметь частные IP-адреса, которые не отображаются непосредственно в сети. IP-адреса обычно назначаются сервером в локальной сети, и IP-адрес устройства может меняться при подключении к новой сети. Подробнее о назначении адресов и частных IP-адресах мы поговорим позже.

Данные, передаваемые через межсетевой уровень, называются *пакетом*, который заключен в кадр канального уровня. Рисунок 11-8 иллюстрирует идею о том, что пакет помещается в раздел данных кадра.

Заголовок IP-пакета содержит IP-адрес источника и IP-адрес назначения. Заголовок также содержит информацию, описывающую пакет, такую как используемая версия IP и длина заголовка. Раздел данных IP-пакета содержит полезную нагрузку, которую несет уровень IP.

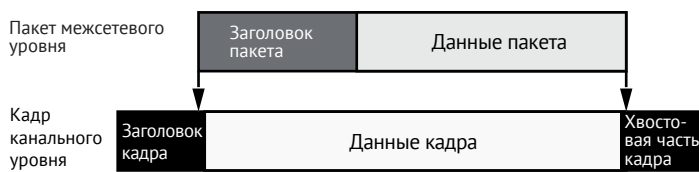


Рис. 11-8. Пакет содержится в секции данных кадра

Сегодня в интернете используются две версии Internet Protocol. *Internet Protocol Version 4 (IPv4)* является доминирующей версией, а другая активная версия – это *Internet Protocol Version 6 (IPv6)*. Вам может быть интересно, что произошло с протоколом IPv5. Такого протокола никогда не существовало, но экспериментальный протокол под названием Internet Stream Protocol определил свою IP-версию как 5, и поэтому IPv5 был пропущен при разработке преемника IPv4. Существенное различие между IPv4 и IPv6 заключается в размере IP-адреса. Адрес IPv4 имеет длину 32 бита, в то время как адрес IPv6 составляет 128 бит. Это различие позволяет использовать гораздо большее количество адресов в IPv6. Это изменение размера адреса призвано помочь решить проблему с относительно небольшим количеством адресов IPv4. В этой книге мы сосредоточимся на адресах IPv4 (и будем называть их просто *IP-адресами*), поскольку они по-прежнему являются основным средством адресации в интернете.

32-битный IP-адрес обычно отображается в точно-десятичном виде, т. е. 32 бита разделяются на четыре группы по 8 бит в каждой, 8-битные числа отображаются в десятичной системе счисления (а не в шестнадцатеричной или двоичной), и четыре десятичных числа разделяются точками. Пример IP-адреса, отображаемого в точно-десятичном виде – 192.168.1.23. Каждое 8-битное десятичное число можно назвать *октетом*.

Компьютеры, подключенные к одной локальной сети, имеют IP-адреса, начинающиеся с одинаковых старших битов, и считаются находящимися в одной *подсети*. Компьютеры, находящиеся в одной подсети, могут напрямую взаимодействовать друг с другом на канальном уровне, поскольку они работают в одной физической сети. Компьютеры, находящиеся в разных подсетях, должны передавать свой трафик через маршрутизатор (роутер) – устройство, соединяющее подсети и работающее на межсетевом уровне.

Подсеть делит IP-адрес на две части: *сетевой префикс*, который разделяет все устройства одной подсети, и *идентификатор узла*, который уникален для каждого узла в этой подсети. Количество битов, входящих в сетевой префикс, зависит от конфигурации сети.

Рассмотрим пример. Предположим, что подсеть использует 24-битный сетевой префикс, оставляя нам 8 бит для представления узла. Также предположим, что узел в этой подсети использует IP-адрес из примера, приведенного ранее, – 192.168.1.23. С данным IP-адресом и сетевым префиксом IP-адрес можно разбить на части, как показано на рис. 11-9.

В этом примере все узлы в локальной подсети имеют IP-адрес, который начинается с 192.168.1. Каждый узел имеет свое значение последнего октета, а данному конкретному узлу присваивается 23. В этом примере используется 24-битная длина префикса, т. е. префикс точно совпадает с первыми тремя октетами IP-адреса.

Это хороший пример, но длина префикса не всегда совпадает с границами октета. Например, 25-битный префикс будет также включать первый бит последнего октета, оставляя только 7 бит для идентификации узла.

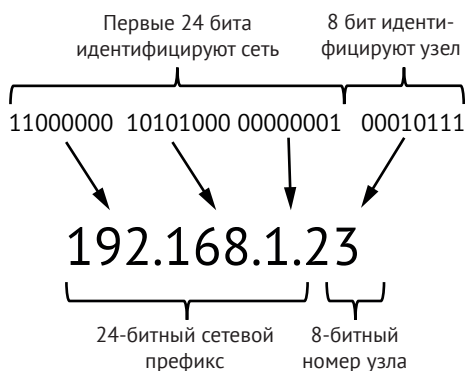


Рис. 11-9. Пример IP-адреса с использованием 24-битного сетевого префикса

Количество битов, зарезервированных для сетевого префикса, обычно выражается одним из двух способов. Метод *бесклассовой адресации* (*Classless Inter-Domain Routing, CIDR*) указывает IP-адрес, за которым следует косая черта (/), а затем количество битов, используемых для сетевого префикса. В нашем примере это будет 192.168.1.23/24. Другим распространенным способом представления количества битов префикса является *маска подсети* – 32-битное число, в котором двоичная 1 ис-

пользуется для каждого бита, являющегося частью сетевого префикса, а 0 – для каждого бита, являющегося частью номера узла. Маски подсетей также записываются в точечно-десятичном виде, поэтому в нашем примере с 24-битным сетевым префиксом маска подсети будет равна 255.255.255.0, как показано на рис. 11-10.



Рис. 11-10. 24-битный сетевой префикс, выраженный в виде маски подсети

Давайте посмотрим, как это может быть полезно на практике. Допустим, ваш компьютер имеет IP-адрес 192.168.0.133 и маску подсети 255.255.255.224 или по методу CIDR записывается как 192.168.0.133/27. Ваш компьютер хочет подключиться к другому компьютеру с IP-адресом 192.168.0.84. Как упоминалось ранее, два компьютера могут общаться напрямую, если они находятся в одной подсети, а если нет, то они должны проходить через маршрутизатор. Поэтому ваш компьютер должен определить, находится ли другой компьютер в той же подсети. Как он может это сделать?

Выполнение побитового логического И из IP-адреса и его маски подсети дает первый адрес в подсети. Этот первый адрес, где все биты номера узла равны 0, служит идентификатором для самой подсети. Его обычно называют *сетевым идентификатором*. Два компьютера, имеющих общий сетевой идентификатор, находятся в одной подсети. Узел может выполнить эту операцию логического И как для своего собственного IP-адреса, так и для IP-адреса, к которому он хочет подключиться, чтобы проверить, имеют ли они общий сетевой идентификатор и, следовательно, находятся ли они в одной подсети. Давайте попробуем сделать это с примером IP-адреса нашего компьютера, как показано здесь:

---

IP = 192.168.0.133	= 11000000.10101000.00000000.10000101
MASK = 255.255.255.224	= 11111111.11111111.11111111.11100000
AND = 192.168.0.128	= 11000000.10101000.00000000.10000000 = The network ID

---

Теперь выполним ту же операцию для второго компьютера из нашего примера:

---

IP = 192.168.0.84	= 11000000.10101000.00000000.01010100
MASK = 255.255.255.224	= 11111111.11111111.11111111.11100000
AND = 192.168.0.64	= 11000000.10101000.00000000.01000000 = The network ID

---



Как видно из примера, в результате этой операции было получено два разных сетевых идентификатора (192.168.0.128 и 192.168.0.64). Это означает, что второй компьютер не находится в той же подсети, что и ваш. Чтобы обмениваться информацией, эти компьютеры должны передавать свои сообщения через маршрутизатор, соединяющий две подсети.

### УПРАЖНЕНИЕ 11-1:

#### Какие IP находятся в одной подсети?

Находится ли IP-адрес 192.168.0.200 в той же подсети, что и ваш компьютер? Предположим, что ваш компьютер имеет IP-адрес 192.168.0.133 и маску подсети 255.255.255.224.

Вот другой способ взглянуть на это: сетевой префикс описывает диапазон адресов, которые могут использоваться в подсети. Первый адрес в этом диапазоне определяется как биты сетевого префикса, за которыми следуют все двоичные 0 для идентификатора узла. Продолжая наш пример с компьютером с адресом 192.168.0.133, первый адрес в его подсети – 192.168.0.128. Последний адрес в диапазоне – это биты сетевого префикса, за которыми следуют все двоичные 1 для идентификатора узла. В нашем примере это 192.168.0.159. Первый и последний адреса имеют особое значение – первый идентифицирует сеть, последний является *широковещательным адресом* (используется для отправки сообщения всем узлам в подсети). Все адреса между ними могут использоваться для узлов в подсети. IP-адрес из нашего примера 192.168.0.133 явно находится в этом диапазоне (от 192.168.0.128 до 192.168.0.159), в то время как другой компьютер с IP-адресом 192.168.0.84 находится вне этого диапазона.

Вы также можете использовать количество битов, зарезервированных для идентификатора узла, чтобы определить, сколько IP-адресов доступно для узлов в подсети.

В нашем примере 27 бит зарезервированы для сетевого префикса, оставляя 5 бит для идентификаторов узлов. Эти 5 бит дают нам 32 возможных адреса узлов, так как  $2^5$  равно 32. Однако, как упоминалось ранее, первый и последний адреса имеют специальное назначение, поэтому реально только 30 узлов могут быть идентифицированы с помощью такого сетевого префикса. Это согласуется с нашими предыдущими выводами, что первый идентификатор узла равен 128, а  $128 + 32$  дает нам 160, что является первым адресом в следующей подсети, поэтому 159 будет последним номером узла в нашем диапазоне<sup>1</sup>.

<sup>1</sup> Надо отметить, что автор явно искусственно усложнил задачу: в реальности по крайней мере домашние маршрутизаторы обычно настроены в соответствии со значением маски 255.255.255.0, т. е. в подсети вашего Wi-Fi-роутера могут находиться  $256 - 2 = 254$  узла с номерами от 1 до 254 (см. пример исследования конфигурации в проекте № 30). – *Прим ред.*

## Транспортный уровень

Транспортный уровень обеспечивает канал связи, который приложения могут использовать для отправки и получения данных. Существует два широко используемых протокола транспортного уровня: *Transmission Control Protocol (TCP)* и *User Datagram Protocol (UDP)*. TCP обеспечивает надежное соединение между двумя узлами. Он гарантирует, что ошибки сведены к минимуму, данные поступают в порядке, потерянные данные отправляются повторно и т. д. Данные, передаваемые с помощью TCP, называются *сегментами*. С другой стороны, UDP – это протокол «негарантированной доставки», т. е. его доставка ненадежна. UDP предпочтительнее, когда скорость важнее надежности. Данные, передаваемые с помощью UDP, называются *датаграммами*. Оба протокола занимают свое место, но для простоты изложения в оставшейся части главы я буду рассматривать только TCP. Рисунок 11-11 иллюстрирует идею о том, что сегмент TCP помещается в раздел данных пакета, который в свою очередь помещается в раздел данных кадра.

Как мы видели ранее, канальный уровень включает MAC-адрес назначения в заголовок кадра для идентификации интерфейса локальной сети, а межсетевой уровень включает IP-адрес назначения в заголовок пакета для идентификации узла в интернете. Этой информации достаточно, чтобы доставить пакет к определенному устройству в интернете. Когда пакет достигает узла назначения, заголовок транспортного уровня включает *номер порта* сети назначения, идентифицирующего конкретную службу или процесс, который и получит данные. Узел с одним IP-адресом может иметь несколько активных портов, каждый из которых используется для выполнения различных видов деятельности в сети.

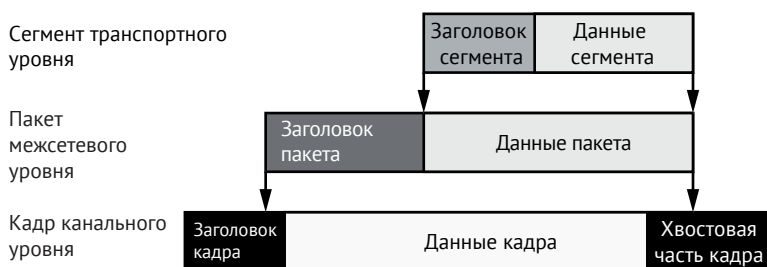


Рис. 11-11. Сегмент TCP содержится в разделе данных IP пакета

Если использовать аналогию, то IP-адрес похож на почтовый адрес офисного здания, а номер сетевого порта – на номер кабинета сотрудника в этом офисном здании.

IP-адрес однозначно идентифицирует хост-компьютер, так же как почтовый адрес однозначно идентифицирует офисное здание. С использованием интернет-протокола пакет может быть доставлен на хост так же, как по-

ссылка может быть доставлена в офисное здание. Однако, как только пакет попадает на компьютер, операционная система должна решить, что с ним делать. Пакет предназначен не для самой ОС, а для какого-то процесса, запущенного на компьютере. Точно так же посылка, пришедшая в офисное здание, скорее всего, предназначена не для разбирающего почту работника, а для кого-то другого в здании. Операционная система проверяет номер порта и доставляет входящие данные процессу, прослушивающему указанный порт, точно так же, как разбирающий почту работник проверяет имя или номер офиса на посылке, чтобы доставить ее нужному человеку.

Сетевые порты в диапазоне от 0 до 1 023 называются *общезвестными портами*, в то время как порты в диапазоне от 1 024 до 49 151 могут быть зарегистрированы в Internet Assigned Numbers Authority (IANA) и называются *зарегистрированными портами*. Порты со значением более 49 151 являются *динамическими портами*. Технически любой процесс с достаточными привилегиями может прослушивать любой порт, который еще не используется в системе, потенциально игнорируя типичный вариант использования данного номера порта. Однако, когда клиентское приложение хочет подключиться к удаленной службе на другом компьютере, ему необходимо знать, какой порт использовать, поэтому имеет смысл стандартизировать номера портов. Например, веб-серверы обычно используют порт 80 и порт 443 (для шифрованных соединений). Веб-браузер предполагает, что он должен использовать порт 80 или 443, если нет других указаний.

## УПРАЖНЕНИЕ 11-2:

### Исследование распространенных портов

Найдите номера портов для распространенных протоколов прикладного уровня. Каковы номера портов для системы доменных имен (Domain Name System, DNS), для безопасной оболочки (Secure Shell, SSH) и простого протокола передачи почты (Simple Mail Transfer Protocol, SMTP)? Эту информацию можно найти в интернете с помощью поиска или посмотреть в реестре IANA здесь: <http://www.iana.org/assignments/port-numbers>. В списках IANA иногда используются неожиданные термины для названия служб. Например, DNS указывается просто как «домен» («domain»).

Серверы используют общезвестные порты, чтобы облегчить клиентам подключение. Однако в большинстве случаев сетевая связь является двусторонней (клиент посылает запрос, а сервер отвечает), поэтому клиент должен иметь открытый порт, чтобы иметь возможность получать данные от сервера. Клиенту достаточно временно открыть такой порт, чтобы осуществить взаимодействие с сервером. Такие порты называются *эфемерными* и назначаются сетевыми компонентами операционной системы. Например, клиентский веб-браузер подключается к веб-серверу по порту 80, и эфемерный порт на клиенте также открыт, допустим, порт

номер 61 348. Клиент отправляет свой веб-запрос на порт 80 на сервере, а сервер отправляет свой ответ на порт 61 348 на клиенте.

IP-адрес плюс номер порта образуют *конечную точку*, а экземпляр конечной точки называется *сокетом*. Сокет может прослушивать новые соединения или представлять установленное соединение. Если несколько клиентов подключаются к одной конечной точке, каждый из них имеет свой собственный сокет.

#### ПРИМЕЧАНИЕ

Смотрите проект № 31 на стр. 316, где вы сможете посмотреть использование портов вашего Raspberry Pi.

## Прикладной уровень

*Прикладной уровень* – последний, самый верхний уровень набора интернет-протоколов. В то время как три нижних уровня обеспечивают общую основу для коммуникации через интернет, протоколы прикладного уровня направлены на выполнение конкретной задачи. Например, веб-серверы используют *протокол передачи гипертекста (HyperText Transfer Protocol, HTTP)* для получения и обновления веб-контента. Серверы электронной почты используют *простой протокол передачи почты (Simple Mail Transfer Protocol, SMTP)* для отправки и получения сообщений электронной почты. Серверы передачи файлов используют *протокол передачи файлов (File Transfer Protocol, FTP)* для, как вы уже догадались, передачи файлов! Другими словами, на прикладном уровне мы получаем протоколы, которые описывают поведение приложений, в то время как нижние уровни стека являются «водопроводом», который позволяет приложениям делать то, что они хотят делать посредством интернета. На рис. 11-12 представлен полный вид четырех уровней.

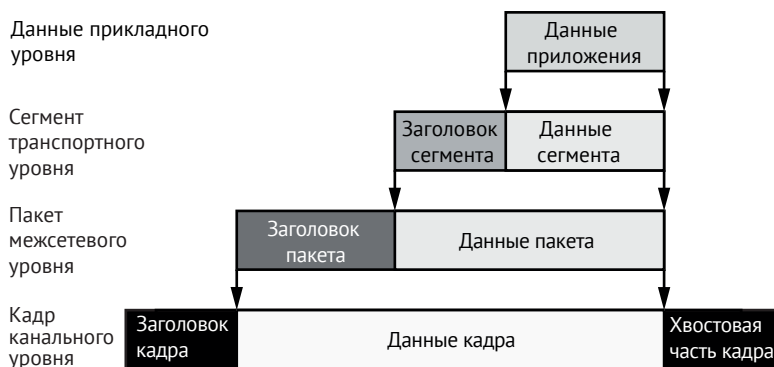


Рис. 11-12. Данные прикладного уровня содержатся в секции данных сегмента

На рис. 11-12 показано, как каждый уровень помещается в полезную нагрузку данных нижнего слоя. Собрав все слои вместе на рис. 11-13, мы можем увидеть представление того, что содержится в кадре, отправленном на устройство в интернете.

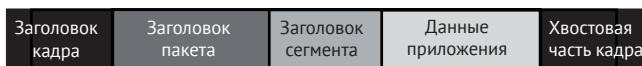


Рис. 11-13. Кадр, содержащий IP пакет, сегмент TCP и данные приложения

Мы рассмотрели содержимое сетевого кадра снизу вверх, начиная с уровня, наиболее близкого к аппаратному. Когда кадр принимается узлом, он обрабатывается им в том же порядке, начиная с канального уровня и заканчивая прикладным. И наоборот, когда кадр отправляется с узла, он собирается в обратном порядке. Процесс подготавливает данные приложения, которые заключаются в сегмент, пакет и, наконец, в кадр.

## Путешествие по интернету

Теперь, когда вы знакомы с каждым из четырех уровней сетевой модели TCP/IP, давайте рассмотрим пример того, как данные путешествуют по интернету. Мы увидим, как различные устройства на этом пути взаимодействуют с каждым из сетевых уровней. На рис. 11-14 показано, как клиент в левом верхнем углу взаимодействует с сервером в левом нижнем углу.

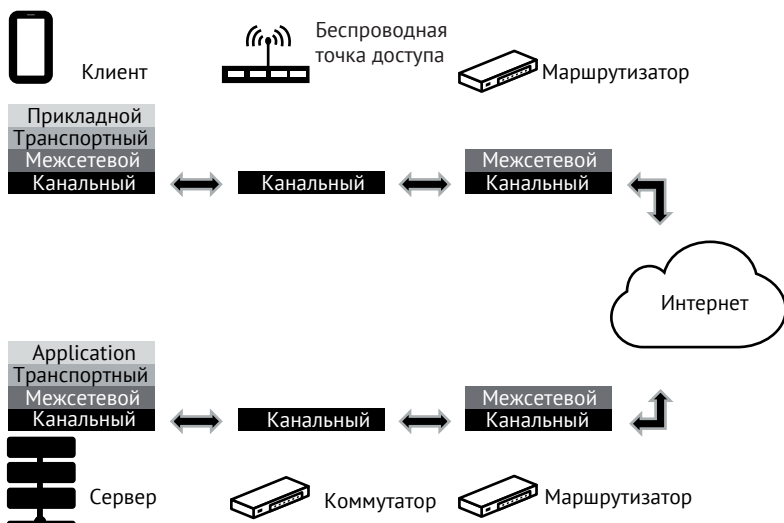


Рис. 11-14. Различные устройства взаимодействуют на разных уровнях сетевого стека

Я представлю сценарий, показанный на рис. 11-14. Клиентское устройство (верхний левый угол диаграммы) подключено к беспроводной сети Wi-Fi. Эта сеть подключена к интернету через маршрутизатор. Где-то в другом месте находится сервер (нижняя левая часть диаграммы).

мы), который имеет проводное подключение к интернету через коммутатор и маршрутизатор. Пользователь клиентского устройства открывает веб-браузер и запрашивает веб-страницу, расположенную на сервере. Для простоты предположим, что клиент уже знает IP-адрес сервера.

Веб-браузер на клиенте «говорит» на языке HTTP, протоколе прикладного уровня, поэтому он формирует HTTP-запрос, предназначенный для сервера назначения. Затем браузер передает HTTP-запрос программному стеку TCP/IP операционной системы, запрашивая доставку данных на сервер, в частности IP-адрес сервера и порт 80, стандартный порт для HTTP. Программный стек TCP/IP на клиентской операционной системе затем упаковывает полезную информацию HTTP в сегмент TCP (транспортный уровень), устанавливая порт назначения на 80 в заголовке сегмента.

При необходимости TCP делит данные прикладного уровня на несколько сегментов, каждый из которых имеет свой собственный заголовок. Затем программное обеспечение межсетевого уровня на клиенте упаковывает сегмент TCP в IP-пакет, который включает IP-адрес назначения сервера в заголовок пакета. При необходимости IP разделяет пакет на несколько меньших фрагментов для подготовки к передаче по каналному уровню. На канальном уровне клиента IP-пакет упаковывается в кадр с MAC-адресом локального маршрутизатора в заголовке. Этот кадр передается по беспроводной сети с помощью оборудования Wi-Fi клиентского устройства.

Беспроводная точка доступа получает этот кадр. Точка доступа, работая на канальном уровне, отправляет кадр маршрутизатору. Маршрутизатор изучает пакет на межсетевом уровне, чтобы определить IP-адрес назначения. Чтобы достичь сервера, запрос должен пройти через несколько маршрутизаторов в интернете. Локальный маршрутизатор упаковывает пакет в новый кадр с новым MAC-адресом назначения (адрес следующего маршрутизатора) и отправляет новый кадр дальше. Этот процесс маршрутизации проходит через несколько маршрутизаторов в интернете, пока запрос не достигнет маршрутизатора в подсети, к которой подключен сервер.

Последний маршрутизатор упаковывает пакет в кадр, подходящий для локальной сети сервера. Заголовок этого кадра включает MAC-адрес сервера. Коммутатор в подсети сервера смотрит на MAC-адрес в кадре и пересылает кадр через соответствующий физический порт. Коммутатору нет необходимости смотреть на более высокие уровни. Сервер получает кадр, и драйвер сетевого интерфейса передает TCP/IP-пакет в программный стек TCP/IP, который в свою очередь передает данные HTTP процессу, прослушивающему TCP-порт 80. Программное обеспечение веб-сервера, прослушивающее порт 80, обрабатывает запрос. Обработка включает в себя ответ клиенту, и для этого весь процесс повторяется снова, только в обратном порядке.

#### **ПРИМЕЧАНИЕ**

*Смотрите проект № 32 на стр. 318, где вы сможете проследить маршрут от вашего Raspberry Pi до хоста в интернете.*

## Основополагающие возможности интернета

В то время как TCP/IP обеспечивает необходимые условия для надежной передачи данных через интернет, другие протоколы предоставляют дополнительные фундаментальные возможности интернета. Эти возможности реализуются в виде протоколов прикладного уровня. Давайте рассмотрим два таких протокола (DHCP и DNS) и систему преобразования IP-адресов (NAT).

### Протокол динамической настройки узла (DHCP)

Каждому узлу в интернете необходим IP-адрес, маска подсети и IP-адрес его маршрутизатора (также называемого *шлюзом по умолчанию*) для связи с другими узлами. Как назначаются IP-адреса? Устройству может быть присвоен *статический IP-адрес*, который требует, чтобы кто-то отредактировал конфигурацию устройства и задал его IP-адрес вручную. Это иногда полезно, но требует от пользователя убедиться, что данный IP-адрес еще не занят и действителен для данной подсети.

Большинство конечных пользователей не обладают достаточным опытом для ручной настройки параметров IP на своих устройствах и не хотят иметь дело с этим трудоемким процессом. К счастью, большинство IP-адресов назначается динамически с помощью *протокола динамической настройки узла* (*Dynamic Host Configuration Protocol, DHCP*). Благодаря DHCP, когда устройство подключается к сети, оно получает IP-адрес и соответствующую информацию без вмешательства пользователя.

Чтобы DHCP был доступен в сети, одно из устройств в сети должно быть настроено как *DHCP-сервер*. Этот сервер имеет запас IP-адресов, которые он может назначать устройствам. Работа DHCP показана на рис. 11-15.

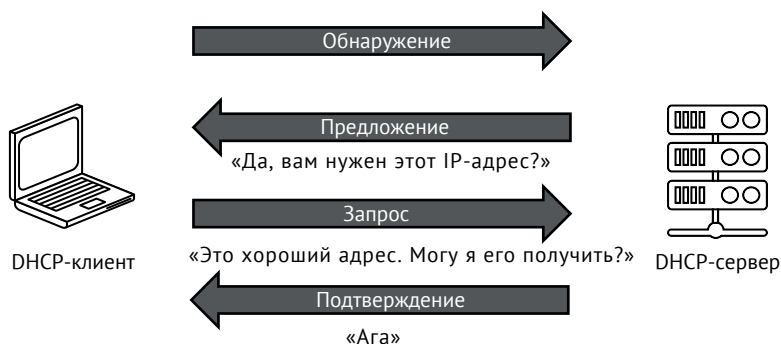


Рис. 11-15. Диалог по DHCP

Давайте рассмотрим рис. 11-15. Когда устройство подключается к сети, оно рассылает сообщение, чтобы обнаружить какие-нибудь DHCP-серверы. *Широковещательное сообщение* – это особый вид пакета, который адресован всем узлам локальной сети. Когда сервер DHCP получает это широковещательное сообщение, он предлагает IP-адрес



клиентскому устройству. Если клиент хочет принять предложенный IP-адрес, он отвечает серверу запросом на этот адрес. Затем DHCP-сервер подтверждает запрос, и IP-адрес назначается клиенту. IP-адрес предоставляется клиенту *в аренду*, и срок его действия истекает, если клиент не продлевает аренду.

**ПРИМЕЧАНИЕ**

*Обратитесь к проекту № 33 на стр. 319, где вы сможете увидеть IP-адрес, который ваш Raspberry Pi арендовал с помощью DHCP.*

## **Частные IP-адреса и преобразование сетевых адресов**

Количество доступных IP-адресов ограничено, поэтому большинство домашних интернет-провайдеров (internet service providers, ISP) назначает клиенту только один IP-адрес. Этот IP-адрес присваивается устройству, непосредственно подключенному к сети провайдера, обычно маршрутизатору. Однако многие клиенты имеют несколько устройств в своей домашней сети. Давайте рассмотрим, как несколько устройств могут совместно использовать один публичный IP-адрес, используя частные IP-адреса и преобразование сетевых адресов.

Определенные диапазоны IP-адресов считаются *частными IP-адресами* – адресами, предназначенными для использования в частных сетях, например в домах или офисах, где устройства не подключены напрямую к интернету. Любой адрес, который соответствует шаблону 10.x.x.x, 172.16.x.x или 192.168.x.x, является частным IP-адресом. Любой может использовать эти диапазоны IP-адресов, не спрашивая разрешения. Загвоздка в том, что частные IP-адреса немаршрутизируемые, т. е. их нельзя использовать в публичном интернете. Сервер DHCP в домашней сети может назначать эти адреса, не беспокоясь о том, что другие сети используют те же адреса. В отличие от публичных IP-адресов, которые должны быть уникальными, частные IP-адреса предназначены для одновременного использования в нескольких частных сетях. Не имеет значения, если несколько сетей используют одни и те же адреса, поскольку эти адреса все равно никогда не будут видны за пределами частной сети. Частные IP-адреса решают проблему предоставления провайдером только одного публичного IP-адреса для дома или предприятия, но как они могут быть полезны, если они не маршрутизируются в интернете?

*Преобразование сетевых адресов (Network Address Translation, NAT)* позволяет устройствам в частной сети, часто домашней, использовать один и тот же публичный IP-адрес в интернете. Когда пакеты проходят через маршрутизатор NAT, маршрутизатор изменяет информацию об IP-адресе в этих пакетах. Когда пакет, исходящий из частной домашней сети, попадает на NAT-маршрутизатор, он изменяет поле IP-адреса источника, чтобы он соответствовал публичному IP-адресу, как показано на рис. 11-16.



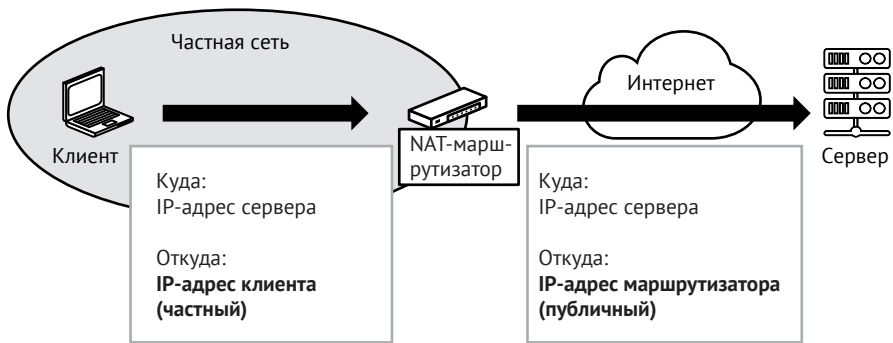


Рис. 11-16. NAT-маршрутизатор заменяет частные IP-адреса собственным публичным IP-адресом

Когда ответ приходит обратно на маршрутизатор, он устанавливает IP-адрес назначения на частный адрес узла, который отправил запрос. Таким образом, весь трафик из дома кажется исходящим с одного публичного IP-адреса, даже если на самом деле в частной сети работает несколько устройств. NAT также имеет дополнительное преимущество в плане безопасности: устройства в частной сети не имеют прямого доступа к публичному интернету, поэтому злоумышленник в интернете не может инициировать подключение непосредственно к частному устройству. Большинство маршрутизаторов, продаваемых потребителям для домашнего использования, являются NAT-маршрутизаторами, часто со встроенными возможностями беспроводной точки доступа.

Частные IP-адреса полезны не только для домашних сетей, но и для предприятий, которые не хотят, чтобы их компьютеры находились в открытом доступе в сети. Многие корпоративные сети используют *прокси-сервер*, а не NAT-маршрутизатор.

Прокси-сервер похож на NAT-маршрутизатор тем, что позволяет устройствам в частной сети получать доступ в интернет. Но прокси-сервер отличается тем, что обычно работает на прикладном уровне, а не на межсетевом. Прокси-серверы также обычно предоставляют дополнительные функции, такие как аутентификация пользователей, регистрация трафика и фильтрация содержимого.

#### ПРИМЕЧАНИЕ

Обратитесь к проекту № 34 на стр. 320, где вы сможете узнать, является ли назначенный вашему устройству IP-адрес публичным или частным IP-адресом.

## Система доменных имен

Мы узнали, что узлы (хосты) в интернете идентифицируются по IP-адресам. Однако большинство пользователей интернета редко, если вообще когда-либо, имеют дело непосредственно с IP-адресами. Хотя IP-адреса хорошо подходят для компьютеров, они не очень удобны для пользователей. Никто не хочет запоминать наборы из четырех цифр,

разделенных точками. К счастью, у нас есть *система доменных имен* (*Domain Name System, DNS*), которая облегчает нам работу. DNS – это интернет-сервис, который сопоставляет имена с IP-адресами. Это позволяет нам обращаться к хосту по имени, например *www.example.com*, а не по IP-адресу.

Полное DNS-имя компьютера известно как *полное доменное имя*, или *FQDN* (*Fully Qualified Domain Name*). Такое имя, как *travel.example.com*, является FQDN. Это имя состоит из короткого локального имени хоста (*travel*) и суффикса домена (*example.com*). Термин «имя хоста» часто используется как синоним для обозначения либо короткого имени компьютера, либо FQDN. В дальнейшем в этом разделе мы будем использовать *имя хоста* для обозначения FQDN компьютера. *Домен*, например *example.com*, представляет собой группу сетевых ресурсов, управляемых организацией. И *example.com*, и *travel.example.com* являются доменными именами. Первое представляет сетевой домен, второе – конкретный хост в этом домене<sup>1</sup>.

Программное обеспечение должно иметь возможность запрашивать DNS для преобразования имен хостов в IP-адреса – это называется *разрешением* имени хоста. Для обеспечения этой функциональности хосты конфигурируются со списком IP-адресов DNS-серверов. Этот список обычно предоставляется DHCP, и обычно он состоит из DNS-серверов,

---

<sup>1</sup> Несколько забегая вперед, тем не менее стоит сразу уточнить, что, согласно принятой спецификации DNS, нет каких-то специальных позиций, представляющих «конкретный хост», выделенных в доменном имени. Каждая составляющая FQDN является доменным именем, отличающимся уровнем и, как правило, одновременно адресом соответствующего хоста. В приведенном автором примере *com* является доменом верхнего (первого) уровня, *example* – второго, *travel* – третьего. Любое полное доменное имя указывает на определенный хост в сети (сервер, маршрутизатор либо просто компьютер), обладающий собственным публичным IP-адресом и является, по сути, псевдонимом этого адреса, введенным для удобства доступа (подробности см. далее в тексте). Несколько упрощая, можно сказать, что имена «промежуточных» уровней нередко также являются адресами таких хостов, несущих свои собственные ресурсы. Это, например, отдельный сайт *example.com*, который может быть главным по отношению к *travel.example.com*, а может быть и никак не связан с ним или вовсе отсутствовать. Исключением являются только домены первого уровня (*top-level domain, TLD*, см. далее), указывающие на хосты, могущие нести только DNS-информацию о доменах второго уровня, но не самостоятельные ресурсы. В этой связи часто говорят о «зоне» – в случае приведенного примера можно сказать, что домен второго уровня *example* размещен в зоне *com*, а, например, в доменном имени *yandex.ru* домен *yandex* размещен в зоне *ru*. Уровней доменных имен может быть в принципе сколько угодно, причем только организация или частное лицо, владеющие доменом, например, второго уровня (*example*), вправе создавать в нем домены третьего уровня (*travel*), но владелец домена первого уровня (*com*) не вправе вмешиваться в эту деятельность (средневековое правило «вассал моего вассала не мой вассал»). В том числе это правило относится и к организациям ICANN и IANA – распорядителям корневого домена (см. далее), которые не вправе вмешиваться в деятельность организаций, ответственных за домены первого уровня (TLD). – *Прим. ред.*

поддерживаемых интернет-провайдером или работающих в локальной сети. Когда клиент хочет подключиться к серверу по имени, он запрашивает у DNS-сервера IP-адрес, соответствующий этому имени. Сервер отвечает запрошенным IP-адресом, если это возможно. Это проиллюстрировано на рис. 11-17.

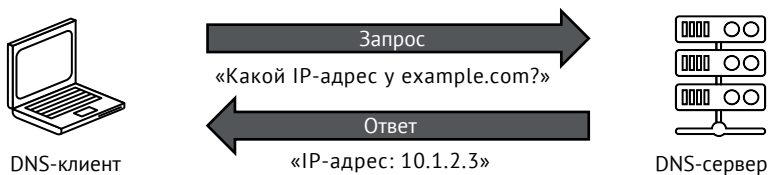


Рис. 11-17. Упрощенный запрос DNS. IP-адрес *example.com* не является реальным

Как только клиент получил IP-адрес сервера, он начинает взаимодействовать с ним, используя этот IP-адрес, как было описано ранее. Я слышал, что DNS называют телефонным справочником интернета, хотя эта аналогия может оказаться неудачной для некоторых читателей, поскольку телефонные справочники уже не так распространены, как раньше!

Вы можете предположить, что между IP-адресами и именами существует взаимно однозначное соответствие. На самом деле это не так. Имя может быть сопоставлено с несколькими IP-адресами. В этом случае разные клиенты запрашивают DNS для определенного имени, и все они могут получить в ответ разные IP-адреса. Это полезно в ситуациях, когда нагрузку на определенную службу необходимо распределить между несколькими серверами. Это может быть сделано географически, так что клиенты в Европе, например, получают IP-адрес, отличный от клиентов в Азии, что позволяет клиентам в каждом регионе подключаться к IP-адресу сервера, который находится физически близко к ним.

Возможна и обратная ситуация: несколько имен могут быть привязаны к одному IP-адресу. В этом случае запрос на разные имена может вернуть один IP-адрес. Это полезно, когда на сервере размещается несколько экземпляров одного и того же типа услуг, каждый из которых идентифицируется по имени. Это часто встречается в веб-хостинге, где на одном сервере размещается несколько веб-сайтов, каждый из которых идентифицируется по имени DNS.

Информация в DNS называется *записью*. Существуют различные типы записей, наиболее простой является *запись А*, которая просто сопоставляет имя хоста с IP-адресом. Другими примерами являются записи *CNAME* (*canonical name* – каноническое имя), которые сопоставляют одно имя хоста с другим, и записи *MX* (*mail exchanger* – почтовый обменник), используемые для служб электронной почты.

Ни одна организация не захочет брать на себя задачу управления множеством записей DNS, которые существуют сегодня. К счастью, в этом нет необходимости, так как DNS реализован таким образом, что позволяет разделить ответственность. Такое DNS-имя, как *www.example.com*, на самом деле представляет собой иерархию записей, и разные DNS-

серверы отвечают за ведение записей на разных уровнях иерархии. Иерархия DNS применительно к *www.example.com* показана на рис. 11-18<sup>1</sup>.

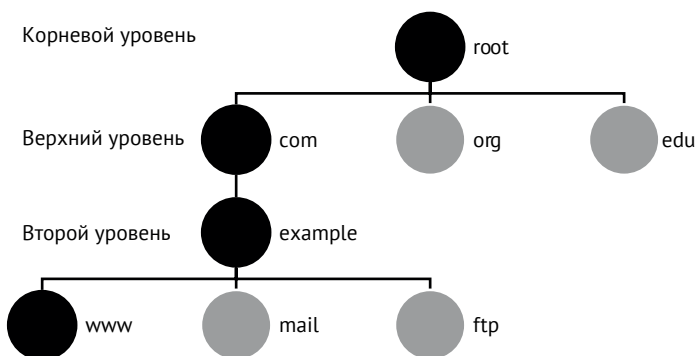


Рис. 11-18. Пример иерархии DNS (*www.example.com* выделен)

На вершине этого иерархического дерева находится *корневой домен*. Корневой домен не имеет текстового представления в DNS-имени<sup>2</sup>, как *www.example.com*, но он является важной частью иерархии DNS.

