Брюс Смит

АССЕМБЛЕР ДЛЯ RASPBERRY PI

Практическое руководство

4-е издание

+ Материалы на www.bhv.ru

УДК 004.4
ББК 32.973.26-018.2
С50

Смит Б.

С50 Ассемблер для Raspberry Pi. Практическое руководство: Пер. с англ. — 4-е изд.— СПб.: БХВ-Петербург, 2022. — 320 с.: ил.

ISBN 978-5-9775-6801-2

Рассмотрены основы программирования на языке ассемблера для процессоров ARM на примере Raspberry Pi с операционной системой Raspberry Pi OS. Приведены подробные сведения об архитектуре и особенностях ARM, вызовах операционной системы. Подробно описан синтаксис ассемблера для ARM. Рассмотрены компоновщик GCC, отладка с GDB, использование функций языка C в ассемблере с помощью библиотеки libc. Описаны функции GPIO, система команд ARM Neon и команды Thumb. Все разделы снабжены практическими примерами. Книга ориентирована на начинающих разработчиков, желающих освоить программирование на языке ассемблера для устройств с архитектурой ARM.

Электронный архив на сайте издательства содержит исходный код программ из книги.

Для начинающих программистов

УДК 004.4
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Михаила Райтмана*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-0-6480987-3-7 (англ.)
ISBN 978-5-9775-6801-2 (рус.)

Оглавление

2. Начало

Ассемблер как язык позволяет вам обращаться к родному языку Raspberry Pi — *машинному коду*. Это собственный язык ARM-чипа, который по сути есть разум и душа вашей компьютерной системы. ARM расшифровывается как Advanced RISC Machine (продвинутая машина с сокращенным набором команд), и именно этот чип заведует всем, что происходит с вашей Raspberry Pi.

Микропроцессоры, такие как ARM, управляют использованием и передачей данных. Процессор также часто называют ЦП — *центральным процессором*, и данные, которые обрабатывает ЦП, текут сквозь него в виде непрерывного, почти бесконечного потока единиц и нулей. Порядок этих единиц и нулей не случайный, и определенная их последовательность превращается в набор определенных действий. Это похоже на азбуку Морзе, где правильная последовательность точек и тире превращается в осмысленные буквы и слова.

`-.-. .-.. . ...- . .-.` (здесь закодировано слово CLEVER, умный)

Числа со смыслом

Сама программа, написанная машинным кодом, представляет собой бессчетное множество строк из единиц и нулей. Вроде вот этого:

```
11010111011011100101010100001011
01010001011100100110100011111010
01010100011001111111001010010100
10011000011101010100011001010001
```

Понять, что именно означают эти числа, почти невозможно (или возможно, но потребовало бы очень много времени). Ассемблер помогает преодолеть эти проблемы.

Язык ассемблера — это своего рода сокращенная форма, которая позволяет писать программы машинного кода с помощью английской лексики. А *ассемблер* — это программа, которая переводит программу на языке ассемблера в машинный код, избавляя нас от трудоемкой и рутинной расшифровки цифр. Программа на языке ассемблера часто представляет собой обычный текстовый файл, который читается

компилятором и преобразуется в двоичный эквивалент (из единиц и нулей). Программа на языке ассемблера называется *входным*, или *исходным*, файлом, а машинный код — *объектным файлом*. Ассемблер переводит (или компилирует) исходный файл в объектный файл.

Язык ассемблера написан с использованием мнемоники. *Мнемоника* — это механизм быстрого обучения и запоминания, который основывается на ассоциациях между легко запоминающимися последовательностями букв и информацией, которую нужно запомнить. В естественном языке ту же функцию выполняют аббревиатуры. Например, чтобы запомнить цвета радуги, вы можете использовать фразу: «Каждый Охотник Желает Знать, Где Сидит Фазан», и взять первую букву каждого слова.

В среде SMS-сообщений и интернет-чатов тоже возник свой мнемонический язык. Благодаря ему текстовые сообщения становятся короче и компактнее. Например, «мп» может означать «мои поздравления», «спс» — «спасибо», а «дв» — «до встречи».

Команды ARM

У микросхемы ARM есть определенный набор команд машинного кода, которые она понимает. Именно таким операциям (или «кодам операций») и их использованию посвящена эта книга. ARM — это всего лишь один из типов микропроцессоров. Существуют и другие типы, и у каждого из них свой уникальный набор команд.

Нельзя просто так взять программу машинного кода, написанную для ARM, и успешно запустить ее на другом микропроцессоре. Она либо не сработает так, как ожидалось, либо вообще не запустится. Но понятия и концепции, о которых мы поговорим, будут применимы в работе с большинством других видов микропроцессоров и в аналогичных задачах. Если вы научитесь программировать на одном ассемблере, с программированием на других будет уже проще. По сути, в новом языке нужно будет просто выучить новый набор команд, причем многие из них будут похожи на уже известные вам.

