КНИЖНАЯ ПОЛКА ИСТОВОГО ИНЖЕНЕРА

Даниэле Лакамера
под науч. ред. А.Ю. Романова

АРХИТЕКТУРА ВСТРАИВАЕМЫХ СИСТЕМ

РАЗРАБОТКА ЗАЩИЩЕННЫХ
И ПОДКЛЮЧЕННЫХ УСТРОЙСТВ
С ПРИМЕРАМИ КОДА НА C

ВШЭ • ВЫСШАЯ ШКОЛА ЭКОНОМИКИ
МИЭМ
ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Даниэле Лакамера

Архитектура встраиваемых систем

Daniele Lacamera

Embedded Systems Architecture

Design and write software for embedded devices to build safe and connected systems

BIRMINGHAM—MUMBAI

Даниэле Лакамера

Архитектура встраиваемых систем

Разработка защищенных и подключенных устройств с примерами кода на С

Москва, 2023

УДК 004.04
ББК 32.371
Л19

Научный редактор

Романов А. Ю. – канд. техн. наук, доцент Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики».

Лакамера Д.

Л19 Архитектура встраиваемых систем / пер. с англ. В. С. Яценкова; под науч. ред. А. Ю. Романова. – М.: ДМК Пресс, 2023. – 332 с.: ил.

ISBN 978-5-93700-206-8

Книга, которую вы держите в руках, продолжает серию «Книжная полка Истового Инженера», которая издается при поддержке компании YADRO. Это издание подготовлено к публикации Московским институтом электроники и математики им. А. Н. Тихонова НИУ ВШЭ совместно с «ДМК Пресс».

В книге описываются принципы работы и взаимодействия различных компонентов в реальных системах. Представлен общий обзор процесса разработки встраиваемых систем; показано, как настроить среду разработки, рассматриваются структура, механизмы загрузки и управление памятью встраиваемой системы. Вы изучите программный интерфейс и драйверы устройств, узнаете, как устанавливать связь через TCP/IP, как повысить безопасность устройств интернета вещей. Наконец, вы на практике познакомитесь с многопоточными операционными системами, самостоятельно разработав планировщик, и научитесь использовать механизмы доверенного выполнения с аппаратной поддержкой.

Издание предназначено для программистов и инженеров, желающих освоить область разработки встраиваемых систем.

УДК 004.04
ББК 32.371

ISBN 978-1-80323-954-5 (англ.)

ISBN 978-5-93700-206-8 (рус.)

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Благодарности

Здесь будут фамилии тех, кто помогал изданию этой книги, прислав в издательство найденные ошибки или ссылку на подозрительные материалы.

Автор

Даниэле Лакамера – технолог и разработчик программного обеспечения. Он является экспертом в области операционных систем и TCP/IP, имеет более 20 академических публикаций по оптимизации транспортных протоколов. Начал свою карьеру в качестве разработчика ядра Linux и свой первый вклад внес в версию Linux 2.6.

С 2012 года работает над архитектурами на основе микроконтроллеров, уделяя особое внимание проектированию, разработке и интеграции программного обеспечения для встраиваемых систем. Является активным участником многих проектов свободного программного обеспечения и соавтором стека TCP/IP и операционной системы (ОС) POSIX для устройств Cortex-M, которые распространяются под лицензией GPL. В настоящее время его деятельность сосредоточена на безопасности IoT, криптографии, безопасной загрузке и пользовательских транспортных протоколах.

Рецензент

Марко Оливерио получил докторскую степень в Университете Калабрии за диссертацию по защите ОС от атак по побочным каналам и аппаратных атак. Некоторые из его научных публикаций были представлены на значимых конференциях, посвященных ОС. После защиты докторской диссертации он начал работать разработчиком микропрограмм и встраиваемых систем, участвуя в нескольких проектах с открытым исходным кодом.

Научный редактор русского перевода

Романов Александр Юрьевич (https://www.hse.ru/staff/a.romanov) –научный редактор русского перевода данной книги, доцент Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики» (МИЭМ НИУ ВШЭ). С 2014 г. работает в МИЭМ НИУ ВШЭ, где возглавляет лабораторию САПР (https://miem.hse.ru/edu/ce/cadsystem), специализирующуюся на проектной деятельности, а также разработке цифровых систем на ПЛИС/микроконтроллерах, робототехнических комплексов, аппаратных реализаций систем искусственного интеллекта, многопроцессорных систем, систем удаленного доступа к лабораторному оборудованию и т. д. В 2015 г. защитил диссертацию в Институте проблем проектирования в микроэлектронике РАН (г. Зеленоград), является автором более 200 научных статей, патентов и книг. Преподает встраиваемые системы с 2009 г.

Предисловие

За последние два десятилетия встраиваемые микроконтроллерные системы получили широкое распространение благодаря достижениям производителей микросхем и разработчиков микроэлектроники, стремящихся к увеличению вычислительной мощности и уменьшение размера логических узлов микропроцессоров и периферийных устройств.

Проектирование, реализация и интеграция программных компонентов для этих систем в большинстве случаев требуют прямого обращения к аппаратным функциям, когда приложения выполняются в одном потоке и нет операционной системы, обеспечивающей абстракции для доступа к функциям процессорного ядра и внешним периферийным устройствам. По этой причине разработка встраиваемых систем считается отдельной областью в индустрии разработки программного обеспечения, в которой подход разработчика и рабочий процесс должны быть соответствующим образом адаптированы к особенностям среды выполнения.

Эта книга кратко объясняет особенности аппаратной архитектуры типичной встраиваемой системы, знакомит с инструментами и методологиями, необходимыми для начала разработки архитектуры целевой системы, а затем проводит читателей через взаимодействие с системными функциями и периферийными устройствами. Некоторые области, такие как энергоэффективность и подключение к коммуникационным сетям, рассматриваются подробнее, чтобы дать более полное представление о методах, используемых для проектирования маломощных и подключенных систем. Далее в книге рассматривается более сложный проект, включающий упрощенную операционную систему реального времени, которая строится шаг за шагом, начиная с реализации отдельных системных компонентов. Наконец, во втором издании добавлен подробный анализ реализации TrustZone-M – технологии TEE, представленной ARM в новейшем семействе встраиваемых микроконтроллеров.

В этой книге основной акцент сделан на конкретных механизмах защиты и безопасности, включая практические примеры использования технологий, призванных повысить устойчивости системы к ошибкам программирования в коде приложения или даже к злонамеренным попыткам нарушить ее целостность.

Для кого эта книга

Если вы программист или инженер, который хочет освоить область разработки и программирования встраиваемых систем, эта книга для вас. Вы также найдете эту книгу полезной, если вы неопытный или начинающий про-

граммист встраиваемых систем, желающий расширить свои знания. Более опытным разработчикам встраиваемого программного обеспечения наша книга может пригодиться для обновления знаний о драйверах устройств, безопасности памяти, безопасной передаче данных, разделении привилегий и безопасных областях выполнения кода.