Корневой домен содержит записи для всех *доменов верхнего уровня* (*top-level domain*, TLD), таких как *.com*, *.org*, *.edu*, *.net* и т. д. По состоянию на 2020 год в мире насчитывается 13 корневых серверов имен, каждый из которых отвечает за знание деталей всех серверов доменов верхнего уровня. Допустим, вы хотите найти запись в домене, который заканчивается на *.com*. Корневой сервер может направить вас к серверу TLD, который знает о доменах под *.com*.

DNS-сервер верхнего уровня отвечает за знание обо всех доменах второго уровня в своей иерархии. DNS-сервер верхнего уровня для *.com* может указать вам на DNS-сервер второго уровня для *example.com*. DNS-серверы доменов второго уровня ведут записи о хостах и доменах третьего уровня, которые относятся к доменам второго уровня. Это означает, что DNS-сервер(ы) для *example.com* отвечает за ведение записей для таких хостов, как *www.example.com* и *mail.example.com*. Эта схема продолжается, позволяя создавать вложенные домены. После регистрации

<sup>1</sup> Отметим, что часто встречающаяся запись «www» в начале доменного имени не несет никакой технической нагрузки, являясь обычным именем домена следующего (обычно третьего) уровня в отношении основного домена. Если главный сайт размещается на *www.example.com*, то при обращении по основному имени (*example.com*) происходит переадресация пользователя на *www.example.com*. Чаще бывает наоборот, поскольку когда-то имя домена *www* имело тот смысл, что именно по этому адресу размещали сайты, подчиняющиеся правилам, разработанным для Всемирной паутины (World Wide Web, см. следующую главу). Теперь же им подчиняются все сайты почти без исключения, и такое выделение потеряло всякое практическое значение. Тем не менее огромное большинство сайтов имеет *www*-псевдоним и сегодня – традиция! – *Прим. ред.*

<sup>2</sup> Общий для всех корневой домен нулевого уровня обозначается точкой в конце доменного имени (так что правильная формальная запись полного имени выглядит как *www.example.com.*), но конечную точку, которая присутствует всегда, при записи доменных имен принято опускать. – *Прим. ред.*

домена в домене верхнего уровня владелец этого домена может создавать столько записей, сколько необходимо для его домена.

Как упоминалось ранее, когда компьютеру нужно найти IP-адрес для FQDN, он посылает запрос на настроенный DNS-сервер. Что делает DNS-сервер с этим запросом? Если сервер недавно просматривал запрашиваемую запись, он может иметь копию этой записи в своем кеше, и тогда он может немедленно вернуть IP-адрес клиенту. Если у DNS-сервера нет ответа в кеше, он может запросить другие DNS-серверы, чтобы получить ответ. Для этого нужно начать с корня и пройти по иерархии серверов, чтобы найти нужную запись. После того как сервер получил запись, он может поместить ее в кеш, чтобы немедленно отвечать на будущие запросы по этой записи. В конечном итоге кешированная запись удаляется, чтобы гарантировать, что сервер всегда предоставляет достаточно свежие данные.

#### ПРИМЕЧАНИЕ

*Смотрите проект № 35 на стр. 321, где вы сможете искать информацию в DNS.*

## Сеть – это вычисления

Давайте рассмотрим, как интернет вписывается в более широкую картину вычислительных технологий, которую мы уже разобрали в этой книге. Тема сетей может показаться отклонением от главной темы книги, но на самом деле она не так уж далека от вычислительной техники в целом. Интернет состоит из аппаратного и программного обеспечения, работающих вместе для обеспечения связи между устройствами. Данные, передаваемые через интернет, сводятся к нулям и единицам, представленным в различных формах, таких как напряжение на проводе. С точки зрения компьютера сетевой интерфейс, такой как адаптер Wi-Fi или Ethernet, – это просто еще одно устройство ввода/вывода. Операционная система взаимодействует с такими адаптерами через драйверы устройств, также ОС включает программные библиотеки, которые позволяют приложениям легко взаимодействовать через интернет. Сетевые устройства, такие как маршрутизаторы и коммутаторы, тоже являются компьютерами, хотя и узкоспециализированными. Интернет и сети в целом – это расширение локальных вычислений, позволяющее передавать и обрабатывать данные за пределами одного устройства.

## Выводы

В этой главе мы рассмотрели интернет – глобально объединенную систему компьютерных сетей, которые используют набор общих протоколов. Вы узнали о четырех уровнях набора интернет-протоколов – канальном уровне, межсетевом уровне, транспортном уровне и прикладном уровне. Вы увидели, как данные перемещаются через интернет и как устройства взаимодействуют на разных уровнях. Вы узнали, как DHCP предо-

ставляет данные о конфигурации сети, как NAT позволяет устройствам в частных сетях подключаться к интернету и как DNS предоставляет читаемые имена, которые можно использовать вместо IP-адресов. В следующей главе вы узнаете о Всемирной паутине (World Wide Web), наборе ресурсов, передаваемых по протоколу HTTP через интернет.

## ПРОЕКТ № 29: Изучение канального уровня

Необходимые условия: Raspberry Pi, работающий под Raspberry Pi OS. Я рекомендую вам пролистать приложение В и прочитать весь раздел «Raspberry Pi», если вы этого еще не сделали.

В этом проекте вы будете использовать Raspberry Pi для проверки канального уровня вашей локальной сети. Начнем со следующей команды, которая выводит MAC-адрес вашего Ethernet-адаптера:

---

```
$ ifconfig eth0 | grep ether
```

---

Вывод должен выглядеть примерно следующим образом:

---

```
ether b8:27:eb:12:34:56 txqueuelen 1000 (Ethernet)
```

---

В этом примере MAC-адрес – b8:27:eb:12:34:56. Это шестнадцатеричное представление 48-битного числа. Помните, что каждый шестнадцатеричный символ представляет собой 4 бита, поэтому 12 символов × 4 бита = 48 бит.

Первые 24 бита MAC-адреса представляют поставщика/производителя оборудования. Это число известно как *уникальный идентификатор организации* (*Organizationally Unique Identifier, OUI*) и выдается Институтом инженеров электротехники и электроники (*Institute of Electrical and Electronics Engineers, IEEE*). В данном случае OUI равен B827EB, который присвоен Raspberry Pi Foundation. Текущий список OUI можно посмотреть здесь: <http://standards-oui.ieee.org/oui.txt>.

Адаптер Wi-Fi вашей Raspberry Pi имеет свой собственный MAC-адрес. Посмотрите его следующим образом:

---

```
$ ifconfig wlan0 | grep ether
ether b8:27:eb:78:9a:bc txqueuelen 1000 (Ethernet)
```

---

В моей системе OUI (первые 24 бита MAC-адреса) адаптера Wi-Fi совпадает с OUI адаптера Ethernet. Это происходит потому, что оба адаптера являются внутренним оборудованием Raspberry Pi и используют OUI для Raspberry Pi Foundation.

С вашего Raspberry Pi вы также можете посмотреть MAC-адреса других устройств в локальной сети. Для этого можно использовать инструмент

arp-scan, который пытается подключиться к каждому компьютеру в вашей локальной сети и получить его MAC-адрес.

Сначала установите этот инструмент:

---

```
$ sudo apt-get install arp-scan
```

---

Затем выполните эту команду (это строчная буква l в конце команды, а не цифра 1):

---

```
$ sudo arp-scan -l
```

---

Вы должны получить список IP-адресов (о которых мы рассказываем в этой главе) и MAC-адресов плюс столбец, в котором делается попытка сопоставить MAC-префикс с производителем. Я получил 10 результатов для моей локальной сети, часть из которых я не сразу распознал. Вы можете увидеть некоторые повторяющиеся результаты, обозначенные символом DUP в третьем столбце. В возвращаемом списке обычно не указывается адрес компьютера, с которого выполнялось сканирование.

В третьем столбце могут быть результаты, которые отображаются как (Unknown). Это означает, что инструмент arp-scan не смог сопоставить номер OUI с известным производителем, возможно, потому, что инструмент использует устаревшую версию списка OUI. Вы можете попытаться исправить это, загрузив текущий список номеров OUI из IEEE, а затем запустив сканирование снова, как показано ниже:

---

```
$ get-oui  
$ sudo arp-scan -l
```

---

Когда я вижу несколько устройств в своей домашней сети, которые не могу сразу идентифицировать, у меня тут же возникает желание выяснить, что это за устройства! В качестве бонусного задания для вас определите каждое устройство, полученное от arp-scan. Это может быть непрактично, если вы используете этот инструмент в сети, которую не контролируете (например, в кафе или библиотеке), но, если вы находитесь дома, это можно сделать. Скорее всего, вам придется войти в систему на каждом устройстве в вашей сети и покопаться в его настройках, чтобы найти IP-адрес или MAC-адрес и посмотреть, совпадает ли он с одной из записей, полученных с помощью arp-scan. Подсказка: используйте утилиту ifconfig в Linux или Mac или ipconfig в Windows. А на мобильных устройствах ищите сетевые настройки в пользовательском интерфейсе.

## ПРОЕКТ № 30: Изучение межсетевого уровня

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы рассмотрите межсетевой уровень с помощью Raspberry Pi. Давайте начнем со следующей команды, которая выведет список всех сетевых интерфейсов на вашем устройстве и соответствующие им IP-адреса.

---

```
$ ifconfig
```

---

Скорее всего, вы увидите три интерфейса: `eth0`, `lo` и `wlan0`. Интерфейс `lo` – это особый случай, это интерфейс *loopback*. Он используется для процессов, запущенных на Pi, которые хотят обмениваться информацией друг с другом с помощью TCP/IP, но без фактической отправки трафика по сети. То есть трафик остается на устройстве. Интерфейс *loopback* имеет IP-адрес 127.0.0.1. Это специальный адрес, который не маршрутизируется и не может быть использован в качестве адреса в локальной подсети, поскольку любая попытка доставки сообщений на этот адрес приводит к тому, что сообщения возвращаются обратно на отправляющий компьютер. Другими словами, каждый компьютер считает 127.0.0.1 своим собственным IP-адресом. Как мы отмечали в предыдущем проекте, `eth0` – это проводной интерфейс Ethernet, а `wlan0` – беспроводной интерфейс Wi-Fi. Если вы подключены к сети через один из этих интерфейсов или через оба, то должны увидеть IP-адрес рядом с текстом `inet` в выводе `ifconfig`. Вы также можете увидеть IPv6-адрес, указанный рядом с `inet6`. Вот пример вывода `ifconfig` для `wlan0`:

---

```
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 192.168.1.138 netmask 255.255.255.0 broadcast 192.168.1.255
        inet6 fe80::8923:91b2:13e0:ed2a prefixlen 64 scopeid 0x20<link>
```

---

В этом выводе видно, что назначенный IP-адрес – 192.168.1.138. Значение `netmask` (маска подсети) – 255.255.255.0, а широковещательный (`broadcast`) адрес – 192.168.1.255.

Команда `ifconfig` дает нам информацию о различных сетевых интерфейсах на Raspberry Pi, но она не показывает нам, как настроена маршрутизация. Давайте посмотрим на это с помощью команды `ip route`. Я привел здесь примеры вывода, ваши результаты могут отличаться.

---

```
$ ip route
default via 192.168.1.1 dev wlan0 src 192.168.1.138 metric 303
192.168.1.0/24 dev wlan0 proto kernel scope link src 192.168.1.138 metric 303
```

---

Вывод этой команды может быть немного сложным для интерпретации. Если коротко, то в первой строке указан маршрут по умолчанию. Это место, куда следует отправлять пакеты, если не существует определенного



маршрута. В данном конкретном примере каждый пакет, не соответствующий определенному правилу маршрутизации, должен быть отправлен на адрес 192.168.1.1. Это означает, что 192.168.1.1 – это IP-адрес локального маршрутизатора, также известного как *шлюз по умолчанию*.

Следующая строка – это запись маршрутизации, которая говорит вам, что любой пакет, отправленный на IP-адрес в диапазоне 192.168.1.0/24, должен быть отправлен через устройство wlan0. Это адаптер Wi-Fi в локальной подсети. Другими словами, это правило маршрутизации гарантирует, что связь с IP-адресами в локальной подсети происходит напрямую, без прохождения через маршрутизатор.

Подводя итог, можно сказать, что любой пакет, отправленный на IP-адрес, который попадает в диапазон 192.168.1.0/24, должен быть отправлен непосредственно на адрес назначения через интерфейс wlan0. Любой другой трафик использует маршрут по умолчанию, который отправляет трафик на маршрутизатор по адресу 192.168.1.1. В итоге трафик локальной подсети отправляется непосредственно на целевое устройство, а трафик на устройства в других подсетях, вероятно, в интернете, отправляется на шлюз по умолчанию.

## ПРОЕКТ № 31: Изучение использования портов

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы увидите, какие сетевые порты используются на Raspberry Pi. Затем изучите порты на других компьютерах. Начнем со следующей команды, которая покажет вам прослушиваемые и установленные TCP-сокеты на Raspberry Pi.

```
$ netstat -nat
```

Давайте разберем опции `-nat`, используемые в команде. Буква `n` в опции указывает, что для отображения номеров портов следует использовать числовой вывод. Буква `a` означает, что следует показать все соединения (как прослушиваемые, так и установленные), а `t` означает ограничить вывод только TCP. На моем устройстве список выглядит следующим образом:

Active Internet connections (servers and established)					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN
tcp	0	36	192.168.1.138:22	192.168.1.125:52654	ESTABLISHED
tcp	0	0	192.168.1.138:22	192.168.1.125:51778	ESTABLISHED
tcp6	0	0	:::22	:::*	LISTEN

Здесь вы видите четыре сокета, все они связаны с SSH<sup>1</sup>. Я могу сказать, что они связаны с SSH, потому что все сокеты используют порт 22. Я включил SSH на своем Raspberry Pi, чтобы разрешить удаленные терминальные соединения. Первая и последняя строки показывают, что Pi прослушивает порт 22 для новых входящих SSH-соединений, используя как TCP, так и TCP через IPv6. Средние две строки показывают, что у меня есть два установленных SSH-соединения с этим устройством, оба с моего ноутбука (с IP 192.168.1.125) на Pi (с IP 192.168.1.138). Обратите внимание, что оба установленных соединения идут на один и тот же порт сервера на Pi (22), тогда как порт клиента на моем ноутбуке меняется (52654 и 51778), поскольку это эфемерные порты.

Выполните команду снова, на этот раз добавив опцию `p` и префикс `sudo`:

---

```
$ sudo netstat -natp
```

---

Это даст вам тот же список, но с идентификатором процесса (PID) и именем программы, к которой принадлежит сокет. Любой трафик, отправленный на сокет, направляется на PID, который обрабатывает трафик и отвечает по мере необходимости. На моем компьютере я вижу, что программа, использующая этот порт, – это `sshd` – служба для SSH.

Теперь, когда вы изучили, какие порты используются на вашем Raspberry Pi, давайте проверим порты на удаленном компьютере. Для этого вы будете использовать инструмент `nmар`, который сначала должен быть установлен на вашем Raspberry Pi:

---

```
$ sudo apt-get install nmap
```

---

После установки инструмента выберите целевой хост, который хотите просканировать. Это может быть либо устройство в вашей сети (например, маршрутизатор или ноутбук), либо хост в интернете. Обратите внимание, что многократное сканирование хоста, который вы не контролируете, может показаться подозрительным для администраторов этого сервера, поэтому я настоятельно рекомендую сканировать только те устройства, которые принадлежат вам.

В моем случае я решил просканировать свой шлюз по умолчанию, который находится по адресу 192.168.1.1. Следующая команда `nmар` сканирует открытые TCP-порты по указанному IP-адресу. Попробуйте сделать это на своей Raspberry Pi, заменив IP-адрес на адрес устройства, которое вы хотите просканировать. Если вы хотите просканировать свой собствен-

---

<sup>1</sup> SSH (*Secure Shell* – безопасная оболочка) – разновидность сетевого протокола прикладного уровня, разработанная для безопасного соединения в интернете. Шифрует весь трафик (включая и передаваемые пароли), позволяет безопасно передавать практически любой другой сетевой протокол. Не путать с SSL – протоколом, также призванным обеспечить безопасное соединение, но в другой области (для доступа к сайтам по *https*, см. далее). – Прим. ред.

ный маршрутизатор, см. проект № 30, где рассказывается, как получить IP-адрес вашего шлюза по умолчанию.

---

```
$ nmap -sT 192.168.1.1
```

---

Частичный список результатов моего сканирования показал следующие порты:

---

PORT	STATE	SERVICE
53/tcp	open	domain
80/tcp	open	http

---

Это говорит о том, что устройство работает не только как маршрутизатор, но и как DNS-сервер (порт 53) и веб-сервер (порт 80). Это нормально для домашнего маршрутизатора предоставлять эти услуги.

## ПРОЕКТ № 32: Прослеживание маршрута до хоста в интернете

Необходимые условия: Raspberry Pi, работающий под Raspberry Pi OS.

В этом проекте вы изучите маршрут, который проходит пакет от вашего Raspberry Pi до хоста в интернете. Сначала вам нужно выбрать хост в интернете. Это может быть веб-сайт, например *www.example.com*, или IP-адрес или FQDN любого известного вам хоста в интернете. После выбора хоста введите следующую команду, заменив *www.example.com* на имя или IP-адрес хоста, который вы хотите посмотреть.

---

```
$ traceroute www.example.com
```

---

Инструмент *traceroute* пытается показать маршрутизаторы, которые встречаются на пути следования пакета через интернет. Вывод следует читать построчно. Каждая строка последовательно пронумерована и показывает имя (если доступно) и IP-адрес маршрутизатора, встреченного на данном этапе пути пакета. Если через некоторое время ответа не будет, в выводе отображается звездочка (\*) и происходит переход к следующему маршрутизатору. Вы также можете увидеть более одного IP-адреса в строке, что указывает на несколько возможных маршрутов.

## ПРОЕКТ № 33: Узнайте свой арендованный IP-адрес

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы посмотрите информацию об аренде IP-адреса вашего Raspberry Pi, полученного от сервера DHCP. Конечно, это предполагает, что ваш Raspberry Pi настроен на использование DHCP (по умолчанию), а не статического IP-адреса. Для этого посмотрите системный журнал:

---

```
$ cat /var/log/syslog | grep leased
```

---

Ожидайте увидеть результаты, похожие на следующие:

---

```
Jan 24 19:17:09 pi dhcpcd[341]: eth0: leased 192.168.1.104 for 604800 seconds
```

---

Здесь видно, что IP-адрес 192.168.1.104 был арендован у DHCP-сервера для использования на сетевом интерфейсе eth0, интерфейсе Ethernet на Raspberry Pi. Ваш вывод, скорее всего, показывает другой IP-адрес и, возможно, другой интерфейс, может быть, wlan0.

По умолчанию файл *syslog* периодически очищается, а его содержимое перемещается в резервный файл. Из-за этого вы можете не увидеть запись DHCP в файле *syslog*. Вы можете освободить текущий IP-адрес, запросить новый и снова найти запись об аренде следующим образом:

---

```
$ sudo dhclient -r wlan0
$ sudo dhclient wlan0
$ cat /var/log/syslog | grep leased
```

---

Замените wlan0 на eth0, если вы хотите сделать это для проводного Ethernet, а не для Wi-Fi.

## ПРОЕКТ № 34: Является ли IP вашего устройства публичным или частным?

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы узнаете, является ли IP-адрес вашего Raspberry Pi публичным или частным. Если ваше устройство имеет частный IP-адрес, вы также узнаете публичный IP-адрес, который используется для связи через интернет. Как и раньше, можете использовать следующую утилиту для просмотра назначенного IP-адреса (адресов) вашего устройства.

---

```
$ ifconfig
```

---

При поиске назначенного IP-адреса устройства вы, скорее всего, увидите запись 127.0.0.1, которую можете игнорировать, поскольку она используется для loopback (см. проект № 30). Как упоминалось ранее, любой адрес, который соответствует шаблону 10.x.x.x, 172.16.x.x или 192.168.x.x, является частным IP-адресом. Теперь, даже если у вас есть частный IP-адрес, вроде одного из перечисленных, при доступе к ресурсам в интернете вы также косвенно используете публичный IP-адрес. Это адрес, который видят веб-сайты или другие интернет-сервисы, когда вы к ним подключаетесь. Если вы работаете в домашней сети, этот публичный IP-адрес, скорее всего, присвоен вашему маршрутизатору. Если вы работаете в корпоративной сети, этот публичный IP-адрес может быть присвоен прокси-устройству на конце вашей корпоративной сети. В любом случае весь сетевой трафик из вашей локальной сети в интернет идет с этого публичного адреса.

Чтобы найти публичный IP-адрес, который ваше устройство использует при подключении к устройству в интернете, можно зайти на ваш маршрутизатор или прокси-сервер и проверить его сетевую конфигурацию. Если вы знаете, как запросить эту информацию у своего маршрутизатора или прокси-сервера, смело делайте это. Однако, поскольку каждая модель сетевого устройства несколько отличается, я не буду здесь описывать все шаги.

Более универсальным вариантом является запрос к онлайн-сервису, который может выдать ваш текущий публичный IP-адрес. Это возможно, поскольку каждый интернет-сервер, к которому подключается ваше устройство, знает ваш IP-адрес, просто нужно найти службу, которая готова сообщить, какой IP-адрес она видит. Если на вашем устройстве запущен веб-браузер, возможно, проще всего сделать запрос в Google, например «мой IP-адрес». Обычно это позволяет получить нужную информацию.

Если вы работаете с терминала, например на Raspberry Pi, можете использовать утилиту `curl`, чтобы сделать HTTP-запрос к веб-сайту, который вернет ваш текущий IP-адрес. Ниже приведено несколько примеров сервисов, которые доступны для этого на момент написания книги:

```
$ curl http://ipinfo.io/ip
$ curl http://checkip.amazonaws.com/
$ curl http://ipv4.icanhazip.com/
$ curl http://ifconfig.me/ip
```

Любой из них должен вернуть ваш публичный IP-адрес в окно терминала. Сравните этот адрес с адресом, который вы получили ранее из `ifconfig`. Если они одинаковы, то вашему устройству напрямую присвоен публичный IP-адрес. Если они отличаются, то, скорее всего, вашему устройству присвоен частный IP-адрес, и вы подключаетесь к интернету через NAT-маршрутизатор или прокси-сервер.

## ПРОЕКТ № 35: Поиск информации в DNS

Необходимые условия: Raspberry Pi под Raspberry Pi OS.

В этом проекте вы будете использовать Raspberry Pi для запроса записей DNS. Давайте начнем с поиска IP-адреса веб-сайта. Для этого вы будете использовать утилиту `host`. Следующая команда выдает IP-адрес для `www.example.com`, что является именем хоста интересующего меня сайта. Не стесняйтесь подставить имя другого хоста, который вы хотите проверить.

```
$ host www.example.com
```

Вы должны увидеть вывод, содержащий IP-адрес хоста. Вы также можете увидеть IPv6-адрес. В зависимости от запрашиваемого имени хоста вы можете получить на выходе несколько записей, поскольку имя DNS может соответствовать нескольким IP-адресам. Также можете узнать, что введенное вами имя на самом деле является псевдонимом для другого имени, которое в свою очередь сопоставлено с IP-адресом.

DNS также позволяет осуществлять обратный поиск, когда вы указываете IP-адрес, а вам возвращается имя хоста. Это не всегда работает, поскольку для этого необходимо наличие записей DNS. Чтобы попробовать, просто используйте `host` с IP-адресом. В следующей команде замените `a.b.c.d` на ваш публичный IP-адрес, который вы нашли в проекте № 34, или на любой другой публичный IP-адрес, который хотите запросить. Опять же, это работает только для IP-адресов, которые имеют записи DNS для поддержки обратного поиска.

```
$ host a.b.c.d
```

По умолчанию утилита `host` использует DNS-сервер, на который настроено ваше устройство. Вы также можете запросить определенный DNS-сервер с помощью утилиты `host`, указав IP-адрес этого сервера. Интернет-провайдеры предоставляют своим клиентам услуги DNS, однако существует множество бесплатных альтернативных DNS-серверов. Например, на момент написания этой книги Google предоставляет DNS-сервер по адресу 8.8.8.8, а Cloudflare предоставляет DNS-сервер по адресу 1.1.1.1. Если вы хотите использовать DNS-сервер по адресу 1.1.1.1 для проверки `www.example.com`, можете ввести следующее:

---

```
$ host www.example.com 1.1.1.1
```

---

Как и раньше, будет выведена информация об IP-адресе, а также текст, указывающий, какой DNS-сервер был использован для поиска.

Если вам интересны подробности DNS-запроса, можете использовать опцию `-v` в команде `host`, которая обеспечивает подробный вывод.

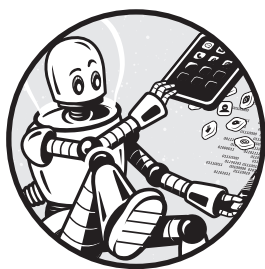
---

```
$ host -v www.example.com
```

---

# 12

## ВСЕМИРНАЯ ПАУТИНА



В предыдущей главе описывался интернет – глобально связанная совокупность компьютерных сетей, которые используют один и тот же набор протоколов. Всемирная паутина – это система, построенная на основе интернета и настолько популярная, что ее часто путают с самим интернетом. В этой главе мы погрузимся в детали Всемирной паутины. Сначала рассмотрим ее ключевые атрибуты и связанные с ней языки программирования, а затем рассмотрим веб-браузеры и веб-серверы.

### Обзор Всемирной паутины

*Всемирная паутина (World Wide Web), которую часто называют просто веб, представляет собой набор ресурсов, передаваемых с помощью протокола передачи гипертекста (HyperText Transfer Protocol, HTTP) через интернет. Веб-ресурс – это все, к чему можно получить доступ через интернет таким способом, например документ или изображение.*

Компьютер или программа, на котором размещаются веб-ресурсы, называется *веб-сервером*, а *веб-браузер* – это тип приложения, обычно используемый для доступа к содержимому в интернете. Браузеры используются для просмотра документов, известных как *веб-страницы*, а совокупность связанных между собой веб-страниц называется *веб-сайтом*.



Всемирная паутина является распределенной, адресуемой и связанной. Давайте начнем с рассмотрения каждого из этих основных атрибутов.

## Распределенная паутина

Всемирная паутина является *распределенной*. Никакая централизованная организация или система не управляет тем, какой контент может быть опубликован в веб. На любом компьютере, подключенном к интернету, может работать веб-сервер, и владелец такого компьютера может сделать доступным любой контент, который он пожелает. Тем не менее организации или страны могут блокировать доступ пользователей к определенному контенту в интернете, а правительства могут закрывать сайты, на которых размещается незаконный контент. За исключением этих случаев Всемирная паутина является открытой платформой для публикации всего, что люди пожелают опубликовать, и ни одна организация не контролирует доступность контента в глобальном масштабе.

## Адресуемая паутина

Во Всемирной паутине используются *унифицированные указатели ресурсов* (*Uniform Resource Locators, URLs*), чтобы дать каждому ресурсу в веб уникальный адрес, который включает в себя как его местоположение, так и способ доступа к нему. URL обычно называют *веб-адресами* или просто *адресами*. Для наглядного представления структуры этих адресов давайте воспользуемся URL-адресом вымышленного туристического сайта, как показано на рис. 12-1. Этот URL идентифицирует страницу с информацией о путешествии в Каролины<sup>1</sup>.

`http://travel.example.com/destinations/carolinas?location=b`

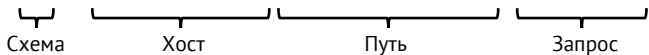


Рис. 12-1. Пример URL-адреса

URL состоит из нескольких частей. *Схема* URL определяет протокол прикладного уровня для доступа к ресурсу. В данном случае это протокол HTTP, который мы рассмотрим более подробно позже. Символ двоеточия (:) указывает на конец части, обозначающей схему доступа к ресурсу. После двух косых черт (/ /) следует основная часть указателя на ресурс, адресуемый URL. В данном примере часть указателя содержит DNS-имя хоста, на котором размещается ресурс, *travel.example.com*. Здесь также может быть использован IP-адрес вместо доменного имени. В этом разделе может содержаться и другая информация, помимо имени хоста, например имя пользователя (перед хостом и со знаком @) или номер порта (после хоста и с двоеточием). Далее идет часть URL, называемая *путь*,

<sup>1</sup> Имеются в виду американские штаты Северная Каролина и Южная Каролина, часто объединяемые в одну территорию. Один из центров расселения прибывающих европейцев по территории США в XIX веке, традиционный туристический объект. – *Прим. ред.*

которая определяет местоположение ресурса на веб-сервере. Путь URL аналогичен пути в файловой системе, выстраивающей ресурсы в логическую иерархию. В нашем примере путь `/destinations/carolinas` подразумевает, что на сайте есть коллекция страниц, описывающих туристические направления (*destinations*), а конкретная страница, указанная в URL, – это страница о Каролинах. Мы можем обоснованно предположить, что если бы на сайте была страница, описывающая Флориду как место назначения, то ее можно было бы найти по адресу `/destinations/florida`.

Наконец, часть под названием *запрос* в URL действует как модификатор ответа ресурса, возвращаемого клиенту. В нашем примере запрос указывает, что на странице *carolinas* должны быть показаны места на пляже (*beach*). Формат и значение запроса URL варьируется от сайта к сайту.

В этот URL вложено много информации, поэтому позвольте мне повторить простым языком, как его читать. На компьютере под названием *travel.example.com* запущен веб-сайт. Сайт работает по протоколу HTTP, поэтому при подключении к сайту используйте этот протокол. На этом сайте есть страница под названием *carolinas*, часть коллекции *destinations*. Строка запроса направляет страницу на показ только тех мест, которые находятся на пляже.

URL-адрес не обязательно должен включать все элементы, приведенные в примере на рис. 12-1. Он также может содержать некоторые элементы, не включенные в этот пример. С другой стороны, вполне допустим URL, включающий только протокол и имя хоста, например <http://travel.example.com>. В этом случае сайт предоставляет свою страницу, установленную в нем по умолчанию, поскольку путь не указан.

### УПРАЖНЕНИЕ 12-1: Определение частей URL-адреса

Для следующих URL-адресов определите схему (протокол), имя пользователя, хост, порт, путь и запрос. Не все URL-адреса включают все эти части.

- `https://example.com/photos?subject=cat&color=black`
- `http://192.168.1.20:8080/docs/set5/two-trees.pdf`
- `mailto:someone@example.com`

Вы можете проверить свои ответы в приложении А.

Веб-браузер обычно отображает текущий URL в адресной строке, как показано на рис. 12-2.

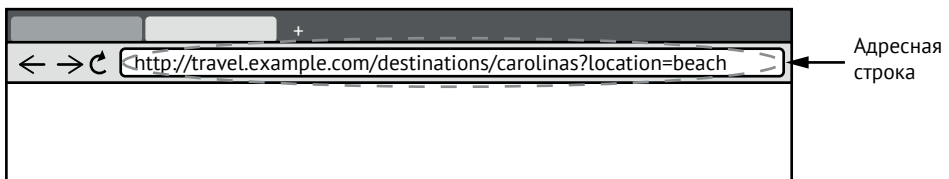


Рис. 12-2. Адресная строка

Сегодня в браузерах принято исключать протокол, двоеточие и косые черты из представления URL в адресной строке. Это не означает, что эти элементы URL больше не используются браузером. Браузер просто пытается упростить вид для пользователей. Специфика отображения URL продолжает меняться с течением времени, и различные браузеры ведут себя по-разному.

На рис. 12-3 показаны примеры того, как Google Chrome (версия 77) отображает URL в своей адресной строке.

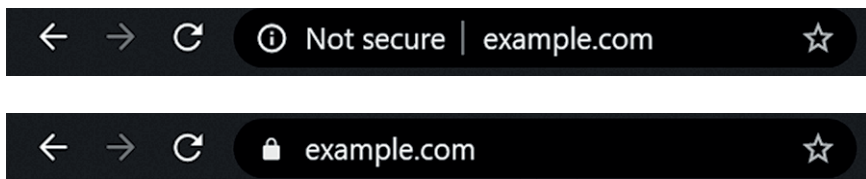


Рис. 12-3. Адресная строка Chrome

Верхнее изображение на рис. 12-3 показывает адресную строку при загрузке HTTP-сайта. Chrome не отображает префикс `http://` в своей адресной строке. Обратите внимание на текст *Not secure*. На нижнем рисунке показана адресная строка при загрузке сайта HTTPS. HTTPS – это защищенная версия HTTP. Chrome опускает префикс `https://`, но отображает значок замка, указывающий на то, что это сайт HTTPS.

Мы до сих пор обсуждали URL-адреса в контексте веб-страниц, но URL-адреса распространяются и на другие ресурсы в интернете. Например, картинка, демонстрируемая на веб-странице, имеет свой собственный URL, так же как и файл *скрипта*<sup>1</sup> или файл данных XML. Веб-браузер показывает в адресной строке только URL веб-страницы, но типичная веб-страница ссылается на различные другие ресурсы по URL, которые автоматически загружаются браузером.

Иногда нет необходимости включать в URL протокол, имя хоста или даже полный путь. Когда в URL отсутствует один или несколько этих элементов, он называется *относительным URL*. Относительный

<sup>1</sup> *Скрипт* (сценарий) – текст программы на специальном скриптовом языке (например, PHP, VBScript или JavaScript). Для скриптовых языков характерно существование исполняемой программы в виде исходного кода в обычном текстовом формате. Понятие сценарного (скриптового) языка программирования близко к понятию интерпретируемого языка; например, Python также можно рассматривать как скриптовый язык. Таким образом, понятие «скриптовый язык» относится не к устройству языка, а скорее к особенностям применения создаваемого с его помощью программного кода. Типичным примером скриптов могут служить издавна известные пользователям Windows пакетные bat-файлы (файлы интерпретатора командной строки). Применительно к веб-программированию программа-скрипт может существовать в виде отдельного файла или в виде вставки в код веб-страницы. С помощью файлов, содержащих скрипты, реализуется, например, настраиваемый функционал браузеров, а с помощью скриптов, встроенных в веб-страницу, организованы интерактивные функции сайтов. Подробнее о языке JavaScript см. далее в разделе «Языки Всемирной паутины». – *Прим. ред.*

URL интерпретируется как относительный по отношению к контексту, в котором он находится. Например, если на веб-странице используется URL типа `/images/cat.jpg`, браузер, загружающий страницу, считает, что протокол и имя ресурса, на котором размещена фотография кошки, совпадают со протоколом и именем хоста самой страницы.

## Связанная паутина

Природа URL-адресов, когда каждый ресурс в интернете имеет уникальный адрес, позволяет одному веб-ресурсу легко ссылаться на другой. Отсылка с одного веб-документа на другой известна как *гиперссылка* или просто *ссылка*. Такие ссылки являются односторонними, любая веб-страница может ссылаться на другую страницу без разрешения или взаимной ссылки. Эта система страниц, ссылающихся друг на друга, и образует «паутину» во Всемирной паутине. Документы, подобные веб-страницам, которые могут быть связаны гиперссылками, называются *гипертекстовыми документами*.

## Веб-протоколы

Для передачи данных по Всемирной паутине используется *протокол передачи гипертекста* (*HyperText Transfer Protocol*, *HTTP*) и его защищенный вариант *HTTPS*.

### HTTP

Несмотря на свое название, HTTP предназначен не только для передачи гипертекста, он используется для чтения, создания, обновления и удаления всех ресурсов в интернете. HTTP обычно опирается на TCP/IP. TCP обеспечивает надежную передачу данных, а IP – адресацию хостов. Сам HTTP основан на модели *запроса и ответа*. Запрос HTTP отправляется на веб-сервер, а сервер посылает на него ответ.

Каждый HTTP-запрос включает *HTTP-метод*, который описывает, какое действие клиент запрашивает у сервера.

Вот некоторые часто используемые методы HTTP:

GET – получение ресурса без его изменения;

PUT – создание или изменение ресурса по определенному URL-адресу на сервере;

POST – создание ресурса на сервере в качестве дочернего элемента существующего URL;

DELETE – удаление ресурса с сервера.

Любой метод HTTP может быть использован на любом ресурсе, но сервер, на котором размещен конкретный ресурс, часто не разрешает использовать некоторые методы на этом ресурсе. Например, большинство веб-сайтов не позволяет клиентам удалять ресурсы. Те, которые позволяют удалять ресурсы, почти всегда требуют, чтобы пользователь вошел в систему под учетной записью, имеющей право удалять содержимое.

Наиболее часто используемым методом на типичном веб-сайте является GET. Когда веб-браузер переходит на веб-сайт, он выполняет GET на запрашиваемой странице. Эта страница может содержать ссылки на скрипты, изображения и т. д., и браузер также использует метод GET для получения этих ресурсов, прежде чем страница будет полностью отображена.

Каждый ответ HTTP включает *код состояния HTTP*, который описывает ответ сервера. Каждый код состояния представляет собой трехзначное число, где старшая цифра указывает на общий класс состояния. Ответы в диапазоне 100 являются информационными. Ответы в диапазоне 200 означают успех. Ответы в диапазоне 300 указывают на перенаправление. Ответы в диапазоне 400 означают ошибку на стороне клиента – запрос не был правильно сформирован клиентом. Ответ в диапазоне 500 означает, что сервер столкнулся с ошибкой.

Некоторые часто используемые коды состояния HTTP:

**200** – Success (успех). Сервер смог выполнить запрос;

**301** – Moved Permanently (перемещено навсегда). Браузер должен перенаправить запрос на другой URL, указанный в ответе;

**401** – Unauthorized (не авторизован). Требуется авторизация;

**403** – Forbidden (запрещено). Пользователь не имеет доступа к запрашиваемому ресурсу;

**404** – Not Found (не найдено). Сервер не нашел запрашиваемый ресурс;

**500** – Internal Server Error (внутренняя ошибка сервера). На сервере произошло что-то непредвиденное.

HTTP довольно легко понимать. Он использует понятный человеку текст для описания запросов и ответов. Первая строка запроса включает метод HTTP, URL ресурса и запрашиваемую версию HTTP. Например:

---

```
GET /documents/hello.txt HTTP/1.1
```

---

Это просто означает, что клиент просит сервер отправить ему содержимое */documents/hello.txt*, используя HTTP версии 1.1. После строки запроса запрос HTTP обычно включает поля заголовков, которые предоставляют дополнительную информацию о запросе, и необязательное тело сообщения.

Похожим образом и ответ HTTP использует простой текстовый формат. Первая строка включает версию HTTP, код состояния и фразу ответа. Вот пример первой строки ответа HTTP:

---

```
HTTP/1.1 200 OK
```

---

В этом примере ответа сервер указывает код состояния 200 и фразу ответа *OK*. Как и HTTP-запросы, ответы также могут включать значения заголовков и тело сообщения. На рис. 12-4 представлен более подробный, но все еще упрощенный пример HTTP-запроса и ответа.