Микропроцессоры в ходе своей работы перемещают и обрабатывают данные, поэтому неудивительно, что многие команды машинного кода посвящены именно этим операциям и у большинства *наборов команд* (собирательный термин для этих мнемоник) есть команды для сложения и вычитания чисел. Мнемоника языка ассемблера, используемая для этих операций, обычно выглядит так:

```
ADD
SUB
```

Это простой пример, и многие другие мнемоники ARM тоже просты, если рассматривать их в отдельности. Но вот последовательность строк, объединенная в цельную программу, выглядит уже сложнее. Разбив программу на составные части, можно без труда понять, как все работает.

Мнемоника языка ассемблера обычно состоит из трех символов, но иногда их бывает и больше. Как и в любом новом деле, вам нужно будет немного привыкнуть, но, если вы проработаете примеры, приведенные в этой книге, и примените их в своих собственных экспериментах, у вас не должно возникнуть особых проблем.

Так, `MOV` — мнемоника команды MOVe (переместить). Эта команда берет информацию из одного места и перемещает ее в другое место. Проще простого, да?

Процесс преобразования

Когда вы разработали свою программу на языке ассемблера, ее нужно преобразовать в машинный код с помощью *компилятора ассемблера*. Например, когда ассемблер встречает мнемонику `MOV`, он подставит вместо нее число, которое для процессора представляет собой инструкцию. Ассемблер сохраняет собранный машинный код в виде файла в памяти, после чего этот код можно будет запускать или выполнять. В процессе сборки программы ассемблер также проверяет ее синтаксис, чтобы убедиться в том, что все написано правильно. Если он обнаружит ошибку, то сообщит вам об этом и скажет ее исправить. Исправив проблему, вы можете еще раз попробовать собрать программу. Обратите внимание, что проверка синтаксиса гарантирует лишь правильность написания самих команд ассемблера. Но никто не проверит за вас логику программы, и поэтому, если вы написали сами команды правильно, но суть изложили неверно, программа соберется, но сработает неверно. Например, вам нужно сложение, а вы вместо этого вы запрограммировали вычитание!

Написать программу на ассемблере можно разными способами. Первые чипы ARM были разработаны Acorn и, как и стоило ожидать, они появились в ряде компьютеров на базе Acorn, работающих под управлением ОС RISC. К ним относились Archimedes и RISC PC. На этих машинах использовался компилятор BBC BASIC, который был чем-то новым в том смысле, что позволял писать программы на языке ассемблера как расширение BBC BASIC. Вы и сейчас можете писать этим методом, установив RISC OS на свою Raspberry Pi.

Как вы уже поняли, в этой книге мы предполагаем, что вы используете операционную систему Raspberry Pi и программное обеспечение GNU GCC. Есть и другое ПО для ассемблера, причем большая часть его бесплатна, и вы без труда найдете его в Интернете. Основным преимуществом программирования на GCC является то, что этот компилятор также может собирать программы, написанные на языке C.

Эта книга не посвящена программированию на C, но по ряду причин нам будет полезно познакомиться с некоторыми его моментами, и даже эта малость может помочь вам в написании программ. Подробнее мы поговорим об этом в книге на примерах позже.

В целом вам ничто не мешает попробовать любой другой или вообще все ассемблеры, и знания, которые вы получите в этой книге, помогут вам в этом.

А зачем вообще машинный код?

На этот вопрос ответить нетрудно. По сути, все, что делает ваш Raspberry Pi, выполняется с использованием машинного кода. Программируя напрямую в машинном коде, вы работаете на самом фундаментальном уровне работы Raspberry Pi.

Когда вы используете язык высокого уровня, такой как BBC BASIC или Python, все его операции все равно превращаются в машинный код всякий раз, когда вы запускаете программу. Это занимает некоторое время, хотя для нас эти доли секунды и незаметны. Но тем не менее процесс преобразования, или интерпретации, замедляет работу программного обеспечения. Фактически даже самые эффективные языки могут быть более чем в 30 раз медленнее, чем их эквивалент машинного кода, и это еще если повезет!

Если же вы программируете на машинном коде, программа будет работать намного быстрее, поскольку здесь преобразование не выполняется. Когда вы запускаете ассемблер, процесс преобразования происходит один раз, но однажды преобразованный машинный код можно многократно запускать напрямую. Необязательно запускать ассемблер каждый раз. Если вы довольны работой своей программы, то можете сохранить машинный код и использовать именно его. Вы также можете сохранить исходную программу на языке ассемблера и работать с ней, особенно если хотите что-то поменять в ее коде позже.

Языковые уровни

Языки вроде C или Python называются *языками высокого уровня*. На языках высокого уровня часто легче писать, поскольку их синтаксис похож на английский язык, а еще в них есть команды, которые выполняют большую и сложную последовательность действий, для подробного расписывания которой потребовался бы длинный список команд машинного кода. Машинный код — это *язык низкого уровня*, поскольку он работает в самых внутренностях компьютера, описывая каждое действие, и из-за этого его труднее понимать.

Это, конечно, преимущество языков высокого уровня по сравнению с языками низкого уровня. Но пока вы набираетесь опыта в языке ассемблера, ничто не мешает вам создавать библиотеки подпрограмм для выполнения конкретной задачи и просто добавлять их в свои программы по мере их написания. Впрочем, если вы углубитесь в мир ARM, то выяснится, что нужные библиотеки уже существуют.