Краткое содержание книги

Глава 1 представляет собой введение во встраиваемые системы на базе микроконтроллеров. В ней также раскрывается тематический охват книги, от широкого определения встраиваемых систем до более конкретной архитектуры, на которой базируются все практические примеры – 32-битные микроконтроллеры с отображением физической памяти.

Глава 2 описывает используемые инструменты и рабочий процесс разработки. Это введение в наборы инструментов, отладчики и эмуляторы, которые можно использовать для создания кода в двоичном формате для последующей загрузки и запуска на целевой платформе.

Глава 3 посвящена стратегиям и методологиям командной разработки и тестирования. Эта глава содержит описание процессов, которые обычно применяются при разработке и тестировании программного обеспечения для встраиваемых систем.

В главе 4 детально рассмотрен этап загрузки (запуска, boot-up) встроенной системы, анализируются этапы загрузки и загрузчики. Она содержит подробное описание кода запуска и механизмов, используемых для разделения программного обеспечения на несколько этапов загрузки.

В главе 5 описаны некоторые оптимальные стратегии управления памятью. Показаны наиболее распространенные ловушки и дается объяснение, как избежать ошибок выделения памяти, которые могут привести к непредсказуемому поведению или полному выходу системы из строя.

В главе 6 рассматривается доступ к выводам GPIO и другим встроенным периферийным устройствам общего назначения. Это первое взаимодействие целевой платформы с внешним миром, использующее электрические сигналы для выполнения простых операций ввода/вывода.

Глава 7 проведет вас через интеграцию контроллеров последовательной шины (UART, SPI и I2C). Детальный анализ наиболее распространенных протоколов связи по шине сопровождается подробными примерами кода, необходимого для взаимодействия с приемопередатчиками, обычно доступными во встроенных системах.

В главе 8 рассматриваются методы снижения энергопотребления в энергосберегающих системах. Проектирование встраиваемых систем с низким и сверхнизким энергопотреблением требует выполнения определенных шагов для достижения требуемого уровня экономичности.

В главе 9 представлены протоколы и интерфейсы, необходимые для построения распределенных и подключенных систем. Системы IoT должны взаимодействовать с удаленными конечными точками, применяя стандартные сетевые протоколы, реализованные с использованием сторонних биб-

лиотек. Особое внимание уделяется защите связи между конечными точками с помощью защищенных сокетов.

В главе 10 объясняется принцип функционирования многозадачной операционной системы через реализацию планировщика задач в реальном времени. В этой главе предлагаются три подхода к реализации операционных систем для микроконтроллеров с нуля с использованием различных планировщиков.

В главе 11 описаны механизмы TEE, обычно доступные во встраиваемых системах, и приводится пример запуска кода в защищенных и незащищенных областях с использованием ARM TrustZone-M. На современных микроконтроллерах TEE предоставляет возможность защитить определенные области памяти или периферийных устройств, ограничив их доступ из незащищенной области выполнения.

Как получить максимальную пользу от этой книги

Ожидается, что вы владеете языком C и понимаете, как работают компьютерные системы. Для запуска кода практических примеров требуется рабочий компьютер с установленной платформой GNU/Linux. Для полного понимания реализованных механизмов иногда потребуется пошаговое изучение примеров кода. Вам предлагается изменять, улучшать и повторно использовать предоставленные примеры, применяя полученные знания.

ПО и оборудование, рассмотренные в этой книге	**Необходимая операционная система**
Набор STM32F4DISCOVERY	
Плата STM32L5 Nucleo-144	
Инструменты разработки GNU ARM	Windows, macOS, Linux
GNU Make	
OpenOCD	

Дополнительные инструкции по использованию необходимых инструментов разработки приведены в главе 2.

Если вы используете цифровую версию этой книги, вы можете скачать файловый архив книги с сайта издательства. Это поможет вам избежать возможных ошибок, связанных с копированием и вставкой кода.

Используемые соглашения

В этой книге используется ряд соглашений по оформлению текста.

`Код в тексте`: обозначает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, пользовательский ввод и дескрипторы. Пример: «При вызове из

командной строки должен быть указан один файл конфигурации с несколькими платформами и конфигурациями платы разработки, расположенными в каталоге /scripts».

Блок кода оформлен следующим образом:

```
/* Переход к небезопасной области кода */
asm volatile("mov r12, %0" ::"r"
     ((uint32_t)app_entry - 1));
asm volatile("blxns r12" );
```

Когда необходимо привлечь внимание читателя к определенной части блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
Secure Area 1:
  SECWM1_PSTRT : 0x0 (0x8000000)
  SECWM1_PEND : 0x39 (0x8039000)
```

Ввод или вывод командной строки записывается следующим образом:

```
$ renode /opt/renode/scripts/single-node/stm32f4_discovery.resc
```

Команды для консоли отладчика записываются следующим образом:

```
> add-symbol-file app.elf 0x1000
> bt full
```

Советы и важные примечания

Оформлены, как показано здесь.

Предисловие от научного редактора русского перевода

Дорогие друзья!
Книга, которую вы держите в руках, продолжает серию «Книжная полка истового инженера», которую выпускает издательство «ДМК Пресс» совместно с Московским институтом электроники и математики им. А. Н. Тихонова НИУ ВШЭ при поддержке группы компаний YADRO (yadro.com).

Если вы интересуетесь цифровой электроникой, разработкой на ПЛИС, проектированием на языках описания аппаратуры Verilog или VHDL, то вы, скорее всего, уже знакомы с книгами по цифровому синтезу, такими как: Д. Харрис и С. Л. Харрис «Цифровая схемотехника и архитектура компьютера», «Цифровая схемотехника и архитектура компьютера: RISC-V», «Цифровая схемотехника и архитектура компьютера. Дополнение по архитектуре ARM»; «Цифровой синтез: практический курс» (под ред. А. Ю. Романова и Ю. В. Панчула), Ф. Бруно «Программирование FPGA для начинающих» и др. (воспользуйтесь QR-кодами, чтобы узнать об упомянутых книгах).

Эта книга несколько отличается от упомянутых выше и посвящена программированию на классическом языке C++, продолжая тренд, заданный еще одной книгой из этой серии – «C++20 в деталях» (автор: Р. Гримм). Только, если в той книге анализируются последние новшества и высокоуровневые возможности C++, то в этой, наоборот, рассмотрены особенности программирования на самом низком уровне встраиваемых систем.

Прочитав эту книгу вы узнаете, как устроены встраиваемые системы, как организовано программирование, верификация и отладка для таких систем, как происходит загрузка программного кода, управление памятью, взаимодействие с периферией, как устроены стандартные шинные интерфейсы, как управлять питанием и энергосбережением, как устроена многопоточность на уровне операционной системы, как реализовано управление доверен-

ной средой выполнения, и, наконец, как строятся распределенные системы и системы Интернета вещей. То есть все то, что еще называют системным программированием, когда разработчик своей программой взаимодействует с устройством непосредственно без абстрагирования и безопасного кода.

Интересно и подробно излагаемый материал дополнен хорошими и рабочими примерами программного кода, доступными для скачивания из репозитория.

Эта книга будет интересна тем, кому уже надоели проекты на Arduino и хочется перейти к чему-то более мощному, как микропроцессоры от STM, но нет желания отдавать львиную долю производительности на надстройки в виде полноценной операционной системы реального времени на Raspberry PI или Jetson Nano. Эта книга для тех, кто хочет понять, как устроена операционная система изнутри, как работают привычные механизмы распараллеливания задач и разделение памяти. Она для тех, кто хочет разрабатывать компактный, энергоэффективный, производительный и безопасный код на самом низком уровне.

С другой стороны, эта книга будет интересна и программистам кода, выполняемого на высоком уровне. Она позволит лучше понять то системное программное и аппаратное окружение, в котором выполняется высокоуровневая программа, как устроены драйвера устройств и системные библиотеки, и расширить свой кругозор.

Таким образом, аудитория этой книги: широкий круг читателей от студентов технических вузов, до разработчиков аппаратуры и программистов различных профилей. Она требует наличия базовых знаний синтаксиса, основных концепций и принципов программирования на C++. Также для лучшего освоения материала и апробации приведенных примеров может потребоваться отладочная плата с STM32.

В добрый путь!

Александр Юрьевич Романов,
к. т. н., доцент ДКИ МИЭМ НИУ ВШЭ,
преподаватель курсов «Проектирование систем на кристалле»,
«Системное проектирование цифровых устройств»
и «Системы искусственного интеллекта»
г. Москва, Россия

Часть I

ВВЕДЕНИЕ В РАЗРАБОТКУ ВСТРАИВАЕМЫХ СИСТЕМ

В первой главе этой книги представлен общий обзор встраиваемых систем и показано, чем разработка таких систем отличается от других направлений инженерной деятельности, с которыми вы можете быть знакомы. Вторая глава помогает превратить рабочий компьютер инженера в настоящую лабораторию для разработки аппаратного и программного обеспечения и оптимизировать действия, необходимые для создания, тестирования, отладки и развертывания встроенного программного обеспечения.

Первая часть состоит из следующих глав:

- главы 1 «Встроенные системы с практической точки зрения»;
- главы 2 «Рабочая среда и оптимизация рабочего процесса».

Глава 1

Встраиваемые системы с практической точки зрения

Проектирование и разработка программного обеспечения для встраиваемых систем сопряжены с иным набором задач, чем традиционная разработка приложений высокого уровня.

Первая глава знакомит читателей с основными понятиями, компонентами и платформами, которые будут встречаться на протяжении всей книги.

В этой главе обсуждаются следующие темы:

- определение предметной области;
- универсальный ввод/вывод;
- интерфейсы и периферия;
- подключенные системы;
- введение в механизмы изоляции;
- эталонная платформа.

1.1. Определение предметной области

Встраиваемые системы – это вычислительные устройства, которые выполняют определенные задачи без прямого или постоянного взаимодействия с пользователем. Из-за разнообразия рынков и технологий они имеют разные формы и размеры, но преимущественно это устройства небольшого размера с ограниченным количеством ресурсов.

В этой книге основные понятия встраиваемых систем будут проанализированы с точки зрения разработки программных компонентов, взаимодействующих с ресурсами и периферийными узлами устройства. Начнем с определения области применения методов и архитектурных шаблонов, описанных в нашей книге, в рамках более широкого определения встраиваемых систем.

1.1.1. Встраиваемые Linux-системы

Значительную часть рынка встраиваемых систем занимают устройства, аппаратных ресурсов которых достаточно для запуска разновидностей операционной системы GNU/Linux. Рассмотрение таких систем выходит за рамки этой книги, поскольку их разработка в значительной степени опирается на различные стратегии комбинирования и интеграции компонентов. Типичная аппаратная платформа, на которой может работать система на основе ядра Linux, оснащена достаточно большим объемом оперативной памяти (до нескольких гигабайт) и достаточным объемом энергонезависимой памяти для хранения всех программных компонентов, предоставляемых в дистрибутиве GNU/Linux.

Кроме того чтобы менеджер памяти Linux предоставлял отдельные виртуальные адресные пространства каждому процессу в системе, устройство должно быть оснащено *блоком управления памятью* (memory management unit, MMU) – аппаратным компонентом, помогающим операционной системе (ОС) преобразовывать физические адреса в виртуальные, и наоборот, во время выполнения прикладной программы.

Устройства с такими характеристиками нужны далеко не всегда, поэтому для снижения производственных затрат на практике стараются использовать системы с более простой конструкцией.

Производители оборудования и разработчики микросхем постоянно ищут новые методы повышения производительности систем на основе микроконтроллеров. За последнее десятилетие появились новые поколения платформ, позволяющие снизить стоимость оборудования, сложность встроенного ПО, размер и энергопотребление, сохраняя при этом набор функций, наиболее интересных для рынка встраиваемых систем.