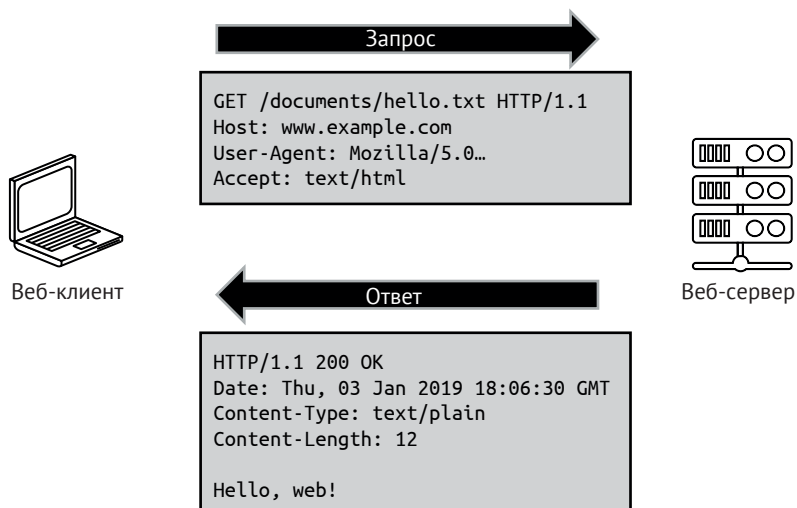


Рис. 12-4. Упрощенная схема запроса и ответа HTTP

#### ПРИМЕЧАНИЕ

Обратитесь к проекту № 36 на стр. 349, где вы можете посмотреть на сетевой трафик HTTP.

## HTTPS

Безопасный вариант HTTP, известный как *HTTPS* (*HyperText Transfer Protocol Secure*), широко используется во Всемирной паутине для шифрования данных, передаваемых через интернет. *Шифрование* – это процесс кодирования данных в формат, который невозможно прочитать. *Расшифровка* – это обратный процесс шифрования, в результате которого зашифрованные данные снова становятся доступными для чтения. Криптографические алгоритмы шифруют и расшифровывают данные с помощью секретной последовательности байтов, известной как *криптографический ключ*. Поскольку ключи могут храниться в секрете, сам алгоритм может быть хорошо известен.

HTTPS использует два вида шифрования. *Симметричное шифрование* использует один общий ключ как для шифрования, так и для расшифровки сообщения. *Асимметричное шифрование* использует два ключа (*пару ключей*): *открытый ключ* используется для шифрования данных, а *закрытый* – для их расшифровки.

Асимметричное шифрование позволяет свободно распространять открытый ключ, чтобы любой мог зашифровать и отправить данные, в то время как закрытый ключ передается только доверенным лицам, которые должны иметь возможность получать и расшифровывать данные.

Без HTTPS веб-трафик передается «в чистом виде», т. е. он не зашифрован и может быть перехвачен или изменен в процессе передачи злоумышленниками. HTTPS помогает снизить эти риски. При использовании HTTPS весь запрос HTTP шифруется, включая URL, заголовок и тело. То же самое относится и к ответу HTTPS, он полностью зашифрован. HTTPS принимает HTTP-запрос и шифрует его с помощью про-

токола под названием *протокол защиты транспортного уровня* (*Transport Layer Security, TLS*). В прошлом использовался аналогичный протокол под названием *уровень защищенных сокетов* (*Secure Sockets Layer, SSL*), но из-за проблем с безопасностью он был упразднен в пользу TLS. Когда мы говорим о HTTPS, мы имеем в виду HTTP, зашифрованный с помощью TLS.

Когда начинается HTTPS-сессия, клиент подключается к серверу с помощью сообщения *client hello* с подробной информацией о том, как он хочет безопасно общаться. Сервер отвечает сообщением *server hello*, которое подтверждает, как будет происходить обмен данными. Сервер также отправляет набор байтов, известный как *цифровой сертификат*, который включает открытый криптографический ключ сервера, используемый для асимметричного шифрования. Затем клиент проверяет, действителен ли сертификат сервера. Если да, клиент шифрует строку байтов с помощью открытого ключа сервера, а затем отправляет зашифрованное сообщение на сервер. Сервер расшифровывает байты с помощью своего закрытого ключа. Сервер и клиент используют информацию, которой они обменялись ранее, для вычисления общего секретного ключа, используемого для симметричного шифрования. Как только клиент и сервер получают общий ключ, этот ключ используется для шифрования и расшифровки всех сообщений между клиентом и сервером в течение всего сеанса.

HTTPS ранее использовался только в ограниченных случаях, для веб-сайтов, работающих с особо конфиденциальной информацией. Однако в настоящее время Всемирная паутина переходит в состояние, когда HTTPS является скорее нормой, чем исключением. Растет убеждение, что преимущества HTTPS в плане безопасности и конфиденциальности имеют смысл для большинства, если не для всего трафика в веб. Google поощряет эти изменения, помечая HTTP-сайты как «небезопасные» в Chrome и используя наличие HTTPS как положительный сигнал для своей поисковой системы, что помогает повысить рейтинг HTTPS-сайтов при поиске в Google.

#### ПРИМЕЧАНИЕ

Смотрите проект № 37 на стр. 351, где вы сможете настроить простой веб-сервер в своей локальной сети.

## Поиск в паутине

Для многих людей отправной точкой для работы во Всемирной паутине является поиск. Вместо того чтобы переходить по определенному URL, пользователь набирает в браузере несколько слов для поиска и смотрит, что появится в ответ. Дизайн браузеров способствует этому, поскольку браузеры обычно используют также и адресную строку в качестве окна поиска. Даже когда пользователь хочет посетить определенный сайт, он часто выполняет поиск этого сайта и затем нажимает на полученную ссылку, а не вводит полный URL в адресной строке.

Такой дизайн повышает удобство использования браузера, хотя и стирает различия между терминами «URL» и «поиск», «браузер» и «поисковая система».



Несмотря на распространенность и удобство поиска в веб, возможность поиска не является неотъемлемой чертой Всемирной паутины. Не существует стандартной спецификации того, как должен работать поиск. Это означает, что поиск, одна из ключевых особенностей веб, зависит от нестандартных, частных поисковых систем. На момент написания этой книги Google доминирует в сфере веб-поиска, и, хотя существуют хорошие альтернативные поисковые системы, их использование в мире составляет лишь малую часть от использования Google.

## Языки Всемирной паутины

Любой контент, который можно сохранить в виде файла, может быть размещен во Всемирной паутине. Например, на веб-сервере может быть размещена коллекция файлов Excel, которые можно загрузить с веб-сайта и открыть в Excel. Однако веб-браузер – это гораздо больше, чем просто инструмент для загрузки файлов, которые можно открыть в других приложениях. Веб-браузер не только загружает контент, но и отображает веб-страницы. Эти страницы могут быть простыми документами или интерактивными веб-приложениями. Чтобы сделать это возможным, браузеры понимают три компьютерных языка, которые используются для создания веб-сайтов.

**Язык разметки гипертекста (HyperText Markup Language, HTML)** – определяет структуру веб-страницы. Другими словами, он определяет, *что находится на странице*. Например, HTML может указать, что на веб-странице есть кнопка.

**Каскадные таблицы стилей (Cascading Style Sheets, CSS)** – определяют внешний вид веб-страницы. Другими словами, они определяют, *как выглядит страница*. Например, CSS может указать, что вышеупомянутая кнопка имеет ширину 30 пикселей и синий цвет.

**JavaScript** – определяет поведение веб-страницы. Другими словами, он определяет, *как функционирует страница*. Например, JavaScript можно использовать для сложения двух чисел при нажатии на кнопку.

Эти три языка используются вместе для создания контента в веб. Стоит отметить, что веб-браузеры также способны отображать и некоторые другие типы данных, в частности определенные форматы изображений, видео и аудио, но мы не будем подробно останавливаться на них. Теперь давайте погрузимся в каждый из трех основополагающих языков веб: HTML, CSS и JavaScript.

## Структурирование веб с помощью HTML

HTML – это язык разметки, который описывает структуру веб-страницы. Обратите внимание, что HTML не является языком программирования. Язык программирования описывает операции, которые должен выполнять компьютер, в то время как язык разметки описывает структуру данных. В случае с HTML данные представляют собой веб-страницу. Веб-страница



может содержать различные элементы, такие как абзацы, заголовки и изображения. Вот пример простой веб-страницы, описанной на языке HTML.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Cat</title>
  </head>
  <body>
    <h1>Thoughts on a Cat</h1>
    <p>This is a cat.</p>
    
  </body>
</html>
```

Вы увидите ряд элементов, заключенных в знаки «меньше» (<) и «больше» (>). Это *теги* HTML – наборы текстовых символов, используемые для определения частей HTML-документа. В качестве примера можно привести тег, используемый для обозначения начала абзаца – <p>. Для обозначения конца абзаца используется соответствующий тег </p>. Обратите внимание на косую черту в конечном теге, которая отличает его от начального тега. *Элемент HTML* – это часть страницы, начинающаяся с начального тега, заканчивающаяся конечным тегом и включающая текст между тегами. Вот пример элемента HTML: <p> This is a cat. </p>. На самом деле не все элементы требуют конечного тега. Например, элемент *img*, используемый для представления изображения, не нуждается в конечном теге. Вы можете увидеть это в предыдущем примере кода HTML.

На рис. 12-5 показано, как этот пример HTML может быть отображен в веб-браузере.

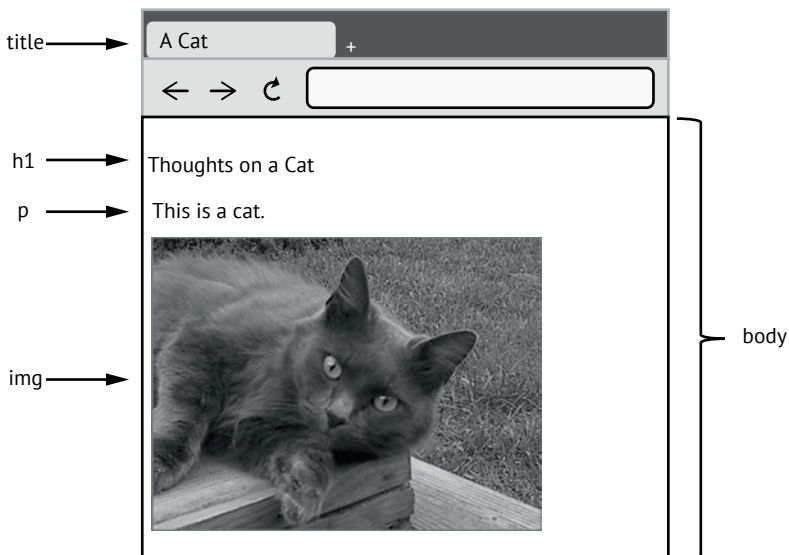


Рис. 12-5. Наш пример веб-страницы, отображаемой в веб-браузере

В документ намеренно не включена информация о том, как он должен быть представлен, поэтому браузер использует шрифт и размер по умолчанию для заголовка и абзаца.

В этом примере браузер также по умолчанию использовал черный текст на белом фоне – опять же, это не было специально указано в документе. Поскольку этот пример HTML не содержит информации о стилях, разные браузеры могут выбрать разное отображение этой страницы.

Рассмотрим пример HTML-кода более подробно. Первая строка HTML-документа объявляет, что файл является HTML-документом, следующим образом: `<!DOCTYPE html>`. После этого HTML-документ структурируется в виде дерева с родительскими и дочерними элементами. Тег `<html>` является родительским тегом верхнего уровня, все заключено между `<html>` и `</html>`. Эти два тега можно интерпретировать как «HTML начинается здесь» и «HTML заканчивается здесь». Это имеет смысл, так как все в HTML-документе должно быть HTML! Тег `<html>` также содержит атрибут `lang`, который определяет язык этого документа как `en`, код для английского языка.

Элемент `html` имеет два дочерних элемента: *head* (голова) и *body* (тело). Элементы, содержащиеся в *head* (между `<head>` и `</head>`), описывают документ, тогда как элементы в *body* (между `<body>` и `</body>`) составляют содержимое документа.

В нашем примере *head* содержит два элемента: *meta*, описывающий набор символов, используемый для кодирования нашего документа, и *title* – заголовок. Браузеры обычно отображают текст заголовка на вкладке страницы и используют его в качестве имени по умолчанию, когда пользователь добавляет страницу закладку или избранное. Поисковые системы используют текст заголовка при отображении результатов. По этим причинам веб-разработчикам важно давать содержательные заголовки своим страницам.

В нашем примере *body* включает тег `<h1>`, который используется для элемента заголовка первого уровня. Доступны теги заголовков от `<h1>` до `<h6>`, где *h1* предназначен для использования в качестве самого высокого уровня заголовков разделов, а *h6* – в качестве самого низкого уровня. После заголовка следует абзац, обозначенный тегом `<p>`, а после него включается изображение с помощью тега `<img>`. Обратите внимание, что байты самого изображения не присутствуют в HTML. Вместо этого тег `<img>` просто ссылается на файл изображения по относительному URL (*cat.jpg*). Чтобы полностью загрузить эту страницу, браузеру необходимо сделать отдельный HTTP-запрос для загрузки изображения. В данном примере URL-адрес изображения – это просто имя файла, т. е. файл размещен на том же сервере и по тому же пути, что и сам документ. Если бы изображение было размещено в другом месте, можно было бы использовать URL с указанием пути или имени сервера. Тег `<img>` также имеет атрибут `alt`, который содержит альтернативный текст, описывающий изображение. Он используется в случаях, когда изображение не может быть отображено, например когда используется браузер, работающий только с текстом, или устройство чтения с экрана, которое читает содержимое страницы вслух.

Вы могли заметить, что в предыдущем коде HTML использовались отступы, чтобы показать вложенность различных элементов на странице. Например, теги `<h1>` и `<p>` отступают на один уровень, показывая, что они являются дочерними элементами тега `<body>`. Это обычная практика в веб-разработке для улучшения читабельности HTML, но она не обязательна. На самом деле пробельные символы за пределами одного пробела или табуляции не имеют значения в HTML-документе! Мы можем удалить все лишние пробелы, табуляции и переносы строк, оставив HTML на одной строке, и документ будет отображаться в браузере точно так же. Веб-браузеры игнорируют лишние пробелы, поэтому расстановка элементов на странице полезна только для разработчиков.

#### ПРИМЕЧАНИЕ

*Обратитесь к проекту № 38 на стр. 353, где вы сможете заставить локальный веб-сайт возвращать документ, структурированный с помощью HTML, а не как простой текст.*

Элементы HTML, которые мы рассмотрели, составляют лишь небольшой процент от общего числа элементов, распознаваемых веб-браузерами. Мы не будем здесь подробно описывать весь HTML, он хорошо представлен в интернете. Спецификации HTML ранее поддерживались двумя организациями: Консорциумом Всемирной паутины (World Wide Web Consortium, W3C) и Рабочей группой по технологиям гипертекстовых веб-приложений (Web Hypertext Application Technology Working Group, WHATWG). Последней основной версией HTML, получившей статус «Рекомендовано» от W3C, был HTML5. В 2019 году эти две организации договорились, что дальнейшее развитие стандарта HTML будет осуществляться в основном WHATWG в рамках так называемого «Живого стандарта HTML» (*HTML Living standard*), который постоянно поддерживается.

Современные браузеры стараются поддерживать как текущие, так и более старые версии HTML, поскольку многие веб-материалы были написаны с учетом более ранних стандартов HTML. В прошлом браузеры вводили нестандартные элементы HTML, часть из которых со временем стали стандартными, а другие вышли из употребления и потеряли поддержку. Разработчики веб-браузеров должны балансировать между инновациями и соблюдением стандартов, поддерживая при этом менее совершенный HTML, который иногда встречается в веб. Веб-браузеры постоянно развиваются, и разные браузеры иногда по-разному отображают один и тот же контент. Это означает, что веб-разработчики должны регулярно тестировать свои работы на нескольких браузерах, чтобы обеспечить стабильное поведение.

## Стилизация веб-страниц с помощью CSS

В нашем предыдущем примере HTML мы использовали теги, которые описывали структуру документа, но эти теги не передавали никакой информации о том, как документ должен быть представлен. Это было

сделано намеренно, мы хотим разделить структуру и стиль. Разделение этих двух понятий позволяет отображать один и тот же контент разными стилями в разных контекстах. Например, большая часть веб-контента должна по-разному отображаться на большом экране ПК и на маленьком экране мобильного телефона.

*Каскадные таблицы стилей* (*Cascading Style Sheets, CSS*) – это язык, используемый для описания стиля веб-страницы. Таблица стилей представляет собой список правил. Каждое правило описывает стиль, который должен быть применен к определенной части страницы. Каждое правило включает селектор, который указывает, к каким элементам на странице должен быть применен стиль. Термин «каскадный» означает возможность применения нескольких правил к одному и тому же элементу. Давайте рассмотрим простой пример:

---

```
p {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 11pt;
  margin-left: 10px;
  color: DimGray;
}

h1 {
  font-family: 'Courier New', Courier, monospace;
  font-size: 18pt;
  font-weight: bold;
}
```

---

В этом примере правила стиля определены для элементов абзаца (p) и заголовка уровня 1 (h1). Когда этот CSS применяется к странице, все абзацы на этой странице используют указанный шрифт размером 11 пунктов, левый отступ в 10 пикселей и серый цвет текста. А заголовки h1 используют указанный полужирный шрифт и размер шрифта 18 пунктов. Обратите внимание, что `font-family` – это список шрифтов, а не один шрифт. Это означает, что веб-браузер должен попытаться найти подходящий шрифт, начиная с самого левого шрифта и продвигаясь вправо, пока не будет найден подходящий. Не на каждом клиентском устройстве установлен первый вариант шрифта, указание нескольких шрифтов увеличивает вероятность того, что подходящий шрифт будет доступен.

Таблицу стилей можно применить к веб-странице несколькими способами. Один из вариантов – включить правила CSS в элемент стиля на странице (тег `<style>`). Например:

---

```
<style>p {color: red};</style>
```

---

Это не идеальный вариант, поскольку стиль и структура теперь тесно связаны. Лучшим вариантом является задание правил CSS в отдельном файле, также размещенном в веб. При таком подходе HTML и CSS оста-

ются полностью разделенными, и это позволяет нескольким HTML-файлам использовать одну и ту же таблицу стилей. Таким образом, мы можем изменить правило CSS, и оно будет применяться сразу к нескольким страницам. Один элемент в разделе *head* HTML может быть использован для применения правил таблицы стилей из файла CSS, например, так (где *style.css* – это URL файла CSS, который нужно применить):

---

```
<link rel="stylesheet" type="text/css" href="style.css">
```

---

Если мы применим эту таблицу стилей к нашему примеру страницы с кошкой, увидим изменения в заголовке и тексте абзаца, как показано на рис. 12-6.

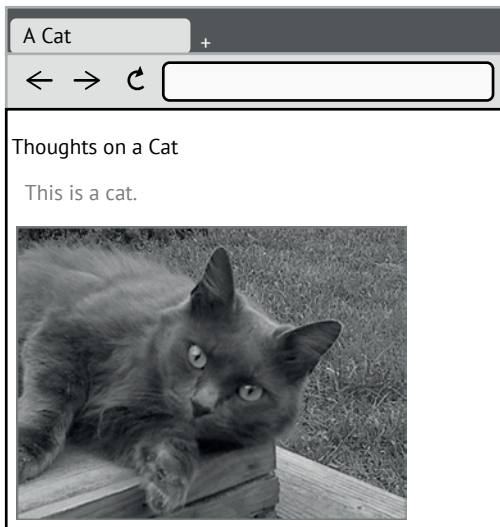


Рис. 12-6. Наш пример веб-страницы с примененным CSS

**ПРИМЕЧАНИЕ**

*Смотрите проект № 39 на стр. 355, где вы сможете обновить вашу веб-страницу с кошкой с помощью некоторого CSS.*

Этот пример CSS прост, но CSS позволяет создавать и намного более сложные стили. Если вы знакомы с удивительным разнообразием визуальных стилей, которые можно найти во Всемирной паутине, то вы уже видели силу CSS в действии<sup>1</sup>.

---

<sup>1</sup> Отметим, что оформление HTML-документов в нужном стиле возможно не только с помощью CSS. Вначале никаких CSS не существовало, потому большинство структурообразующих тегов (`<p>`, `<body>`, `<table>` и т. п.) имеют богатый набор атрибутов, предоставляющих широкий спектр возможностей для оформления страницы (см. любой онлайн-справочник по HTML). В раннем вебе (90-е годы) эти возможности широко использовались, в частности, возникла привычка оформления веб-страниц в нужном формате с помощью

## Создание скриптов с помощью JavaScript

Веб изначально задумывался как средство обмена информацией через гипертекстовые документы. HTML дает нам такую возможность, а CSS предоставляет метод управления представлением таких документов. Однако веб превратился в платформу для интерактивного контента, и JavaScript стал стандартным средством для обеспечения этой интерактивности. *JavaScript* – это язык программирования, который позволяет веб-страницам реагировать на действия пользователей и программно выполнять различные задачи. С JavaScript веб-браузер становится не просто средством чтения документов, а полноценной платформой для разработки приложений.

JavaScript – это интерпретируемый язык, он не компилируется в машинный код перед передачей в браузер. Веб-серверы размещают код JavaScript в текстовом формате, а браузер загружает этот код и интерпретирует его во время выполнения. Тем не менее некоторые браузеры используют *компилятор just-in-time (JIT)*, который компилирует JavaScript во время выполнения, что приводит к повышению производительности. Некоторые разработчики *минифицируют*<sup>1</sup> JavaScript перед его развертыванием, удаляя пробелы, комментарии и в целом уменьшая размер скрипта. Минификация JavaScript может улучшить время загрузки сайта. Минификация это не то же самое, что компиляция, минифицированный файл представляет все еще высокоуровневый, а не скомпилированный машинный код.

Синтаксис JavaScript похож на синтаксис языка C и других языков, заимствованных из C (таких как C++, Java и C#). Однако это сходство поверхностно, поскольку JavaScript сильно отличается от этих языков. И пусть вас не путает название, JavaScript имеет мало общего с языком Java. Язык является объектно-ориентированным, но в основе его лежат *прототипы*, а не классы. То есть существующий объект, а не класс служит шаблоном для других объектов.

JavaScript взаимодействует с HTML-страницей, используя предоставляемое браузером представление страницы, называемое *объектной моделью документа (Document Object Model, DOM)*. DOM представляет собой иерархическую древовидную структуру элементов страницы, которую можно программно изменять. Обновление элемента в DOM заставляет браузер обновить этот элемент на отображаемой веб-странице. JavaScript включает методы для работы с DOM, и, используя эти методы,

---

таблиц (тег `<table>`). Таблицы стилей CSS предоставляют намного более гибкие возможности, и сегодня оформление сайта с помощью атрибутов и таблиц повсеместно признано моветоном. Тем не менее о такой возможности не следует забывать: вам может потребоваться оформить один-единственный *html*-документ в объеме странички, и это проще сделать в едином файле по-старинке, чем городить таблицы стилей, обычно размещающиеся в дополнительных файлах.

<sup>1</sup> Минификация (*minification*) – процесс, направленный на уменьшение размера исходного кода путем удаления ненужных символов без изменения его функциональности. – *Прим. ред.*

код JavaScript может как реагировать на события, происходящие на странице (например, нажатие кнопки), так и изменять содержимое отображаемой страницы.

Давайте рассмотрим часть простого скрипта, который взаимодействует со страницей из нашего примера. Скрипт добавляет текст «Meow!» в абзац нашей страницы каждый раз, когда кликают по фотографии кошки (или касаются ее на сенсорном экране).

---

```
document.getElementById('cat-photo').onclick = function() {  
    document.getElementById('cat-para').innerHTML += ' Meow!';  
};
```

---

Первая строка здесь добавляет обработчик событий, который запускается при нажатии на фотографию кошки (onclick). Код обработчика события определен в следующей строке, и он указывает браузеру добавить текст «Meow!» в абзац. Поскольку этот код определен как обработчик события, он запускается только при наступлении события – клика по изображению. Обратите внимание, что сценарий ссылается на фотографию и абзац по идентификаторам `cat-photo` и `cat-para`, соответственно. Элементам HTML можно присваивать идентификаторы, это позволяет легко ссылаться на них в программном коде. Наш скрипт будет работать, только если мы добавим эти идентификаторы в наш HTML. Вот обновленный HTML-код, который ссылается на скрипт (с именем `cat.js`) и добавляет необходимые идентификаторы.

---

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="utf-8">  
    <title>A Cat</title>  
    <link rel="stylesheet" type="text/css" href="style.css">  
    <script src="cat.js"></script>  
  </head>  
  <body>  
    <h1>Thoughts on a Cat</h1>  
    <p id="cat-para">This is a cat.</p>  
      
  </body>  
</html>
```

---

После сохранения кода скрипта под именем `cat.js` и обновления HTML, как показано выше, перезагрузка страницы и клик на изображении кошки будет добавлять «Meow!» в наш абзац. Если мы кликнем изображение несколько раз, то в итоге получим нечто подобное тому, что показано на рис. 12-7.



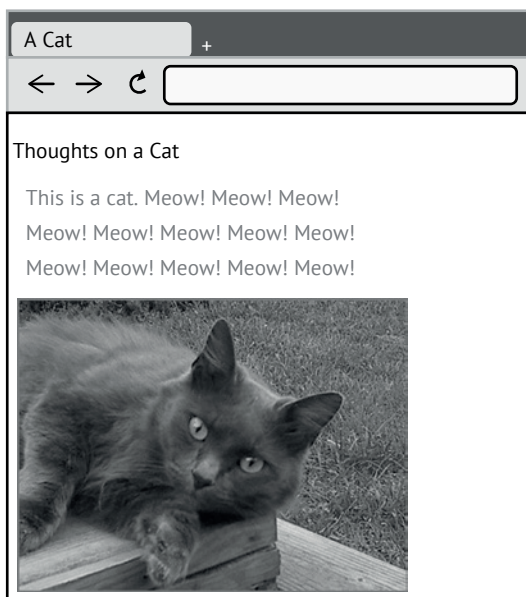


Рис. 12-7. Наш пример веб-страницы после выполнения кода JavaScript для добавления текста

#### ПРИМЕЧАНИЕ

Обратитесь к проекту № 40 на стр. 356, где вы сможете обновить веб-страницу с помощью JavaScript.

JavaScript можно использовать для создания полноценных приложений, запускаемых в веб-браузере. Предыдущий пример – лишь самая малость из того, на что он способен. JavaScript стандартизирован в спецификации, известной как *ECMAScript*. Различные браузеры используют скриптовые движки, которые пытаются полностью или частично соответствовать регулярно обновляемому стандарту ECMAScript.

## Структурирование данных в веб с помощью JSON и XML

Веб-сайты – не единственный тип контента, доступный во Всемирной паутине. *Веб-служба* предоставляет данные по протоколу HTTP и предназначена для программного взаимодействия в отличие от веб-сайта, который возвращает HTML (и связанные с ним ресурсы) и предназначен для использования пользователем через веб-браузер. Большинство конечных пользователей никогда напрямую не взаимодействуют с веб-службами, хотя веб-сайты и приложения, которые мы используем, часто основаны на веб-службах.

Представьте, что вы управляете веб-сайтом с информацией о местных музыкальных группах, выступающих в вашем городе. Сайт содержит про-



филь каждой группы, включая информацию об ее участниках, историю, место выступления и т. д. Конечный пользователь может зайти на ваш сайт и легко найти информацию о своих любимых музыкантах. Теперь, допустим, к вам обратился разработчик приложения, который хочет включить последнюю информацию с вашего сайта в свое приложение. Однако это приложение имеет свое собственное представление о подаче информации, которое радикально отличается от вашего, и разработчики не хотят просто отображать ваши веб-страницы в своем приложении. Им нужен способ получить доступ к базовым данным на вашем сайте. Они могут попытаться прочесть ваши веб-страницы посредством программы и извлечь необходимую информацию, но этот процесс сложен и чреват ошибками, особенно если макет вашего сайта меняется.

Вы можете значительно облегчить работу разработчика, предоставив веб-службу, которая представляет данные с вашего сайта в формате, отличном от HTML. Хотя HTML и обеспечивает определенную структуру, это структура, которая описывает текстовый документ (заголовки, абзацы и т. д.) и дает мало представления о типах данных, на которые ссылается этот документ. HTML имеет смысл для человека, но его трудно анализировать программному обеспечению. Так какой же формат должна использовать ваша веб-служба для структурирования данных о группах? Наиболее распространенными форматами данных общего назначения, используемыми сегодня веб-службами, являются XML и JSON.

*Расширяемый язык разметки (Extensible Markup Language, XML)* существует с 1990-х годов и является популярным средством обмена данными через веб. Как и HTML, это язык разметки на основе текста, но вместо набора предопределенных тегов XML позволяет использовать пользовательские теги, которые описывают ваши данные. В случае с нашей вымышленной службой информации о музыкальных группах мы можем определить тег `<band>` и тег `<concert>`. Давайте рассмотрим воображаемую музыкальную группу, описанную с помощью XML:

---

```
<band name="The Highbury Musical Club">
  <bandMembers>
    <member name="Jane Fairfax" instrument="Piano" />
    <member name="Emma Woodhouse" instrument="Guitar" />
    <member name="Harriet Smith" instrument="Percussion" />
    <member name="Frank Churchill" instrument="Vocals" />
  </bandMembers>
  <upcomingConcerts>
    <concert location="Donwell Abbey" date="August 14, 2020" />
    <concert location="Hartfield" date="November 20, 2020" />
  </upcomingConcerts>
</band>
```

---

Как видите, конкретные теги XML и их атрибуты адаптированы к нашим потребностям, а общая структура начальных тегов, конечных тегов и древовидной иерархии соответствует HTML. Гибкость XML, где теги могут быть определены произвольно, означает, что и производитель, и пользователь XML должны договориться о предполагаемых

тегах и их значениях. Это справедливо и для HTML, но в случае HTML все стороны пользуются стандартом. В случае XML стандартизирован только общий формат, а конкретные теги варьируются.

XML является популярным методом обмена данными в веб, и многие веб-службы используют XML в качестве основного средства представления данных. Однако XML многословен, и его правильный синтаксический анализ может оказаться сложным.

*JavaScript Object Notation (JSON)*, как и XML, является методом описания данных в текстовом формате. JSON избегает использования тегов разметки и вместо этого использует стиль, аналогичный синтаксису JavaScript для описания объектов, откуда и происходит его название. В JSON объекты заключаются в фигурные скобки ({ and }), а массивы (коллекции объектов) – в квадратные ([ and ]). Его синтаксис короче, чем у XML, что помогает уменьшить размер данных, передаваемых по сети. Популярность JSON возросла в 2010-х годах, когда он начал вытеснять XML в качестве предпочитаемого формата данных для новых веб-сервисов. Вот та же воображаемая музыкальная группа, описанная в JSON:

---

```
{
  "name": "The Highbury Musical Club",
  "bandMembers": [
    {
      "name": "Jane Fairfax",
      "instrument": "Piano"
    },
    {
      "name": "Emma Woodhouse",
      "instrument": "Guitar"
    },
    {
      "name": "Harriet Smith",
      "instrument": "Percussion"
    },
    {
      "name": "Frank Churchill",
      "instrument": "Vocals"
    }
  ],
  "upcomingConcerts": [
    {
      "location": "Donwell Abbey",
      "date": "August 14, 2020"
    },
    {
      "location": "Hartfield",
      "date": "November 20, 2020"
    }
  ]
}
```

---

И XML, и JSON игнорируют лишние пробельные символы, поэтому, как и в HTML, мы можем удалить все лишние пробелы, табуляции и переносы строк, не влияя на интерпретацию данных. Это позволяет получить довольно компактное отображение данных, особенно в случае JSON.

XML и JSON не являются форматами, предназначенными для прямого отображения в веб-браузере. Открытие содержимого JSON или XML в некоторых браузерах может привести к тому, что браузер что-то покажет (возможно, слегка отформатированную версию данных), но на самом деле JSON и XML не предназначены для прямого потребления веб-браузерами. Они предназначены для чтения кодом, который в свою очередь делает что-то полезное с этими данными. Возможно, этот код – приложение для смартфона, которое показывает информацию о том, какие группы играют поблизости, как в нашем примере. Или же код – это клиентский JavaScript, который преобразует JSON в HTML для отображения в браузере.

## Веб-браузеры

Теперь, когда мы рассмотрели языки, используемые для описания веб, давайте взглянем на программное обеспечение клиентской части – на веб-браузеры. Первый веб-браузер назывался *WorldWideWeb* (не путать с темой этой главы). Он был разработан Тимом Бернерсом-Ли в 1990 году. Этот первый браузер был клиентом для первого веб-сервера, *CERN httpd*. Через несколько лет *WorldWideWeb* был вытеснен браузером *Mosaic*, который помог популяризировать Всемирную паутину. Следующим крупным браузером стал *Netscape Navigator*, который также пользовался большой популярностью. В 1995 году компания Microsoft выпустила свой первый браузер, *Internet Explorer*, как прямого конкурента *Netscape Navigator*, и *Internet Explorer* стал доминирующим браузером своего времени. Сегодня ситуация с браузерами значительно изменилась, и на момент написания этой книги доминирующими браузерами являются *Google Chrome*, *Apple Safari* и *Mozilla Firefox*.

### Визуализация страницы

Давайте теперь рассмотрим процесс, через который проходит веб-браузер для визуализации страницы. Типичное посещение веб-сайта начинается с запроса страницы сайта, заданной по умолчанию (например, <http://www.example.com/>), или запроса конкретной страницы сайта (например, <http://www.example.com/animals/cat.html>). Пользователь может ввести этот URL непосредственно в адресной строке, или он может попасть на него, перейдя по ссылке. В любом случае браузер запрашивает содержимое указанного URL. Предполагая, что URL является действительным и представляет собой веб-страницу, сервер отвечает в виде HTML.

Затем веб-браузер должен взять возвращенный HTML и создать DOM-представление страницы.

HTML может содержать ссылки на другие ресурсы, такие как изображения, скрипты и таблицы стилей. Каждый из этих ресурсов имеет свой URL, и браузер делает отдельные запросы для каждого ресурса, как показано на рис. 12-8.

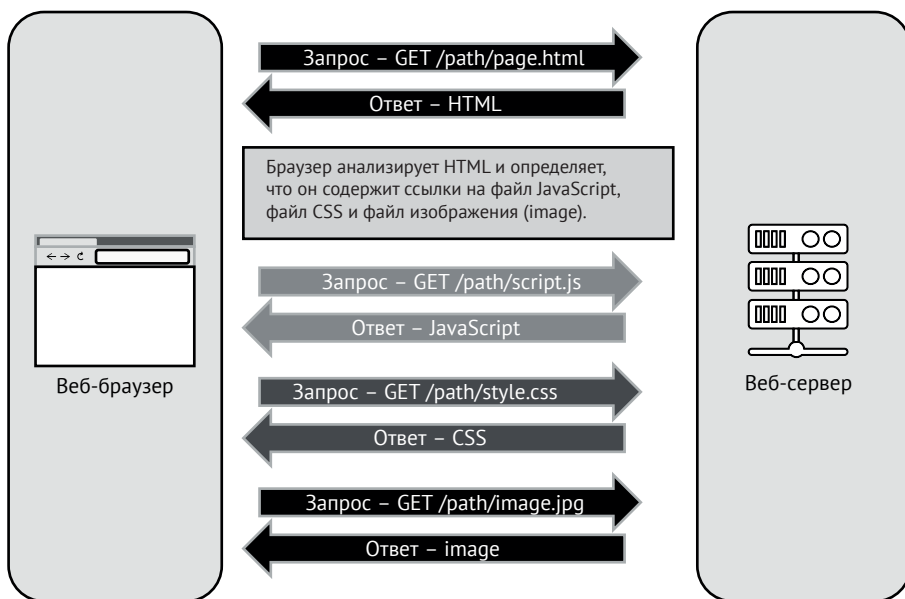


Рис. 12-8. Веб-браузер запрашивает страницу и связанный с ней контент

После того как браузер получил различные ресурсы страницы, он отображает HTML, используя любой заданный CSS для определения подходящего представления. Любые скрипты передаются на выполнение движку JavaScript. Код JavaScript может немедленно вносить изменения в страницу или регистрировать обработчики событий, которые запускаются позже при наступлении определенных событий. Код JavaScript может также запрашивать данные у веб-службы и использовать их для обновления страницы.

Веб-браузеры состоят из браузерного движка (для HTML и CSS), движка JavaScript и пользовательского интерфейса, который связывает все вместе. Хотя пользовательский интерфейс определяет внешний вид браузера (например, внешний вид кнопки **Назад** и адресной строки), именно браузерный движок и движок JavaScript определяют представление и поведение веб-сайтов (сюда входят такие вещи, как расположение страницы и ее реакция на ввод). Поскольку каждый браузерный движок и движок JavaScript работают немного по-разному, веб-страница может выглядеть или вести себя по-разному при просмотре в разных браузерах. В идеале все браузеры должны отображать содержимое одинаково, в точности так, как задумал разработчик сайта, но это не всегда так. На момент написания этой книги только три основных браузерных движка находятся в активной разработке: WebKit, Blink и Gecko.

*WebKit* – это движок браузера и движок JavaScript для браузера Safari от Apple. Он также используется в приложениях, размещенных в магазине приложений iOS App Store, поскольку Apple требует, чтобы все приложения для iOS, отображающие веб-контент, использовали этот движок.

*Blink*, являющийся ответвлением (форком) *WebKit*, представляет собой движок браузера для проекта с открытым исходным кодом *Chromium*, который также включает движок V8 JavaScript. *Chromium* является основой для Google Chrome и Opera. В декабре 2018 года компания Microsoft объявила, что браузер *Microsoft Edge* также будет основан на *Chromium*. Microsoft решила прекратить разработку собственных движков браузеров и движков JavaScript. В результате остается только один крупный браузер, который не восходит к *WebKit*, – это Mozilla Firefox, который имеет собственный движок браузера *Gecko* и движок JavaScript *SpiderMonkey*.

#### ПРИМЕЧАНИЕ

*Разветвление (форк) программного обеспечения происходит, когда разработчики делают копию исходного кода проекта, а затем вносят в эту копию изменения. Это позволяет исходному и ответвленному проектам сосуществовать как отдельным программным проектам.*

## Строка агента пользователя (User Agent String )

Формальный технический термин для веб-браузера – «агент пользователя» (*user agent*). Этот термин может применяться и к другому программному обеспечению (все, что действует от имени пользователя), но здесь мы говорим именно о веб-браузерах. Этот термин встречается в технической документации по Всемирной паутине, хотя за пределами официального общения он используется редко. Тем не менее одно из мест, где этот термин используется на практике, – это *строка агента пользователя* (*user agent string*). Когда браузер делает запрос к веб-серверу, он обычно включает значение строки заголовка User-Agent, которое описывает браузер. В качестве примера здесь приведена строка агента пользователя, отправленная браузером Chrome (версия 71) на Windows 10:

---

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36

---

Эта строка может показаться противоречивой. Что же все это значит?

Первая запись, Mozilla/5.0, осталась с первых дней существования веб. Mozilla – это имя агента пользователя для Netscape Navigator, и многие сайты специально искали «Mozilla» в строке агента пользователя как индикатор, чтобы отправить в браузер самую современную версию своего сайта. В то время другие браузеры тоже хотели получить лучшие версии сайтов, поэтому они называли себя Mozilla, даже если они вовсе не были Mozilla. Сегодня, когда практически каждый браузер иденти-

фицирует себя как Mozilla, мы считаем эту часть строки агента пользователя довольно бессмысленной.