Написанные на ассемблере программы легко переносить на другие компьютеры или системы, работающие на чипе ARM. Нужно лишь загрузить файл ассемблера в компилятор на новой машине, собрать код в программу машинного кода и запустить ее.

Компилятор GNU GCC есть практически на всех разновидностях микропроцессоров, поэтому, познакомившись с GCC, вы сможете перенести свои новые навыки на другие системы, если захотите.

Если раскрыть возможности чипа ARM на полную катушку, вы даже можете напрямую передавать и запускать машинный код. Эти возможности невероятно кру-

ты, особенно учитывая тот факт, что почти каждый существующий сегодня смартфон или планшет работает на чипах ARM!

На орбиту!

Чтобы еще раз подчеркнуть возможности чипа ARM и смартфонов в целом, вспомним, что на орбиту Земли было выведено целое новое поколение спутников под названием CubeSats (рис. 2.1). Они небольшие (порядка 10 см) и выполняют определенные задачи. Космический центр Суррея на юге Великобритании разработал несколько спутников CubeSat, которые работают на смартфонах Android. Эти спутники стоят около 100 тыс. долларов каждый, что гораздо меньше, чем стоимость аппаратов прошлого поколения. Опять же, вычислительная мощность одного современного смартфона, возможно, в десятки тысяч раз больше, чем у компьютеров всех лунных миссий Аполлона, вместе взятых! И вся эта мощь — вот она, прямо в Raspberry Pi.

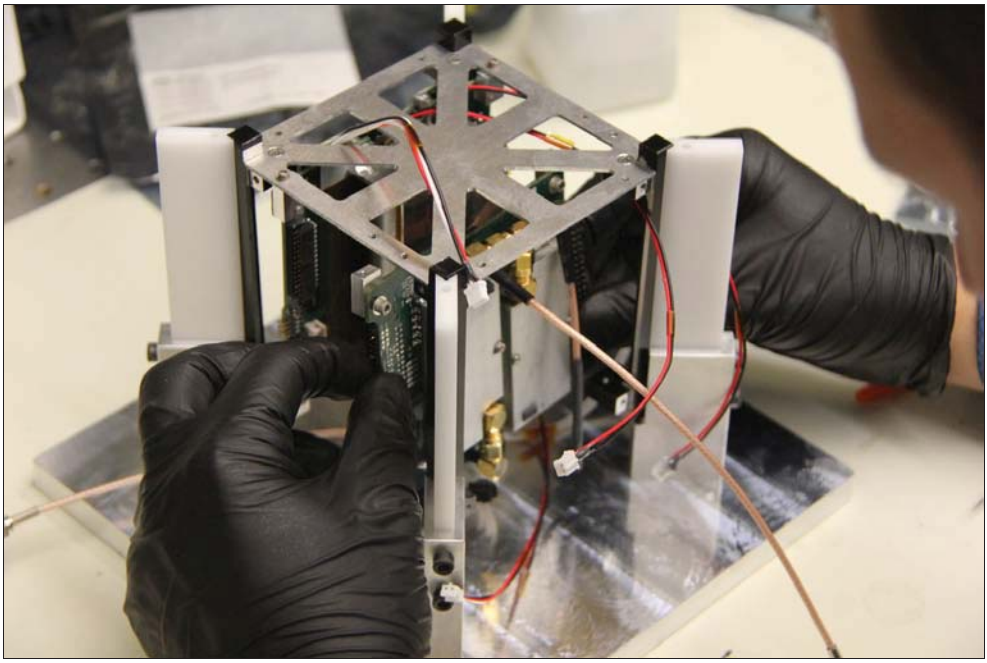

Рис. 2.1. Изготовление CubeSat

Но не все коту масленица! У плат есть различия в версиях ЦП. Как и в случае с программным обеспечением, чип ARM постоянно совершенствовался и возникали проблемы с новыми версиями. Однако базовый набор команд остается прежним, поэтому «перенос» будет не так сложен, как может показаться. Реальной проблемой он становится только в том случае, если вы используете более продвинутые функции микропроцессора. Для нашего вводного руководства эти изменения не актуальны, так что в этой книге все подойдет к вашей Raspberry Pi.

RISC и наборы команд

Буква R в аббревиатуре ARM означает RISC. Это тоже сокращение от Reduced Instruction Set Computing (вычисления с сокращенным набором команд). Все процессоры работают на машинном коде, и каждая из команд машинного кода или код операции выполняет свою определенную задачу. Объединенные вместе, команды образуют набор команд.

Идея RISC заключалась в создании небольшого оптимизированного набора команд. У этого подхода есть несколько преимуществ — меньше придется запоминать. Но при этом образуется большее разнообразие в их использовании.

Структура ассемблера

Программирование на любом языке, как и общение на любом языке, подчиняется некоторому ряду правил. Эти правила определяются структурой и синтаксисом языка, который мы используем. Чтобы писать программы, нам нужно знать синтаксис языка и правила, которые определяют структуру программы.

Самый простой способ разработать программу — просто написать список задач, который она должна выполнить. Программа начинается с начала и выполняется команда за командой, пока не дойдет до конца. Другими словами, все команды выполняются по очереди, пока не будет достигнут конец программы. Это работает, но весьма неэффективно.