В некоторых случаях встраиваемые вычислительные устройства должны выполнять ряд задач в течение короткого, измеримого и предсказуемого промежутка времени. Такие устройства представляют собой *системы реального времени* (real-time system, RTS), принципиально отличающиеся от привычных нам многозадачных систем, таких как настольные компьютеры, серверы и мобильные телефоны.

Обработка в реальном времени – это цель, которую очень трудно, если вообще возможно, достичь на встраиваемых платформах Linux. Ядро Linux не предназначено для работы в жестком режиме реального времени, и даже если применяются патчи планировщика ядра, помогающие удовлетворить требования к работе в реальном времени, результаты несопоставимы со специализированными системами.

Встраиваемые устройства с батарейным и извлекаемым из среды питанием[1] по определению должны обладать низким энергопотреблением и энергосберегающими технологиями беспроводной связи. Большое количество ресурсов и повышенная аппаратная сложность систем на базе Linux не всегда

---

[1] Energy-harvesting devices – устройства со сверхнизким энергопотреблением, способные извлекать и накапливать электрическую энергию из внешних источников (свет, тепло, электромагнитные поля).

позволяют достичь необходимого уровня энергопотребления по сравнению с более простыми специализированными устройствами.

В этой книге рассматриваются встраиваемые системы на основе 32-разрядных микроконтроллеров, которые способны выполнять однопоточные приложения без ОС (bare-metal, на «голом железе»), а также поддерживают минимальные ОС реального времени. Такие системы широко применяются в промышленном производстве изделий, предназначенных для конкретных целей. В последнее время они все чаще служат аппаратной основой для более общих, многоцелевых платформ разработки.

1.1.2. 8-разрядные микроконтроллеры

В прошлом на рынке встраиваемых систем доминировали 8-битные микроконтроллеры. Их несложная конструкция позволяет разрабатывать небольшие приложения, которые могут выполнять набор предопределенных задач. Но они слишком просты и обладают небольшим количеством ресурсов, недостаточным для реализации даже примитивной операционной системы. Вдобавок характеристики 32-разрядных микроконтроллеров значительно эволюционировали, и теперь они охватывают все варианты использования 8-разрядных микроконтроллеров в том же диапазоне цены, размеров и энергопотребления.

В настоящее время 8-разрядные микроконтроллеры в основном занимают нишу на рынке обучающих платформ, предназначенных для ознакомления любителей и новичков с основами программирования для электронных устройств. В этой книге мы не рассматриваем 8-разрядные платформы, поскольку им не хватает характеристик, позволяющих использовать системное программирование, многопоточность и расширенные функции для создания профессиональных встраиваемых систем.

В контексте данной книги термин «встраиваемые системы» используется для обозначения класса устройств, которые основаны на микроконтроллерах, обладающих ограниченными ресурсами, но позволяют создавать системы реального времени с помощью функций, предоставляемых аппаратной архитектурой и системным программированием.

1.1.3. Аппаратная архитектура

Архитектура встраиваемой системы сосредоточена вокруг ее *микроконтроллера*, также иногда называемого *блоком микроконтроллера* (microcontroller unit, MCU). Обычно это одна интегральная схема, содержащая процессор, ОЗУ, флеш-память, последовательные приемники и передатчики и другие базовые компоненты. Рынок предлагает множество различных вариантов архитектур микроконтроллеров, поставщиков, ценовых диапазонов, функций и интегрированных ресурсов. Обычно стараются разрабатывать недорогие автономные системы с низким потреблением энергии и ресурсов на

одной интегральной схеме. Поэтому их часто называют *системой на чипе* (System-on-Chip, SoC), или *однокристалльной системой*.

Из-за разнообразия процессоров, памяти и интерфейсов, которые можно интегрировать в чип, не существует единой эталонной архитектуры микроконтроллеров. Тем не менее некоторые архитектурные компоненты являются общими для широкого спектра моделей и брендов и даже для различных архитектур процессоров.

Одни микроконтроллеры предназначены для специализированных приложений и оснащены различными интерфейсами для связи с периферийными устройствами и внешним миром. Другие ориентированы на решения с минимальными затратами на оборудование или с жестко ограниченным потреблением энергии.

Тем не менее практически каждый современный микроконтроллер имеет следующий минимальный набор компонентов:

- микропроцессор;
- оперативная память (ОЗУ);
- флеш-память;
- последовательные приемопередатчики.

Кроме того, все больше устройств нуждаются во встроенном доступе к сети для связи с другими устройствами и шлюзами. Некоторые микроконтроллеры поддерживают либо хорошо зарекомендовавшие себя стандарты, такие как интерфейсы Ethernet или Wi-Fi, либо специальные протоколы, разработанные в соответствии с требованиями к встраиваемым системам, такие как радиоинтерфейсы с частотой до 1 ГГц или шина *локальной сети контроллеров* (controller area network, CAN), и частично или полностью реализованные в рамках интегральной схемы микроконтроллера.

Все компоненты должны иметь общую шину с процессором, который отвечает за координацию работы периферийной логики. ОЗУ, флеш-память и управляющие регистры приемопередатчиков отображаются в одном и том же физическом адресном пространстве (рис. 1.1).

Адреса, по которым отображаются ОЗУ и флеш-память, зависят от конкретной модели микроконтроллера и обычно указаны в техническом описании. Микроконтроллер выполняет код на своем родном машинном языке – последовательности инструкций, переданных в виде двоичного исполняемого файла. Формат команд и файла зависит от архитектуры микроконтроллера. Исполняемый файл обычно получается в результате компиляции и сборки исходного кода программы с последующим преобразованием выходного файла компилятора в машинный код целевого микропроцессора.

Часть процессора предназначена для выполнения инструкций, которые хранятся в двоичном формате непосредственно в ОЗУ, а также во внутренней флеш-памяти. Обычно выполнение кода начинается с нулевого адреса в памяти или со специального адреса, указанного в техническом описании микроконтроллера. Микропроцессор быстрее всего извлекает и выполняет код из ОЗУ, но постоянная прошивка хранится во флеш-памяти, емкость которой обычно больше, чем ОЗУ почти у всех микроконтроллеров, и позволяет сохранять данные при выключении питания и перезагрузках.

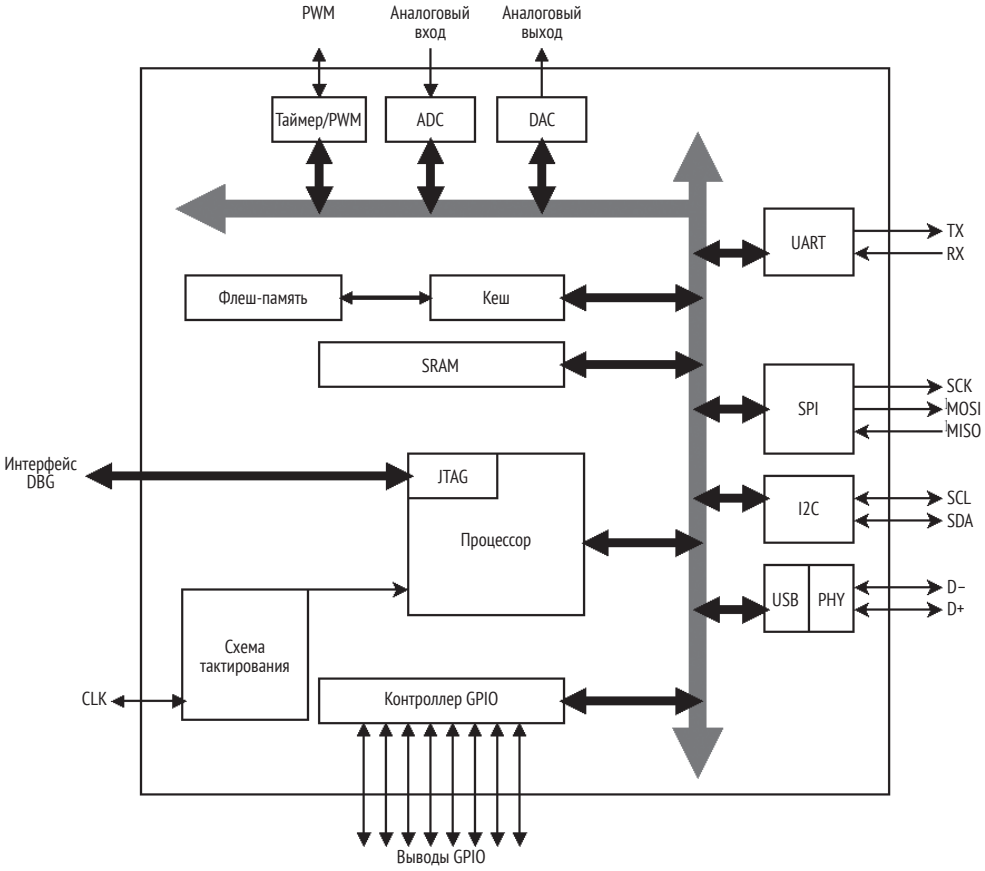

Рис. 1.1 ❖ Упрощенная блок-схема компонентов универсального микроконтроллера

Для компиляции исполняемого файла и загрузки его во флеш-память микроконтроллера требуется компьютер (хост-машина), оснащенный необходимыми аппаратными и программными средствами. Для создания правильного машинного кода из текста программы на языке высокого уровня необходимо указать компилятору характеристики и требования целевого микроконтроллера. По многим уважительным причинам наиболее популярным языком программирования для разработки программного обеспечения встраиваемых систем является C, хотя это не единственный доступный вариант. Для программирования встраиваемых систем можно использовать языки и более высокого уровня, такие как Rust или C++.