Следующий раздел в скобках, (Windows NT 10.0; Win64; x64), указывает платформу, на которой работает браузер.

За ним следует движок браузера, в данном случае AppleWebKit/537.36. Как уже упоминалось ранее, движок Blink в Chrome является ответвлением WebKit и по-прежнему идентифицирует себя как таковой. Следующий текст – (KHTML, like Gecko) – является лишь продолжением этого, KHTML – это старый движок, на котором был основан WebKit.

Теперь мы переходим к фактическому названию и версии браузера – Chrome/71.0.3578.98.

Наконец, мы видим странное упоминание браузера Apple Safari/537.36, включенное для сайтов, которые предоставляют Safari особое обращение. Включая этот текст, Chrome пытается убедиться, что эти сайты отправляют ему тот же контент, который получает Safari.

Это довольно сложный способ идентификации Chrome, но другие браузеры делают то же самое для обеспечения совместимости со всеми видами веб-сайтов. Эта сложность является досадным побочным эффектом исторически сложившихся разных возможностей различных браузеров и веб-сайтов, которые пытались отправлять адаптированные версии своего контента в зависимости от конкретного браузера. Браузеры развивались так, что сегодня различия в возможностях браузеров стали меньше. Однако многие веб-сайты не эволюционировали и по-прежнему отправляют контент, адаптированный под конкретные браузеры, заставляя современные браузеры продолжать обманывать старые сайты и думать, что они взаимодействуют с другим браузером.

## Веб-серверы

До сих пор мы концентрировали внимание в основном на технологиях, используемых на клиентской стороне веб. Веб-браузеры говорят на общей тройке языков: HTML, CSS и JavaScript. А что же на стороне веб-сервера? Какие языки и технологии используются для работы веб-серверов? Если коротко, то любой язык программирования или технология могут быть использованы на веб-сервере, если только эта технология может взаимодействовать по HTTP и возвращать данные в формате, понятном клиенту.

В целом веб-сайты могут быть как статическими, так и динамическими. *Статический веб-сайт* возвращает HTML, CSS или JavaScript, которые были созданы заранее. Как правило, содержимое сайта хранится в файлах на сервере, и сервер просто возвращает содержимое этих файлов без изменений. Это означает, что любая необходимая обработка во время выполнения должна быть реализована в JavaScript, который запускается в браузере. С другой стороны, *динамический веб-сайт* выполняет обработку на сервере, генерируя HTML при поступлении запроса.

На заре развития Всемирной паутины почти все было статическим. Страницы представляли собой простой HTML, и интерактивность была невелика. Со временем разработчики начали добавлять код, который выполнялся на веб-сервере, позволяя серверу возвращать динамический контент или принимать загрузку файла или отправку формы от пользователя. Эта тенденция продолжилась, и стало обычным явлением, когда запросы проходят через обработку на стороне сервера, прежде чем сервер отвечает на них.

Давайте рассмотрим, как обычно осуществляется обработка на стороне сервера на динамическом сайте, как показано на рис. 12-9. Предположим, что динамический сайт, представленный на рис. 12-9, – это блог. Браузер делает запрос на запись в блоге. Когда веб-сервер получает запрос на запись в блог, он считывает запрошенный URL и определяет, что ему необходимо сгенерировать HTML. Затем код на сервере запрашивает базу данных (которая может находиться на этом веб-сервере или на другом сервере), извлекает соответствующие текстовые данные блога, форматирует этот текст как HTML, а затем отвечает клиенту этим HTML. Такой подход полезен, поскольку позволяет управлять содержимым сайта отдельно от его кода, однако динамические сайты имеют и некоторые недостатки. Повышенная сложность на сервере означает больше работы по настройке, более медленный отклик во время выполнения, потенциально большую нагрузку на сервер и повышенный риск проблем с безопасностью.

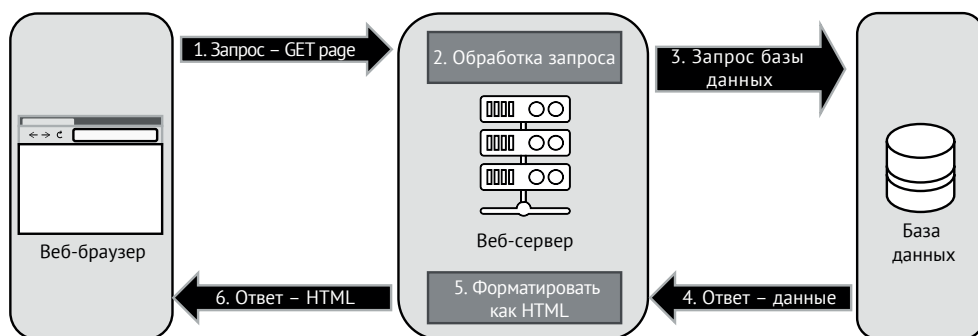


Рис. 12-9. Типичный динамический веб-сайт обрабатывает запрос

В последнее время наблюдается тенденция возврата к статическим сайтам, где это возможно. Поток запроса страницы на статическом сайте показан на рис. 12-10.

Как показано на рис. 12-10, обработка статического сайта на стороне сервера существенно проще по сравнению с динамическим сайтом. Обработка на стороне сервера статического сайта сводится к возврату статического файла, соответствующего запрошенному URL. Контент уже создан, серверу не нужно получать необработанные данные и форматировать их. Уменьшение сложности на стороне сервера обычно означает создание более простых, быстрых и безопасных сайтов.



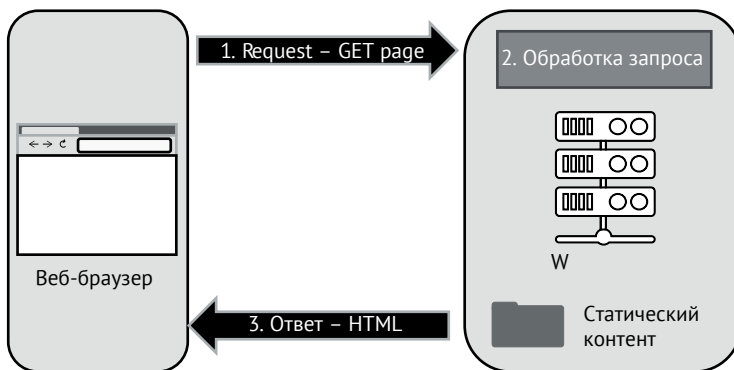


Рис. 12-10. Статический сайт обрабатывает запрос

Важно понимать, что в этом контексте термины «статический» и «динамический» используются с точки зрения сервера, а не пользователя. Контент статического сайта поступает в неизменном виде из файлов на сервере, в то время как контент динамического сайта генерируется на сервере. Эти термины не описывают восприятие сайта пользователем и не определяют такие вещи, как является ли сайт интерактивным или обновляется ли его содержимое автоматически. Эти эффекты могут быть достигнуты с помощью JavaScript в браузере, иногда в сочетании с отдельной веб-службой, независимо от того, является ли сам сайт статическим или динамическим.

Если вы размещаете статический сайт, то все, что вам нужно, это программное обеспечение веб-сервера, которое может отвечать на запросы ваших статических файлов и обслуживать содержимое этих файлов. Никакого специального кода не требуется. Для размещения статических сайтов существует множество программных пакетов и онлайн-сервисов. Как правило, программное обеспечение для обслуживания статического сайта настроено так, чтобы указывать на каталог файлов на сервере, и когда поступает запрос на определенный файл, сервер просто возвращает содержимое этого файла. Например, если файлы для сайта *example.com* находятся на сервере в каталоге под названием */websites/example*, то запрос *http://example.com/images/cat.jpg* будет соответствовать */websites/example/images/cat.jpg*. Веб-сервер просто считывает соответствующий файл из своего локального каталога и возвращает клиенту байты, содержащиеся в этом файле. Веб-сайт, разрабатываемый в проектах № 37–40, является примером статического сайта.

Если вы создадите динамический веб-сайт или веб-службу, то вы можете использовать существующее программное обеспечение, которое управляет контентом и обслуживает динамические страницы, либо вы можете написать собственный код, генерирующий веб-контент. Если вы решите писать собственный код, то обнаружите, что веб-разработка на стороне сервера сильно отличается от веб-разработки на стороне клиента. Для веб-сервера можно использовать любой язык программирования, любую операционную систему и любую платформу. Все что



угодно, лишь бы веб-сервер отвечал по протоколу HTTP и возвращал данные в формате, понятном клиенту! Клиенту все равно, какие технологии были использованы для создания HTML или JavaScript, ему просто нужен ответ в формате, который он может обработать.

Поскольку для клиентов не имеет значения, какая технология используется на стороне веб-сервера, разработчикам, желающим написать работающий на сервере код, доступно множество вариантов. Веб-разработка на стороне клиента ограничивается трио HTML, CSS и JavaScript<sup>1</sup>, в то время как веб-разработка на стороне сервера может вестись на Python, C#, JavaScript, Java, Ruby, PHP и других языках. Веб-разработка на стороне сервера часто включает в себя взаимодействие с какой-либо базой данных. Точно так же, как любой язык программирования может быть использован на сервере, любой тип базы данных может быть использован для веб-разработки на стороне сервера.

## Выводы

В этой главе мы рассмотрели Всемирную паутину – комплекс распределенных, адресуемых, связанных ресурсов, передаваемых по HTTP через интернет. Вы узнали, как веб-страницы структурируются с помощью HTML, оформляются с помощью CSS и создаются с помощью JavaScript. Мы рассмотрели веб-браузеры, которые используются для доступа к контенту, и разобрали веб-серверы – программное обеспечение, на котором размещаются веб-ресурсы. В следующей главе мы рассмотрим некоторые тенденции в современной вычислительной технике, и у вас будет возможность завершить итоговый проект, который свяжет воедино различные концепции, встречающиеся в этой книге.

---

<sup>1</sup> К клиентским языкам еще относят VisualBasicScript (VBS), однако его популярность не идет ни в какое сравнение с популярностью JavaScript. – *Прим. ред.*

## ПРОЕКТ № 36: Исследование трафика HTTP

В этом проекте вы будете использовать Google Chrome или Chromium для изучения HTTP-трафика между веб-браузером и веб-сервером. Вы можете использовать Chrome на Windows PC или Mac или использовать веб-браузер Chromium на Raspberry Pi. Следующие шаги предполагают, что вы используете Raspberry Pi, но процесс будет аналогичен и на Windows PC или Mac, просто используйте Chrome вместо Chromium.

1. Если вы не используете графический рабочий стол на Raspberry Pi, перейдите на него сейчас. В отличие от предыдущих проектов этот проект не может быть осуществлен через окно терминала.
2. Нажмите **Raspberry** (значок в левом верхнем углу) -> **Internet** -> **Chromium Web Browser**.
3. Перейдите на веб-сайт, например <http://www.example.com>.
4. Нажмите клавишу **F12** (или **CTRL-SHIFT-I**), чтобы открыть инструменты разработчика (**DevTools**), как показано на рис. 12-11.

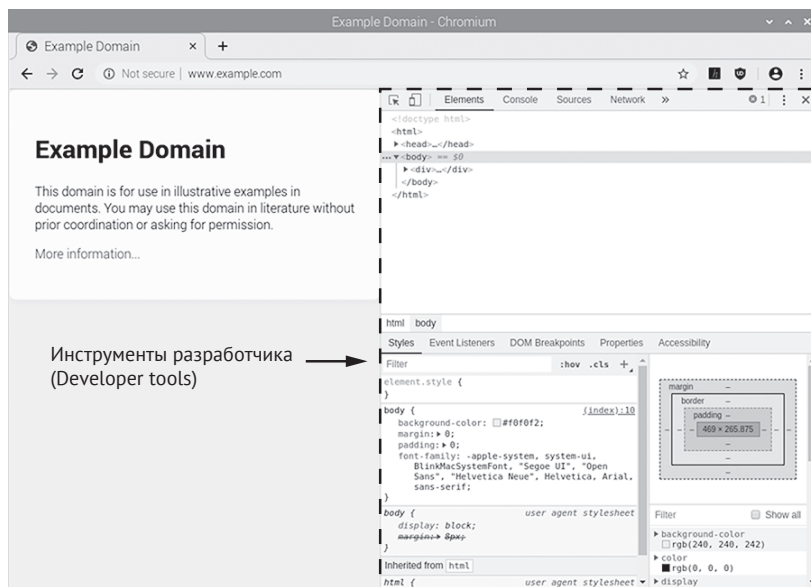


Рис. 12-11. Инструменты разработчика в Chromium

5. В меню **DevTools** выберите пункт меню **Network**.
6. Нажмите **F5** (или нажмите значок перезагрузки), чтобы перезагрузить страницу. Вы увидите HTTP-запросы, которые были сделаны для загрузки страницы, которую вы сейчас просматриваете.
7. Если вы действительно используете [www.example.com](http://www.example.com), то, скорее всего, увидите довольно скучные запросы. Если вы хотите увидеть что-то более интересное, посетите более сложный сайт и наблюдайте за сетевыми запросами, как показано на рис. 12-12.

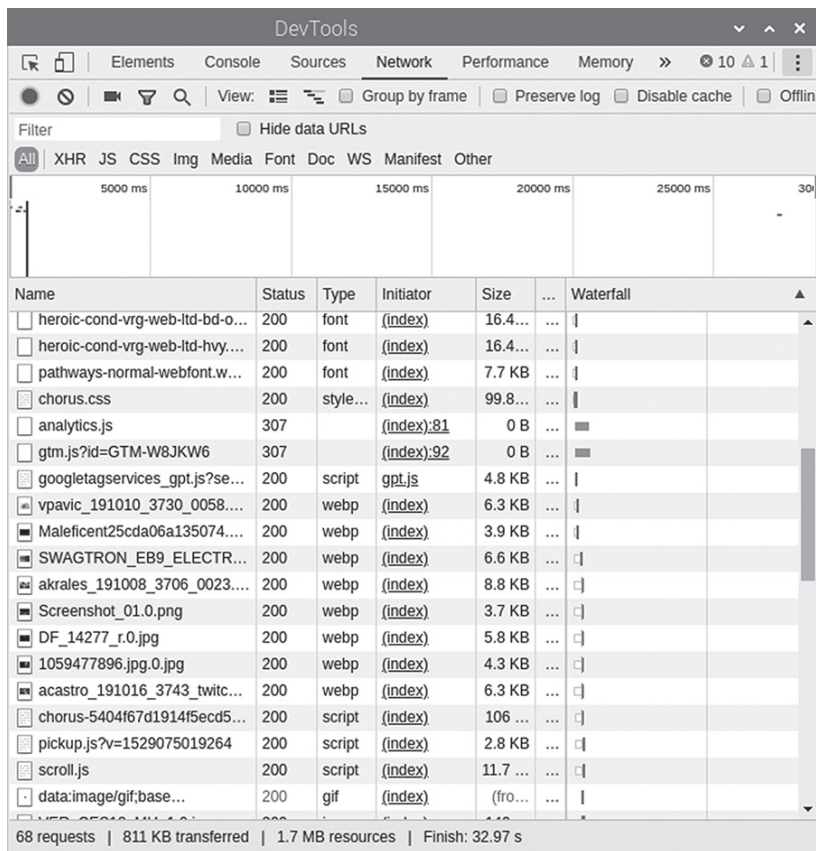


Рис. 12-12. Пример HTTP-трафика для веб-сайта, показанный в DevTools Chromium

8. Каждая строка представляет собой запрос к веб-серверу. Вы можете увидеть имя ресурса, который был запрошен, статус запроса (200 означает успех) и многое другое.

9. Вы можете щелкнуть по каждой строке и просмотреть специфику запроса, например заголовки и возвращаемый контент.

Я рекомендую вам попробовать это на нескольких сайтах, чтобы получить представление о количестве запросов, сделанных для сайта. Вы можете быть удивлены тем, как много контента передается!

## ПРОЕКТ № 37:

### Запуск собственного веб-сервера

В этом проекте вы настроите Raspberry Pi на работу в качестве веб-сервера. Для этого вы будете использовать Python 3, поэтому можете выполнять эти шаги на любом устройстве с установленным Python 3, хотя приведенные здесь шаги были написаны с учетом Raspberry Pi. Наш простой веб-сайт будет возвращать содержимое файла при получении запроса.

Откройте окно терминала и создайте каталог, где будут храниться файлы, которые будет обслуживать ваш сайт, а затем установите этот новый каталог в качестве текущего.

---

```
$ mkdir web
$ cd web
```

---

Когда запрос делается к корню вашего сайта, программное обеспечение веб-сервера ищет файл с именем *index.html* и возвращает содержимое этого файла клиенту. Давайте создадим очень простой файл *index.html*:

---

```
$ echo "Hello, Web!" > index.html
```

---

Эта команда создает текстовый файл с именем *index.html*, в котором содержится текст «Hello, Web!» Вы можете просмотреть содержимое текстового файла, чтобы убедиться, что он был создан успешно, открыв файл в текстовом редакторе, или вывести его содержимое в терминале, как показано ниже:

---

```
$ cat index.html
```

---

После того как файл создан, давайте воспользуемся встроенным веб-сервером Python, чтобы передать ваше сообщение «Hello, Web!» всем, кто подключится.

---

```
$ python3 -m http.server 8888
```

---

Эта команда указывает Python запустить HTTP-сервер на порт 8888. Давайте проверим, работает ли это так, как нам нужно. Откройте еще одно окно терминала на Raspberry Pi. В этом втором окне терминала введите следующую команду, чтобы сделать GET-запрос к корню вашего нового сайта:

---

```
$ curl http://localhost:8888
```

---

Утилита `curl` может использоваться для выполнения HTTP GET-запросов, а `localhost` – это имя хоста, обозначающее компьютер, который вы используете в данный момент. Эта команда указывает утилите `curl` выполнить HTTP GET к порту 8888 на локальном компьютере. В ответ вы должны увидеть текст «Hello, Web!» Кроме того, в исходном терминале вы должны увидеть, что пришел запрос GET.

Теперь давайте попробуем подключиться к вашему сайту через веб-браузер. На рабочем столе Raspberry Pi откройте веб-браузер Chromium. В адресной строке введите `http://localhost:8888`. В браузере должен появиться текст с вашего сайта, как показано на рис. 12-13.

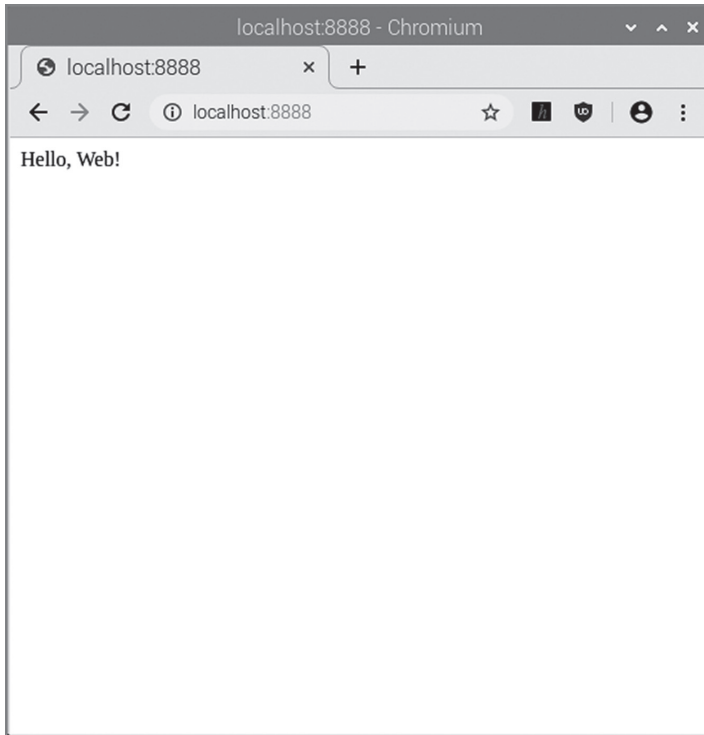


Рис. 12-13. Подключение к локальному веб-серверу с помощью браузера Chromium

Теперь попробуйте подключиться к вашему веб-сайту с другого устройства. Чтобы это сработало, второе устройство должно находиться в той же сети, что и ваш Raspberry Pi. Например, они оба должны находиться в одной сети Wi-Fi. Или если у вашего Raspberry Pi есть публичный IP-адрес (см. проект № 34 на стр. 352), то ваш сайт будет доступен любому устройству в интернете! Сначала узнайте IP-адрес вашего Raspberry Pi, выполнив следующую команду во втором окне терминала:

---

```
$ ifconfig | grep inet
```

---

Скорее всего, будет возвращено несколько IP-адресов. Вы не можете использовать 127.0.0.1 при подключении с удаленного устройства, поэтому выберите другой IP-адрес, назначенный вашей Raspberry Pi. Получив IP-адрес, откройте браузер на другом устройстве. Это может быть смартфон, ноутбук или любое другое устройство в вашей сети, имеющее веб-браузер. В окне браузера введите в адресную строку следующее: `http://w.x.y.z:8888` (замените `w.x.y.z` на IP-адрес вашего устройства). Нажмите **ENTER** или соответствующую кнопку в браузере, чтобы перейти по этому адресу. В браузере должно появиться сообщение «Hello, Web!»

Если это не сработало и у вашего Raspberry Pi нет публичного IP-адреса, убедитесь, что оба устройства находятся в одной физической локальной сети. Кроме того, иногда веб-сервер Python перестает отвечать на новые запросы. Если веб-сервер перестал отвечать, вы можете перезапустить его.

Чтобы остановить веб-сервер, перейдите в терминал, где была выполнена команда `http.server`, и нажмите **CTRL-C** на клавиатуре. Затем перезапустите сервер, снова выполнив команду `python3 -m http.server 8888` (нажмите на клавиатуре на стрелку вверх, чтобы получить последнюю команду).

После того как сайт заработает, попробуйте отредактировать файл `index.html` и изменить сообщение так, как вам хочется. Для этого вы можете использовать текстовый редактор по вашему выбору. Когда файл `index.html` будет обновлен, перезагрузите веб-страницу в браузере, чтобы увидеть изменения!

Если вы не хотите, чтобы другие устройства могли получить доступ к вашему сайту, можете ограничить доступ, чтобы ответ получали только запросы от самого Raspberry Pi. Это можно сделать, запустив веб-сервер Python с опцией `--bind`, вот так:

---

```
$ python3 -m http.server 8888 --bind 127.0.0.1
```

---

Чтобы запустить веб-сервер с опцией `--bind`, сначала необходимо остановить все запущенные экземпляры веб-сервера (нажмите **CTRL-C** на клавиатуре).

## ПРОЕКТ № 38: Возврат HTML с вашего веб-сервера

Необходимое условие: проект № 37.

В этом проекте вы обновите свой локальный веб-сервер, чтобы он возвращал HTML вместо простого текста. Используйте текстовый редактор по вашему выбору, откройте файл `index.html` (который был создан в проекте № 37) и замените весь текст в файле на следующий HTML. Это тот же самый HTML-код, который обсуждался в этой главе. Вам не нужно беспокоиться об отступах в каждой строке, поскольку лишние пробелы в HTML не имеют значения.

---

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Cat</title>
  </head>
  <body>
    <h1>Thoughts on a Cat</h1>
    <p>This is a cat.</p>
    
  </body>
</html>
```

---

После обновления файла вы снова будете использовать встроенный веб-сервер Python. Если он еще не запущен, запустите его с помощью этой команды. Перед выполнением команды убедитесь, что окно терминала находится в каталоге `web`.

---

```
$ python3 -m http.server 8888
```

---

Теперь используйте веб-браузер для подключения к вашему веб-серверу, как делали в проекте № 37. Вы должны увидеть отображение страницы, но без фотографии кошки. Если вы посмотрите в окно терминала, где выполняли команду Python для веб-сервера, то должны увидеть неудачную попытку получить фотографию кошки, как показано ниже:

---

```
192.168.1.123 - - [31/Jan/2020 17:38:56] "GET /cat.jpg HTTP/1.1" 404 -
```

---

Код ошибки 404 означает, что ресурс не может быть найден, что вполне логично, учитывая, что у вас нет файла с именем `cat.jpg` в этом каталоге! Почему веб-браузер вообще запросил фотографию кошки? Если вы посмотрите на HTML-код страницы, то увидите тег HTML `<img>`, который направляет браузер на показ изображения `cat.jpg`. Браузер запрашивает изображение, но не может его получить, поскольку файл отсутствует.

Давайте решим проблему с отсутствующим изображением кошки. Вам нужно загрузить изображение кошки (или на самом деле изображение чего угодно) в формате JPEG и сохранить его как `~/web/cat.jpg`. Чтобы упростить эту задачу, можете загрузить изображение, использованное в этой главе, с помощью следующей команды. Перед выполнением команды убедитесь, что ваш терминал находится в каталоге `web`.

---

```
$ wget https://www.howcomputersreallywork.com/images/cat.jpg
```

---

Теперь изображение `cat.jpg` должно находиться в вашем веб-каталоге. Перезагрузите страницу в веб-браузере, чтобы увидеть изображение кошки

на странице. Напоминание: если веб-сервер завис, перезапустите его, как описано в проекте № 37.

Стоит отметить, что вы можете не только просмотреть изображение кошки на странице, но и запросить его непосредственно с сервера, поскольку оно имеет свой собственный URL. Попробуйте ввести в браузере следующий URL (заменяв *SERVER* на имя хоста или IP-адрес, который вы использовали для своего сайта): *http://SERVER:8888/cat.jpg*. Вы должны увидеть изображение кошки, отображаемое в браузере за пределами веб-страницы. Каждый ресурс, на который ссылается веб-страница, имеет свой собственный URL, и к нему можно получить прямой доступ!

## ПРОЕКТ № 39: добавление CSS на ваш сайт

Необходимые условия: проект № 38.

В этом проекте вы будете использовать CSS для придания стиля вашему сайту. Во-первых, используйте текстовый редактор по вашему выбору, чтобы создать файл с именем *style.css* в каталоге *web*. Этот файл будет содержать ваши правила CSS. Убедитесь, что файл с именем *style.css* сохранен в директории *web* вместе с файлами *index.html* и *cat.jpg*. Содержимое *style.css* должно быть следующим:

```
p {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 11pt;
    margin-left: 10px;
    color: DimGray;
}

h1 {
    font-family: 'Courier New', Courier, monospace;
    font-size: 18pt;
    font-weight: bold;
}
```

После создания *style.css* откройте файл *index.html* для редактирования, как это было сделано в предыдущем проекте. Оставьте существующий HTML на месте. Мы просто хотим добавить одну строку в раздел *head*, как показано здесь:

```
<head>
  <meta charset="utf-8">
  <title>A Cat</title>
  <link rel="stylesheet" type="text/css" href="style.css">❶
</head>
```



После того как вы внесете изменения ❶ в файл *index.html*, запустите свой веб-сервер (если он еще не запущен) и перезагрузите страницу в веб-браузере. Вы увидите, что стиль страницы обновился. Напоминание: если веб-сервер завис, перезапустите его, как описано в проекте № 37.

Не стесняйтесь редактировать файл *style.css*, чтобы попробовать разные стили. Может быть, вы хотите сделать шрифт абзаца огромным или другого цвета! Отредактируйте стиль по своему вкусу, сохраните *style.css* и перезагрузите страницу в браузере.

Если вы не видите отражения обновлений в браузере, это может быть связано с тем, что ваш браузер загружает кешированную копию вашего сайта, а не последнюю версию. Попробуйте открыть страницу в новой вкладке или вообще перезапустить браузер. Вы также можете указать браузеру обходить локальный кеш при перезагрузке. Для этого перейдите на страницу, а затем нажмите **CTRL-F5**, чтобы заставить страницу перезагрузиться. Это работает в большинстве браузеров на Windows и Linux. На Mac можно принудительно обновить страницу в Chrome и Firefox с помощью **CMD-SHIFT-R**. Иногда требуется несколько обновлений, прежде чем браузер отобразит последнюю версию сайта.

## ПРОЕКТ № 40: Добавьте JavaScript на свой сайт

Необходимое условие: проект № 39.

В этом проекте вы будете использовать JavaScript, чтобы сделать сайт интерактивным. Сначала с помощью текстового редактора по своему выбору создайте файл *cat.js* в каталоге *web*. Этот файл будет содержать код JavaScript. Убедитесь, что файл назван *cat.js* и сохранен в каталоге *web* вместе с файлами *index.html* и *cat.jpg*. Содержимое файла *cat.js* должно быть следующим:

```
document.addEventListener('DOMContentLoaded', function() {
    document.getElementById('cat-photo').onclick = function() {
        document.getElementById('cat-para').innerHTML += ' Meow!';
    };
});
```

После сохранения файла *cat.js* откройте файл *index.html* для редактирования, как вы это делали в предыдущем проекте. Оставьте существующий HTML на месте и внесите изменения, показанные здесь:

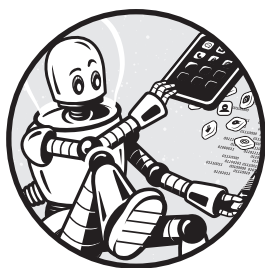
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A Cat</title>
    <link rel="stylesheet" type="text/css" href="style.css">
    <script src="cat.js"></script>❶
  </head>
  <body>
    <h1>Thoughts on a Cat</h1>
    <p id="cat-para"❷>This is a cat.</p>
    
  </body>
</html>
```

Эти изменения ссылаются на скрипт ❶ и присваивают идентификаторы абзацу ❷ и изображению ❸.

После внесения этих изменений в файл *index.html* запустите веб-сервер (если он еще не запущен) и перезагрузите страницу в веб-браузере. Теперь вы должны иметь возможность кликнуть (или коснуться) фотографию кошки и увидеть слово «Meow!», добавленное к абзацу. Напоминание: если веб-сервер завис, перезапустите его, как описано в проекте № 37.

# 13

## СОВРЕМЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ ТЕХНОЛОГИИ



В этой главе представлен обзор нескольких областей современных вычислительных технологий. Учитывая разнообразие и широту охвата вычислительной техники, у меня был широкий выбор тем.

Выбранные мною области ни в коем случае не являются исчерпывающим списком интересных вещей, происходящих сегодня в вычислительной технике. Они представляют собой горстку тем, которые, на мой взгляд, заслуживают вашего внимания. В этой главе мы рассмотрим приложения, виртуализацию, облачные вычисления, биткойн и многое другое. В заключение разберем последний проект, который объединяет многие темы, затронутые в этой книге.

### Приложения

С первых дней появления компьютеров люди называли *приложениями* (*applications*) программы, которые используются непосредственно пользователями. Для удобства английский термин был сокращен до *app*, и в прошлом эти два английских термина были взаимозаменяемы. Однако с тех пор, как в 2008 году компания Apple открыла магазин приложений для iPhone

(App Store), слово *app* приобрело особое значение. Хотя и не существует стандартного технического определения того, что делает программу приложением, как правило, приложения имеют ряд общих характеристик<sup>1</sup>.

Приложения предназначены для конечных пользователей. Приложения часто разработаны для мобильных устройств, таких как смартфон или планшет. Приложения обычно распространяются через цифровой интернет-магазин (*магазин приложений*), такой как Apple App Store, Google Play Store или Microsoft Store. Приложения имеют ограниченный доступ к системе, на которой они работают, и часто должны объявлять, какие конкретные возможности им требуются для работы. Приложения, как правило, используют сенсорные экраны в качестве основного средства пользовательского ввода. Термин «приложение», когда он используется отдельно, обычно подразумевает программное обеспечение, установленное на устройстве, которое напрямую использует API операционной системы. Другими словами, термин «приложение» обычно означает *нативное приложение* (*native app*), т. е. приложение, созданное для конкретной операционной системы. В отличие от этого *веб-приложения* разработаны с использованием веб-технологий (HTML, CSS и JavaScript) и не привязаны к конкретной ОС. На рис. 13-1 представлен общий вид нативных приложений и веб-приложений.



Рис. 13-1. Нативные приложения создаются для конкретной ОС. Веб-приложения создаются с использованием веб-технологий

Как показано на рис. 13-1, нативные приложения обычно устанавливаются из магазина приложений и предназначены для использования возможностей конкретной операционной системы. Веб-приложения обычно запускаются с веб-сайта и разработаны с использованием веб-технологий. Веб-приложения запускаются в браузере или другом процессе, который отображает веб-контент. Давайте рассмотрим нативные и веб-приложения более подробно.

<sup>1</sup> В русском языке не сложилось отдельного термина для мобильных приложений, каким стало английское сокращение *app*. Однако, *app* – это не совсем обычная программа, такая же, какие читатель создавал с помощью С и Python в главе 9. Поэтому в русских текстах не всегда можно отличить, когда речь идет о мобильном приложении (*app*), а когда – о пользовательских программах (*application*). В частности, в этом разделе («Приложения») далее автор ведет речь исключительно о мобильных приложениях (*app*), а в следующем («Виртуализация и эмуляция») – уже об обычных программах. – Прим. ред.

## Нативные приложения

Как уже упоминалось ранее, нативные приложения создаются для конкретной операционной системы. Apple App Store и последовавшие за ним аналогичные магазины приложений открыли новую эру разработки нативных приложений, предоставив разработчикам новые платформы, новые методы распространения своего программного обеспечения и новые способы зарабатывать деньги на программном обеспечении.

Разработка нативных приложений в наше время в основном сосредоточено на двух платформах: iOS и Android. Конечно, программное обеспечение разрабатывается и для других операционных систем, но зачастую оно не обладает типичными характеристиками приложений (дружественность к мобильным устройствам, сенсорный ввод, распространение через магазин приложений и т. д.)

Android и iOS отличаются языками программирования, API и многим другим. Поэтому для написания приложения, работающего и на iOS, и на Android, требуется либо ведение отдельных баз кода, либо использование *кросс-платформенного фреймворка*<sup>1</sup>, такого как Xamarin, React Native, Flutter или Unity. Эти кроссплатформенные решения абстрагируют базовые детали API каждой операционной системы, давая разработчикам возможность писать код, который может быть внедрен для работы на нескольких платформах. Многие нативные приложения также полагаются на веб-службы, и разработчики приложений должны не только писать и поддерживать код для iOS и Android, но и создавать такие веб-службы или интегрироваться с ними.

Разработка кросс-платформенного приложения с подключением к интернету требует значительных усилий и опыта! В прошлом разработчики часто фокусировались только на одной платформе, например Windows PC или Mac. Сегодня разработчикам, ориентированным на несколько платформ и веб, приходится гораздо сложнее. Конкуренция платформ, в общем, хороша для пользователей, но это означает больше работы для разработчиков.

Интересно, что нынешняя ситуация с разработкой приложений могла бы сложиться совершенно иначе. Когда в январе 2007 года был анонсирован iPhone, Стив Джобс (в то время генеральный директор Apple) сказал следующее о разработке приложений для iPhone сторонними разработчиками:

«Весь движок Safari находится внутри iPhone. И поэтому вы можете писать прекрасные приложения Web 2.0 и Ajax, которые выглядят и ведут себя точно так же, как приложения на iPhone. И эти приложения могут прекрасно интегрироваться с сервисами iPhone. Они могут звонить, отправлять электронную почту, искать местоположение на Google Maps. И знаете что? Вам не нужен SDK!»

<sup>1</sup> Фреймворк (от *framework* – букв. остов, каркас) – специальное программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта. – Прим. ред.

*SDK (software development kit) – это набор средств разработки, используемый разработчиками для создания приложений для определенной платформы.*

Исходя из этой цитаты, первоначальный план Apple по разработке приложений сторонними разработчиками заключался в том, чтобы просто позволить им создавать сайты, похожие на приложения, которые могли бы использовать возможности iPhone. Разработка нативных приложений ограничивалась бы приложениями, которые Apple разработала и включила в комплект поставки iPhone, например приложениями «Камера», «Почта» и «Календарь».

В то время использование веба в качестве платформы для разработки приложений не было распространено. Позиция Apple была ориентирована на будущее. К сожалению, технологии, лежащие в основе веба в 2007 году, были недостаточно зрелыми, чтобы позиционировать веб как настоящую платформу для приложений. В октябре 2007 года Apple изменила свою позицию, объявив, что позволит разработчикам создавать нативные приложения для iPhone. В 2008 году компания Apple открыла магазин App Store как единственный поддерживаемый механизм распространения нативных приложений для iPhone среди пользователей.

Изменение политики Apple пошло на пользу компании, поскольку App Store стал источником дохода для Apple. Регистрация в качестве разработчика App Store платная, плюс Apple получает процент от каждой продажи. App Store и нативная разработка для iPhone также открыли дверь для эксклюзивного контента, приложений, которые работают только на устройствах Apple.

App Store также предоставляет преимущества конечным пользователям. Полезен тщательно подобранный список приложений с рейтингами, и магазин обеспечивает меры, внушающие доверие потребителям. Приложения, которые попадают в App Store, должны соответствовать определенным стандартам качества. Централизованный платежный сервис означает, что пользователям не нужно предоставлять свои платежные данные нескольким компаниям. Приложения обновляются автоматически, что является преимуществом перед традиционным программным обеспечением для ПК, хотя и не является преимуществом перед вебом, поскольку веб-приложения также обновляются без участия пользователя.

После успеха Apple App Store другие компании создали аналогичные цифровые магазины для распространения программного обеспечения. Google Play Store, Microsoft Store и Amazon Appstore работают по модели, схожей с магазином Apple, и предоставляют аналогичные преимущества. Хотя эта система в целом хорошо сработала для этих компаний и для конечных пользователей, она также создала сложную среду для разработчиков: несколько магазинов, несколько платформ и различные технологии. Каждый цифровой магазин имеет свои требования, которым должны соответствовать разработчики приложений, и каждый магазин получает процент от выручки.

## Веб-приложения

Одновременно с ростом нативных приложений веб превратился в платформу, вполне способную работать с приложениями. Была представлена усовершенствованная версия HTML, известная как HTML5, а веб-браузеры стали более способными и последовательными в обработке контента. Разработчики браузеров сделали свои реализации JavaScript совместимыми со стандартом ECMAScript 5, обеспечив лучшую основу для кода JavaScript. Помимо обновлений браузеров, сообщество веб-разработчиков приняло (и продолжает принимать) концепцию, известную как *адаптивный веб-дизайн*, подход, который обеспечивает хорошее отображение веб-контента независимо от размера экрана, на котором он отображается. Используя технологии адаптивного дизайна, веб-разработчики могут поддерживать единый веб-сайт, который хорошо работает на различных устройствах, вместо того чтобы создавать отдельные веб-сайты, предназначенные для разных устройств. Кроме того, в последние годы было выпущено множество *фреймворков для веб-разработки*, таких как Angular и React. Эти фреймворки облегчают разработчикам написание и поддержку *веб-приложений* – веб-сайтов, которые ведут себя как приложения.

Разработчики поняли, что современные веб-технологии можно использовать для создания вещей, очень похожих на нативные приложения, и многие разработчики создают веб-сайты, которые функционируют как приложения. Некоторые разработчики вообще отказались от нативных приложений и создают только веб-приложения. Преимущества такого подхода в том, что веб-приложение будет работать на любом устройстве с современным браузером, а код нужно написать только один раз. Однако у веб-приложений тоже есть некоторые недостатки. Веб-приложения не имеют доступа к полному спектру возможностей устройств, работают медленнее, чем нативные приложения, требуют, чтобы пользователь находился в сети, и, как правило, не представлены в магазинах приложений.