Современные языки более структурированы и позволяют писать программы в виде множества независимо выполняемых процедур, или подпрограмм. Эти подпрограммы вызываются из основной программы по мере необходимости. Таким образом, основная программа управляет потоком выполнения и запускает операции, которые подпрограммам недоступны.

Программы, написанные с помощью подпрограмм, получаются меньше и лучше поддаются редактированию. В линейной программе нам, вероятно, пришлось бы много раз повторять большие участки кода, чтобы выполнить поставленную задачу.

В листинге 2.1 показан некоторый псевдокод, на примере которого я иллюстрирую, как может выглядеть структурированная программа. В этом примере команды программы написаны прописными буквами. Разделам кода подпрограммы даются имена, которые пишутся в нижнем регистре и, как это ни парадоксально, с точкой в начале записи. Весь поток программы содержится в шести строках, начинающихся с раздела `.main` и заканчивающихся словом `END`. Программа получается короткой, но читается ясно, и с первого взгляда становится понятно, что в ней происходит. Подпрограммам специально даются осмысленные имена.

В этом примере основная программа просто вызывает некоторые подпрограммы. В идеальном мире мы бы писали именно такие программы, поскольку такой подход также упрощает тестирование этих подпрограмм по отдельности, прежде чем мы включим их в основную программу. Это позволяет гарантировать, что наша программа будет работать правильно.

Листинг 2.1. Псевдокод, иллюстрирующий структурированный подход к программированию

```
.main
     DO getkeyboardinput
     DO displayresult
     DO getkeyboardinput
     DO displayresult
END
.getkeyboardinput
     ; Instructions to read input from keyboard
RETURN
.displayresult
     ; print the result on the screen
RETURN
```

Ошибки на пути

Есть довольно серьезная проблема, с которой вы обязательно столкнетесь при изучении любой новой программы, — поиск ошибок. Этот процесс еще называют *отладкой*. Я гарантирую (и многократно убеждался в этом), что первый вариант вашей программы всегда будет работать неправильно. Вы будете медитировать над кодом до вечерней зорьки, но ошибки так и не найдете. Затем вы будете настаивать на своей правоте и валить все на компьютер. И лишь потом на вас снизойдет озарение, и вы увидите, что ошибка все это время была на самом видном месте.

Создав подпрограмму, а затем протестировав ее отдельно и убедившись, что она работает, вы будете знать, что и в основной программе все будет работать правильно и что сединой вы раньше времени не обзаведетесь.

Кросс-компиляторы

С этим термином вы, вероятно, будете сталкиваться часто. Компилятор GCC встречается на множестве компьютеров, и даже на тех, которые не работают на чипе ARM. Вы можете писать и скомпилировать ассемблер ARM вообще на другом компьютере! Но запустить собранный машинный код уже не сможете. Придется сперва перенести его с главной машины на целевую (например, Raspberry Pi). GCC — не единственный компилятор для Raspberry Pi и не единственный кросс-компилятор. Есть и много других. Загляните на форумы на сайте Raspberry Pi для получения подобной информации, и еще можете поискать в Интернете, если интересно.

Чипы Raspberry Pi ARM

Чип ARM, используемый в Raspberry Pi Zero, A, B, A+, B+ — это (приведем полное название) мультимедийный процессор Broadcom BCM2835 System-on-Chip. Термин «System-on-Chip» (система на кристалле, SoC) означает, что чип содержит почти

все необходимое для запуска Raspberry Pi в единой структуре (и именно поэтому у Raspberry Pi столь малые габариты). В BCM2835 используется дизайн ARM11, который построен на наборе команд ARMv6.

В Raspberry Pi 2 используется чип SoC BCM2836. В нем есть все функции BCM2835, но один ARM11 с тактовой частотой 700 МГц заменен на четырехъядерный ARM Cortex-A7 с тактовой частотой 900 МГц, а все остальное остается прежним. Этот чип быстрее, в нем больше памяти, и он может запускать достаточно серьезное программное обеспечение, такое как Windows 10 и весь спектр дистрибутивов ARM GNU/Linux.

В основе Raspberry Pi 3 лежит чип ARM v8, опять же с использованием структуры SoC и с еще большей частотой 1,2 ГГц. В его основе четыре высокопроизводительных процессора ARM Cortex-A53, работающих в тандеме. А еще это 64-битный процессор, который может работать как в состоянии AArch32, так и в AArch64.

В Raspberry Pi 4 используется Broadcom 2711, который стал еще быстрее — 1,5 ГГц. ARMv8 — это четырехъядерный процессор A72, 64-битный процессор, который может работать как в состоянии AArch32, так и в AArch64. Версия 8 Гбайт позволяет использовать память полной 64-разрядной версии для эффективной работы и обработки приложений — ничуть не хуже обычного ПК!

Что-то я увлекся терминами. Пока вы еще новичок, не стоит слишком забивать голову. По мере того как вы начнете узнавать больше о Raspberry Pi, все эти вещи станут вашей второй натурой. Мы вернемся к SoC в конце книги и объясним эту концепцию более подробно.