Примечание

В этой книге основное внимание будет уделено программированию на C, потому что он менее абстрактный, чем любой другой язык высокого уровня, что упрощает понимание поведения базового оборудования при просмотре кода.

Все современные платформы встроенных систем также имеют по крайней мере один механизм (например, JTAG) для отладки и загрузки программного обеспечения во флеш-память. При доступе к интерфейсу отладки с хост-машины отладчик может взаимодействовать с блоком точек останова (breakpoint) в процессоре, прерывая и возобновляя выполнение, а также выполнять чтение и запись любого адреса в памяти.

Значительную часть программирования встраиваемых систем составляет реализация обмена данными с периферийными устройствами через собственные интерфейсы микроконтроллера. Для разработки встраиваемого программного обеспечения требуются базовые знания в области электроники, способность разбираться в схемах и технических описаниях, а также уверенное обращение с измерительными приборами, такими как логические анализаторы или осциллографы.

1.1.4. Типичные затруднения

Разработка встраиваемых систем требует постоянного внимания к спецификациям и аппаратным ограничениям. Это постоянный поиск компромисса между наиболее эффективным способом выполнения конкретных задач и использованием очень ограниченных ресурсов. Разработчики встраиваемых систем постоянно встречаются с типичными затруднениями, которые редко встречаются в других средах. Вот несколько примеров таких затруднений:

- во флеш-памяти не хватает места для реализации новой функции;
- недостаточно оперативной памяти для хранения сложных структур или копирования больших буферов данных;
- не хватает быстродействия процессора для своевременного выполнения всех необходимых вычислений и обработки данных;
- устройство с питанием от электрической батареи потребляет слишком много электроэнергии.

Кроме того, операционные системы персональных компьютеров и смартфонов широко используют MMU – компонент процессора, который позволяет выполнять преобразования между физическими и виртуальными адресами на аппаратном уровне.

MMU является необходимой абстракцией для реализации разделения адресного пространства между задачами, а также между задачами и самим ядром. Микроконтроллеры не имеют MMU, и обычно у них нет достаточного объема энергонезависимой памяти для хранения ядер, приложений и библиотек. По этой причине встраиваемые системы часто работают в рамках одной задачи, при этом основной цикл программы выполняет всю обработку данных и обмен данными в определенном порядке. Некоторые устройства могут запускать встроенные ОС, которые намного проще, чем их аналоги для компьютеров или смартфонов.

Разработчики компьютерных приложений часто рассматривают базовую систему просто как готовую среду выполнения с определенными свойствами,

в то время как при разработке встраиваемых систем нередко приходится создавать с нуля все компоненты, от процедуры загрузки до логики приложения. Во встраиваемой среде различные программные компоненты теснее связаны друг с другом из-за отсутствия более сложных абстракций, таких как разделение памяти между процессами и ядром ОС.

Разработчики, которые впервые обращаются к встраиваемым системам, обычно обнаруживают, что отладка некоторых систем представляет собой более сложный процесс, чем просто запуск программы и наблюдение за результатами. Это особенно верно для систем, которые имеют ограниченные возможности взаимодействия с человеком или полностью их лишены.

Для успешного программирования встраиваемых систем нужен правильно организованный рабочий процесс, включающий в себя четко определенные тестовые примеры, список ключевых показателей эффективности системы, полученных в результате анализа спецификаций для выявления возможностей компромиссов, инструменты и процедуры для выполнения всех необходимых измерений и хорошо отлаженный этап прототипирования.

В этом контексте особого внимания заслуживает безопасность. Как обычно, при разработке кода, выполняемого на системном уровне, разумно помнить о глобальных последствиях возможных ошибок. Код большинства встроенных приложений выполняется с расширенными привилегиями, и неправильное поведение одной задачи может повлиять на стабильность и целостность всей микропрограммы. Некоторые платформы предлагают специальные механизмы защиты памяти и встроенного разделения привилегий, которые полезны для построения отказоустойчивых систем даже при отсутствии полноценной ОС, основанной на разделении адресных пространств процессов.

1.1.5. Многопоточность

Одним из преимуществ использования микроконтроллеров, предназначенных для построения встраиваемых систем, является возможность запуска логически разделенных задач в отдельных исполнительных устройствах за счет разделения ресурсов по времени.

Самая распространенная разновидность встраиваемых программ представляет собой последовательно выполняемый код на основе главного цикла, где модули и компоненты образуют интерфейсы обратного вызова. Но современные микроконтроллеры предлагают разработчикам систем специальные средства для создания многозадачной среды и запуска логически разделенных приложений.

Эти средства особенно удобны при разработке более сложных систем реального времени; знакомство с ними помогает понять модель безопасности, основанную на изоляции процессов и сегментации памяти.

1.2. ОЗУ

«640 Кбайт памяти должно хватить всем».

– Билл Гейтс
(основатель и бывший директор Microsoft)

За последние три десятилетия это известное утверждение неоднократно цитировали, чтобы подчеркнуть технологический прогресс и выдающиеся достижения индустрии ПК. Сегодня многие программисты воспринимают эту фразу исключительно как шутку, но именно на такие объемы памяти приходится ориентироваться разработчикам встраиваемых систем спустя более 30 лет после первоначального выпуска MS-DOS.

Хотя сегодня большинство встраиваемых систем оснащены более объемной памятью, особенно благодаря наличию интерфейсов для подключения внешней памяти DRAM, самые простые устройства, которые можно запрограммировать на C, имеют всего 4 Кбайта ОЗУ для реализации всей системной логики. При проектировании встраиваемой системы необходимо тщательно оценивать объем памяти, потенциально необходимой для операций, выполняемых системой, и выделяемой под буферы для связи с периферийными модулями и внешними устройствами.

Модель памяти встраиваемой системы проще, чем у ПК и мобильных устройств. Доступ к памяти обычно осуществляется на физическом уровне, поэтому все указатели в коде сообщают физическое расположение данных, на которые они указывают. В современных ПК за преобразование физических адресов в виртуальное представление запущенных задач отвечает операционная система.

Преимущество прямого адресного доступа к физической памяти в системах без MMU заключается в более простой трансляции адресов при кодировании и отладке. С другой стороны, некоторые базовые функции, реализованные в любой современной ОС, такие как своп (process swapping) и динамическое изменение размера адресных пространств, во встраиваемых системах становятся громоздкими, а иногда и невозможными.