Чтобы устранить некоторые недостатки веб-приложений, *прогрессивные веб-приложения* (*Progressive Web Apps, PWA*) предлагают набор технологий и рекомендаций, которые помогают преодолеть разрыв между нативными и веб-приложениями. PWA – это просто веб-сайт с несколькими дополнительными функциями, которые помогают ему быть более похожим на приложение. Прогрессивное веб-приложение должно обслуживаться по HTTPS, корректно отображаться на мобильных устройствах, иметь возможность работать в режиме офлайн после загрузки, предоставлять браузеру манифест с описанием приложения и быстро переходить от одной страницы к другой. Для конечного пользователя работа PWA должна быть такой же адаптивной и естественной, как и работа нативного приложения. Если сайт соответствует критериям PWA, современные веб-браузеры предоставляют пользователям возможность добавить иконку PWA на домашний экран или рабочий стол. Это означает, что пользователи могут запускать веб-приложение так же, как и нативное приложение. Приложение открывается в собствен-

ном окне, а не в окне браузера, и в целом ведет себя так же, как нативное приложение.

PWA потенциально могут предложить большие преимущества разработчикам, которые хотят использовать веб-технологии для своих приложений, но не хотят создавать несколько приложений для разных платформ. Однако у PWA все же есть некоторые недостатки. Существенным из них является то, что PWA не появляются в магазинах приложений. Мобильные операционные системы уже много лет приучают пользователей к тому, что приложения должны быть получены через магазины приложений. Пользователи не привыкли переходить на веб-страницу, чтобы получить приложение. На момент написания этой книги только Microsoft Store позволяет размещать PWA непосредственно в магазине. Другие платформы предполагают, что PWA будут устанавливаться из браузера или переустанавливаться в нативное приложение, которое отображает веб-контент. Такое перепакованное приложение может быть затем представлено в магазине. Еще одним потенциальным недостатком является то, что PWA могут выглядеть не как нативные приложения. Обычно они выглядят одинаково на всех платформах, хотя некоторые могут считать это положительным моментом. PWA по-прежнему не обладают производительностью нативного приложения или доступом ко всем возможностям базовой платформы, но это не обязательно является проблемой, так как это зависит от потребностей приложения.

## Виртуализация и эмуляция

Когда компьютер не является физическим устройством? Конечно, когда это виртуальный компьютер! *Виртуализация* – это процесс использования программного обеспечения для создания виртуального представления компьютера. Связанная с ней технология, *эмуляция*, позволяет приложениям<sup>1</sup>, разработанным для определенного типа устройства, работать на совершенно другом типе устройства. В этом разделе мы рассмотрим как виртуализацию, так и эмуляцию.

### Виртуализация

Виртуальный компьютер, известный как *виртуальная машина* (*virtual machine, VM*), работает под управлением операционной системы, как и физический компьютер. В свою очередь приложения работают под управлением этой операционной системы. С точки зрения приложения, виртуализированное оборудование действует как физический компьютер. Виртуализация позволяет реализовать несколько полезных сценариев. Компьютер с одной операционной системой может работать под управлением другой операционной системы на виртуальной машине. Например, компьютер под управлением Windows может запу-

---

<sup>1</sup> Напомним, что здесь под приложением опять понимается обычная программа, а не мобильное приложение, как в предыдущем разделе. – *Прим. ред.*



скасть экземпляр Linux на виртуальной машине. Виртуальные машины также позволяют центрам обработки данных размещать несколько виртуальных серверов на одном физическом сервере.

Это дает возможность хостинговым компаниям легко и быстро предоставлять выделенные серверы своим клиентам, если клиент не против виртуального сервера. Для виртуальных машин можно легко делать резервные копии, восстанавливать их и разворачивать.

*Гипервизор (Hypervisor)* – это программная платформа, на которой работают виртуальные машины. Существует два типа гипервизоров, как показано на рис. 13-2.

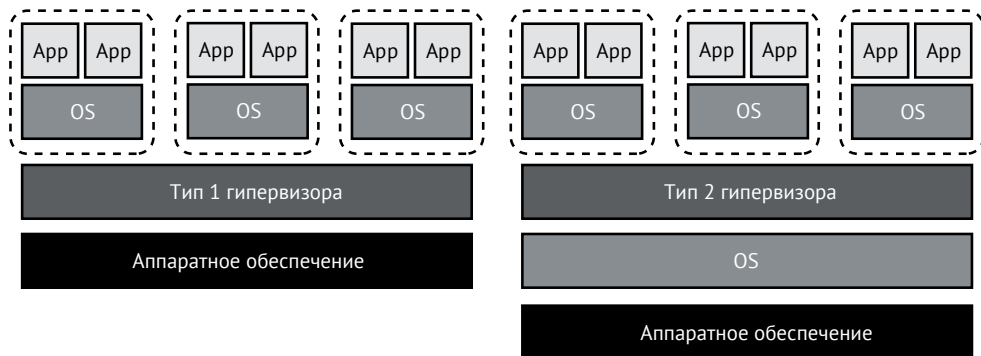


Рис. 13-2. Гипервизоры типа 1 и типа 2

Как показано в левой части рис. 13-2, гипервизор может напрямую взаимодействовать с базовым оборудованием, фактически располагая гипервизор ниже ядра в технологическом стеке. Гипервизор общается с физической аппаратурой и предоставляет виртуализированное оборудование ядру ОС. Такой гипервизор называется *гипервизором первого типа*. В отличие от него *гипервизоры второго типа*, показанные справа на рис. 13-2, работают как приложение в операционной системе. Microsoft Hyper-V и VMware ESX относятся к гипервизорам первого типа, а VMware Player и VirtualBox – к гипервизорам второго типа.

Еще один популярный подход к виртуализации – использование контейнеров. *Контейнер* представляет собой изолированную среду пользовательского режима, в которой можно запускать приложения. В отличие от виртуальной машины контейнер разделяет ядро с основной ОС и с другими контейнерами, запущенными на том же компьютере. Процесс, запущенный в контейнере, может видеть только часть ресурсов, доступных на физической машине. Например, каждому контейнеру может быть предоставлена своя изолированная файловая система. Контейнеры обеспечивают изоляцию VM без издержек, связанных с запуском отдельного ядра для каждой VM. В целом контейнеры ограничены запуском той же операционной системы, что и хост, поскольку ядро является общим. Некоторые контейнерные технологии, например OpenVZ, используются для виртуализации всей части пользовательского режима операционных систем, в то время как другие, например Docker, используются для запуска отдельных приложений в изолированных контейнерах.

## ПРИМЕЧАНИЕ

Вы можете вспомнить, что процесс операционной системы также описывался как «контейнер» в главе 10, но это не то же самое, что контейнер виртуализации.

## Эмуляция

Эмуляция – это использование программного обеспечения для того, чтобы заставить один тип устройства вести себя как другой тип устройства. Эмуляция и виртуализация похожи тем, что обе они предоставляют виртуальную среду для работы программного обеспечения, но если виртуализация предлагает часть базового оборудования, то эмуляция представляет виртуальное оборудование, которое *не похоже* на используемое физическое оборудование. Например, виртуальная машина или контейнер, работающий на процессоре x86, запускает программное обеспечение, скомпилированное для x86, напрямую используя физический процессор. В отличие от этого *эмулятор* (программа, выполняющая эмуляцию), работающий на аппаратном обеспечении x86, может запускать программное обеспечение, скомпилированное для совершенно другого процессора. Эмуляторы часто предоставляют и другое виртуальное оборудование, помимо процессора.

Например, полный эмулятор для Sega Genesis (видеоигровой системы 1990-х годов) будет эмулировать процессор Motorola 68000, звуковой чип Yamaha YM2612, контроллеры ввода и любое другое оборудование, которое можно найти в Sega Genesis. Во время работы такой эмулятор переводит инструкции процессора, изначально предназначенные для работы на Sega Genesis, в возможности, реализованные в коде x86. Это влечет за собой значительные нагрузки, поскольку каждая инструкция процессора должна быть переведена, но достаточно быстрый современный компьютер все равно может эмулировать гораздо более медленную Sega Genesis на полной скорости. В результате появляется возможность запускать программное обеспечение, предназначенное для одной платформы, на совершенно другой платформе, как показано на рис. 13-3.

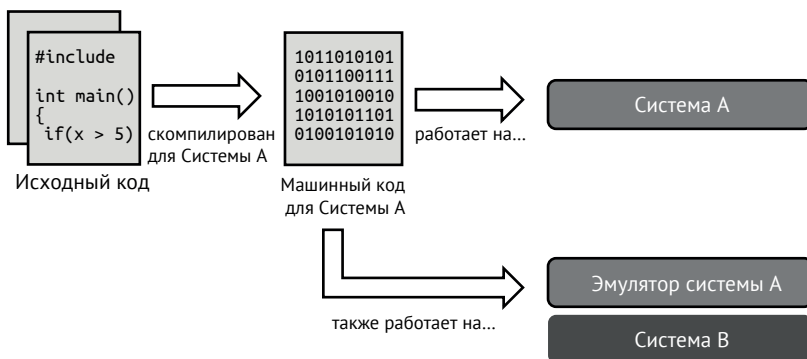


Рис. 13-3. Код, скомпилированный для системы A, может работать на эмуляторе для системы A

Эмуляция играет важную роль в сохранении программного обеспечения, разработанного для устаревших платформ. По мере старения вычислительных платформ становится все труднее найти работающее оборудование. Эмуляция повсеместно используется разработчиками программного обеспечения, которые хотят перенести старое программное обеспечение на современную платформу. Оригинальный исходный код может быть утерян, или его модернизация может оказаться обременительной. В таких случаях применение эмулятора позволяет первоначально скомпилированному коду работать на новой платформе без изменений.

### Процессные виртуальные машины (process virtual machines)<sup>1</sup>

Существует еще один тип виртуальных машин, который имеет некоторые общие черты с эмуляторами. *Процессная виртуальная машина* запускает приложение в среде выполнения, которая абстрагируется от базовой операционной системы. Она похожа на эмулятор тем, что предоставляет платформу для выполнения, которая отделена от аппаратного обеспечения и OS, на которых она работает. Однако, в отличие от эмулятора, процессная VM не пытается имитировать реальное оборудование. Скорее, она предоставляет среду, предназначенную для выполнения платформенно-независимого программного обеспечения. Как мы обсуждали в главе 9, Java и .NET используют процессные виртуальные машины, выполняющие байткод.

## Облачные вычисления

*Облачные вычисления* – это предоставление вычислительных услуг через интернет. В этом разделе мы рассмотрим различные типы облачных вычислений, но сначала давайте быстро пройдемся по истории удаленных вычислений.

### История удаленных вычислений

С начала развития вычислительной техники мы можем наблюдать колебания от удаленных централизованных вычислений (доступ к серверам с терминалов) к локальным вычислениям (настольные

<sup>1</sup> Для того, что в среде специалистов именуется *process virtual machines*, был предложен специальный термин по-русски – *языковая виртуальная машина*, *языковая ВМ*, т. е. эмулирующая среду выполнения программы для определенного языка (в отличие от *системной ВМ*, предоставляющей доступ к реальному или виртуальному оборудованию). Однако термин не является общепринятым, и здесь приведен буквальный перевод английского термина. – *Прим. ред.*

компьютеры), а теперь снова к удаленным (веб), доступ к которым осуществляется с интеллектуальных локальных устройств (например, смартфонов). Сегодня многие приложения используют комбинацию удаленных и локальных вычислений. В случае с веб-приложениями часть кода выполняется в браузере, а часть – на веб-сервере. Устройства, которые мы носим сегодня в кармане, значительно мощнее, чем комнатные компьютеры первых дней развития вычислительной техники, однако большая часть того, что мы хотим делать на этих устройствах, связана с взаимодействием с другими компьютерами, поэтому вполне логично, что ответственность за обработку данных должна быть распределена между локальными устройствами и удаленными серверами.

С возрождением удаленных вычислений у организаций возникла потребность в обслуживании серверов. В прошлом это означало покупку физического сервера, его настройку в соответствии с потребностями, подключение к сети и предоставление ему возможности работать где-нибудь в чулане. Организация имела физический доступ к машине и полный контроль над ее конфигурацией. Однако обслуживание сервера (или целого парка серверов) может быть сложным и дорогостоящим мероприятием. Сюда входят затраты на приобретение и обслуживание аппаратного обеспечения, обновление программного обеспечения, решение проблем безопасности и планирования мощностей, управление конфигурацией сети и т. д. Часто набор навыков и знаний, необходимых для этой работы, не очень хорошо согласуется с целями организации. Даже компания, ориентированная на технологии, не всегда хочет заниматься обслуживанием серверов. Именно здесь на помощь приходят облачные вычисления.

*Облачные вычисления* предоставляют удаленные вычислительные возможности через интернет (*облако*). Базовое оборудование обслуживается компанией, предоставляющей облачные услуги (*облачный провайдер*), освобождая организацию или пользователя, нуждающегося в таких возможностях (*потребитель облачных услуг*), от необходимости содержать серверы. Облачные вычисления позволяют приобретать вычислительные услуги по мере необходимости. Для потребителя облачных вычислений это означает отказ от контроля над некоторыми вещами и доверие третьей стороне в предоставлении надежных услуг. Облачные вычисления имеют множество форм, давайте рассмотрим некоторые из них.

## **Категории облачных вычислений**

Различные категории облачных вычислений обычно определяются границей, которая разделяет ответственность между поставщиком и потребителем облачных услуг. На рис. 13-4 представлен обзор четырех категорий облачных вычислений (IaaS, PaaS, FaaS и SaaS) и соответствующих им областей ответственности. Мы рассмотрим каждую из этих категорий вкратце.

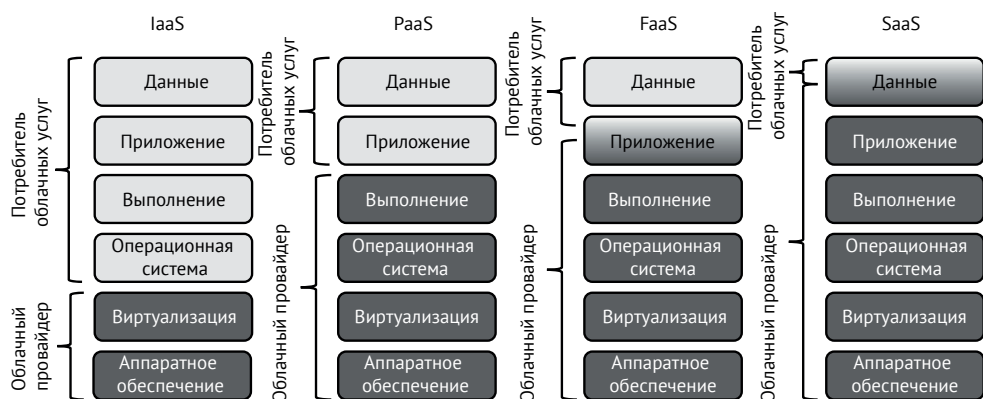


Рис. 13-4. Распределение ответственности в различных типах облачных вычислений

Вертикальные стеки на рис. 13-4 представляют собой компоненты, необходимые для работы приложения. Независимо от того, какая категория облачных вычислений используется, все компоненты должны присутствовать. Разница между категориями заключается в том, кто отвечает за управление каждым компонентом, облачный поставщик или потребитель облачных услуг. Различные компоненты в каждом стеке должны выглядеть знакомыми, поскольку мы уже рассматривали эти темы. Однако *время выполнения* требует пояснений. *Среда выполнения* – это среда, в которой выполняется приложение, включая все необходимые библиотеки, интерпретаторы, виртуальные машины и т. д. Давайте теперь рассмотрим четыре категории облачных вычислений, показанные на рис. 13-4, двигаясь слева направо.

*Инфраструктура как услуга (Infrastructure as a Service, IaaS)* – это модель облачных вычислений, в которой поставщик облака управляет только аппаратным обеспечением и виртуализацией, позволяя пользователю управлять операционной системой, средой выполнения, кодом приложения и данными.

Потребитель IaaS обычно получает подключенный к интернету виртуальный компьютер для использования по своему усмотрению, как правило, в качестве сервера. Этот виртуальный компьютер обычно реализуется как виртуальная машина на основе гипервизора или как контейнер пользовательской части дистрибутива Linux. Потребитель виртуального сервера IaaS имеет доступ к операционной системе виртуального компьютера и может настраивать его по своему усмотрению. Это дает потребителю максимальную гибкость, но это также означает, что ответственность за поддержку программного обеспечения системы (включая операционную систему, программное обеспечение сторонних производителей и т. д.) полностью ложится на плечи пользователя. IaaS предоставляет виртуальный компьютер, а потребитель несет ответственность за все, что работает на этом компьютере. Вот несколь-

ко примеров IaaS: Amazon Elastic Compute Cloud (EC2), Microsoft Azure Virtual Machines и Google Compute Engine.

*Платформа как услуга (Platform as a Service, PaaS)* дает облачному провайдеру больше ответственности. В случае PaaS провайдер управляет не только оборудованием и виртуализацией, но и операционной системой и средой выполнения, которую хочет использовать потребитель. Потребитель облачных услуг PaaS разрабатывает приложение, предназначенное для работы на выбранной им облачной платформе, используя различные возможности, присущие только этой платформе. Пользователям PaaS не нужно заботиться о поддержке базовой ОС или среды выполнения. Потребитель облачных услуг может просто сосредоточиться на коде своего приложения. Хотя облачный провайдер абстрагирует детали базовой системы, потребителю все равно необходимо управлять ресурсами, предоставляемыми провайдером для работы его приложения. Это включает в себя объем необходимого хранилища и тип выделенных виртуальных машин. PaaS предоставляет управляемую платформу для выполнения кода, а потребитель отвечает за приложение, которое работает на этой платформе. Вот некоторые примеры PaaS: Amazon Web Services Elastic Beanstalk, Microsoft Azure App Service и Google App Engine.

*Функция как услуга (Function as a Service, FaaS)* еще на один шаг дальше модели PaaS. Она не требует от потребителя развертывания полного приложения или предварительной подготовки экземпляров платформы. Вместо этого потребителю нужно только разместить свой код (функцию), который запускается в ответ на определенные события. Например, разработчик может написать функцию, которая возвращает расстояние до ближайшего продуктового магазина. Эта функция может запускаться в ответ на отправку веб-браузером своих текущих GPS-координат на URL. Эта событийно ориентированная модель означает, что облачный провайдер отвечает за вызов кода потребителя по требованию. Потребителю облачных услуг больше не нужно постоянно запускать код приложения, ожидая запросов. Это может упростить работу для потребителя и снизить затраты, хотя это может означать более медленное время отклика при поступлении запроса, если функциональный код еще не запущен.

FaaS является разновидностью *бессерверных вычислений*, модели облачных вычислений, в которой потребителям не нужно заниматься управлением серверами или виртуальными машинами. Конечно, этот термин является неправильным, серверы на самом деле необходимы для выполнения кода, просто потребителю не нужно о них думать! FaaS предоставляет событийно-ориентированную платформу для запуска кода, а потребитель отвечает за код, который запускается в ответ на события. Некоторые примеры FaaS включают Amazon Web Services Lambda, Microsoft Azure Functions и Google Cloud Functions.

*Программное обеспечение как услуга (Software as a Service, SaaS)* – это принципиально иной тип облачных услуг. SaaS предоставляет потребителю приложение, которое полностью управляется в облаке. В то время как IaaS, PaaS и FaaS предназначены для команд разработчиков программного обеспечения, которые хотят запускать свой собственный код в об-

лаке, SaaS предоставляет конечным пользователям или организациям уже готовое облачное приложение. Сегодня так много программного обеспечения работает в облаке, что это может показаться непримечательным, но это контрастирует с тем, как пользователь или организация устанавливают и обслуживают программное обеспечение на своих локальных устройствах и в сети. SaaS предоставляет полное приложение, управляемое в облаке, а потребитель отвечает только за данные, которые он хранит в этом приложении. Даже управление данными частично осуществляется провайдером, включая детали хранения, резервного копирования и т. д. Некоторые примеры SaaS включают Microsoft 365, Google G Suite и Dropbox.

Среди основных провайдеров облачных услуг можно назвать Amazon Web Services, Microsoft Azure, Google Cloud Platform, IBM Cloud, Oracle Cloud и Alibaba Cloud<sup>1</sup>.

## Невидимый веб и темный веб

Вам наверняка попадались новости о гнусных событиях в *даркнете* (*темном вебе*)<sup>2</sup> или в невидимом вебе. К сожалению, эти два термина часто путают, но они имеют разные значения. Веб можно разделить на три больших сегмента: видимая, невидимая и темная части, как показано на рис. 13-5.

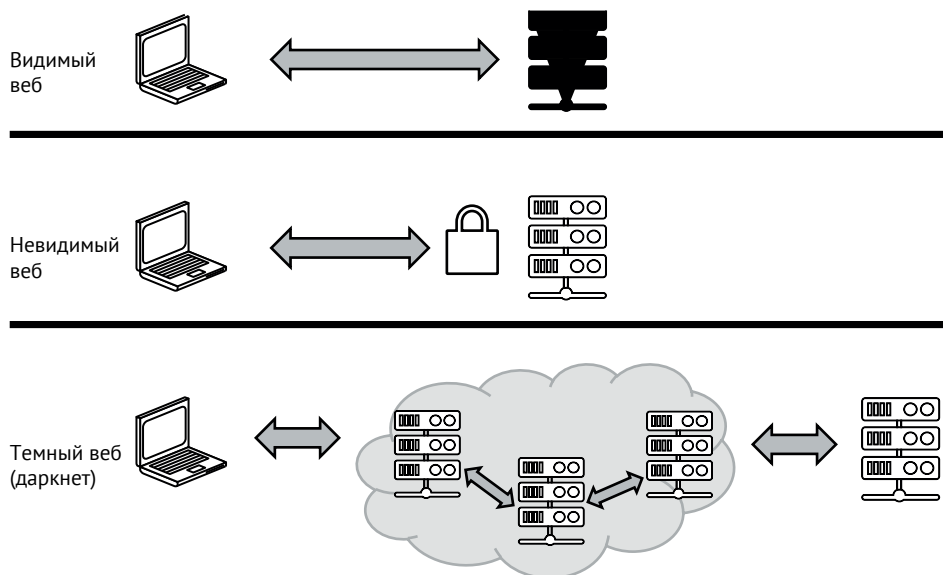


Рис. 13-5. Видимый, невидимый и темный веб

<sup>1</sup> В России крупнейшие облачные провайдеры – SberCloud, «Яндекс», Mail.Ru, МТС, Ростелеком и некоторые другие. – *Прим. ред.*

<sup>2</sup> *Даркнет* (*darknet* – темная сеть) – общепринятое название скрытой части интернета. Автор предпочитает название «темный веб» (*dark web*), в переводе используются оба термина. – *Прим. ред.*



Контент, который находится в свободном доступе для всех желающих, является частью *видимого веба*. Публичные блоги, новостные сайты и публичные сообщения в Twitter – все это примеры видимого веб-контента. Видимый веб индексируется поисковыми системами, и иногда его и определяют как контент, который можно найти с помощью поисковой системы.

*Невидимый веб* – это веб-контент, доступ к которому невозможен без авторизованного входа на сайт или в веб-службу. Большинство пользователей интернета регулярно обращается к невидимому содержанию. Проверка банковского баланса, чтение электронной почты через сайт типа Gmail, вход в Facebook, просмотр истории покупок на Amazon – все это примеры действий в невидимой части сети. Невидимый веб – просто контент, который не находится в открытом доступе и для доступа к которому, как правило, требуется определенный пароль. Большинство пользователей не хочет, чтобы их электронная почта или банковские счета были в открытом доступе, поэтому есть веская причина, по которой этот тип контента не является публичным и не может быть проиндексирован поисковыми системами.

*Даркнет (темный веб)* – это веб-контент, для доступа к которому требуется специализированное программное обеспечение. Вы не можете получить доступ к темной сети, используя только стандартный веб-браузер. Наиболее распространенной технологией даркнета является *Tor* (*The Onion Router* – букв. *луковый маршрутизатор*). С помощью системы шифрования и ретрансляторов Тор обеспечивает анонимный доступ в вебу, не позволяя провайдеру следить за тем, какие сайты посещает пользователь, и не позволяет сайтам узнать IP-адрес своего посетителя. Кроме того, Тор позволяет пользователям получать доступ к скрытым сайтам, известным как *onion services*, которые вообще не могут быть доступны без Тор, эти сайты являются частью даркнета. Тор скрывает IP-адреса *onion services*, делая их также анонимными. Как вы возможно догадываетесь, анонимность темной сети иногда используется в преступных целях. Однако есть и легитимные способы использования конфиденциальности, обеспечиваемой темной сетью, например в политических дискуссиях. Я рекомендую соблюдать осторожность при доступе к контенту в даркнете.

## Биткоин

*Криптовалюта* – это цифровой актив, предназначенный для использования в финансовых операциях в качестве замены традиционной валюты, такой как доллар США. Пользователи криптовалют поддерживают баланс этой валюты, практически как в традиционном банке, и могут тратить ее на товары и услуги. Некоторые пользователи относятся к криптовалютам в первую очередь как к инвестициям, а не как к средству торговли, поэтому для таких пользователей они больше похожи на золото. В отличие от традиционных валют криптовалюты, как правило, децентрализованы, и ни одна организация не контролирует их использование.



## Основы биткоина

Появившись в 2009 году, *биткоин* стал первой децентрализованной криптовалютой, и сегодня он является самой известной. С тех пор появилось большое количество альтернативных криптовалют (известных как *альткоины*<sup>1</sup>), но ни одна из них не смогла оспорить господство биткоина. Основная денежная единица биткоина также называется просто *биткоин*, сокращенно *BTC*.

Биткоин и подобные ему криптовалюты основаны на технологии *блокчейн* (*block chain* – цепь из блоков). В блокчейне информация группируется в структуры данных, называемые *блоками*, а блоки связаны между собой в хронологическом порядке. То есть вновь созданный блок добавляется в конец блокчейна. В случае с биткоином в блоках хранятся записи о транзакциях, отслеживающие движение биткоинов. На рис. 13-6 показан блокчейн биткоина.

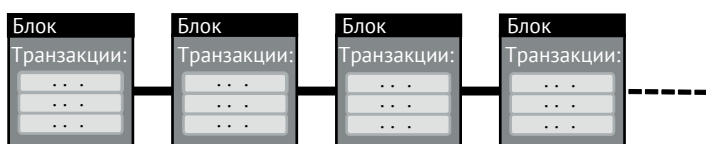


Рис. 13-6. Блокчейн биткоина связывает хронологические блоки записей о транзакциях

Блокчейн работает через сеть, такую как интернет, с несколькими компьютерами, работающими вместе для обработки транзакций и обновления блокчейна. Компьютеры, которые работают вместе для обработки транзакций биткоина, называются *биткоин-сетью*. Компьютер, подключенный к биткоин-сети, называется *узлом*, и определенные узлы хранят копию блокчейна. Единой главной копии не существует. Шифрование и расшифровка используются для обеспечения целостности транзакций и предотвращения фальсификации данных в блокчейне. Однажды записанные данные блокчейна незыблемы – их нельзя изменить. Блокчейн биткоина – это публичный, децентрализованный, неизменяемый журнал транзакций. Этот журнал используется для записи всех событий, происходящих в биткоин-сети, таких как перевод биткоинов.

## Биткоин-кошельки

Биткоины конечного пользователя хранятся в *биткоин-кошельке*. Хотя если быть точнее, то биткоин-кошелек хранит коллекцию пар криптографических ключей, как показано на рис. 13-7.

<sup>1</sup> *Coin* – монета, медяк. – Прим. ред.

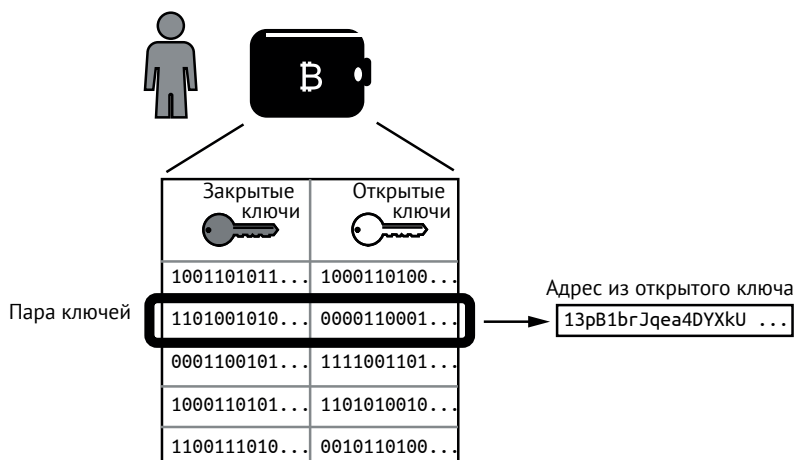


Рис. 13-7. Биткоин-кошелек содержит пары ключей. Биткоин-адрес формируется на основе открытого ключа

Как показано на рис. 13-7, каждая пара ключей в кошельке состоит из двух чисел – закрытого и открытого ключей. *Закрытый ключ* – это случайно сгенерированное 256-битное число. Это число должно храниться в секрете, любой, кто знает закрытый ключ, может потратить биткоины, связанные с этой парой ключей. *Открытый ключ*, который используется для получения биткоинов, является производным от закрытого ключа.

При получении биткоинов открытый ключ представляется в виде *биткоин-адреса* – текстовой строки, сгенерированной на основе открытого ключа. Вот пример биткоин-адреса: 13pB1brJqea4DYXkUKv5n-44HCgBkJNa2v1.

Допустим, у меня есть один биткоин, который я хочу отправить вам. Этот биткоин связан с адресом, который я контролирую. То есть у меня есть закрытый ключ для этого адреса. Если вы дадите мне текстовое строковое представление биткоин-адреса, который вы контролируете, я смогу отправить свой биткоин на ваш адрес. Вам не нужно (и не следует) отправлять мне свой закрытый ключ. Я могу отправить вам свой биткоин, потому что у меня есть закрытый ключ для моего адреса, который позволяет потратить мой биткоин. Но я не могу перевести ни одного биткоина с вашего адреса, потому что у меня нет вашего закрытого ключа.

## Биткоин-транзакции

Давайте рассмотрим подробнее, как это работает. Передача биткоинов называется *транзакцией*. Чтобы отправить биткоины, программное обеспечение кошелька создает транзакцию с указанием деталей перевода, делает цифровую подпись закрытым ключом и передает транзакцию в биткоин-сеть. Компьютеры в биткоин-сети проверяют транзакцию и добавляют ее в новый блок блокчейна. На рис. 13-8 показана биткоин-транзакция.

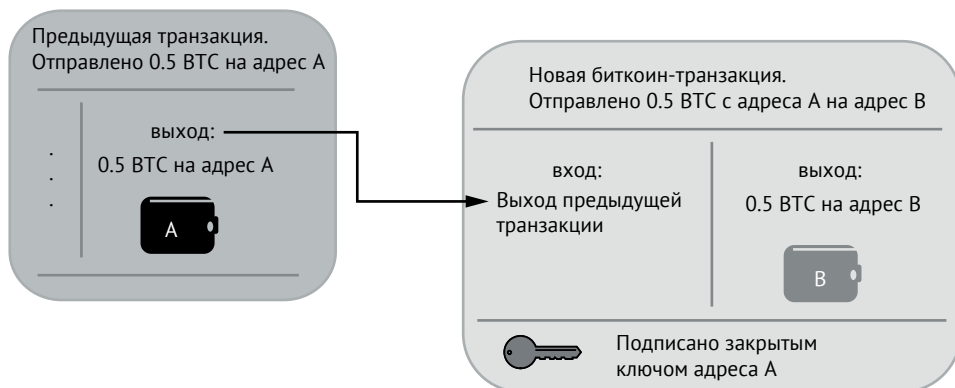


Рис. 13-8. Биткоин-транзакция переводит 0,5 BTC на адрес B (без учета комиссии за транзакцию)

Как показано на рис. 13-8, транзакция содержит входы и выходы, представляющие, откуда и куда поступает биткоин. Слева на этом рисунке показана предыдущая транзакция, где отображен только выход, поскольку вход предыдущей транзакции не имеет значения для нашего обсуждения. В предыдущей транзакции 0,5 BTC было отправлено на адрес A.

В правой части рис. 13-8 показана новая транзакция, которая перемещает 0,5 BTC с адреса A на адрес B. Для простоты эта транзакция имеет только один вход и один выход. Вход представляет собой источник переводимых биткоинов. Можно было бы ожидать, что это будет биткоин-адрес, но это не так. Вместо этого входом является выход предыдущей транзакции. Допустим, адрес A – это мой адрес, и я хочу отправить 0,5 BTC на ваш адрес, адрес B.

Теперь я знаю, что ранее на мой адрес было отправлено 0,5 BTC, поэтому я могу использовать выход предыдущей транзакции в качестве входа для новой транзакции, что позволит мне отправить вам эти 0,5 BTC. Выходная часть транзакции содержит адрес, на который отправляется биткоин.

Вы можете подумать, что на адресе имеется баланс биткоинов, однако количество биткоинов, связанных с адресом, не хранится в биткоин-кошельке, и баланс не хранится непосредственно в блокчейне. Вместо этого в блокчейне хранится история транзакций, связанных с этим адресом, и на основании этой истории можно рассчитать баланс для определенного адреса. Напомним, что в биткоин-кошельках просто хранятся ключи, позволяющие осуществлять транзакции.

## Майнинг биткоинов

Процесс поддержания блокчейна биткоина известен как *майнинг биткоина* (*mining* – добыча ископаемых). Компьютеры со всего мира добавляют блоки транзакций в блокчейн, эти компьютеры называются *майнерами*. На рис. 13-9 показан процесс майнинга биткоинов.

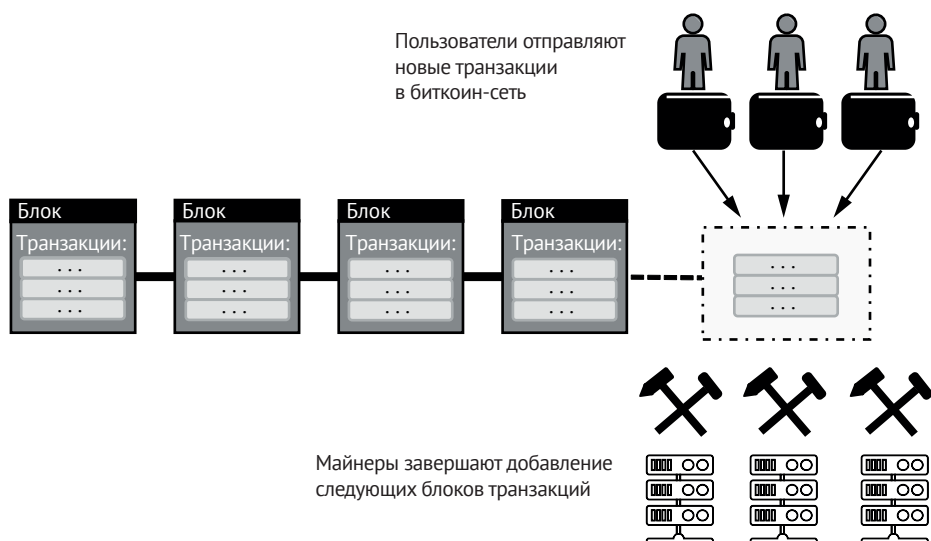


Рис. 13-9. Майнинг биткоина

Для того чтобы добавить блок транзакций в блокчейн, майнер должен проверить включенные в блок транзакции (убедиться в том, что каждая транзакция синтаксически правильна, что входные монеты не были еще потрачены и т. д.), а также решить сложную вычислительную задачу. Требование к майнерам решить такую задачу предотвращает фальсификации в блокчейн, поскольку изменение блока потребует решения задачи для измененного блока и для каждого блока, следующего за ним в блокчейне. Такая система решения сложной задачи в качестве средства предотвращения нежелательного поведения называется *доказательство выполнения работы* (*proof of work*).

Решение вычислительной задачи требует значительного количества вычислений методом проб и ошибок. Такое решение трудно получить, но легко проверить.

Первый майнер, выполнивший задачу, получает сумму биткоинов. Таким образом генерируются и вводятся в систему новые биткоины. Этим майнинг биткоинов похож на традиционный майнинг – майнеры выполняют работу и при удачном стечении обстоятельств могут «озолотиться». Помимо того, что майнер получает новые добытые биткоины, он также может требовать *плату* за каждую транзакцию, включенную в блок, которая вычитается из общей суммы биткоинов, отправленных в транзакции. Биткоин спроектирован таким образом, что в общей сложности может быть добыт только 21 млн монет. Когда это число будет достигнуто, майнеры биткоина перестанут получать биткоины, а для финансирования своей деятельности будут полагаться на комиссию за транзакции.

## НАЧАЛО БИТКОИНА

Блокчейн биткоина зародился в 2009 году, когда был добыт первый блок, известный как *genesis block*. Этот блок был добыт Сатоси Накамото, которому приписывают изобретение биткоина. Предполагается, что Сатоси Накамото – это псевдоним, на момент написания этой книги личность данного человека оспаривается.

Чтобы майнинг биткоинов был прибыльным, затраты на эксплуатацию оборудования для майнинга не должны превышать стоимость добытых биткоинов. Оборудование для майнинга биткоинов, как правило, потребляет много электроэнергии, поэтому счета за электричество для добывающих биткоины могут быть высокими. Изначально биткоин добывался на обычных компьютерах, но сегодня для максимально быстрой добычи используется специализированное дорогостоящее оборудование (помните, награда достается тому, кто первым решит задачу). Эти затраты, а также непостоянство цены биткоина означают, что майнинг биткоина не является гарантированным путем к прибыли!

Блокчейн биткоина является публичным, все транзакции могут быть просмотрены любым человеком. Однако блокчейн не содержит записей о личности людей, перечисляющих биткоины. Поэтому, хотя баланс адреса и история транзакций публичны, нет простого способа привязать этот адрес к человеку. По этой причине биткоин привлекателен для тех, кто хочет сохранить анонимность, например для тех, кто ведет коммерческие сайты в темной сети.

Технология блокчейн тесно связана с криптовалютами, где она используется в качестве бухгалтерской книги, но блокчейн можно использовать и для других целей. Любая система, которой необходима устойчивая к взлому история записей, может использовать блокчейн. Время покажет, будет ли биткоин или другие криптовалюты успешными в долгосрочной перспективе, но независимо от этого мы можем увидеть, как технология блокчейн используется другими новыми способами.

## Виртуальная и дополненная реальность

Две технологии, которые способны кардинально изменить наше взаимодействие с компьютером, – это виртуальная реальность (*virtual reality*, VR) и дополненная реальность (*augmented reality*, AR). *Виртуальная реальность* – это форма вычислительных технологий, которая погружает пользователя в трехмерное виртуальное пространство, обычно отображаемое с помощью шлема.

VR позволяет пользователю взаимодействовать с виртуальными объектами с помощью различных методов ввода, включая взгляд пользователя, голосовые команды и специализированные портативные контроллеры. В отличие от этого *дополненная реальность* накладывает

виртуальные элементы на реальный мир либо через шлем, либо при помощи портативного устройства, например смартфона или планшета. VR погружает пользователя в другой мир, AR изменяет реальный мир.