Кстати, частота в один мегагерц (1 МГц) соответствует миллиону циклов в секунду. Скорость микропроцессоров, называемая *тактовой частотой*, часто измеряется в МГц. Например, микропроцессор, работающий на частоте 700 МГц, выполняет 700 млн циклов в секунду. 1,2 ГГц (гигагерц) — это 1,2 млрд циклов в секунду. Позже мы увидим, как эта скорость влияет на выполнение команд.

Еще прозвучал термин «четырехъядерный». Четырехъядерный процессор имеет четыре независимых блока, называемых «ядрами», которые одновременно читают и выполняют команды. В целом четырехъядерный процессор будет работать быстрее, чем двухъядерный или одноядерный. Каждая открытая программа может работать на собственном ядре, поэтому, если задачи разделены между ядрами, скорость будет лучше. Эта так называемая параллельная обработка является основной особенностью ARM.

3. Проба пера

В этой главе мы пошагово рассмотрим процедуру создания и запуска программы машинного кода, начиная с момента включения Raspberry Pi и заканчивая внесением корректив в уже готовую программу. Первая программа не будет делать ничего фантастического. На самом деле не произойдет вообще ничего, кроме возврата символа приглашения, но даже в написание этой программы вовлечен каждый шаг, который вам нужно знать и использовать при создании и запуске других программ из этой книги.

Прямо сейчас я предполагаю, что у вас есть копия образа Raspberry Pi OS (Raspbian) на SD-карте, вставленной в ваш Raspberry Pi, и что вы использовали ее хотя бы один раз, чтобы выполнить начальную настройку таких важных моментов, как клавиатура и Интернет. Если вы еще этого не сделали, сделайте это сейчас, чтобы можно было продолжить.

Командная строка3. Проба пера

При первой загрузке вы автоматически войдете в систему и попадете на рабочий стол. В правом верхнем углу выполните двойной щелчок мышью на значке монитора. Перед вами откроется окно **Terminal**, и вы попадете в командную строку, где увидите примерно такое приглашение ввода:

```
pi@raspberrypi   $
```

Командная строка — это консоль, в которую вы вводите команды, которые затем выполнит ваша ОС. Командная строка начинается там, где мигает курсор. Консоль ожидает, что вы будете вводить команды, которые затем можно выполнить нажатием клавиши <Return> (также называемой клавишей <Enter>). Попробуйте ввести вот такую команду:

```
dir
```

Ввести ее нужно в точности так, как показано здесь. При нажатии клавиши <Return> перед вами появится список всех каталогов или файлов, которые находятся в текущем каталоге. (С этого момента я не буду каждый раз говорить о нажатии

клавиши <Return>, но вы имейте в виду, что если нужно ввести что-то с клавиатуры, особенно в командной строке, нужно будет нажать клавишу <Return>.)

Теперь введите вот это:

```
Dir
```

Вы получите такой ответ:

```
bash: Dir: command not found
```

Это *сообщение об ошибке*. В командной строке регистр имеет значение, поэтому команды:

```
dir
```

и

```
Dir
```

ОС считает разными, если учитывает регистр. То же самое и с именами файлов:

```
program1
```

и

```
Program1
```

— это разные файлы.

По соглашению о написании команд в командной строке всегда используется нижний регистр. Команды чувствительны к регистру. А в именах файлов могут быть и строчные, и прописные буквы, если вы понимаете разницу. Лучше всегда использовать строчные символы, поэтому не забываем поглядывать на индикатор клавиши <CapsLock>.

Создание исходного файла

Чтобы создать программу с машинным кодом, нужно выполнить процесс из трех действий «написать — собрать — связать», в результате которого мы получим файл, который можно будет запустить. Первый шаг — написать программу на ассемблере. Поскольку именно этот код является источником программы, файл называется *исходным* файлом. У такого файла будет расширение `s` после имени. Например:

```
program1.s
```

Исходные файлы можно писать в любом удобном текстовом редакторе. Существует много отличных и бесплатных редакторов, поэтому стоит уделить время изучению их обзоров и проверить пару вариантов самостоятельно. Возможно, у вас уже есть любимый редактор, и с этим этапом вы разобрались.

В Raspberry Pi OS уже есть набор редакторов, установленных в разделе **Recommended Software**, и вы можете найти их в главном меню приложения (пункт **Raspberry** в меню на рабочем столе). Скорее всего, нужные вам редакторы находятся в списке **Accessories**. Сюда входят (на момент подготовки книги) редакторы Vim, gVim, а также редактор Geany Programmer's Editor (советую попробовать каждый из них и выбрать тот, который вам больше нравится).

Если ни Vim, ни gVim в системе не установлены, вы можете исправить эту проблему с помощью параметра **Recommended Software** или из командной строки, набрав команду:

```
sudo apt-get install vim
```

После этого надо будет ответить на пару запросов: вас могут спросить о необходимости дополнительных функций. Ответ `Y` вполне подойдет. Установка займет несколько минут, а на сайте Vim тем временем можно найти много полезных советов.