Встраиваемые системы требуют очень кропотливой работы с памятью. Программисты, которые привыкли разрабатывать код для персональных компьютеров, ожидают, что среда выполнения обеспечит определенный уровень защиты. Виртуальное адресное пространство не допускает перекрытия областей памяти, а ОС легко обнаруживает несанкционированный доступ к памяти и нарушения сегментации, поэтому оперативно завершает процесс и предотвращает компрометацию всей системы.

Во встраиваемых системах, особенно при разработке кода для «голого железа», границы каждого пула адресов необходимо проверять вручную. Случайное изменение нескольких битов в неправильной области памяти или даже простое обращение к другой области памяти может привести к фаталь-

ной и необратимой ошибке. Система может зависнуть или, в худшем случае, начать вести себя непредсказуемо. При работе с памятью во встраиваемых системах необходимо соблюдать требования безопасности, особенно при работе с жизненно важными устройствами. Выявить ошибки работы с памятью в уже готовой программе обычно намного труднее, чем заставить себя разрабатывать безопасный код и защищать систему от ошибок программиста.

Правильные приемы работы с памятью будут показаны в главе 5.

1.3. Флеш-память

Серверы и персональные компьютеры хранят исполняемые приложения и библиотеки на внешних запоминающих устройствах. Перед выполнением код приложений считывают, преобразуют, возможно, распаковывают и сохраняют в ОЗУ до начала выполнения.

Микропрограмма (прошивка, firmware) встраиваемого устройства представляет собой, как правило, единственный файл двоичного кода, содержащий все программные компоненты и сохраняемый во внутренней флеш-памяти микроконтроллера. Поскольку флеш-память напрямую отображается на фиксированный сегмент адресов в пространстве памяти, процессор может последовательно извлекать и выполнять из нее отдельные инструкции без промежуточных шагов. Этот механизм называется *выполнением на месте* (execute in place, XIP).

Все неизменяемые разделы прошивки не требуют загрузки в оперативную память и доступны через прямую адресацию в пространстве памяти. Сюда входят не только исполняемые инструкции, но и все переменные, помеченные компилятором как константы. С другой стороны, при использовании XIP требуется несколько дополнительных шагов при подготовке образа микропрограммы для сохранения во флеш-памяти, а компоновщик должен быть проинструктирован о различных отображаемых в памяти областях целевого микроконтроллера.

Внутренняя флеш-память, отображаемая в адресное пространство микроконтроллера, недоступна для произвольной записи. Изменение содержимого внутренней флеш-памяти возможно только с использованием блочного доступа из-за аппаратных особенностей устройств флеш-памяти. Чтобы изменить значение одного байта во флеш-памяти, весь блок, содержащий этот байт, необходимо стереть и записать заново. Механизм доступа к блочной флеш-памяти для записи известен как *программирование в приложении* (in-application programming, IAP). Некоторые реализации файловой системы абстрагируют операции записи на блочном флеш-устройстве, создавая временную копию блока, в котором выполняется операция записи.

При выборе компонентов для устройства на базе микроконтроллера очень важно, чтобы размер флеш-памяти соответствовал пространству, занимаемому микропрограммой. Флеш-память остается одним из самых дорогих компонентов микроконтроллера, поэтому для крупномасштабного производства стараются выбирать чипы с памятью минимального размера, в ко-

торую помещается прошивка. Разработка программного обеспечения с жесткими ограничениями на размер машинного кода в настоящее время редко встречается в других областях, но при попытке втиснуть несколько функций в небольшое хранилище без этого не обойтись. Для некоторых архитектур существуют оптимизирующие компиляторы, способствующие уменьшению размера кода при сборке и компоновке встроенного ПО.

Доступ к дополнительной энергонезависимой памяти, находящейся за пределами микросхемы микроконтроллера, обычно можно получить с помощью специальных интерфейсов, таких как *последовательный периферийный интерфейс* (serial peripheral interface, SPI). Технология изготовления внутренней и внешней флеш-памяти различается. Хотя внешняя флеш-память обычно более плотная и менее дорогая, она не допускает прямого отображения памяти в физическом адресном пространстве, что делает ее непригодной для хранения образов микропрограмм. Это связано с тем, что невозможно последовательно выполнить код, извлекающий инструкции, если не использовать механизм загрузки исполняемых символов в ОЗУ – доступ для чтения на устройствах такого типа выполняется по одному блоку за раз. С другой стороны, доступ для записи может быть быстрее по сравнению с IAP, что делает внешние устройства энергонезависимой памяти идеальным местом для длительного хранения данных, которые получаются во время выполнения микропрограмм.

1.4. Универсальный ввод/вывод (GPIO)

Самая главная функция, которую можно реализовать с помощью любого микроконтроллера и благодаря которой они приобрели такую популярность, – это возможность управлять сигналами на заданных выводах интегральной схемы. Микроконтроллер может управлять *цифровым выходом*, устанавливая на нем логический уровень 0 или 1. Аналогичным образом вывод микросхемы можно использовать для считывания логического уровня, когда он настроен как *цифровой вход*. Программа считывает цифровое значение 1, когда приложенное к нему напряжение превышает определенный порог, и 0 в остальных случаях.

1.4.1. АЦП и ЦАП

Некоторые чипы имеют встроенные модули *аналого-цифрового преобразователя* (АЦП), которые способны определять напряжение, подаваемое на вывод, и делать его *выборку* (получать двоичное значение измеряемой величины). АЦП часто используют для измерения мгновенных значений переменного напряжения, вырабатываемого каким-либо внешним устройством. Встроенное программное обеспечение считывает напряжение на *аналоговом входе* микроконтроллера с точностью, зависящей от разрядности АЦП.

Модуль *цифроаналогового преобразователя* (ЦАП) выполняет противоположную работу, преобразуя значение в регистре микроконтроллера в соответствующее напряжение на *аналоговом выходе* чипа.

1.4.2. Таймеры и ШИМ

Микроконтроллеры могут использовать различные способы измерения времени. Обычно они имеют как минимум один интерфейс, основанный на таймере обратного отсчета, который может запускать прерывание и автоматически сбрасываться по истечении заданного периода времени.

Выводы GPIO, сконфигурированные как линии выхода, могут быть запрограммированы для вывода прямоугольного периодического сигнала с предварительно настроенной частотой и длительностью рабочего цикла. Такой способ управления сигналом называется *широтно-импульсной модуляцией* (ШИМ) и имеет несколько применений: от управления выходными периферийными исполнительными устройствами до регулировки яркости светодиодов или даже воспроизведения звукового сигнала через динамик.

Более подробная информация о GPIO, таймерах прерываний и сторожевых таймерах будет рассмотрена в главе 6.

1.5. Интерфейсы и периферийные устройства