Хотя различные попытки создания VR предпринимались на протяжении нескольких десятилетий, только в 2010-х годах VR стал действительно популярным. Google помог популяризировать VR в 2014 году, выпустив Google Cardboard (*cardboard* – картон), названный так в честь идеи, что шлем VR можно создать из картона, линз и смартфона. Специально разработанные приложения Cardboard представляют пользователю VR-контент, отображая контент для левого глаза на одной половине экрана смартфона и контент для правого глаза на другой половине экрана, как показано на рис. 13-10.

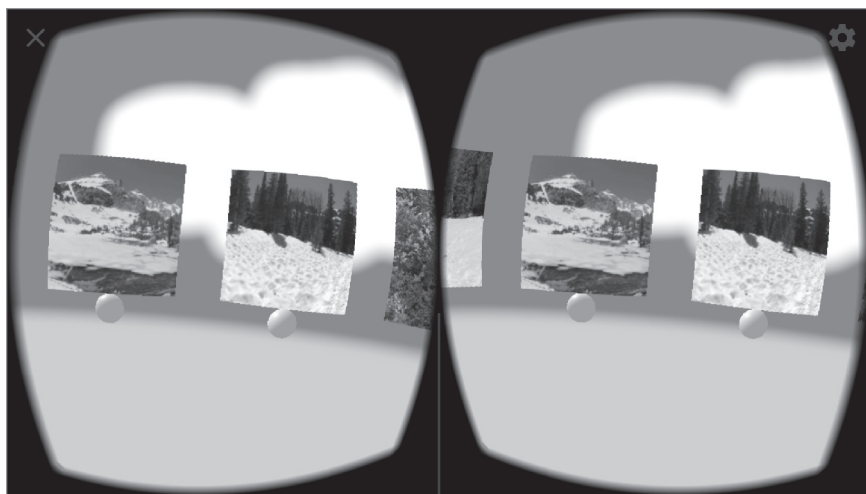


Рис. 13-10. Приложение, разработанное для Google Cardboard, представленное в режиме VR

Приложения, разработанные для Cardboard, полагаются на способность смартфона определять гироскопическое движение, позволяя дисплею обновляться, когда пользователь двигает головой. Такая система имеет 3 степени свободы (*3 degrees of freedom, 3DoF*) и может отслеживать ограниченное движение головы, но не движение в пространстве. Это позволяет пользователю *смотреть* по сторонам, но не перемещаться. Cardboard также поддерживает базовый однокнопочный ввод. Cardboard прост, но эффективен. Он познакомил с VR многих пользователей, которые, возможно, не попробовали бы его другим способом.

Для более захватывающего опыта требуется 6 степеней свободы (*6 degrees of freedom, 6DoF*), когда пользователь может передвигаться в VR, физически перемещая свое тело в реальном пространстве. Некоторые шлемы VR поддерживают 6DoF, а контроллеры VR могут иметь либо 3DoF, либо 6DoF. Контроллер 6DoF, удерживаемый в руке пользователя, может отслеживать свое положение в VR-пространстве, что позволяет более естественно взаимодействовать с VR-средой.

На потребительском рынке появилось множество VR-решений после Google Cardboard. Некоторые из них ориентированы на смартфоны (Samsung Gear VR, Google Daydream).

Другие используют персональный компьютер для обработки данных с подключенным шлемом VR и контроллерами (Oculus Rift, HTC Vive, Windows Mixed Reality). Еще другие представляют собой автономные устройства, не требующие смартфона или ПК (Oculus Go, Oculus Quest, Lenovo Mirage Solo). В целом наивысшую графическую точность обеспечивают решения, подключаемые к ПК, они же являются самыми дорогими, особенно если учесть стоимость требуемого компьютера.

Как упоминалось ранее, дополненная реальность, или AR, является похожей, но другой технологией. В то время как VR пытается полностью погрузить пользователя в виртуальный мир, AR накладывает виртуальные элементы на реальный мир. Это может быть реализовано с помощью мобильного устройства, где камера, расположенная на задней панели, используется для наблюдения за реальным миром, а смоделированные элементы накладываются на то, что видит камера. Передовые технологии AR позволяют программному обеспечению распознавать физические объекты в помещении, чтобы наложенные виртуальные элементы могли беспрепятственно взаимодействовать с окружающей средой. В базовой форме AR реализован в мобильных приложениях, но более полно он реализован в специальных устройствах, таких как Google Glass, очки Magic Leap и Microsoft HoloLens. Такие AR-устройства надеваются на голову и накладывают сгенерированную компьютером графику на поле зрения пользователя. Пользователи могут взаимодействовать с виртуальными элементами с помощью различных методов, например посредством голосовых команд или движения рук.

Различные технологии VR и AR (вместе называемые XR) представляют собой множество платформ для разработчиков программного обеспечения. Многие разработчики VR полагаются на существующие игровые движки, которые обычно используются для создания 3D-игр, такие как игровой движок Unity или игровой движок Unreal. Эти движки уже знакомы разработчикам игр, и они позволяют относительно легко создавать свое программное обеспечение для многих VR платформ. Веб-разработчики могут создавать контент VR и AR с помощью API JavaScript, известных как *WebVR* и *WebXR*. Из них WebVR появился первым и был ориентирован именно на VR. За ним последовал WebXR с поддержкой как AR, так и VR.

## Интернет вещей

Традиционно мы считаем, что серверы предоставляют услуги в интернете, а пользователи взаимодействуют с этими серверами через подключенные к интернету персональные компьютерные устройства, такие как ПК, ноутбуки и смартфоны. В последние годы мы наблюдаем рост числа новых типов устройств, подключаемых к интернету, – колонок, телевизоров, термостатов, дверных звонков, автомобилей, лампо-



чек и т. д.! Эта концепция подключения всех видов устройств к интернету называется *интернетом вещей* (*internet of things, IoT*).

Стоимость и физические размеры электронных компонентов снижаются, Wi-Fi и мобильный доступ в интернет широко распространены, и потребители ожидают, что их устройства будут становиться «умнее». Все это способствовало тенденции подключения всего к интернету. IoT-устройства, как правило, не работают без какой-либо веб-службы, поддерживающей их, поэтому развитие облачных вычислений также способствовало распространению интернета вещей. Для потребителей устройства IoT занимают видное место в «умном доме», где можно отслеживать и контролировать все типы домашних устройств. В бизнес-сфере устройства IoT можно найти в производстве, здравоохранении, транспорте и т. д.

Хотя эти типы подключенных устройств приносят очевидные преимущества, они также создают риски. Безопасность таких устройств вызывает особую озабоченность. Не все IoT-устройства хорошо защищены от атак злоумышленников. Даже если данные на самом устройстве не представляют интереса для злоумышленника, устройство может послужить плацдармом для преступника в ранее хорошо защищенной сети, или оно может использоваться в качестве отправной точки для удаленной атаки на другую цель. Для потребителей устройства IoT кажутся достаточно безобидными, и вопросы безопасности часто не стоят на первом месте при подключении такого устройства к домашней сети.

Конфиденциальность – еще один риск, связанный с устройствами IoT. Многие из этих устройств по своей природе собирают данные. Эти данные часто отправляются в облачную службу для обработки. Насколько конечные пользователи могут доверять организациям, которые работают с этими сервисами и имеют дело с личными данными пользователей? Даже самая благонамеренная организация может стать жертвой утечки данных, и пользовательские данные могут быть раскрыты неожиданным образом. Такие устройства, как «умные колонки», должны постоянно находиться в режиме прослушивания в ожидании устных команд. Это создает риск случайной записи частных разговоров. Современные читатели романа Джорджа Оруэлла «1984» могут найти определенную иронию в том, что сегодняшние потребители с готовностью обмениваются приватностью на удобство.

Еще один риск, связанный с IoT-устройствами, заключается в том, что их полная функциональность часто зависит от облачных сервисов. Если интернет-соединение устройства прервется, оно может временно утратить свою функциональность. Более серьезной проблемой является то, что производитель устройства может однажды навсегда отключить сервис, поддерживающий его работу. В этот момент «умное» устройство превратится в «глупое»!

#### ПРИМЕЧАНИЕ

*Ознакомьтесь с проектом № 41 на стр. 381, где вы сможете использовать полученные знания об аппаратном и программном обеспечении и об интернете для создания подключенного к сети IoT-устройства «торговый автомат»*



## Выводы

В этой главе мы рассмотрели множество тем, связанных с современными вычислительными технологиями. Вы узнали о мобильных приложениях, как о нативных, так и о веб-приложениях. Вы изучили, как виртуализация и эмуляция позволяют компьютерам запускать программное обеспечение на виртуализированном оборудовании. Вы увидели, как облачные вычисления предоставляют новые платформы для запуска программного обеспечения. Вы узнали, чем отличаются видимая, невидимая и темная сети, а также как криптовалюты, такие как биткоин, позволяют создавать децентрализованные платежные системы. Мы коснулись виртуальной и дополненной реальности и рассмотрели, как они позволяют создавать уникальные пользовательские интерфейсы. Вы узнали об IoT и получили возможность создать подключенный к интернету «торговый автомат».

Поскольку мы приблизились к концу этой книги, давайте рассмотрим некоторые основные вычислительные концепции и посмотрим, как они сочетаются друг с другом. Компьютеры – это двоичные цифровые устройства, где все представлено в виде нулей или единиц, включено (*on*) или выключено (*off*). Двоичная логика, также известная как булева логика, обеспечивает основу для вычислительных операций. Компьютеры создаются с помощью цифровых электрических схем, в которых уровни напряжения представляют двоичные состояния – низкое напряжение соответствует 0, а высокое напряжение – 1.

Цифровые логические вентили – это транзисторные схемы, позволяющие выполнять булевы операции, такие как И и ИЛИ. Логические вентили могут быть скомпонованы для создания более сложных схем, таких как счетчики, устройства памяти и схемы сложения. Эти типы схем обеспечивают концептуальную основу для аппаратного обеспечения компьютера: центральный процессор (CPU), выполняющий инструкции, память с произвольным доступом (RAM), хранящая инструкции и данные при подключенном питании, и устройства ввода/вывода (I/O), взаимодействующие с внешним миром.

Компьютеры являются программируемыми, они могут выполнять новые задачи без изменения аппаратного обеспечения. Инструкции, которые указывают компьютеру, что делать, называются программным обеспечением или кодом. Процессоры выполняют машинный код, в то время как разработчики программного обеспечения обычно пишут исходный код на языке программирования более высокого уровня. Компьютерные программы обычно выполняются под управлением операционной системы – программного обеспечения, которое взаимодействует с компьютерным оборудованием и обеспечивает среду для выполнения программ. Компьютеры обмениваются данными с помощью интернета – глобально связанного комплекса компьютерных сетей, использующих общий набор протоколов TCP/IP. Интернет используется для создания Всемирной паутины, комплекса распределенных, адресуемых, связанных ресурсов, передаваемых по протоколу HTTP через ин-

тернет. Все эти технологии обеспечивают среду, в которой могут процветать современные компьютерные инновации.

Я надеюсь, что эта книга дала вам более полное представление о том, как работают компьютеры. Мы охватили большой объем информации, но при этом лишь прошлись по верхам большинства тем. Если какая-то область привлекла ваше внимание, я бы посоветовал вам продолжить изучение этой темы – читайте о ней в интернете, посещайте уроки, смотрите видео или купите другую книгу! Вам предстоит открыть для себя еще множество знаний о вычислительной технике.

## ПРОЕКТ № 41: Использование PYTHON для управления схемой торгового автомата

Предварительные условия: проекты № 7 (на стр. 139) и № 8 (на стр. 140), в которых вы собрали схему торгового автомата. Raspberry Pi под Raspberry Pi OS. Я рекомендую вам перейти к приложению B и прочитать весь раздел «Raspberry Pi» на стр. 416, если вы этого еще не сделали.

В этом проекте вы будете использовать полученные знания об аппаратном и программном обеспечении и интернет для создания подключенного к сети IoT-устройства «торговый автомат». Еще в главе 6 вы собрали схему торгового автомата, используя кнопки, светодиод и цифровые логические вентили. В этом проекте вы усовершенствуете данное устройство. Вы сохраните кнопки и светодиод, но замените логические вентили на код Python, работающий на Raspberry Pi. Это позволит вам легко добавлять разные функции в программное обеспечение, например вы сможете подключиться к устройству по сети.

Для этого проекта вам понадобятся следующие компоненты:

- макетная плата;
- светодиод;
- токоограничивающий резистор для использования со светодиодом, приблизительно 220 Ом;
- два переключателя или кнопки, которые подходят для макетной платы;
- провода-перемычки, в том числе 4 провода «папа–мама»;
- Raspberry Pi.

### **GPIO**

Помимо крошечного размера и низкой стоимости, у Raspberry Pi есть еще одна особенность, которая отличает его от обычного компьютера, – выводы его GPIO-портов. Каждый контакт ввода/вывода общего назначения (*general purpose input/output*, *GPIO*) может использоваться как электрический вход или выход. Когда вывод работает как вход, код, запущенный на Raspberry Pi, может считать уровень высоким при напряжении 3,3 В или низким при напряжении 0 В. Raspberry Pi даже имеет подтягивающий и за-

земляющий внутренние резисторы, включающиеся программным путем, поэтому вам больше не придется добавлять такие резисторы к кнопкам. Когда вывод работает как выход, он может быть установлен на высокий (3,3 В) или низкий (0 В) уровень напряжения, и все это контролируется программой. Некоторые выводы всегда установлены на землю, 5 или 3,3 В. В программном обеспечении порты GPIO обозначаются номерами. На рис. 13-11 показаны обозначения выводов GPIO для платы Raspberry Pi.

Как видно на рис. 13-11, номера выводов GPIO не соответствуют номерам контактов разъема. Контакты, показанные на сером поле, просто пронумерованы от 1 до 40, начиная с верхнего левого и заканчивая нижним правым. Например, второй сверху контакт в левой колонке – это GPIO 2, а его номер равен 3. Когда вы ссылаетесь на эти выводы GPIO в коде, вам нужно использовать номер GPIO.

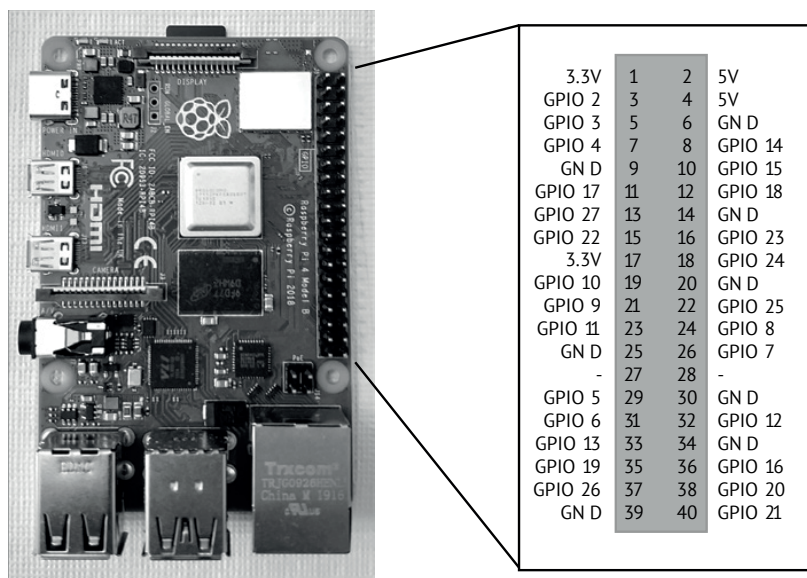


Рис. 13-11. Контакты GPIO Raspberry Pi

## ПОСТРОЕНИЕ СХЕМЫ

Прежде чем писать код, подключите компоненты схемы к макетной плате и к выводам GPIO Raspberry Pi, как показано на рис. 13-12.

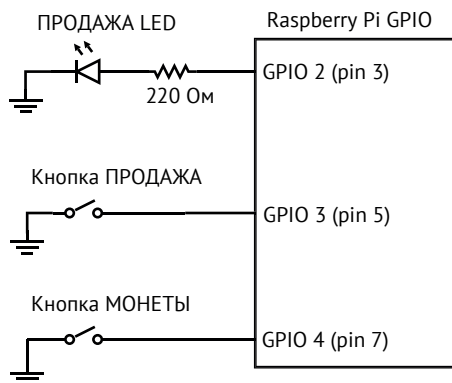


Рис. 13-12. Схема торгового автомата на Raspberry Pi

Я бы рекомендовал выключить питание Raspberry Pi, прежде чем подключать что-либо к выводам GPIO. Для соединения контактов GPIO с макетной платой используйте провод-перемычку «папа–мама». Вы можете подключить конец провода «мама» к контакту GPIO, а конец провода «папа» – к макетной плате.

Если использовать номера контактов, показанные на рис. 13-12, то ПРОДАЖА LED, кнопка ПРОДАЖА и кнопка МОНЕТЫ будут подключены к трем последовательным портам GPIO на Raspberry Pi. Вам также необходимо подключить любой из контактов GND (я бы рекомендовал номер 9) к минусовой шине макетной платы, чтобы вы могли легко подключить кнопки и светодиод к земле.

Вы могли заметить, что входные кнопки этой схемы подключены иначе, чем кнопки, которые вы использовали в проектах № 7 (на стр. 139) и № 8 (на стр. 140). В тех проектах вы подключали кнопку к 5 В с одной стороны и к заземляющему резистору / входному контакту с другой стороны. Схема была спроектирована таким образом, что разомкнутые контакты кнопки должны подавать низкое напряжение, а замкнутые – высокое. Здесь все наоборот – замкнутые контакты подают низкий уровень напряжения, а разомкнутые – высокий<sup>1</sup>. Внутри Raspberry Pi подтягивает вывод GPIO к высокому уровню, когда контакты кнопки разомкнуты (или ничего не подключено).

На рис.13-13 показана эта схема, собранная на макетной плате.

<sup>1</sup> Следует отметить, что такое подключение более соответствует обычной схемотехнической практике: замыкать кнопку полагается на землю, в отпущенном состоянии на ней высокий уровень за счет внешнего или внутреннего подтягивающего резистора. Это обусловлено тем обстоятельством, что во многие традиционные контроллеры (например, в ATmega, лежащих в основе Arduino) встроены только подтягивающие резисторы, а заземляющие отсутствуют. В Raspberry Pi на каждом GPIO-порту имеются оба типа встроенных резисторов, но лучше поступать всегда одинаково – так меньше вероятность наклепить ошибок в программе. – *Прим. ред.*

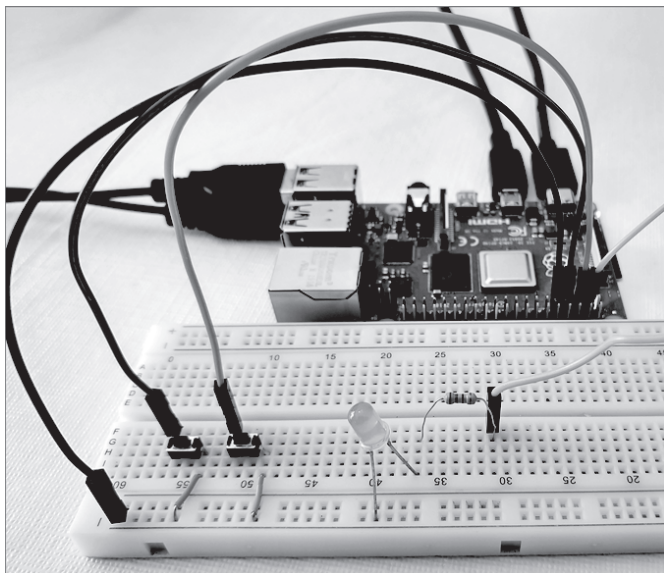


Рис. 13-13. Схема на макетной плате торгового автомата на Raspberry Pi

После того как схема подключена и вы проверили соединения, включите питание Raspberry Pi. Вы можете увидеть включение светодиода, и это нормально, поскольку вы еще не выполнили никакого кода, чтобы установить светодиод в определенное состояние.

### ПРОВЕРЬТЕ ВАШУ СХЕМУ

Перед тем как перейти к коду торгового автомата, давайте напишем простую программу, чтобы проверить правильность подключения схемы и дать вам представление о работе с GPIO в Python. Для взаимодействия с выводами GPIO мы будем использовать *GPIO Zero*, библиотеку Python, которая упрощает работу с физическими устройствами, такими как кнопки и светодиоды. С помощью любого текстового редактора создайте новый файл с именем *gpiotest.py* в корне домашней папки. Введите в текстовый редактор следующий код на языке Python. Отступы имеют значение в Python, поэтому убедитесь, что вы делаете их должным образом.

---

```
from time import sleep❶
from gpiozero import LED, Button❷

button = Button(3)❸
led = LED(2)❹

while True:❺
    led.off()
    button.wait_for_press()
    led.on()
    sleep(1)
```

---

Эта простая программа импортирует функцию `sleep` ❶ и классы `LED` и `Button` из библиотеки `GPIO Zero` ❷. Затем она создает переменную с именем `button`, которая представляет физическую кнопку на `GPIO 3` ❸. Аналогично создается переменная `led` для представления светодиода, подключенного к `GPIO 2` ❹. Затем программа входит в бесконечный цикл ❺, который выключает светодиод, ждет нажатия кнопки, а затем включает светодиод на одну секунду, после чего снова проходит цикл.

После сохранения файла его можно запустить с помощью интерпретатора Python следующим образом:

---

```
$ python3 gpiotest.py
```

---

При запуске программы сначала ничего не должно произойти, за исключением того, что светодиод может погаснуть, если до этого он был включен. Если нажать на кнопку, подключенную к `GPIO 3`, светодиод должен включиться на одну секунду, а затем выключиться. Вы можете повторять это до тех пор, пока работает программа.

В нашей простой программе не было предусмотрено никакого изящного способа выхода, поэтому для завершения программы нажмите **CTRL-C** на клавиатуре. Когда вы выходите из программы таким образом, интерпретатор Python покажет вам «трассировку» (traceback) последних вызовов функций – это нормально.

Если программа работает не так, как ожидалось, перепроверьте введенный код и просмотрите раздел «Поиск и устранение неисправностей в схемах» на стр. 414.

### **ПРОГРАММА ДЛЯ ТОРГОВОГО АВТОМАТА**

В проектах № 7 (на стр. 139) и № 8 (на стр. 140) логика торгового автомата управлялась SR-защелкой, вентилем И (AND) и конденсатором. Теперь все это можно заменить программой на Raspberry Pi. Эта новая конструкция также позволяет избавиться от светодиода `МОНЕТЫ LED`. Раньше светодиод `МОНЕТЫ LED` загорался, если была опущена монета. В новой конструкции программа вместо этого печатает количество монет. Каждый раз, когда опускается монета, счетчик монет должен увеличиваться на единицу, а каждый раз, когда происходит операция продажи, счетчик должен уменьшаться на единицу.

Требования к этому устройству будут следующими:

- нажатие кнопки `МОНЕТЫ` увеличивает счетчик монет на один;
- нажатие кнопки `ПРОДАЖА` имитирует продажу товара. Если счетчик монет больше 0, светодиод `ПРОДАЖА LED` кратковременно загорается, и счетчик уменьшается на единицу. Если счетчик монет равен 0, при нажатии кнопки `ПРОДАЖА` ничего не происходит;
- каждое нажатие кнопки, будь то `МОНЕТЫ` или `ПРОДАЖА`, заставляет программу печатать текущее количество монет.

С помощью текстового редактора по вашему выбору создайте новый файл с именем *vending.py* в корне домашней папки. Введите в текстовый редактор следующий код на языке Python:

```
from time import sleep❶
from gpiozero import LED, Button

vend_led = LED(2)❷
vend_button = Button(3)
coin_button = Button(4)
coin_count = 0❸
vend_count = 0

def print_credits():❹
    print('Credits: {0}'.format(coin_count - vend_count))

def coin_button_pressed():❺
    global coin_count
    coin_count += 1
    print_credits()

def vend_button_pressed():❻
    global vend_count
    if coin_count > vend_count:
        vend_count += 1
        print_credits()
        vend_led.on()
        sleep(0.3)
        vend_led.off()

coin_button.when_pressed = coin_button_pressed❼
vend_button.when_pressed = vend_button_pressed

input('Press Enter to exit the program.\n')❽
```

Во-первых, код импортирует функцию `sleep`, класс `LED` и класс `Button` ❶, все это будет использовано далее в программе. Далее объявляются три переменные, которые представляют физические компоненты, подключенные к выводам GPIO – `vend_led` на GPIO 2, `vend_button` на GPIO 3 и `coin_button` на GPIO 4 ❷. Переменная `coin_count` объявлена для отслеживания количества нажатий на кнопку `МОНЕТЫ`, а переменная `vend_count` – для отслеживания количества раз, когда происходила операция продажи ❸. Эти две переменные используются для подсчета количества монет.

Функция `print_credits` ❹ печатает количество доступных монет, которое является просто разницей между `coin_count` и `vend_count`.

Функция `coin_button_pressed` ❺ – код, который запускается при нажатии кнопки `МОНЕТЫ`. Она увеличивает на один `coin_count` и печатает количество монет. Выражение `global coin_count` позволяет изменять глобальную переменную `coin_count` в функции `coin_button_pressed`.

Функция `vend_button_pressed` ⑥ – это код, который запускается при нажатии кнопки ПРОДАЖА. Если еще остались монеты (`coin_count > vend_count`), то функция увеличивает на один `vend_count`, печатает количество монет и включает светодиод на 0,3 с.

Установка `coin_button.when_pressed = coin_button_pressed` ⑦ связывает функцию `coin_button_pressed` с `coin_button` на GPIO 4 таким образом, что функция запускается при нажатии кнопки. Аналогично `vend_button_pressed` связывается с `vend_button`.

Наконец, мы вызываем функцию `input` ⑧. Эта функция печатает сообщение на экране и ждет, пока пользователь нажмет клавишу **ENTER**. Это простой способ поддержания работы программы. Без этой строки кода программа дошла бы до конца и прекратила работу до того, как пользователь получил бы возможность взаимодействовать с кнопками.

После сохранения файла вы можете запустить его с помощью интерпретатора Python следующим образом:

---

```
$ python3 vending.py
```

---

Когда вы запустите программу, в окне терминала сразу же должно появиться сообщение *Press Enter to exit the program*. В этот момент попробуйте нажать кнопку МОНЕТЫ, которая подключена к GPIO 4. Вы должны увидеть, как программа выводит *Credits: 1*. Затем попробуйте нажать кнопку ПРОДАЖА. Светодиод должен кратковременно загореться, и программа должна напечатать *Credits: 0*. Попробуйте нажать кнопку ПРОДАЖА еще раз, ничего не должно произойти. Попробуйте нажимать кнопки МОНЕТЫ и ПРОДАЖА несколько раз и убедитесь, что программа работает так, как ожидалось. Когда вы закончите тестирование, нажмите **ENTER** для завершения программы.

Как видите, Raspberry Pi или аналогичное устройство может воспроизвести в программном виде ту же логику, которую мы ранее реализовали в аппаратном виде. Однако программное решение гораздо легче модифицировать. Новые функции могут быть добавлены путем изменения нескольких строк кода, а не путем добавления новых микросхем и проводов. Raspberry Pi на самом деле является излишне мощным для того, что мы хотели здесь сделать. То же самое можно было бы реализовать с помощью менее мощного вычислительного устройства при еще меньших затратах, но принцип тот же.

## ИЮТ-ТОРГОВЫЙ АВТОМАТ

Допустим, оператор торгового автомата хочет иметь возможность проверять состояние автомата удаленно через интернет. Поскольку вы используете Raspberry Pi для логики торгового автомата, можете сделать еще один шаг вперед и превратить его в торговый автомат IoT! Вы можете добавить в программу простой веб-сервер, который позволит кому-либо подключиться к IP-адресу устройства через веб-браузер и посмотреть, сколько раз была опущена монета и сколько раз происходила торговая операция.



В Python это сделать относительно просто, поскольку в нем есть простая библиотека веб-сервера *http.server*. Вам просто нужно создать HTML, содержащий данные, которые вы хотите отправить, и написать обработчик входящих GET-запросов. Также необходимо запустить веб-сервер при запуске программы.

Используйте текстовый редактор по вашему выбору для редактирования существующего файла *vending.py* в корне вашей домашней папки. Начните со вставки следующего оператора `import` в качестве первой строки файла (оставив весь существующий код нетронутым, просто сместив его на строку вниз):

---

```
from http.server import BaseHTTPRequestHandler, HTTPServer
```

---

Затем удалите всю строку `input` в нижней части файла и добавьте этот код в конец файла:

```
HTML_CONTENT = """
<!DOCTYPE html>
<html>
  <head><title>Vending Info</title></head>
  <body>
    <h1>Vending Info</h1>
    <p>Total Coins Inserted: {0}</p>
    <p>Total Vending Operations: {1}</p>
  </body>
</html>
"""

class WebHandler(BaseHTTPRequestHandler):❶
    def do_GET(self):❷
        self.send_response(200)❸
        self.send_header('Content-type', 'text/html')❹
        self.end_headers()
        response_body = HTML_CONTENT.format(coin_count, vend_count).encode()❺
        self.wfile.write(response_body)

print('Press CTRL-C to exit program.')
server = HTTPServer(('', 8080), WebHandler)❻
try:❼
    server.serve_forever()❸
except KeyboardInterrupt:
    pass
finally:
    server.server_close()❾
```

HTML\_CONTENT – это многолинейная строка, определяющая HTML-код, который программа отправляет по сети. Этот блок HTML-кода представляет собой простую веб-страницу с заголовком `<title>`, заголовком 1-го уровня `<h1>` и двумя абзацами `<p>`, которые описывают состояние торгового автомата. Конкретные значения в этих абзацах представлены в виде

шаблонов {0} и {1}. Эти значения заполняются программой во время ее работы. Поскольку это HTML, интервалы и переносы строк здесь не имеют значения.

Класс `WebHandler` ❶ описывает, как веб-сервер обрабатывает входящие HTTP-запросы. Он наследуется от класса `BaseHTTPRequestHandler`, что означает, что он имеет те же методы и поля, что и `BaseHTTPRequestHandler`. Однако это всего лишь дает общий обработчик HTTP-запросов, вам все равно нужно указать, как ваша программа будет отвечать на конкретные HTTP-запросы.

В данном случае программе нужно отвечать только на HTTP GET-запросы, поэтому в коде определен метод `do_GET` ❷. Этот метод вызывается, когда на сервер приходит GET-запрос, и он отвечает следующим образом:

- код состояния 200, означающий успех ❸;
- заголовок `Content-type: text/html`, который указывает браузеру, что в качестве ответа ожидается HTML ❹;
- HTML-строка, которая была определена ранее, но в которой значения в шаблонах заменены значениями `coin_count` и `vend_count` ❺.

Экземпляр веб-сервера создается с помощью класса `HTTPServer` ❻. Здесь вы указываете, что имя сервера может быть любым и что HTTP-сервер прослушивает порт 8080 (' ', 8080). Здесь также указывается, что для входящих HTTP-запросов нужно использовать класс `WebHandler`.

Веб-сервер запускается с помощью `server.serve_forever()` ❼. Он помещается в блок `try/except/finally` ❼, так что сервер продолжает работать до тех пор, пока не произойдет исключение `KeyboardInterrupt` (осуществляемое посредством **CTRL-C**). Когда это происходит, вызывается `server.server_close()` для очистки, и программа завершается ❽.

После сохранения файла вы можете запустить его с помощью интерпретатора Python следующим образом:

---

```
$ python3 vending.py
```

---

Программа должна вести себя так же, как и раньше, когда вы нажимаете кнопки **МОНЕТЫ** или **ПРОДАЖА**. Однако теперь вы также можете подключиться к устройству через веб-браузер и просмотреть данные о торговом автомате. Для этого вам понадобится устройство в той же локальной сети, что и Raspberry Pi, если только ваш Pi не подключен напрямую к интернету с публичным IP-адресом, в этом случае любое устройство в интернете сможет подключиться к нему. Если у вас нет другого устройства, вы можете запустить веб-браузер на самом Raspberry Pi и позволить Raspberry Pi действовать и как клиент, и как сервер.

Вам нужно найти IP-адрес вашего Raspberry Pi. Мы делали это в проекте № 30 на стр. 315, если вы хотите вспомнить детали, но вот команда, которую вам нужно использовать:

---

```
$ ifconfig
```

---

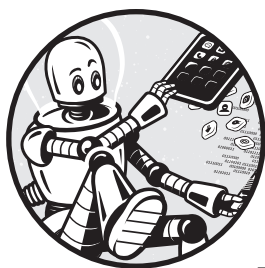
Получив IP-адрес Raspberry Pi, откройте веб-браузер на устройстве, которое вы хотите использовать в качестве клиента. В адресной строке введите `http://IP:8080`, заменив IP на IP-адрес вашей Raspberry Pi. Конечный результат должен выглядеть примерно так: `http://192.168.1.41:8080`. После того как вы введете это в адресную строку браузера, вы увидите, как загрузится веб-страница с подсчетом опущенных монет и торговых операций. Каждый раз, когда вы запрашиваете эту страницу, то должны видеть, как программа Python выводит информацию о запросе на терминал. После загрузки веб-страницы она не будет автоматически перезагружаться, поэтому, если нажмете кнопки **МОНЕТЫ** или **ПРОДАЖА** еще несколько раз и захотите увидеть последние значения, вам нужно будет обновить страницу в браузере. Чтобы остановить программу, используйте **CTRL-C** на клавиатуре.

Вспомните из главы 12, что веб-сайты бывают статическими или динамическими. Сайт, который вы запускали в проектах главы 12, был статическим, так как он выдавал контент, созданный заранее. В отличие от него сайт торгового автомата в этой главе является динамическим. Он генерирует HTML-ответ при поступлении запроса. В частности, перед ответом он обновляет значения количества опущенных монет и количества операций продажи в контенте HTML.

В качестве бонусного задания попробуйте модифицировать программу, чтобы она также отображала значение «количества доступных монет» на веб-странице. Это значение должно совпадать с последним значением количества доступных монет, которое было выведено на терминал.

# Приложение А

## ОТВЕТЫ НА УПРАЖНЕНИЯ



Здесь вы найдете вопросы и ответы к упражнениям, включенным в эту книгу. Некоторые вопросы не имеют единственного правильного ответа, в таких случаях я включил пример ответа. Вы получите максимальную пользу от этих упражнений, если сами придумаете ответ до того, как прочитаете решение здесь!

### **1-2: Двоичное в десятичное**

**Упражнение:** Переведите представленные в двоичной системе числа в их десятичные эквиваленты.

**Решение:**

10 (двоичное) = 2 (десятичное);

111 (двоичное) = 7 (десятичное);

1010 (двоичное) = 10 (десятичное).

### **1-3: Десятичное в двоичное**

**Упражнение:** Переведите представленные в десятичной системе числа в их двоичные эквиваленты.

**Решение:**

3 (десятичное) = 11 (двоичное);

8 (десятичное) = 1000 (двоичное);

14 (десятичное) = 1110 (двоичное).

## 1-4: Из двоичной системы в шестнадцатеричную

**Упражнение:** Переведите эти числа, представленные в двоичном формате, в их шестнадцатеричные эквиваленты. Не переводите в десятичную систему, если это возможно! Вам может помочь табл. 1-5. Цель состоит в том, чтобы перейти непосредственно от двоичной системы к шестнадцатеричной.

**Решение:**

10 (двоичное) = 2 (шестнадцатеричное);

11110000 (двоичное) = F0 (шестнадцатеричное).

## 1-5: Из шестнадцатеричной в двоичную

**Упражнение:** Преобразуйте эти числа, представленные в шестнадцатеричной системе счисления, в их двоичные эквиваленты. Не переводите в десятичную систему счисления, если это возможно! Вам может помочь табл. 1-5. Цель состоит в том, чтобы перейти непосредственно от шестнадцатеричной системы к двоичной.

**Решение:**

1A (шестнадцатеричное) = 0001 1010 (двоичное);

C3A0 (шестнадцатеричное) = 1100 0011 1010 0000 (двоичное).

## 2-1: Создайте собственную систему представления текста

**Упражнение:** Определите способ представления заглавных букв от A до D в виде 8-битных чисел, а затем закодируйте слово *DAD* в 24 бита с помощью вашей системы.

**Бонус:** Представьте ваше закодированное 24-битное число в шестнадцатеричном виде тоже.

В табл. A-1 приведен пример ответа, единственного правильного ответа тут нет.

**Решение:**

**Таблица A-1.** Пользовательская система для представления A–D с помощью байтов

Буква	Двоичное число
A	00000001
B	00000010
C	00000011
D	00000100

DAD по этой системе будет 00000100 00000001 00000100 (пробелы добавлены для наглядности). Запись в шестнадцатеричном виде: 0x040104.

## 2-2: Кодировка и декодировка ASCII

**Упражнение:** Используя табл. 2-1, закодируйте следующие слова в двоичном и шестнадцатеричном коде ASCII, используя по одному байту для каждого символа. Помните, что для заглавных и строчных букв существуют разные значения.

**Решение:**

**Текст:** Hello

**Двоичное:** 01001000 01100101 01101100 01101100 01101111

**Шестнадцатеричное:** 0x48656C6C6F

**Текст:** 5 cats

**Двоичное:** 00110101 00100000 01100011 01100001 01110100 01110011

**Шестнадцатеричное:** 0x352063617473

Обратите внимание, как кодирование «5 cats» дало нам 0b00110101 в качестве двоичного представления знака 5. Это отличается от числа 5, которое равно 0b101. Знак представляет собой символ (5), который мы используем для обозначения числа пять, в то время как число представляет собой количество. Из той же кодировки «5 cats» обратите внимание, что даже символ пробела, который вы, возможно, считали пустым, все равно требует байта для представления.

**Упражнение:** Используя табл. 2-1, декодируйте следующие слова. Каждый символ представлен в виде 8-битного значения ASCII с пробелами для наглядности.

**Решение:**

**Двоичное:** 01000011 01101111 01100110 01100110 01100101 01100101

**Текст:** Coffee

**Двоичное:** 01010011 01101000 01101111 01110000

**Текст:** Shop

**Упражнение:** Используя табл. 2-1, расшифруйте следующее слово. Каждый символ представлен в виде 8-битного шестнадцатеричного значения с пробелами для наглядности.

**Решение:**

**Шестнадцатеричное:** 43 6C 61 72 69 6E 65 74

**Текст:** Clarinet

## 2-3: Создание собственной системы представления градации серого

**Упражнение:** Определите способ представления черного, белого, темно-серого и светло-серого цветов.

**Решение:** Если мы используем двухбитную систему, то четыре уникальных значения для двухбитного числа – это 00, 01, 10 и 11. Каждое из

этих четырех двоичных чисел может быть сопоставлено с цветом: черным, белым, темно-серым и светло-серым – конкретное сопоставление зависит от вас, поскольку вы разрабатываете свою собственную систему. В табл. А-2 приведен пример ответа, так как здесь нет единственно правильного варианта.

**Таблица А-2.** Пользовательская система для представления градации серого цвета

Цвет	Двоичное число
черный	00
темно-серый	01
светло-серый	10
белый	11

## 2-4: Создание собственного подхода к представлению простых изображений

### Упражнение:

**Часть 1.** На основе вашей предыдущей системы представления цветов градации серого разработайте подход к представлению изображения, состоящего из этих цветов. Если вы хотите упростить задачу, можете предположить, что изображение всегда будет иметь размер 4 пикселя на 4 пикселя, как на рис. 2-1.