Если вам больше нравится работать в приложении, возьмите версию Vim с графическим интерфейсом и при необходимости установите ее командой:

```
sudo apt-get install vim-gtk
```

Поскольку вам предстоит провести много времени за программированием на ассемблере, имеет смысл уделить внимание изучению тонкостей Vim. Многие действия и операции, используемые в Vim, выполняются с помощью удобных комбинаций клавиш (особенно в консольной версии). В табл. 3.1 приведены некоторые команды, которые вам необходимо знать. Эта таблица ни в коем случае не является исчерпывающей, а полный набор комбинаций можно найти на сайте Vim. Но для начала хватит и этого.

Таблица 3.1. *Важные команды редактора Vim*

Клавиша	Действие
Команды перемещения курсора	
←	Переместить влево
↓	Переместить вниз
↑	Переместить вверх
→	Переместить вправо
w	Переместить до следующего слова
W	Переместить до следующего слова, отделенного пробелом
e	Переместить в конец слова
E	Переместить в конец слова (без учета пунктуации)
b	Переместить до предыдущего слова
B	Переместить до предыдущего слова (без учета пунктуации)
0 (ноль)	Новая строка
^	Первый непустой символ
$	Конец строки
G	Перейти к команде (например, 5G -- перейти к строке 5) *Примечание*: в команде перехода к команде слева приписывается количество повторений. Например, команда 4j перемещает курсор на 4 строки

Таблица 3.1 *(окончание)*

Клавиша	Действие
Режим вставки — вставка/дополнение текста	
i	Переход в режим вставки с позиции курсора
I	Вставить в начале строки
A	Вставить после курсора
a	Вставить в конце строки
o	Добавить пустую строку после текущей (без нажатия клавиши <Return>)
O	Добавить пустую строку перед текущей
Esc	Выход из режима вставки
Режим команд	
:w	Запись (сохранение) файла без выхода
:wq	Запись файла (сохранение) и выход из Vim
:Q	Выход (не срабатывает при наличии изменений)
:Q!	Выход без сохранения изменений
:set	Установка нумерации строк

Запустив Vim, вы также можете указать имя файла, который хотите создать. Если файл уже существует, он загрузится в окно редактора, и вы сможете работать с ним, как вам нужно. Если такого файла не существует, Vim создаст для вас новый пустой файл с указанным именем. Откройте новое окно терминала и в командной строке введите:

```
vim prog3a.s
```

Обратите внимание, что между `vim` и `prog3.s` есть пробел, а также заметьте, что в конце имени `prog3` есть расширение `s` исходного файла. Соглашение гласит, что расширение `s` обозначает исходный файл на языке ассемблера.

После этого окно станет в основном пустым, за исключением столбца из символов тильды `~`, бегущего по левому краю, и имени файла внизу, а также обозначения того, что это новый файл.

Нажмите клавишу <i>. Обратите внимание что в нижнем левом углу экрана появился текст:

```
-INSERT --
```

Это означает, что мы находимся в режиме вставки. Нажмите клавишу <Esc>. Надпись исчезла. Теперь мы находимся в командном режиме Vim.

Нажатие клавиш <i> и <Esc> станет для вас второй натурой. Когда включен режим вставки, вы можете вводить программу на ассемблере и редактировать ее, пока не

надоест. В командном режиме нажатия клавиш интерпретируются как прямые команды для Vim.

Имена файлов программ — в нашем случае `prog3a.s` — станут для вас привычными. Все программы в этой книге будут именоваться по номерам глав. Таким образом, `prog3a.s`[1] означает, что исходный файл программы взят из *главы 3* книги. Буква `a`[1] предполагает, что это 1-я программа в этой главе. Файл с именем `prog4b.s`[1] будет означать, что это 2-я программа из *главы 4*. Мы будем делать так просто для удобства. Но вы можете использовать любое имя, какое захотите.

Вернитесь в режим вставки (клавишей <i>) и обратите внимание на мигающий курсор в верхнем левом углу экрана. Все, что вы начнете вводить, появится там, где находится курсор. Перепишите туда код из *программы 3.1*. Вам нужно ввести только тот текст, который в этом листинге расположен между двумя пунктирными линиями.

> ***Примечание к русскому изданию***
>
> Напомним, что электронный архив с файлами приведенных в книге программ можно загрузить с FTP-сервера издательства «БХВ» по ссылке: **ftp://ftp.bhv.ru/9785977568012.zip** или со страницы книги на сайте **https://bhv.ru/** (см. *приложение 4*).

ПРОГРАММА 3.1. Простой исходный файл

```
      .global _start
start:
      MOV R0, #65
      MOV R7, #1
      SWI 0
```

Обратите внимание, что из этих пяти строк программы первая, третья, четвертая и пятая строки написаны с отступом. Только вторая строка написана без отступа. Величина отступа, который вы добавляете, и даже место их размещения, на самом деле не имеют значения. Но зато они просто упрощают чтение программы и позволяют выделить различные уровни программы.

Чтобы создать отступ, нажмите клавишу <Tab>. Другие клавиши работают так, как и должны. Например, клавиши со стрелками используются для перемещения, а клавиши <Delete> и <Backspace> — для перемещения и редактирования текста. Но между словами `global` и `_start` имеется пробел, и этот пробел важен. Вскоре мы рассмотрим, что конкретно делает этот код.