Для связи с периферийными устройствами и другими микроконтроллерами в мире встраиваемых систем хорошо зарекомендовали себя несколько стандартов. Некоторые из внешних выводов микроконтроллера можно запрограммировать для связи с внешними периферийными устройствами с использованием определенных протоколов. Вот несколько общих интерфейсов, доступных на большинстве архитектур:

- *асинхронная последовательная связь* на основе UART;
- *шина последовательного периферийного интерфейса* (SPI);
- *шина связи между интегральными схемами* (inter-integrated circuit, I^2C);
- *универсальная последовательная шина* (universal serial bus, USB).

Рассмотрим каждую из них подробно.

1.5.1. Асинхронная последовательная связь на основе UART

Асинхронная связь обеспечивается *универсальным асинхронным приемником-передатчиком* (universal asynchronous receiver-transmitter, UART). Интерфейсы такого типа, широко известные как *последовательные порты*, называются асинхронными, потому что им не нужно использовать общий тактовый

сигнал для синхронизации отправителя и получателя, а вместо этого они работают с предопределенными тактовыми сигналами, которые могут быть согласованы во время обмена данными. Микроконтроллеры могут содержать несколько UART, которые подключаются к определенным выводам при настройке (или подключены постоянно). Асинхронная связь обеспечивается UART как полнодуплексный канал по двум независимым линиям, соединяющим вывод RX каждой конечной точки с выводом TX на противоположной стороне.

Для успешного обмена данными системы на двух конечных точках должны настроить UART, используя одни и те же параметры. К ним относятся кадрирование байтов на линии и частота кадров данных. Все параметры должны быть известны обеим конечным точкам заранее, чтобы правильно установить канал связи. Несмотря на то что последовательная связь на основе UART проще, чем другие типы связи, она по-прежнему широко используется в электронных устройствах, особенно в качестве интерфейса для модемов и приемников GPS. Кроме того, используя последовательные преобразователи TTL-USB, можно легко подключить UART к консоли на хост-компьютере, что часто удобно для обмена данными с устройством (например, при отладке).

1.5.2. SPI

Другим классическим протоколом на основе последовательного канала является SPI. Эта технология, разработанная в конце 1980-х годов, была разработана, чтобы заменить асинхронную последовательную связь с периферийными устройствами путем внесения нескольких улучшений:

- отдельная линия тактовых импульсов для синхронизации конечных точек;
- протокол типа «ведущий–ведомый»;
- связь «один ко многим» по одной трехпроводной шине.

Ведущее устройство, обычно микроконтроллер, делит шину с одним или несколькими ведомыми устройствами. Для запуска связи используется отдельный сигнал *выбора ведомого устройства* (slave select, SS), предназначенный для адресации каждого ведомого устройства, подключенного к шине. Шина использует две независимые линии для передачи данных, по одной в каждом направлении, и общую тактовую линию, которая синхронизирует два конца линии связи. Благодаря тактовым импульсам, которые генерирует ведущее устройство, передача данных более надежна, что позволяет достичь более высоких скоростей обмена данными, чем в обычном UART. Одним из секретов неизменной популярности SPI в нескольких поколениях микроконтроллеров является низкая сложность, необходимая для разработки ведомых устройств, которые могут быть очень простыми и состоять буквально из одного сдвигового регистра. SPI обычно используется в сенсорных устройствах, ЖК-дисплеях, контроллерах флеш-памяти и сетевых интерфейсах.

1.5.3. I²C

Протокол I²C немного сложнее, потому что он разработан с другой целью: соединение нескольких микроконтроллеров, а также нескольких ведомых устройств на одной и той же двухпроводной шине. По двум линиям шины передаются *последовательные тактовые импульсы* (serial clock, SCL) и *последовательные данные* (serial data, SDA). В отличие от SPI или UART, шина I²C является полудуплексной, поскольку два направления потока данных используют поочередно одну и ту же линию. Благодаря 7-битному механизму адресации ведомых устройств, включенному в протокол, он не требует дополнительных сигналов, предназначенных для выбора ведомых устройств. Допускается использование нескольких ведущих устройств на одной линии, при условии что все они следуют логике арбитража в случае конфликтов на шине.

1.5.4. USB

Протокол USB, изначально предназначенный для замены UART и включающий в себя множество протоколов в одном аппаратном разъеме, очень популярен в персональных компьютерах, портативных устройствах и огромном количестве периферийных устройств.

Этот протокол работает в режиме «хост–устройство». Хост-контроллер предоставляет услуги, которые может использовать подключаемое устройство. USB-трансиверы, присутствующие во многих микроконтроллерах, иногда могут работать в обоих режимах. Реализуя верхний уровень стандартов USB, микроконтроллер может эмулировать различные типы устройств, такие как последовательные порты, запоминающие устройства и двухточечные интерфейсы Ethernet. Таким образом получаются различные USB-устройства на базе микроконтроллера, которые можно подключать к хосту.

Если трансивер поддерживает режим хоста, встраиваемая система может работать как USB-хост и к ней можно подключать внешние устройства. В этом случае во встраиваемой системе должны быть реализованы драйверы устройств и приложения для доступа к функциям, предоставляемым устройством.

Когда оба режима реализованы на одном USB-контроллере, трансивер работает в режиме «на ходу» (on-the-go, OTG), а выбор и настройка нужного режима могут быть выполнены во время работы.

Более подробное введение в некоторые из наиболее распространенных протоколов, используемых для связи с периферийными устройствами и другими системами, представлены в главе 7.

1.6. Подключенные системы

Все большее количество встраиваемых устройств, предназначенных для различного применения, теперь способны устанавливать сетевую связь со

своими одноранговыми узлами в ближайшем окружении или со шлюзами, направляющими их трафик в более широкую сеть или интернет. Термин *интернет вещей* (Internet of Things, IoT) принято использовать для описания сетей, в которых эти встраиваемые устройства могут обмениваться данными с использованием интернет-протоколов.

Это означает, что к устройствам IoT можно обращаться в сети так же, как к более сложным системам, таким как ПК или мобильные устройства, и, что наиболее важно, они используют для обмена данными стандартные протоколы транспортного уровня, типичные для интернет-коммуникаций. TCP/IP – это набор протоколов, стандартизированных организацией IETF; он представляет собой детальное описание инфраструктуры для интернета и других автономных локальных сетей.

Интернет-протокол (internet protocol, IP) обеспечивает сетевое подключение, но при условии, что базовый канал обеспечивает связь на основе пакетов и механизмы для управления и регулирования доступа к физической среде. Многие сетевые интерфейсы удовлетворяют этим требованиям. Семейства альтернативных протоколов, несовместимых с TCP/IP, все еще используются в некоторых распределенных встраиваемых системах, но явным преимуществом использования стандарта TCP/IP во встраиваемом устройстве является то, что в случае связи с невстраиваемыми системами нет необходимости в механизме трансляции для отправки кадров данных за пределы локальной сети.