**Решение:** Предположим, что изображение всегда будет 4 пикселя на 4 пикселя, и поэтому нам нужно представить 16 пикселей, по одному цвету на пиксель. Используя ранее определенную систему представления градации серого цвета из табл. А-2, нам потребуется 2 бита для представления цвета каждого пикселя. Таким образом, для представления полного изображения из 16 пикселей с 2 битами на пиксель нам потребуется  $16 \times 2 = 32$  бита всего.

Когда мы кодируем данные в двоичном формате, в каком порядке мы должны представлять 16 пикселей? Это решение в некоторой степени произвольное, но для данного примера давайте расположим данные слева направо, сверху вниз, как показано на рис. А-1.

При использовании подхода, показанного на рис. А-1, когда мы кодируем наши данные в двоичном формате, первые 2 бита будут представлять цвет пикселя 1, следующие 2 бита – цвет пикселя 2 и т. д. Затем мы используем цветовые коды, установленные в предыдущем упражнении, чтобы определить цвет каждого пикселя. Например, если пиксель 1 – белый, пиксель 2 – черный, а пиксель 3 – темно-серый, то первые 6 бит в данных нашего изображения равны 110001.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Рис. А-1. Порядок пикселей на примере формата изображения

**Часть 2.** Используя свой подход из части 1, запишите двоичное представление изображения цветка из рис. 2-1 в главе 2.

**Решение:** Здесь я привожу пример того, как можно решить эту задачу, применяя подход, взятый из части 1 этого упражнения. Для наглядности на рис. А-2 на изображение цветка наложена пронумерованная сетка изображения.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Рис. А-2. Сетка изображения 4×4, наложенная на изображение цветка

Теперь, когда мы присвоили каждому пикселю в сетке номер, можем обратиться к табл. А-2, чтобы назначить двухбитное значение к каждому квадрату, начиная с квадрата 1 и заканчивая квадратом 16. В итоге мы получим следующую двоичную последовательность, которая представляет изображение цветка в градациях серого:

1111110111101101111110101100101



**ПРИМЕЧАНИЕ**

Я создал простую веб-страницу, которая моделирует эту конкретную систему из 16 пикселей и двухбитной градации серого. Протестируйте ее здесь:

<https://www.howcomputersreallywork.com/grayscale/>.

## 2-5: Составление таблицы истинности для логической функции

**Упражнение:** В табл. 2-7 показаны состояния трех входов для логического выражения. Заполните таблицу истинности для функции (А ИЛИ В) И С.

**Решение:**

**Таблица А-3.** (А ИЛИ В) И С

Таблица истинности

A	B	C	Выход (Output)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

## 3-1: Применение закона Ома

**Упражнение:** Посмотрите на электрическую цепь на рис. А-3. Каково здесь значение тока  $I$ ?

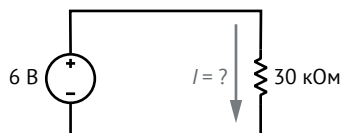


Рис. А-3. Найдите силу тока, используя закон Ома

**Решение:** Закон Ома говорит нам, что ток – это напряжение, деленное на сопротивление. Поэтому  $I$  равен 0,2 мА, как показано здесь:

$$I = \frac{6 \text{ В}}{30\,000 \text{ Ом}} = 0,002 \text{ А} = 0,2 \text{ мА}.$$

### 3-2: Определите падение напряжения

**Упражнение:** Дана электрическая цепь, как на рис. 3-11. Каково будет значение тока  $I$ ? Каково будет падение напряжения на каждом резисторе? Найдите обозначенные напряжения:  $V_A$ ,  $V_B$ ,  $V_C$  и  $V_D$ , каждое из которых измеряется относительно отрицательной клеммы источника питания.

**Решение:** Общее сопротивление составляет  $24\text{ кОм} + 6\text{ кОм} + 10\text{ кОм} = 40\text{ кОм}$ . Это определяет ток в цепи, который мы можем рассчитать по закону Ома:  $10\text{ В} / 40\text{ кОм} = 0,25\text{ мА}$ , как показано на рис. А-4.

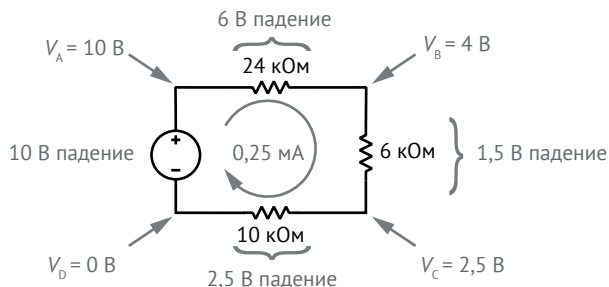


Рис. А-4. Падение напряжения в цепи

Теперь рассчитаем падение напряжения на резисторе  $24\text{ кОм}$ , используя закон Ома:  $V = 0,25\text{ мА} \times 24\text{ кОм} = 6\text{ В}$ . Это означает, что  $V_B$  будет на  $6\text{ В}$  меньше, чем  $V_A$ . То есть  $V_B = 10\text{ В} - 6\text{ В} = 4\text{ В}$ . На резисторе  $6\text{ кОм}$  падает  $0,25\text{ мА} \times 6\text{ кОм} = 1,5\text{ В}$ . Следовательно,  $V_C = V_B - 1,5\text{ В} = 2,5\text{ В}$ . Это оставляет  $2,5\text{ В}$  для падения на резисторе  $10\text{ кОм}$ , которое мы можем вывести из закона напряжения Кирхгофа или рассчитать по закону Ома.

### 4-1: Реализация логического ИЛИ (OR) с транзисторами

**Упражнение:** Нарисуйте схему логической функции ИЛИ (OR), в которой для входов А и В используются транзисторы. Адаптируйте схему на рис. 4-4, в которой используются механические выключатели, но вместо них используйте NPN-транзисторы.

**Решение:** На рис. А-5 показано решение для реализации логического ИЛИ с использованием NPN-транзисторов.

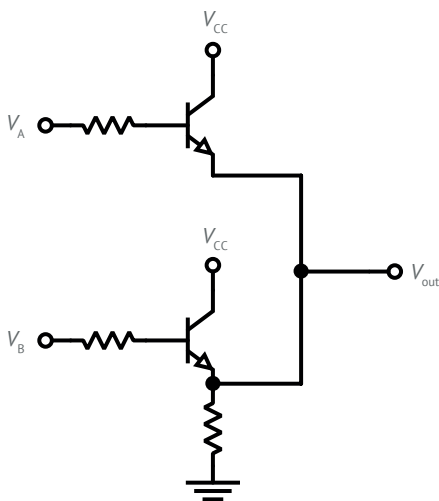


Рис. А-5. Логическое ИЛИ, реализованное с помощью транзисторов NPN

## 4-2: Проектирование схемы с логическими вентилями

**Упражнение:** В упражнении 2-5 главы 2 вы создали таблицу истинности для функции (А ИЛИ В) И С. Теперь, основываясь на этой работе, переведите таблицу истинности и логическое выражение в цифровую схему. Нарисуйте схему логических вентилях (подобную той, что показана на рис. 4-11).

**Решение:** На рис. А-6 показано решение для реализации (А ИЛИ В) И С.

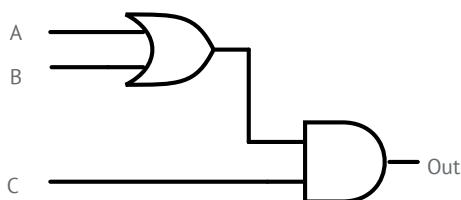


Рис. А-6. Схема логических вентилях для (А ИЛИ В) И С

## 5-1: Практика двоичного сложения

**Упражнение:** Попробуйте решить следующие задачи на сложение.

**Решение:** Ведущие 0 в ответах необязательны.

$$0001 + 0010 = 0011$$

$$0011 + 0001 = 0100$$

$$0101 + 0011 = 1000$$

$$0111 + 0011 = 1010$$

## 5-2: Найдите дополнительный код

**Упражнение:** Найдите 4-битный дополнительный код для 6.

**Решение:** Смотрите рис. А-7.

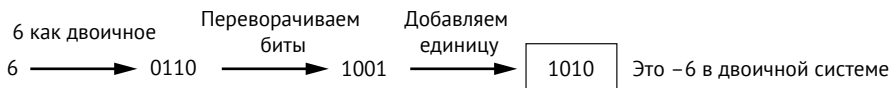


Рис. А-7. Нахождение дополнительного кода для 6

## 5-3: Сложите два двоичных числа и их интерпретируйте их как знаковые и беззнаковые

**Упражнение:** Сложите 1000 и 0110. Интерпретируйте вашу работу через знаковые числа. Затем интерпретируйте их как беззнаковые. Имеют ли результаты смысл?

**Решение:** Смотрите рис. А-8.

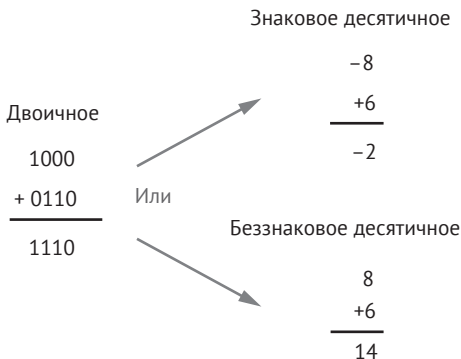


Рис. А-8. Сложение 1000 и 0110

## 7-1: Вычислите необходимое количество битов

**Упражнение:** Используя методы, описанные в главе 7, определите количество битов, необходимое для адресации 4 ГБ памяти. Помните, что каждому байту присваивается уникальный адрес, который является просто числом.

**Решение:** Вспомните, что, согласно главе 1, где представлены префиксы СИ, 1 ГБ памяти – это  $2^{30}$  или 1 073 741 824 байта. Таким образом, 4 ГБ – это число, большее данного в 4 раза, т. е. равное 4 294 967 296 байт. Если взять  $\log_2(4\,294\,967\,296)$ , то получится 32. Таким образом, с 32 битами мы можем представить уникальный адрес для каждого байта в 4 ГБ памяти.

Если ваш калькулятор или приложение не вычисляет функцию двоичного логарифма  $\log_2$ , то можно воспользоваться десятичным  $\lg$ :

$$\log_2(n) = \frac{\log(n)}{\log(2)}.$$

С помощью этой информации вы можете найти  $\log_2(4\,294\,967\,296)$ , взяв  $\lg(4\,294\,967\,296)$  и разделив его на  $\lg(2)$ . Это должно дать результат 32.

Мы также можем прийти к этому решению, используя другой подход. Поскольку адреса памяти присваиваются, начиная с 0, а не с 1, диапазон адресов памяти для 4 ГБ памяти составляет от 0 до 4 294 967 295 (на 1 меньше, чем количество байт). В шестнадцатеричном формате 4 294 967 295 – это 0xFFFFFFFF. Это 8 шестнадцатеричных цифр, а поскольку каждый шестнадцатеричный символ представляет собой 4 бита, мы легко увидим, что требуется  $4 \times 8 = 32$  бита.

## 8-1: Используйте свой мозг в качестве процессора

**Упражнение:** Попробуйте выполнить следующую программу на языке ассемблера ARM в уме или воспользуйтесь карандашом и бумагой:

---

Адрес	На языке ассемблера
0001007c	subs r3, r0, #1
00010080	ble 0x10090
00010084	mul r0, r3, r0
00010088	subs r3, r3, #1
0001008c	bne 0x10084
00010090	---

---

Предположим, что входное значение  $n = 4$  первоначально хранится в  $r0$ . Когда программа дойдет до инструкции по адресу 00010090, вы достигнете конца кода, и в  $r0$  должно быть ожидаемое выходное значение 24. Я рекомендую для каждой инструкции отслеживать значения  $r0$  и  $r3$  до и после ее выполнения. Проработайте все инструкции, пока не дойдете до инструкции по адресу 00010090, и посмотрите, получили ли вы ожидаемый результат. Если все работало правильно, вы должны были пройти через одни и те же инструкции несколько раз, и это не случайно.

**Решение:** После выполнения этого упражнения посмотрите на табл. А-4, чтобы увидеть каждый шаг выполнения ассемблерного кода. Каждая строка в таблице представляет собой выполнение одной инструкции. Для каждой инструкции мы отслеживаем значения  $r0$  и  $r3$ . Стрелка ( $\rightarrow$ ) означает, что значение изменилось с указанного слева на указанное справа. В колонке «Примечания» я использую знак равенства для обозначения «установлено», а не для математической проверки равенства. Например,  $r0 = r3 \times r0$  означает « $r0$  устанавливается равным произведению  $r3$  и  $r0$ ».

**Таблица А-4.** Ассемблерный код для вычисления факториала, шаг за шагом

Адрес	Инструкция	г0	г3	Примечание
		4	?	Вы хотите посчитать факториал 4, поэтому установите г0 = 4 до запуска кода. Значение г3 вначале неизвестно
0001007c	subs г3, г0, #1	4	? → 3	г3 = г0 - 1 = 4 - 1 = 3
00010080	ble 0x10090	4	3	г3 > 0, поэтому перехода нет, вместо этого следуйте к 10084
00010084	mul г0, г3, г0	4 → 12	3	г0 = г3 × г0 = 3 × 4 = 12
00010088	subs г3, г3, #1	12	3 → 2	Уменьшите значение г3 на 1
0001008c	bne 0x10084	12	2	г3 не равно 0, поэтому переходите к 10084
00010084	mul г0, г3, г0	12 → 24	2	г0 = г3 × г0 = 2 × 12 = 24
00010088	subs г3, г3, #1	24	2 → 1	Уменьшите значение г3 на 1
0001008c	bne 0x10084	24	1	г3 не равно 0, поэтому переходите к 10084
00010084	mul г0, г3, г0	24 → 24	1	г0 = г3 × г0 = 1 × 24 = 24
00010088	subs г3, г3, #1	24	1 → 0	Уменьшите значение г3 на 1
0001008c	bne 0x10084	24	0	г3 равно 0, поэтому перехода нет, вместо этого следуйте к 10090
00010090		24	0	Алгоритм завершился, и результат может быть найден в г0, которое сейчас равно 24, как и ожидалось

Надеюсь, эта таблица соответствует результатам, которые вы видели, когда пробовали сделать это самостоятельно. Теперь, когда мы рассмотрели код для  $n = 4$ , подумайте над следующими вопросами.

1. Если мы вычисляем факториал 1, изначально установив  $г0 = 1$ , что произойдет?
2. Математическое определение факториала гласит, что факториал 0 равен 1. Работает ли наш алгоритм с этим сценарием? Какой конкретный результат мы получим, если изначально зададим  $г0 = 0$ ?
3. Возможно, вы заметили, что ожидаемый результат 24 был сохранен в г0 на предпоследней итерации кода. То есть программа проходит через цикл еще раз, но это никак не влияет на значение г0. Как вы думаете, почему код был написан именно так?
4. Учитывая, что мы используем 32-битные регистры, существует ли практический верхний предел для  $n$ ? То есть можно ли задать значение  $n$ , при котором результат будет слишком большим, чтобы поместиться в 32-битный регистр?

Вот ответы на эти вопросы.

1. Первая инструкция subs устанавливает  $г3 = 0$ , и следующая инструкция ble переходит на адрес 0x10090, так как г3 равен 0. В этот момент наш результат в г0 все еще равен 1, что является ожидаемым результатом.

2. Нет, наш алгоритм не сработает. Первая инструкция `subs` устанавливает `г3` в `-1`, а следующая инструкция `ble` переходит на адрес `0x10090`, так как `г3` отрицательно. В этот момент наш результат в `г0` по-прежнему равен `0`, что не является ожидаемым результатом.
3. Факториал числа  $n$  – это произведение целых положительных чисел, меньших или равных  $n$ . Если придерживаться этого определения, то следует умножить `г0` на `1`, даже если это не изменит конечный результат. Это означает один дополнительный цикл в коде, пока `г3` равно `1`. Мы могли бы повысить эффективность кода, пропустив это умножение на `1`, но я оставил его, чтобы следовать математическому определению факториала.
4. Максимальное значение, которое может представлять 32-битное целое число, равно  $2^{32} - 1 = 4\,294\,967\,295$ . Или, если нам нужно представлять также и отрицательные числа, наибольшее значение `2\,147\,483\,647`. Поэтому, если мы попытаемся вычислить факториал, где результат больше, чем примерно `4` млрд (или `2` млрд), мы получим неудовлетворительный результат. Оказывается, что  $n = 12$  – это самое большое значение  $n$ , которое мы можем использовать. Факториал `13` составляет более `6` млрд, что слишком много, чтобы поместиться в 32-битное целое число.

## 9-1: Побитовые операторы

**Упражнение:** Рассмотрите следующие выражения на языке Python. Какими будут значения  $a$ ,  $b$  и  $c$  после выполнения этого кода?

---

```
x = 11
y = 5
a = x & y
b = x | y
c = x ^ y
```

---

**Решение:** На рис. А-9 показано, как работают побитовые операции И (AND), ИЛИ (OR) и Исключающее ИЛИ (XOR) при применении к значениям `11` и `5`.

$x = 11 = 1011$	$x = 11 = 1011$	$x = 11 = 1011$	
$y = 5 = 0101$	$y = 5 = 0101$	$y = 5 = 0101$	
<hr/>	<hr/>	<hr/>	
0001	1111	1110	Исключающее ИЛИ (XOR)
И (AND)	ИЛИ (OR)		

Рис. А-9. Побитовые операции над двумя значениями

Таким образом, значение  $a$  равно `1`. Значение  $b$  равно `15` (`1111` в двоичном виде). Значение  $c$  равно `14` (`1110` в двоичном виде).

# 9-2: Выполните программу на С в уме

**Упражнение:** Попробуйте выполнить предыдущую функцию факториала в уме или с помощью карандаша и бумаги. Предположим, что входное значение  $n = 4$ . Когда функция вернет результат, возвращаемое значение должно быть равно 24. Я рекомендую для каждой строки отслеживать значения  $n$  и  $result$  до и после завершения оператора. Проработайте код до конца цикла `while` и посмотрите, получите ли вы ожидаемый результат.

Обратите внимание, что условие цикла `while` (`--n > 0`) помещает оператор декремента (`--`) перед переменной  $n$ . Это означает, что  $n$  уменьшается на 1, прежде чем его значение сравнивается с 0. Это происходит каждый раз, когда оценивается условие цикла `while`.

```
// Calculate the factorial of n.
int factorial(int n)
{
    int result = n;

    while(--n > 0)
    {
        result = result * n;
    }

    return result;
}
```

**Решение:** Прежде чем читать дальше, я настоятельно рекомендую вам попытаться выполнить это упражнение! Вы узнаете больше, если выполните его самостоятельно. После выполнения этого упражнения посмотрите на табл. А-5, чтобы увидеть каждый шаг выполнения кода из нашего примера на языке С. Стрелка ( $\rightarrow$ ) означает, что значение переменной изменилось со значения слева на значение справа.

**Таблица А-5.** Код для вычисления факториала на С, шаг за шагом

Выражение	Результат	$n$	Примечание
<code>int factorial(int n)</code>	?	4	Мы хотим посчитать факториал 4, поэтому установите $n = 4$ как входное значение нашей функции
<code>int result = n;</code>	$? \rightarrow 4$	4	Сначала устанавливаем <code>result</code> равным $n$
<code>while(--n &gt; 0)</code>	4	$4 \rightarrow 3$	Уменьшаем значение $n$ на 1. $n > 0$ , поэтому переходим к телу цикла <code>while</code>
<code>result = result * n;</code>	$4 \rightarrow 12$	3	<code>result = 4 × 3</code>
<code>while(--n &gt; 0)</code>	12	$3 \rightarrow 2$	Уменьшаем значение $n$ на 1. $n > 0$ , поэтому снова переходим к телу цикла <code>while</code>
<code>result = result * n;</code>	$12 \rightarrow 24$	2	<code>result = 12 × 2</code>
<code>while(--n &gt; 0)</code>	24	$2 \rightarrow 1$	Уменьшаем значение $n$ на 1. $n > 0$ , поэтому снова переходим к телу цикла <code>while</code>



Выражение	Результат	n	Примечание
result = result * n;	24 → 24	1	result = 24 × 1
while(--n > 0)	24	1 → 0	Уменьшаем значение n на 1. n = 0, поэтому выходим из цикла while
return result;	24	0	Мы закончили работу с функцией, и вычисленное значение может быть найдено в переменной result, которая сейчас равна 24, как и ожидалось

Надеюсь, эта таблица соответствует результатам, которые вы получили, когда пробовали сделать это самостоятельно.

## 11-1: Какие IP находятся в одной подсети?

**Упражнение:** Находится ли IP-адрес 192.168.0.200 в той же подсети, что и ваш компьютер? Предположим, что ваш компьютер имеет IP-адрес 192.168.0.133 и маску подсети 255.255.255.224.

**Решение:** Как мы выяснили в главе 11, сетевой идентификатор подсети вашего компьютера – 192.168.0.128. Если два устройства находятся в одной подсети, они будут иметь общую маску подсети и идентификатор сети. Логическое И IP-адреса другого компьютера и маски подсети дает нам сетевой идентификатор.

---

```

IP = 192.168.0.200    = 11000000.10101000.00000000.11001000
MASK = 255.255.255.224 = 11111111.11111111.11111111.11100000
AND  = 192.168.0.192  = 11000000.10101000.00000000.11000000 = The network id

```

---

Идентификатор сети другого компьютера (192.168.0.192) не совпадает с идентификатором сети вашего компьютера (192.168.0.128), поэтому они находятся в разных подсетях. Это означает, что связь между этими хостами должна проходить через маршрутизатор.

## 11-2: Исследование распространенных портов

**Упражнение:** Найдите номера портов для распространенных протоколов прикладного уровня. Каковы номера портов для системы доменных имен (Domain Name System, DNS), для безопасной оболочки (Secure Shell, SSH) и простого протокола передачи почты (Simple Mail Transfer Protocol, SMTP)? Эту информацию можно найти в интернете с помощью поиска или посмотреть в реестре IANA здесь: <http://www.iana.org/assignments/port-numbers>. В списках IANA иногда используются неожиданные термины для названия служб. Например, DNS указывается просто как «домен» («domain»).

Решение:

- DNS: 53
- SSH: 22
- SMTP: 25

## 12-1: Определение частей URL-адреса

**Упражнение:** Для следующих URL-адресов определите схему (протокол), имя пользователя, хост, порт, путь и запрос. Не все URL-адреса включают все эти части.

**Решение:**

*https://example.com/photos?subject=cat&color=black*

**scheme** *https*

**host** *example.com*

**path** *photos*

**query** *subject=cat&color=black*

*http://192.168.1.20:8080/docs/set5/two-trees.pdf*

**scheme** *http*

**host** *192.168.1.20*

**port** *8080*

**path** *docs/set5/two-trees.pdf*

*mailto:someone@example.com*

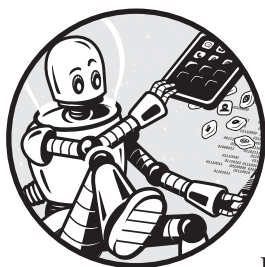
**scheme** *mailto*

**username** *someone*

**host** *example.com*

## Приложение В

# ТЕХНИЧЕСКИЕ СРЕДСТВА



В этом приложении содержится информация, которая поможет вам начать работу с проектами. Мы рассмотрим, как найти электронные компоненты, как обеспечить питание цифровых схем и как настроить Raspberry Pi.

## Покупка электронных компонентов для проектов

Практическая работа с электроникой и программированием поможет вам усвоить концепции, изложенные в этой книге, но необходимость приобрести различные компоненты может показаться пугающей. Этот раздел поможет вам найти электронные компоненты, которые понадобятся для проектов.

Вот полный список всех компонентов, используемых в проектах, если вы захотите заказать все сразу (см. также перечень в описании проекта № 1 и сноску 1 на стр. 93):

- макетные платы (как минимум одна 830-точечная макетная плата. Вы можете обойтись только одной макетной платой, если планируете разбирать каждую схему между упражнениями. Если же вы хотите сохранить собранные схемы, вам понадобится несколько макетных плат);
- резисторы (набор резисторов). Вот конкретные используемые значения: 47 кОм, 10 кОм, 4,7 кОм, 1 кОм, 470 Ом, 330 Ом, 220 Ом;
- цифровой мультиметр;
- 9-вольтовая батарея (типа «Крона»);

- разъем для 9-вольтовой батареи;
- упаковка круглых красных светодиодов 5 или 3 мм;
- два NPN биполярных транзистора, модель 2N2222 в корпусе TO-92 (также известны как PN2222<sup>1</sup>);
- провода-перемычки, предназначенные для использования в макетной плате (как типа «папа–папа», так и типа «папа–мама»);
- кнопки или ползунковые переключатели, которые подходят для макетной платы;
- интегральная схема 7402;
- интегральная схема 7408;
- интегральная схема 7432;
- две интегральные схемы 7473;
- интегральная схема 7486;
- электролитический конденсатор 220 мкФ;
- электролитический конденсатор 10 мкФ;
- 5-вольтовый источник питания (подробнее см. в разделе «Питание цифровых схем» на стр. 410);
- Raspberry Pi и сопутствующие элементы (подробнее см. в разделе «Raspberry Pi» на стр. 416);
- рекомендуется: зажимы типа «крокодил» (с их помощью легче подключить батарею к макетной плате или мультиметр к схеме);
- необязательно: инструмент для зачистки проводов (он может понадобиться, чтобы снять пластик с концов проводов и обнажить медь).

Хотя в этом списке указаны конкретные количества для определенных компонентов, некоторые из них, вероятно, стоит купить в несколько большем количестве на случай повреждения или для экспериментов. Я рекомендую иметь несколько запасных транзисторов и по одной запасной интегральной схеме каждого типа.

## Названия микросхем серии 7400

Поиск подходящей интегральной схемы серии 7400 может оказаться непростой задачей, поскольку эти микросхемы идентифицируются функциональными наименованиями, которые содержат больше информации, чем просто номер 74xx. Серия 7400 состоит из нескольких подсемейств, каждое из которых имеет свою собственную систему наименования компонентов.

Кроме того, производители добавляют к названиям компонентов свои собственные префиксы и суффиксы. Поначалу это может сбить

<sup>1</sup> А также как KSP2222, MPS2222 и пр. Можно заменить на многие другие NPN-типы в таком же корпусе TO-92, например BC337, BC547, BC557, 2N4403 и др. – *Прим. ред.*

с толку, поэтому давайте рассмотрим пример. Недавно я хотел приобрести микросхему 7408, но наименование компонента, который я заказал, было SN74LS08N. Давайте разберем этот номер по составляющим (см. рис. В-1).

SN74LS08N				
Первые 2 цифры обозначают производителя.  SN = Texas Instruments	74 = 7400 серии логических вентилей для коммерческого использования.  54 = Военный класс	Логическое подсемейство.  LS = Lowpower Schottky (маломощный Шоттки)	Функциональный номер компонента.  08 = 4 логических И-вентилей с 2 входами. Произведен от 7408	Суффикс, специфичен для каждого производителя.  N = тип корпуса DIP с монтажом в отверстия

Рис. В-1. Разборка наименования микросхемы серии 7400

Итак, SN74LS08N – это набор четырех И-вентилей 7408 производства Texas Instruments, относящийся к подсемейству маломощных Шоттки, в корпусе с монтажом в отверстия на плате. Нет необходимости вникать в детали термина «маломощный Шоттки», кроме как знать, что это распространенное подсемейство микросхем, которое подходит для наших целей<sup>1</sup>.

Для описанных в этой книге проектов нам надо убедиться, что используются детали, которые совместимы друг с другом. Проекты предполагают, что используются микросхемы, совместимые с исходными логическими уровнями 7400 (5 В). Учитывая те компоненты, что легко доступны, я бы рекомендовал микросхемы из серий LS или HCT. Так что, если вам нужен 7408, можете купить SN74**LS**08N или SN74**HCT**08N. Вообще говоря, в одной схеме можно смешивать детали LS и HCT. Буквы префикса, SN в данном примере, не имеют значения, когда речь идет о совместимости, вам не обязательно покупать все детали от конкретного производителя.

<sup>1</sup> Скорее, тут уместна характеристика «было распространенным». В действительности подсемейство 74LS – результат ранней (середина-конец 1960-х) попытки снизить потребление микросхем транзисторно-транзисторной логики (TTL). В оригинальной TTL-серии 74xx оно было чересчур высоким, что порождало массу неприятных последствий для конструкторов аппаратуры. Снизить потребление удалось, но одновременно заметно упало и быстродействие. Так продолжалось до появления скоростных серий 74НС и 74АС, которые уже не принадлежат к семейству TTL, а основаны на КМОП (CMOS) – комплементарных парах полевых транзисторов. Быстродействующие КМОП-вентили из указанных серий соединяют в себе супернизкое потребление КМОП с быстродействием на уровне 74LS (74НС) и оригинальной 74xx (74АС). Упоминаемая далее автором серия 74HCT является разновидностью 74НС, полностью и без оговорок совместимой с привычной автору TTL, потому и рекомендуется им как возможный вариант. Следует также отметить, что для всех разновидностей 74-й серии имеются отечественные аналоги с метрическим (а не дюймовым) шагом между выводами корпуса (см. ассортимент интернет-магазинов в табл. В-1 далее).

А вот суффикс (N) имеет значение, поскольку он указывает на тип корпуса. Обязательно приобретайте компоненты, которые подходят для макетной платы, микросхемы с суффиксом N отлично подходят.

## Покупка

Если у вас поблизости есть местный магазин электроники, который может предоставить эти детали, то я бы посоветовал вам посетить его. Сотрудники магазина могут помочь вам убедиться, что вы получите то, что вам нужно. Однако я обнаружил, что такие магазины становятся все более редкими, и, возможно, вы не сможете купить все необходимое в своем районе<sup>1</sup>.

Ваш следующий вариант – сделать покупки в интернете. Чтобы облегчить задачу, я создал веб-страницу со ссылками на необходимые комплектующие в различных магазинах: <https://www.howcomputersreallywork.com/parts/>. Или, если вы хотите совершать покупки самостоятельно, в табл. В-1 приведен список некоторых популярных магазинов, где вы можете приобрести комплектующие. Я включил в таблицу примечания по каждому из магазинов. Я не настаиваю конкретно на этих магазинах, вы можете найти запчасти в любом другом месте. И конечно, статус этих интернет-магазинов может измениться после выхода этой книги.

**Таблица В-1.** Покупка электронных комплектующих онлайн<sup>2</sup>

Интернет-магазин	Комментарий
«Чип и дип» (интернет-магазин) <a href="https://www.chipdip.ru">https://www.chipdip.ru</a>	Дорого, но «все есть», плюс масса удобных вариантов оплаты/вывоза/доставки. Образцовый каталог с обильно представленной документацией, может служить справочником по компонентам. Имеется информация о наличии, в том числе и в офлайн-магазинах, возможность заказа практически любых позиций, отсутствующих на складе в данный момент

<sup>1</sup> В России самые популярные из таких магазинов – торговые сети «Чип и дип» и «Вольтмастер», присутствующие во многих городах страны. У них имеются также интернет-магазины (см. табл. В-1). – *Прим. ред.*

<sup>2</sup> В таблице указан перечень некоторых популярных российских интернет-магазинов электронных компонентов. При отборе из многочисленных вариантов обращалось внимание на доступность и ориентированность на непрофессионалов.

Стоит заметить, что последние годы в Рунете активно пропагандируется приобретение электронных компонентов на АлиЭкспресс, где действительно есть все (без кавычек) и притом максимально дешево, но качество и соответствие заказанному никто не гарантирует. Вы быстро обнаружите, что половина приобретенного в итоге отправляется в мусорное ведро, а действительно хорошие вещи стоят неприципиально дешевле, чем в отечественных интернет-магазинах (зато доставляются намного медленнее). Нарботав некоторый опыт, вы, возможно, научитесь приобретать электронные изделия на АлиЭкспресс так, чтобы было и дешевле и не хуже, но первоначально будьте готовы к потерям времени и денег. – *Прим. ред.*

Интернет-магазин	Комментарий
iArduino <a href="https://iarduino.ru">https://iarduino.ru</a>	Масса всего для любительской электроники и недорого, но нет микросхем серий 74 и скудно представлен Raspberry Pi. Информативные описания компонентов с примерами применения и исчерпывающей документацией. Быстрая и отлаженная доставка, но нередко вас ждет безапелляционное «нет на складе»
ДКО «Электронщик» <a href="https://www.electronshik.ru">https://www.electronshik.ru</a>	Разнообразный каталог (1,7 млн наименований!), но неудобные и малочисленные варианты доставки
Вольтмастер <a href="http://www.voltmaster.ru">http://www.voltmaster.ru</a>	Сеть магазинов, подобная «Чипу-дипу» (и с аналогичными ценами), обширный ассортимент. Меньше удобных вариантов доставки
Тиксер <a href="https://tixer.ru">https://tixer.ru</a>	Довольно представительный ассортимент, включает экзотические позиции, которые трудно найти в других местах. Цены ниже «чип-диповских». Удобные варианты доставки, но формирование заказа может затягиваться по срокам

Ряд сайтов посвящен помощи в поиске электронных деталей. Эти сайты предоставляют удобный интерфейс, в котором легко ориентироваться, и позволяют сравнивать цены у нескольких продавцов. Два из них, которые я нашел полезными, – Octopart (<https://octopart.com>) и Findchips (<https://www.findchips.com>)<sup>1</sup>.

## Питание цифровых схем

Логическим микросхемам серии 7400 требуется напряжение 5 В, поэтому использование 9-вольтовой батареи не подходит для питания этих интегральных схем. Давайте рассмотрим несколько вариантов питания микросхем серии 7400.

### Зарядное устройство USB

Многие зарядные устройства для смартфонов, выпущенные после 2010 года, имеют разъем микро-USB. Примерно с 2016 года стали чаще использоваться разъемы USB Type-C (или просто USB-C). К счастью, USB любого из этих разъемов обеспечивает постоянное напряжение 5 В. Поэтому зарядное устройство USB, подобное тому, что показано на рис. В-2, отлично подходит для питания интегральных схем серии 7400. Если вы похожи на меня, возможно, у вас дома уже есть куча старых зарядных устройств для телефонов с микро-USB.

<sup>1</sup> В России из таких ресурсов стоит порекомендовать Яндекс.Маркет (<https://market.yandex.ru>), а также, например, Tiu.ru (<https://tiu.ru/>). – Прим. ред.



Рис. В-2. Зарядное устройство для телефона с разъемом микро-USB

Однако здесь есть проблема: разъем микро-USB не подключается к макетной плате, по крайней мере, без посторонней помощи! Хороший вариант использования зарядного устройства USB с макетной платой, – это приобрести *micro-USB breakout board* (*переходная плата micro-USB*), подобную той, что показана на рис. В-3. Это позволит вам подключить зарядное устройство USB к переходной плате, а переходную плату – к вашей макетной плате. Многие интернет-магазины предлагают такие платы. Может потребоваться некоторая пайка. Эти платы обычно имеют пять контактов, но для данной цели вам нужно обратить внимание только на контакт VCC (5 В) и GND (земля). При подключении к макетной плате не забудьте сориентировать контакты так, чтобы они не были соединены друг с другом, как показано на рис. В-3.

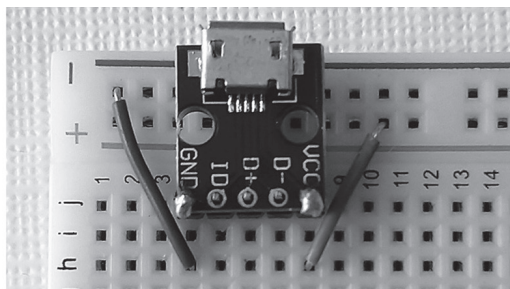
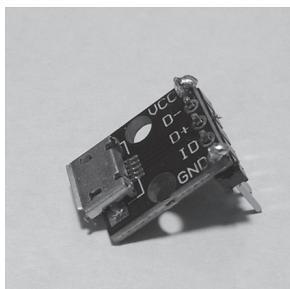


Рис. В-3. Переходная плата *micro-USB*, вставленная в макетную плату (справа)

## Питание для макетной платы

Другой вариант – приобрести блок питания для макетной платы, например DFRobot DFR0140 или YwRobot Power MB V2 545043. Эти удобные устройства подключаются к вашей макетной плате и питаются от встроенного в розетку источника постоянного тока с разъемом 5,5 мм. Источник постоянного тока должен обеспечивать напряжение от 6 до 12 В (обязательно проверьте, какое напряжение допустимо для конкретной платы, которую вы используете). Такие источники постоянного тока довольно часто используются для питания бытовой электроники.



троники, так что, возможно, у вас уже есть несколько таких источников. И этот тип платы просто позволяет легко преобразовать сетевое напряжение в 5 В и подключить к макетной плате. На рис. В-4 показан один из этих распространенных источников питания постоянного тока с разъемом 5,5 мм вместе с блоком питания для макетной платы<sup>1</sup>.

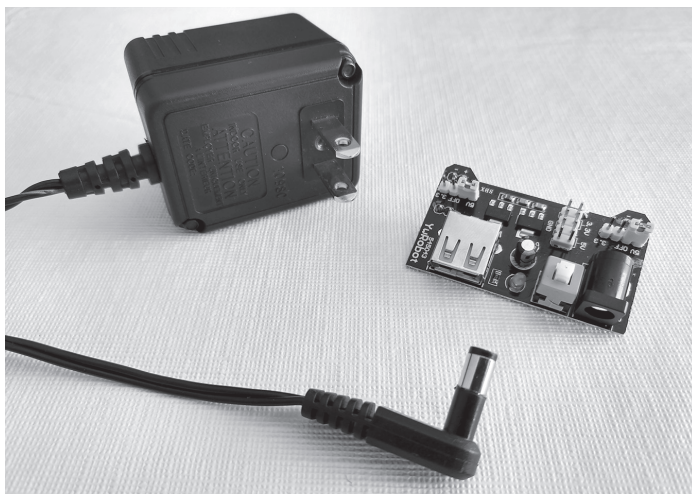


Рис. В-4. Блок питания с разъемом 5,5 мм и блок питания для макетной платы

Одно предостережение: я лично видел, как на этих платах выходил из строя регулятор напряжения, из-за чего плата выдавала напряжение выше 5 В. При подключении одного из этих источников питания не думайте, что выходное напряжение действительно равно 5 В. Проверьте выходное напряжение перед подключением схемы! Использование более низкого входного напряжения постоянного тока также поможет снизить этот риск, поэтому я рекомендую использовать 9-вольтовый или более низковольтный источник питания постоянного тока, учитывая допустимый диапазон от 6 до 12 В. Эти платы также имеют возможность выводить 3,3 В вместо 5 В, что контролируется с помощью переключки на плате, поэтому убедитесь, что переключки установлены в правильном месте.