А пока нажмите клавишу <Esc> и введите:

```
:wq
```

Эта команда сохранит ваш файл и закроет Vim. Теперь вы вернулись в командную строку. Исходный файл готов!

---

[1] Здесь имеется в виду авторская нумерация программ в книге и в электронном архиве (3a, 3b, 4a, 4b). Мы же в книге нумеруем программы более привычной нашему читателю цифровой записью: 3.1, 3.2, 4.1, 4.2 и т. д. (в электронном архиве нумерация остается авторской).

Написанное — исполнить!

Следующим шагом является преобразование исходного файла в исполняемый файл машинного кода. Это делается с помощью двух команд командной строки. В командной строке последовательно введите две команды:

```
as -o prog3a.o prog3a.s
ld -o prog3a prog3a.o
```

Эти две команды сначала собирают, а затем *привязывают* программу на языке ассемблера (о привязывании позже). После этого машинный код готов к выполнению:

```
./<имя файла>
```

Сочетание символов `./` означает «запустить», а имя файла, который нужно запустить, пишется сразу после команды без пробелов. Таким образом, для запуска напишем:

```
./prog3a
```

Когда приглашение появится снова, программа машинного кода будет выполнена. Проще простого!

Выходит, мы только что написали, скомпилировали (собрали и связали) и выполнили программу машинного кода, сделав все основные шаги этого стандартного процесса. Конечно, со временем наши программы будут становиться все сложнее, мы будем шире использовать доступные инструменты, и этот процесс, как мы увидим, станет более сложным.

Ошибки ассемблера

Если в какой-либо момент во время процесса вы получите сообщение об ошибке, или вообще какое-либо сообщение, — тщательно проверьте то, что вы ввели. Сначала внимательно посмотрите на программу на языке ассемблера, затем на команды для сборки и связывания и, наконец, запустите программу. Если произошла ошибка, и вы ее нашли, поздравляем, вы только что отладили свою первую программу на ассемблере.

Если возникает сообщение об ошибке от ассемблера (оно появляется после того, как вы нажали <Return> в конце первой строки), обычно в этом сообщении указывается номер строки, в которой что-то неладно. Даже если вы не знаете, что именно означает полученное сообщение, запишите себе номер строки, а затем откройте исходный файл в Vim. Например, сообщение:

```
prog3a.s:5: Error bad expression
```

говорит об ошибке в строке 5 исходного файла.

Пока ваши файлы еще крошечные, вы можете отсчитать количество строк и найти ту, в которую закралась ошибка. В самом редакторе Vim также есть возможность нумерации строк. В командном режиме Vim введите:

```
:set number
```

Обратите внимание, что в левой части окна появились номера строк. Номера строк не являются частью файла с кодом и отображаются только для справки. На рис. 3.1 показано, как это все это выглядит в редакторе gVim: видны номера строк и то, что Vim работает в режиме вставки. Если вы используете gVim, то в нем различные элементы синтаксиса выделяются разными цветами. Это позволяет легко идентифицировать различные компоненты программы.

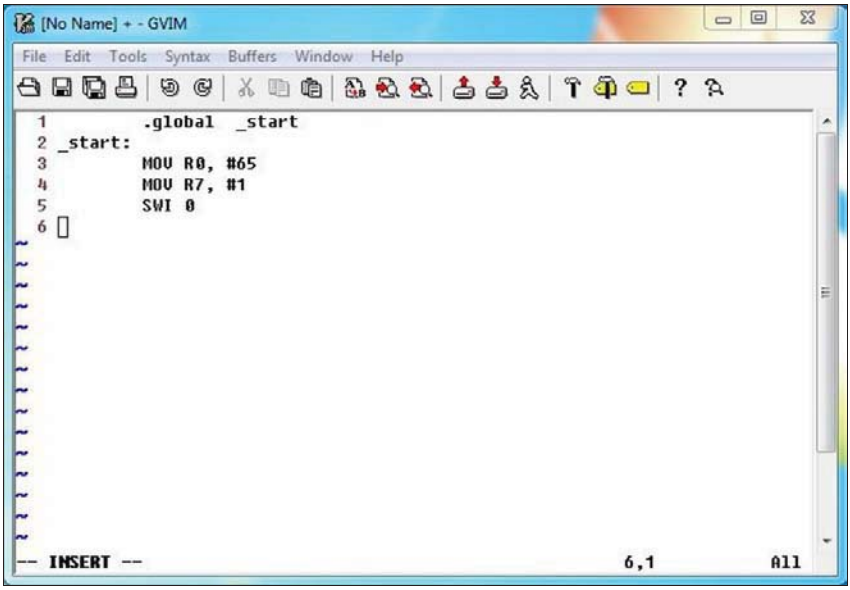

Рис. 3.1. Отображение номеров строк в gVim

Вы можете запустить редактор gVim из командной строки с помощью следующей команды:

```
gvim <имяфайла>
```

Тогда для создания или редактирования файла `prog3a.s` можно выполнить команду:

```
gvim prog3a.s
```

Компоненты

Давайте еще раз взглянем на описанный здесь процесс и попробуем вникнуть в анатомию исходного файла и понять, как все это собралось воедино. Посмотрим еще раз на файл `prog3a.s`. Он состоит всего из пяти строк.