## Питание от Raspberry Pi

Если вы собираетесь купить Raspberry Pi для проектов, о которых идет речь начиная с главы 8, то вам повезло, так как у него есть дополни-

<sup>1</sup> В продаже имеются и готовые адаптеры со встроенной вилкой, подобные зарядникам, сразу выдающие стабилизированное напряжение 5 В на круглый разъем 5,5 мм. Для них достаточно приобрести ответный разъем и присоединить к нему провода со штырьками, вставляющимися в макетную плату. Отдельные платы стабилизаторов при этом не требуются. Обсуждаемый далее Raspberry Pi питается именно от такого адаптера, только с разъемом микро-USB или USB-C вместо круглого 5,5 мм. – *Прим. ред.*

тельное преимущество. Raspberry Pi может использоваться как 5-вольтовый источник питания! Выводы GPIO на Pi имеют различные функции, но для текущей цели вам достаточно знать, что вывод 6 (а также 9, 25 и др.) – это земля, а на вывод 2 подается 5 В. Вы можете подключить эти выводы к макетной плате в качестве источника питания. Схему пинов GPIO см. на рис. 13-11 на стр. 382. Не нужно даже устанавливать никакое программное обеспечение для Raspberry Pi, потому что, как только Pi включится, 5-вольтовый пин будет включен. Просто подключите ваш Pi к питанию. В качестве бонуса выводы 1 и 17 выдают 3,3 В, если это необходимо. Для ясности: при этом вы не используете вычислительную мощность Pi, Raspberry Pi просто действует как источник питания на 5 В.

Существует ограничение на то, какой ток может подавать Raspberry Pi. Адаптер питания, который вы используете для Pi, будет иметь свой максимальный номинальный ток, и сам Pi будет потреблять некоторое количество тока, около 300 мА в спящем режиме. Возможно, это само собой разумеется, но, если вы выберете этот вариант, будьте внимательны, чтобы правильно подключить схему. Вы же не хотите случайно повредить ваш Raspberry Pi! На рис. В-5 показано использование Raspberry Pi в качестве источника питания.

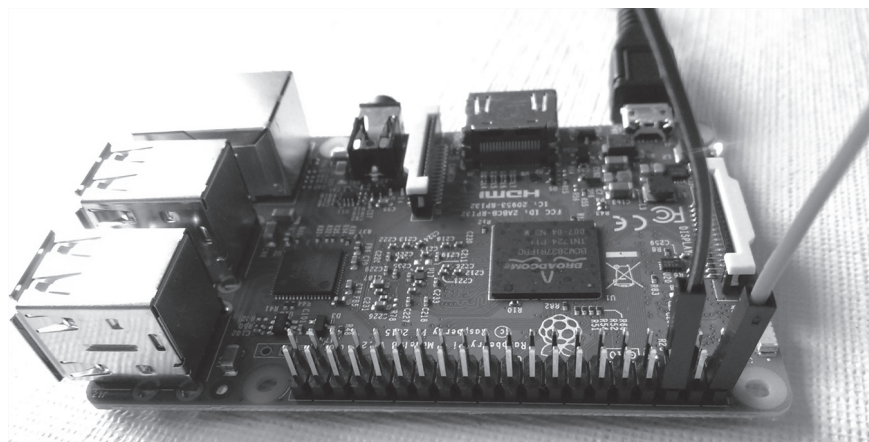


Рис. В-5. Использование Raspberry Pi в качестве источника питания

## Батарейки AA

Для питания цифровой схемы можно также использовать батарейки AA. Одна батарейка AA обеспечивает напряжение 1,5 В, поэтому три батарейки AA можно соединить последовательно, чтобы получить напряжение 4,5 В. Хотя это напряжение меньше рекомендованного для компонентов серии 7400, оно должно подойти для схем, описанных в этой книге. Вы можете купить батарейный отсек для трех батареек AA и подключить его выходные провода к макетной плате, как показано на рис. В-6.



Рис. В-6. Использование трех батареек АА для питания схемы на макетной плате

## Поиск и устранение неисправностей в электронных схемах

Иногда вы собираете схему, ожидая, что она будет работать определенным образом, но в результате получается нечто совершенно иное. Может оказаться, что схема ничего не делает или ведет себя не так, как вы предполагали. Не волнуйтесь, это случается со всеми, кто собирает схемы! Легко допустить ошибку в проводке. Или слабое соединение может вывести все из строя. Устранение неполадок и диагностика проблем в схеме – это ценный навык, который поможет вам расширить понимание того, как все работает. Здесь я расскажу о некоторых методах поиска и устранения неисправностей, которые применяю, когда мои схемы работают не так, как ожидалось.

Если какой-либо компонент в вашей схеме слишком горячий, чтобы к нему прикоснуться, немедленно отключите схему от источника питания. Ошибки подсоединения могут привести к перегреву компонента. Это часто приводит к повреждению компонента, если он остается подключенным более чем на несколько секунд.

Основным инструментом для поиска неисправностей в цепи является мультиметр. С помощью мультиметра можно легко проверить напряжение в различных точках цепи. Спросите себя: «Каким должно быть напряжение в этой или в той точке моей цепи?» Для 5-вольтовых цифровых схем ожидаемое напряжение обычно составляет при-

мерно 0 В или примерно 5 В. Если ваш измерительный прибор показывает неожиданное значение напряжения в любом месте схемы, спросите себя: «Что могло вызвать такое напряжение?» – и проверьте эти моменты.

Для цифровых схем я обычно использую подход «работы в обратном направлении», начиная с компонента, который ведет себя неправильно. Убедитесь, что его выходное напряжение неправильное, а затем проверьте его входы. Нет ли на одном из этих входов неожиданного уровня? Если да, обратитесь к компоненту, который подключен к этому входу, и проверьте его выход. Повторяйте, пока не найдете источник проблемы.

При проверке напряжений я обнаружил, что самый простой подход заключается в подключении черного/минусового/СОМ-провода к точке заземления в цепи и оставлении его там. Если нет очевидного места для подключения провода к земле, просто добавьте провод-перемычку к точке заземления на макетной плате, а затем с помощью зажима «крокодил» подключите этот провод-перемычку к щупу СОМ мультиметра. Закрепив щуп СОМ на земле, вы можете использовать плюсовой щуп, обычно окрашенный в красный цвет, чтобы потыкать в различные точки схемы и проверить напряжение относительно земли.

Еще одна вещь, которую я регулярно проверяю мультиметром во время поиска неисправностей, – сопротивление. Иногда я знаю ожидаемое сопротивление между двумя точками и хочу проверить это значение. Если точки, которые вы измеряете, соединяются более чем одним путем, убедитесь, что вы знаете ожидаемое сопротивление, чтобы правильно интерпретировать результаты измерения.

Обычно я проверяю значение сопротивления, чтобы просто убедиться, что две точки соединены, в этом случае сопротивление должно быть приблизительно 0 Ом. И наоборот, иногда я хочу убедиться, что две точки *не* соединены, и тогда я ожидаю очень высокое сопротивление и разомкнутую цепь. Некоторые измерительные приборы также имеют функцию *прозвонки*, при которой измерительный прибор издает звуковой сигнал, если две точки соединены. Если вы просто проверяете наличие соединения, иногда этот метод предпочтительнее, чем проверка сопротивления.

Вот некоторые специфические вещи, которые необходимо проверять при поиске неисправностей.

**Питание макетной платы.** Имеет ли ваша макетная плата соответствующее напряжение на шинах питания? Напряжение плюсовой шины должно быть равно напряжению от вашего источника (скажем, 9-вольтовой батареи или 5-вольтового источника питания). Обязательно проверьте обе шины питания макетной платы, если они обе используются.

**Соединения на макетной плате.** Убедитесь, что соединения на макетной плате выполнены правильно. Полностью ли вставлены провода, нет ли ослабленных соединений? Дважды проверьте расположение соединений на макетной плате, находятся ли ваши провода в нужных рядах? Не подключено ли что-нибудь лишнее в проверяемом ряду?

**Резисторы.** Правильные ли значения имеют ваши резисторы? При необходимости извлеките каждый из них из схемы и проверьте мультиметром.

**Светодиоды.** Правильно ли ориентированы светодиоды? Более короткий вывод должен быть ориентирован в сторону земли (отрицательного вывода питания).

**Конденсаторы.** Если у вас полярный конденсатор, убедитесь, что положительный и отрицательный выводы ориентированы правильно. Также проверьте значение емкости.

**Интегральные схемы.** Правильно ли подключены ваши микросхемы к земле и положительному напряжению? Полностью ли микросхема установлена на макетной плате, покрывает ли она зазор в центре? Убедитесь, что ваша ИС правильно выровнена, посмотрев на ключ (выемку на корпусе). Используйте ли вы компонент с правильным наименованием типа?

**Переключатели/кнопки цифрового входа.** При использовании заземляющих резисторов: подключен ли один вывод кнопки к положительному напряжению, а другой – через заземляющий резистор к земле? Подключен ли контакт цифрового входа на соответствующей микросхеме к тому же выводу переключателя, что и заземляющий резистор?

## Raspberry Pi

Raspberry Pi – это маленький недорогой компьютер. Он был разработан для преподавания информатики и завоевал популярность среди энтузиастов технологий. Для данной книги я выбрал именно этот компьютер, поэтому здесь мы рассмотрим основы настройки и использования Raspberry Pi.

### Почему Raspberry Pi?

Прежде чем мы перейдем к деталям настройки Raspberry Pi, я хотел бы объяснить, почему выбрал именно этот компьютер для данной книги. Некоторые проекты требуют наличия компьютера для работы. Здесь вы можете спросить себя: «У меня уже есть компьютер, зачем мне еще один?» Конечно, поскольку вы читаете книгу о вычислительной технике, у вас, вероятно, уже есть компьютер, а может быть, и несколько! Однако не все владеют одинаковыми компьютерами, и некоторые типы вычислительных устройств лучше других подходят для обучения. Кроме того, некоторые из проектов в этой книге имеют дело с низкоуровневыми вычислениями, и всем, кто будет их выполнять, потребуется одно и то же устройство.

Raspberry Pi был естественным выбором, поскольку он недорогой и разработан с образовательной целью. Моя цель не в том, чтобы вы перешли на Raspberry Pi в качестве основного компьютера или стали экспертом по Raspberry Pi. Вместо этого мы используем Raspberry Pi для изучения основных понятий, которые вы сможете применить к любо-

му вычислительному устройству. В Raspberry Pi используется процессор ARM, и мы будем использовать *Raspberry Pi OS* (ранее называвшуюся *Raspbian*), версию Linux, оптимизированную для Raspberry Pi.

## Необходимые детали

Прежде всего потребуется приобрести Raspberry Pi и некоторые аксессуары. Вот что вам понадобится.

- **Raspberry Pi.** Обычно они стоят около 35 долл.<sup>1</sup> и могут быть приобретены через интернет. На момент написания этой книги последней моделью является Raspberry Pi 4 Model B, и упражнения в этой книге были протестированы на этой версии и на Raspberry Pi 3 Model B+. Если будет выпущена более новая модель, она, скорее всего, также подойдет, учитывая обратную совместимость Raspberry Pi. Raspberry Pi 4 Model B доступен в нескольких конфигурациях памяти (1ГБ, 2ГБ, 4ГБ и 8ГБ) – для этой книги подойдет любая из них.
- **Блок питания USB-C (только для Raspberry Pi 4).** В Raspberry Pi 4 используется блок питания USB-C. Блок питания должен обеспечивать напряжение 5 В и ток не менее 3 А. Некоторые блоки питания USB-C несовместимы с некоторыми устройствами Raspberry Pi 4, поэтому я рекомендую вам приобрести блок питания USB-C, специально разработанный для Raspberry Pi 4.
- **Блок питания Micro-USB (только для Raspberry Pi 3).** В отличие от Raspberry Pi 4, Raspberry Pi 3 питается от адаптера питания микро-USB, подобного тем, которые используются во многих смартфонах. Если у вас уже есть зарядное устройство для смартфона, оно может подойти для Pi. Только убедитесь, что его разъем действительно микро-USB. Стандартное напряжение на выходе таких зарядных устройств составляет 5 В, но максимальный ток, который они обеспечивают, может быть разным. Для Raspberry Pi 3 рекомендуется использовать источник питания, способный обеспечить ток не менее 2,5 А. Требования к силе тока зависят от того, что подключено к Pi. Поэтому проверьте зарядное устройство вашего смартфона, чтобы узнать, какой ток оно может обеспечить. Возможно, вам потребуется приобрести блок питания микро-USB, предназначенный именно для Pi.
- **Карта памяти Micro-SD, 8 ГБ или больше.** Raspberry Pi поставляется без какого-либо внешнего хранилища, поэтому вам придется добавить его самостоятельно с помощью карты микро-SD. Эти карты обычно используются в смартфонах и фотоаппаратах, так что, возможно, у вас уже есть лишняя карта. В процессе установки Raspberry Pi OS существующие данные будут удалены, поэтому не забудьте сделать резервную копию всего, что хранится на карте микро-SD.

<sup>1</sup> В магазине «Чип и Дип» на момент написания этих строк Raspberry Pi 3 Model B+ выставлен за 6000 руб, Raspberry Pi 4 Model B (2 ГБ) – за 7 150 руб (т. е. около 100 долл.). Но следует учитывать, что «Чип и Дип» не самый дешевый магазин. – Прим. ред.



- **USB-клавиатура и USB-мышь.** Подойдет любая стандартная USB-клавиатура и USB-мышь.
- **Телевизор или монитор с поддержкой HDMI.** Все современные телевизоры и многие компьютерные мониторы поддерживают подключение HDMI.
- **Кабель HDMI.** Raspberry Pi 3 использует стандартный полноразмерный кабель HDMI, но Raspberry Pi 4 имеет порт микро-HDMI. Предполагая, что ваш дисплей имеет полноразмерный HDMI-вход, для Raspberry Pi 4 вам понадобится кабель или переходник с микро-HDMI на HDMI.
- **Дополнительно: корпус для Raspberry Pi.** Он не обязателен, но его удобно иметь. Обратите внимание, что Raspberry Pi 3 и Raspberry Pi 4 имеют разную физическую компоновку, поэтому им нужны корпуса разной формы.

Смотрите подраздел «Покупка» в разделе «Покупка электронных компонентов для проектов» ранее в этом приложении для получения помощи в приобретении этих деталей.

## Настройка Raspberry Pi

На сайте Raspberry Pi (<https://www.raspberrypi.org>) есть подробное руководство по настройке Raspberry Pi. Я не буду описывать здесь все детали, поскольку уже существует документация в интернете, и она меняется со временем. Вместо этого позвольте мне дать краткий обзор необходимых шагов.

У вас есть несколько вариантов установки Raspberry Pi OS на ваш Raspberry Pi. Если у вас есть компьютер с устройством чтения/записи карт микро-SD, то самый простой вариант – использовать *Raspberry Pi Imager*. Вот как использовать этот инструмент, чтобы быстро начать работу с Raspberry Pi.

1. Вставьте карту микро-SD в компьютер.
2. Загрузите Raspberry Pi Imager с сайта <https://www.raspberrypi.org/downloads>.
3. Установите и запустите Raspberry Pi Imager на своем компьютере.
4. Выберите операционную систему: Raspberry Pi OS (32-bit).
5. Выберите SD-карту, которую вы хотите использовать.
6. Нажмите **Записать** (Write), и Raspberry Pi OS будет скопирована на вашу карту микро-SD.
7. Извлеките карту микро-SD из компьютера.
8. Вставьте карту микро-SD в Raspberry Pi.
9. Подключите Raspberry Pi к USB-клавиатуре, USB-мышь, монитору или телевизору с помощью HDMI и, наконец, к источнику питания.
10. Raspberry Pi должен загрузиться в Raspberry Pi OS.

Другим хорошим вариантом установки Raspberry Pi OS является использование *Raspberry Pi New Out of Box Software (NOOBS)*. Чтобы исполь-

зывать NOOBS, загрузите его с сайта <https://www.raspberrypi.org/downloads> и скопируйте на чистую карту микро-SD. Если у вас нет другого компьютера, чтобы сделать это, можете купить карту микро-SD с предварительной загрузкой копии NOOBS. В любом случае, когда NOOBS будет на карте микро-SD, вставьте карту в Raspberry Pi и включите его. Следуйте инструкциям на экране, чтобы установить Raspberry Pi OS.

#### ПРИМЕЧАНИЕ

*На момент написания этой книги 64-битная версия Raspberry Pi OS доступна в виде бета-версии. Однако проекты в этой книге были протестированы на 32-битной Raspberry Pi OS, поэтому я рекомендую вам придерживаться 32-битной версии для этих проектов.*

## Использование Raspberry Pi OS

После того как Raspberry Pi настроен, я рекомендую вам потратить немного времени на ознакомление с пользовательским интерфейсом Raspberry Pi OS. Если вы раньше пользовались компьютером Mac или Windows PC, рабочий стол Raspberry Pi OS должен показаться знакомым. Вы можете открывать приложения в окнах, перемещать эти окна, закрывать их и т. д.

Тем не менее большинство проектов в этой книге не потребует от вас использования графических приложений Pi. Почти все можно сделать из терминала, и большинство проектов требует хотя бы некоторого использования терминала, поэтому давайте потратим минуту на знакомство с ним. На рабочем столе Raspberry Pi OS можно открыть окно терминала, нажав на **Raspberry** (значок в левом верхнем углу) > **Accessories** > **Terminal**, как показано на рис. В-7.

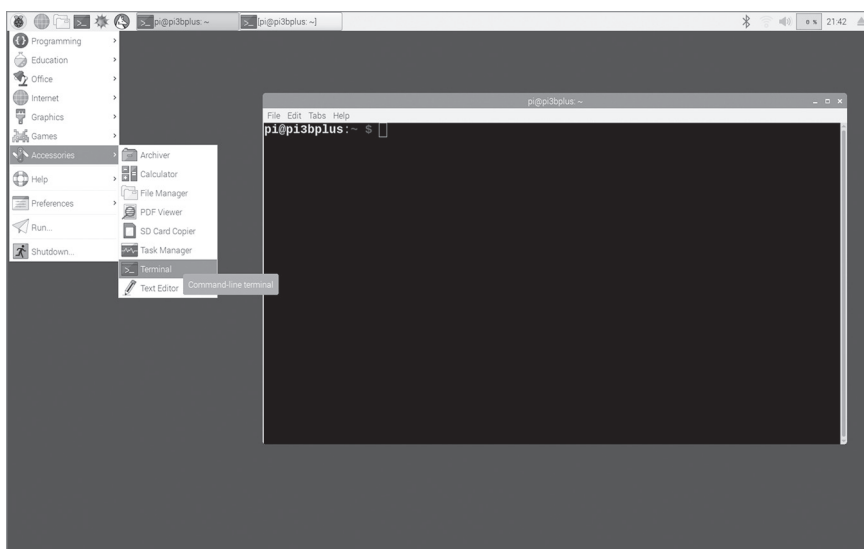


Рис. В-7. Открытие терминала Raspberry Pi



Терминал – это *интерфейс командной строки* (*command line interface, CLI*), где все действия выполняются путем ввода команд. Raspberry Pi OS, как и все версии Linux, имеет отличную поддержку CLI, и вы можете делать практически все что угодно с помощью терминала, если знаете нужные команды. По умолчанию в терминале Raspberry Pi OS запущена оболочка под названием *bash*. *Оболочка* – это пользовательский интерфейс операционной системы, который может быть графическим (как на рабочем столе) или основанным на командной строке. Начальный текст в командной строке *bash* должен выглядеть примерно так:

---

```
pi@raspberrypi:~ $
```

---

Давайте рассмотрим каждую часть этой текстовой строки:

**pi** – имя пользователя, вошедшего в систему. Имя пользователя по умолчанию – «pi»;

**raspberrypi** – имя компьютера, отделено от имени пользователя знаком @;

~ – указывает текущий каталог (папку), в котором вы работаете. Символ ~ имеет особое значение, он обозначает домашний каталог текущего пользователя;

\$ – знак доллара – это подсказка CLI, индикатор того, что вы можете вводить после него свои команды.

В этой книге, когда я перечисляю команды, которые следует набирать в терминале, я использую префикс \$ в строке. В качестве примера эта команда выводит список файлов в текущем каталоге:

---

```
$ ls
```

---

Чтобы запустить команду, *не нужно вводить знак доллара*, просто введите текст, следующий за ним, и нажмите **Enter**. Если вы хотите выполнить команду, которую вводили ранее, можете нажать стрелку вверх на клавиатуре, чтобы просмотреть ранее введенные команды.

Если вы предпочитаете работать в терминале, можете настроить Raspberry Pi на загрузку непосредственно в командную строку, а не на рабочий стол: **Raspberry > Preferences > Raspberry Pi Configuration > System tab > Boot > To CLI**. После того как вы внесете это изменение в конфигурацию, при следующем запуске система будет загружаться непосредственно в CLI, а не на рабочий стол. Если же вы хотите запустить рабочий стол, находясь в среде CLI, просто выполните следующую команду:

---

```
$ startx
```

---

Как пользователь терминала, вы можете управлять Raspberry Pi с другого компьютера или даже с телефона, используя *Secure Shell (SSH)* по

сети. Конечным результатом такого подхода является то, что ваш Pi может работать в любом месте вашей сети даже без подключенного монитора или клавиатуры, а вы можете использовать клавиатуру и монитор другого устройства для управления им. Чтобы осуществить это, необходимо включить SSH на Pi (**Raspberry > Preferences > Raspberry Pi Configuration > Interfaces tab > SSH > Enable**), а затем запустить клиентское приложение SSH на втором устройстве. Я не буду здесь приводить подробные шаги по настройке, в интернете можно найти множество инструкций, как это сделать.

Когда вы закончите работу с Pi на некоторое время, следует аккуратно завершить работу, чтобы избежать повреждения данных, а не просто выключить устройство. С рабочего стола можно выключить систему с помощью команды **Raspberry > Shutdown... > Shutdown**. Или из терминала можно использовать следующую команду для остановки системы:

---

```
$ sudo shutdown -h now
```

---

Вы поймете, что система полностью выключилась, когда на подключенном мониторе больше ничего не будет отображаться, а индикатор активности на плате Raspberry Pi перестанет мигать. После этого можете отключить Pi от питания.

## Работа с файлами и папками

В проектах этой книги регулярно требуется создание или редактирование текстовых файлов, а затем выполнение некоторых команд терминала с участием этих файлов. Давайте поговорим о работе с файлами и папками в Raspberry Pi OS как из командной строки, так и с графического рабочего стола. Операционные системы организуют данные на устройстве памяти, таком как карта микро-SD в Raspberry Pi, с помощью файловой системы. *Файл* – это контейнер с данными, а *папка* (также известная как каталог) – это контейнер, содержащий файлы или другие папки. Структура файловой системы – это *иерархия*, дерево папок. В системах Linux корень этой иерархии обозначается символом /. Корень – это самая верхняя папка, все остальные папки и файлы находятся «под» корнем.

Папка, находящаяся непосредственно под корневой папкой, будет представлена следующим образом: /<имя папки>. Текстовый файл в этой папке будет выглядеть примерно так: /<имя папки>/<имя файла>.txt. Обратите внимание на расширение файла .txt – последнюю часть имени файла. Принято заканчивать имя файла точкой, за которой следует несколько символов, указывающих на тип данных в файле. В случае с текстовыми файлами часто используется txt. Использование расширений файлов не является обязательным требованием, но это общепринятая практика, полезная для упорядочивания данных.

У каждого пользователя в Raspberry Pi OS есть домашняя папка для работы. Пользователь по умолчанию в Raspberry Pi OS называется pi,

а домашняя папка пользователя `pi` находится в `/home/pi`. Пока вы входите в систему под пользователем `pi`, на эту же домашнюю папку можно ссылаться с помощью символа `~`. Допустим, вы создадите в своей домашней папке папку под названием `pizza`. Полный путь к ней будет таким: `/home/pi/pizza`. Или, войдя в систему под именем `pi`, вы можете ссылаться на нее так: `~/pizza`. Давайте попробуем создать папку `pizza` из окна терминала, используя команду `mkdir`, сокращенно от *make directory* (создать папку). Не забудьте нажать **Enter** после ввода команды.

---

```
$ mkdir pizza
```

---

Из терминала вы можете увидеть только что созданную папку с помощью команды `ls`:

---

```
$ ls
```

---

Когда вы введете команду `ls` и нажмете **Enter**, вы должны увидеть папку `pizza` рядом с другими, которые уже присутствовали в вашей домашней папке, такими как *Desktop*, *Downloads* и *Pictures*.

Терминал является не единственным способом просмотра файлов в папке. Вы также можете использовать приложение File Manager (диспетчер файлов), которое можно запустить из **Raspberry > Accessories > File Manager**. Как показано на рис. В-8, приложение File Manager открывается по умолчанию с демонстрацией вашего домашнего каталога.

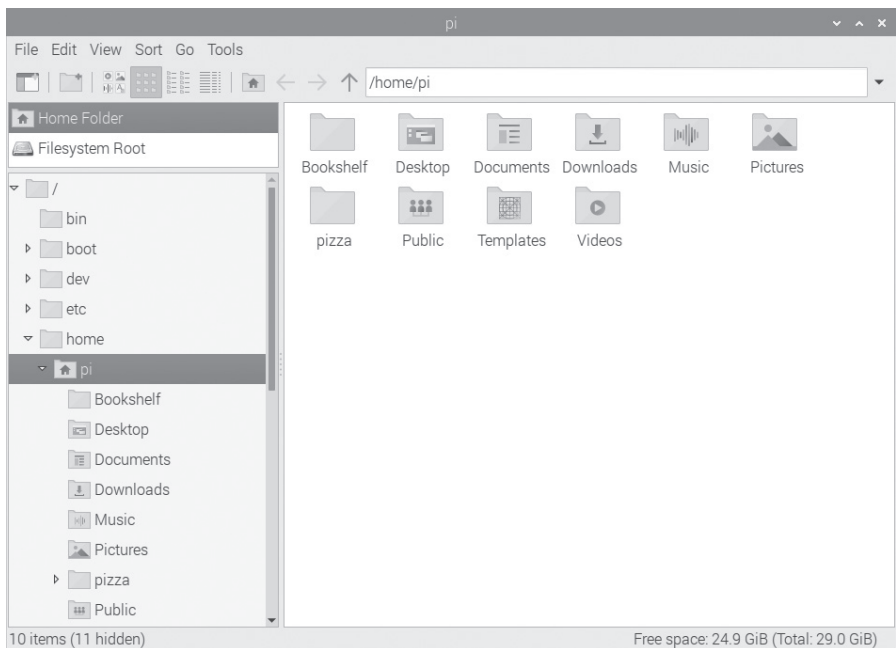


Рис. В-8. Raspberry Pi OS File Manager

В левой части File Manager отображается полная иерархия папок файловой системы, а выбранная в данный момент папка выделена цветом, в данном случае это *pi*. Адресная строка в верхней части */home/pi* указывает на текущую папку. Теперь попробуйте дважды кликнуть по папке *pizza*, она должна быть пустой. Давайте вернемся в окно терминала и создадим несколько файлов в этой папке. Сначала поменяем папки с помощью команды `cd` (для смены каталога), чтобы нашей текущей папкой стала папка *pizza*. Затем создадим два пустых файла с помощью команды `touch`. Наконец, выведем содержимое каталога командой `ls`, ожидая увидеть в списке два новых файла.

---

```
$ cd pizza
$ touch cheese.txt
$ touch crust.txt
$ ls
```

---

Обратите внимание, что, когда вы перешли в папку *pizza*, начальная строка `bash` также должна была измениться. Теперь она должна включать *~/pizza* перед `$`, указывая на текущую папку. Теперь посмотрите на окно приложения File Manager, оно также должно показывать два новых файла в папке *pizza*, как показано на рис. В-9.

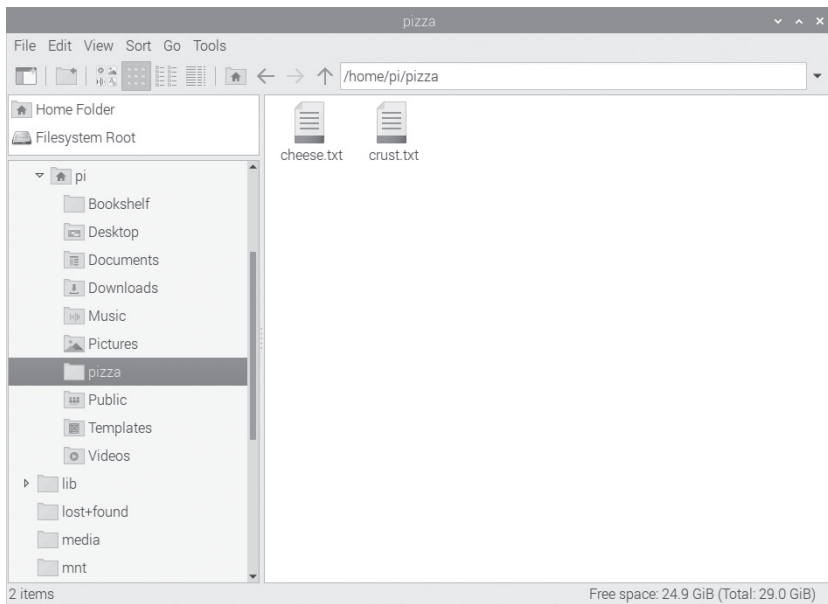


Рис. В-9. Raspberry Pi OS File Manager: файлы в папке *pizza*

Теперь у нас есть два пустых файла в папке *pizza*. Давайте добавим текстовое содержимое в эти файлы. Сначала отредактируем файл *cheese.txt* с помощью текстового редактора командной строки `nano`.

---

```
$ nano cheese.txt
```

---

Открыв окно редактора `nano` в терминале, вы можете набрать текст, который будет сохранен в файле `cheese.txt`. Помните, что `nano` – это приложение командной строки, и вы не можете использовать мышь. Для перемещения курсора необходимо использовать клавиши со стрелками. Попробуйте набрать какой-нибудь текст, как показано на рис. В-10.

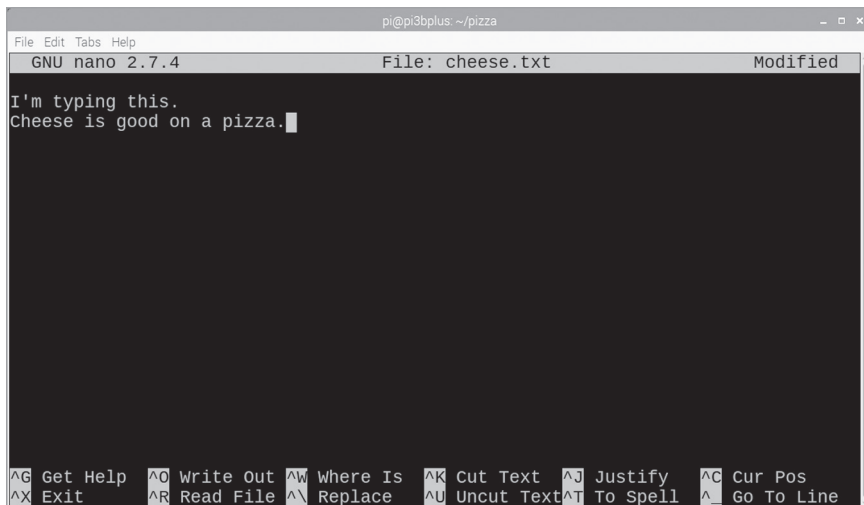


Рис. В-10. Использование `nano` для редактирования `cheese.txt`

После ввода некоторого текста в `nano` нажмите **CTRL-X**, чтобы выйти из `nano`. Редактор предложит вам сохранить вашу работу (Save modified buffer?). Этот вопрос может показаться странным, но не позволяйте термину «buffer» сбивать вас с толку, `nano` просто спрашивает, хотите ли вы сохранить введенный текст в файле. Нажмите **Y**, затем нажмите **Enter**, чтобы принять предложенное имя файла (`cheese.txt`).

Я часто использую `nano`, поскольку он работает из терминала, но вы можете предпочесть редактировать текстовые файлы с помощью графического текстового редактора. Среда Raspberry Pi OS включает удобный текстовый редактор, который можно запустить из **Raspberry > Accessories > Text Editor**. На момент написания этой книги открывался редактор под названием *Mousepad*. Давайте попробуем изменить файл `cheese.txt` в этом текстовом редакторе. Сначала нам нужно открыть файл, выполнив следующие действия в текстовом редакторе: **File > Open > Home > pizza > cheese.txt > Open** (кнопка), как показано на рис. В-11.

После того как откроете файл в текстовом редакторе, вы должны увидеть текст, который набрали ранее. Можете отредактировать его по своему усмотрению. Затем сохраните изменения посредством **File > Save**.

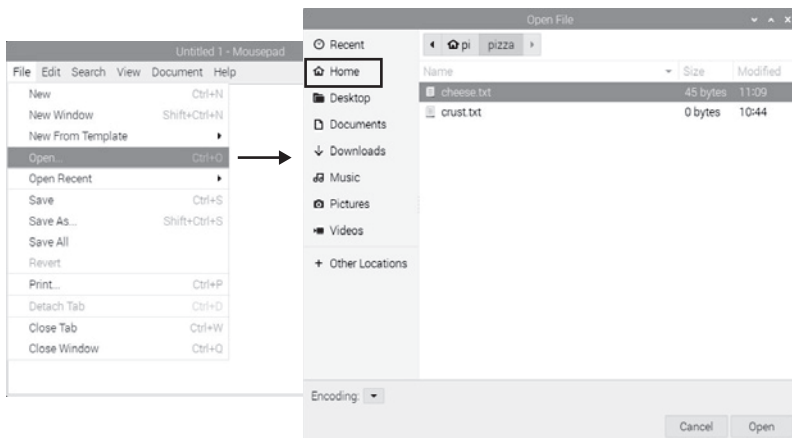


Рис. В-11. Открытие *cheese.txt* в текстовом редакторе

Помимо редактирования существующих файлов, вы также можете использовать текстовый редактор для создания новых. Просто запустите новое окно редактора (**Raspberry > Accessories > Text Editor**) и нажмите **File > Save**, что предложит вам сохранить файл в выбранной вами папке с выбранным вами именем. Вы можете редактировать текст нового файла и сохранять изменения по мере необходимости, снова используя **File > Save**. Или, если хотите сохранить существующий файл с новым именем, выберите команду **File > Save As...** вместо этого.

Если вы предпочитаете использовать `nano` для создания нового файла, то в окне терминала сначала перейдите в папку, в которой хотите сохранить файл (если это необходимо), а затем введите `nano filename`, как показано в этом примере:

---

```
$ nano new-file.txt
```

---

Введите текст, и когда выйдете из `nano`, вам будет предложено сохранить этот новый файл.

Мы только что рассмотрели способы просмотра, редактирования и создания текстовых файлов в Raspberry Pi OS. Если вы хотите оставаться в терминале, то `nano` – это хороший выбор. Если вы предпочитаете рабочий стол, то текстовый редактор *Mousepad* должен удовлетворить ваши потребности. Есть и другие редакторы, входящие в комплект Raspberry Pi OS. *Geany* – это текстовый редактор для программистов, а *Thonny Python IDE* предназначен для программирования на Python. Оба можно найти в разделе **Raspberry > Programming**. В проектах этой книги я оставляю за вами право решать, какой текстовый редактор вы будете использовать.

Вы также можете захотеть управлять файлами и папками другими способами, например перемещать файлы, удалять их и т. д. Все это можно делать из диспетчера файлов, а можно из окна терминала. Вот несколько команд, которые можно использовать в оболочке `bash` для начала:

**cd *folder*** – изменить текущий каталог (папку);

**mkdir *folder*** – создать каталог;

**rm *file*** – удалить файл;

**rm -rf *folder*** – удалить папку и ее содержимое, включая вложенные папки;

**mv *file file2*** – переименовать файл;

**mv *file folder*/** – переместить файл из одного места в другое;

**cp *file folder*/** – копировать файл из одного места в другое.

Книги издательства «ДМК Пресс» можно заказать  
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,  
выслав открытку или письмо по почтовому адресу:  
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.  
При оформлении заказа следует указать адрес (полностью),  
по которому должны быть высланы книги;  
фамилию, имя и отчество получателя.  
Желательно также указать свой телефон и электронный адрес.  
Эти книги вы можете заказать и в интернет-магазине: **[www.a-planet.ru](http://www.a-planet.ru)**.  
Оптовые закупки: тел. (499) 782-38-89.  
Электронный адрес: **[books@aliants-kniga.ru](mailto:books@aliants-kniga.ru)**.

**Мэттью Джаггис**

## **КАК НА САМОМ ДЕЛЕ РАБОТАЮТ КОМПЬЮТЕРЫ**

Главный редактор	<i>Мовчан Д. А.</i>
	<a href="mailto:dmkpress@gmail.com">dmkpress@gmail.com</a>
Зам. главного редактора	<i>Сенченкова Е. А.</i>
Научный редактор	<i>Ревич Ю. В.</i>
Перевод	<i>Плеханова С. Л.</i>
Корректор	<i>Абросимова Л. А.</i>
Верстка	<i>Луценко С. В.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Формат 70×100 1/16.

Гарнитура «NewBaskervilleC». Печать цифровая.

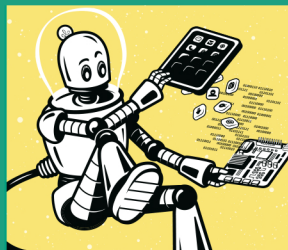
Усл. печ. л. 34,78. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)



Лучше один раз увидеть,  
чем сто раз услышать!

Подробные схемы и рисунки в книге  
помогают прояснить технические  
сложности.



Эта книга по экосистеме персонального компьютера: от оперативной памяти, тактовых сигналов и машинного кода до языков программирования, операционных систем и интернета. Но вы не просто изучите теорию – вы сможете проверить свои знания с помощью упражнений, а также выполните 41 проект для закрепления пройденного.

Создавайте цифровые схемы, сделайте игру-угадайку, переводите десятичные числа в двоичные, изучайте использование виртуальной памяти, пробуйте «мыслить как компьютер», выполняя программу в уме, шаг за шагом!

### **В процессе чтения вы приобретете практические навыки:**

- научитесь пользоваться мультиметром для измерения сопротивления, тока и напряжения;
- соберете полусумматор, чтобы увидеть, как логические операции в аппаратном обеспечении могут быть объединены для выполнения полезных функций;
- напишете программу на языке ассемблера, а затем изучите полученный машинный код;
- научитесь использовать отладчик, разбирать код и взламывать программу, чтобы изменить ее поведение без изменения исходного кода;
- используете сканер портов, чтобы узнать, какие интернет-порты открыты на вашем компьютере;
- запустите свой собственный сервер и пройдете полный курс обучения работе в интернете.

**Мэтью Джастис**, инженер-программист, в течение 17 лет сотрудничал с Microsoft, в частности, занимаясь отладкой ядра Windows, разрабатывая автоматические исправления и руководя группой инженеров, создающих диагностические инструменты и службы. Он работал на всех направлениях – от низкоуровневого программного обеспечения до высокоуровневых веб-приложений.

Интернет-магазин:  
[www.dmkpress.com](http://www.dmkpress.com)  
Оптовая продажа:  
КТК «Галактика»  
[books@aliens-kniga.ru](mailto:books@aliens-kniga.ru)

**ДМК**  
пресс  
издательство  
[www.dmk.pf](http://www.dmk.pf)

ISBN 978-5-97060-973-6



9 785970 609736 >