У любого файла с кодом ассемблера должна быть точка начала, и по умолчанию в ассемблере GCC она выглядит вот так:

```
_start:
```

Первая строка этой программы определяет `_start` как глобальное имя, доступное для всей программы. Позже мы увидим, почему важно сделать это имя глобальным.

Вторая строка определяет, где находится `_start:` в программе. Обратите внимание на использование символа : в конце, которое определяет это имя как «метку». Мы определили `_start` как глобальное имя и теперь отметили, где находится сама метка `_start`.

Следующие три строки написаны на мнемонике ассемблера, причем две из них одинаковые: команда `MOV`. Когда в языке ассемблера используется символ решетки, он обозначает непосредственное значение. Другими словами, число после решетки — это именно то значение, которое и будет использоваться. Первая команда перемещает значение `65` в регистр `0`. Буква `R` обозначает *регистр* — специальное место в микросхеме ARM, о котором мы поговорим чуть позже. Во второй строке значение `1` перемещается в `R7`, или в регистр `7`.

Последняя команда: `SWI 0`. Это специальная команда, которая служит для вызова самой операционной системы Raspberry Pi. В нашем случае она задействуется для выхода из программы машинного кода и передачи управления обратно в командную строку (программа в этот момент должна быть запущена).

Также стоит обратить внимание на регистр символов команд языка ассемблера в наших исходных файлах. Я использую прописные буквы для мнемоник и регистров, но подошли бы и строчные буквы, поскольку внутри исходных файлов регистр символов не имеет значения (в отличие от командной строки в консоли, которая чувствительна к регистру), поэтому

```
MOV R0, #65
```

а также

```
mov r0, #65
```

делают одно и то же. В этой книге я буду использовать прописные буквы. Это упрощает чтение команд в тексте книги и позволяет отличить команды от меток, которые будут написаны в нижнем регистре.

Запустите программу еще раз:

```
./prog3a
```

В командной строке введите:

```
echo $?
```

На экран будет выведено следующее:

```
65
```

Вывелось значение, загруженное в `R0`. Попробуйте изменить `65` на другое число — скажем, `49`. Теперь сохраните код, соберите его заново, заново свяжите и запустите. Если вы сейчас наберете:

```
echo $?
```

будет выведено число `49`. У ОС есть определенные способы возврата информации из программ машинного кода, и мы рассмотрим их позже.

Если вы снова посмотрите на код в файле `prog3a.s`, то увидите, что он состоит из двух отдельных частей. Вверху (в начале) приведены некоторые определения,

а в нижней половине — непосредственно команды на языке ассемблера. Исходные файлы на языке ассемблера всегда состоят из последовательности операторов, записанных по одному в каждой строке. Каждый оператор имеет следующий синтаксис (при этом каких-то его частей может и не быть):

```
<обозначение:> <инструкция>          @ комментарий
```

Все три компонента можно вводить в одной строке или в разных строках. Это дело ваше. Но порядок их должен быть именно таким. Например, команда не может располагаться перед меткой (в той же строке).

Компонент «комментарий» для нас в новинку. Когда ассемблер встречает символ `@`, он игнорирует все написанное после него до конца строки. Такую конструкцию можно использовать для добавления в программу пояснений. Например, вернитесь и отредактируйте код в файле `prog3a.s`, набрав:

```
vim prog3a.s
```

В окне редактора отобразится исходный файл. Курсор будет находиться в верхней части файла. Войдите в режим вставки, создайте новую строку и введите в нее следующее:

```
@ prog3a.s - простенький файл ассемблера
```

Строка комментария с символом `@` в начале полностью игнорируется компилятором. Из-за этого размер исходного файла, конечно, увеличится, но это никак не повлияет на работу исполняемого файла.

Комментарии также можно добавлять с помощью символов `/*` и `*/`, обозначая ими начало и конец комментария:

```
/* Этот комментарий ассемблер проигнорирует */
```

Оба метода приемлемы, и вы можете использовать любой на свой вкус.

Чтобы преобразовать исходный файл в исполняемый, нам потребовалось два шага. Первый:

```
as -o prog3a.o prog3a.s
```

Команда `as` в начале вызывает саму программу ассемблера, которая ожидает после этой команды нескольких аргументов, задающих имена файлов, с которыми она будет работать, а также необходимые действия. Первый из них: `o` — сообщает ассемблеру, что мы хотим создать объектный файл по имени `prog3a.o` из исходного файла `prog3a.s`. Вы можете выбрать другое имя, если хотите. Сами имена не обязательно должны совпадать, но, если они одинаковые, проще найти концы. Расширение в любом случае разное!

Второй и последний шаг: «связать» файл объектного файла и преобразовать его в исполняемый файл с помощью команды `ld` следующим образом:

```
ld -o prog3a prog3a.o
```

Это действие — последняя часть процедуры связки, которая заставляет работать машинный код. В ходе нее из объектного файла, созданного в процессе сборки,