Помимо протоколов связи, которые широко используются в обычных системах, таких как Ethernet или беспроводная локальная сеть, встраиваемые системы используют технологии, специально разработанные для требований, предъявляемых IoT. Были разработаны и введены в действие новые стандарты, обеспечивающие эффективную связь для устройств, обладающих ограниченными аппаратными ресурсами и действующих в условиях строгого энергосбережения.

В последнее время были разработаны новые технологии связи, ориентированные на более низкую скорость передачи данных и высокое энергосбережение. Эти протоколы предназначены для обеспечения узкополосной связи на больших расстояниях. Кадр таких протоколов слишком мал для размещения IP-пакетов, поэтому эти технологии в основном используются для передачи небольших полезных данных, таких как периодические показания датчиков или параметры конфигурации устройства, если доступен двунаправленный канал, и им требуется шлюз-транслятор для преобразования пакетов, потому что на определенном этапе данные могут путешествовать по интернету.

Но взаимодействие с облачными сервисами в большинстве случаев требует подключения всех узлов к интернету и внедрения тех же протоколов связи, которые используются серверами и ИТ-инфраструктурой. Реализовать протокол TCP/IP на встраиваемом устройстве не всегда легко. Несмотря на то что существует несколько доступных открытых библиотек, системный код TCP/IP является сложным, большим по размеру и часто имеет требования к памяти, которых трудно придерживаться.

Вышесказанное относится и к библиотеке Secure Socket Layer (SSL)/Transport Layer Security (TLS), которая обеспечивает конфиденциальность и аутен-

тификацию между двумя конечными точками связи. Выбор правильного микроконтроллера для проекта в данном случае имеет ключевое значение. Если встраиваемая система должна быть подключена к интернету и поддерживать защищенную связь через сокеты, то требования к флеш-памяти и ОЗУ следует тщательно продумать на этапе проектирования, чтобы обеспечить возможность использования сторонних библиотек.

1.6.1. Особенности распределенных систем

Разработка распределенных встраиваемых систем, особенно тех, которые основаны на технологиях беспроводной связи, добавляет ряд интересных проблем.

Некоторые из этих проблем связаны со следующими аспектами:

- выбор подходящих технологий и протоколов;
- ограничения на скорость передачи данных, размер пакета и доступ к среде;
- доступность узлов;
- единичные точки отказа в топологии;
- настройка маршрутов;
- аутентификация задействованных хостов;
- конфиденциальность коммуникаций в среде;
- влияние буферизации на скорость сети, задержку и использование ОЗУ;
- сложность реализации стеков протоколов.

В главе 9 анализируются некоторые технологии канального уровня, применяемые во встраиваемых системах для обеспечения удаленной связи, где поддержка протокола TCP/IP интегрирована в структуру распределенных систем, интегрированных со службами интернета вещей.

1.7. Механизмы изоляции

Некоторые современные микроконтроллеры поддерживают механизм изоляции между доверенным и недоверенным программным обеспечением, работающим в устройстве. Этот механизм основан на расширении процессора, доступном только в некоторых архитектурах. Это расширение обычно основано на своего рода физическом разделении внутри самого процессора между двумя режимами выполнения. Весь код, работающий из недоверенной области в системе, будет иметь ограниченный доступ к ОЗУ, встроенным модулям на чипе и периферийным устройствам, которые должны быть заранее динамически настроены доверенным ПО.

Программное обеспечение, работающее из доверенной области, также может предоставлять возможности, недоступные напрямую из ненадежной области, с помощью вызовов специальных функций, пересекающих границу между областями.

В главе 11 рассмотрена технология, лежащая в основе *доверенной среды выполнения* (trust execution environment, TEE), а также программные компоненты, используемые в реальных встроенных системах для формирования безопасной среды выполнения недоверенных модулей и компонентов.

1.8. Базовая платформа

Если говорить о разработке микроконтроллеров для встраиваемых систем, то в настоящее время предпочтительной стратегией при проектировании встроенного процессорного ядра является использование архитектуры *вычислителя с сокращенным набором команд* (reduced instruction set computer, RISC). Производители интегральных схем рекомендуют придерживаться нескольких базовых архитектур в качестве рекомендаций по выбору состава логики для интеграции в микроконтроллер. Каждая архитектура отличается от других несколькими характеристиками реализации процессорного ядра и включает в себя одно или несколько семейств микропроцессоров, характеризуемых следующими параметрами:

- размер слова, используемый для регистров и адресов (8-, 16-, 32- или 64-битный);
- набор инструкций;
- набор конфигураций;
- порядок следования байтов;
- расширенные функции процессора (контроллер прерываний, FPU, MMU);
- стратегии кеширования;
- устройство конвейера.

Выбор базовой платформы для встраиваемой системы зависит от потребностей вашего проекта. Маломощные и менее функциональные процессоры, как правило, больше подходят для устройств с низким энергопотреблением, имеют меньший корпус микроконтроллера и дешевле. Системы более высокого класса поставляются с большим набором ресурсов, а некоторые из них имеют специальное оборудование для выполнения сложных вычислений (например, модуль с плавающей запятой или аппаратный модуль симметричного шифрования Advanced Encryption Standard (AES) для разгрузки процессорного ядра). Широко распространенные в прошлом 8- и 16-битные ядра постепенно уступают место 32-битным архитектурам, но некоторые успешные проекты остаются относительно популярными на некоторых нишевых рынках и среди радиолюбителей.

1.8.1. Базовая архитектура ARM

ARM является самым распространенным поставщиком базовых платформ на рынке встраиваемых систем: для встраиваемых приложений произведено

более 10 млрд микроконтроллеров на базе ARM. Одним из наиболее интересных проектов ядер в индустрии встраиваемых систем является семейство ARM Cortex-M, в модельный ряд которого входит множество вариантов ядер от экономичных и энергоэффективных до высокопроизводительных ядер, специально разработанных для мультимедийных микроконтроллеров. Несмотря на использование трех различных наборов инструкций (ARMv6, ARMv7 и ARMv8), все процессоры семейства Cortex-M используют один и тот же программный интерфейс, что повышает переносимость программ между микроконтроллерами одного семейства.

Большинство примеров в этой книге основаны на семействе процессоров Cortex-M. Хотя большинство рассматриваемых концепций применимы и к другим семействам, такой выбор базовой платформы открывает возможность для наиболее полного анализа взаимодействия кода программы с аппаратной частью. В частности, в некоторых примерах в этой книге используются специальные инструкции по сборке из набора инструкций ARMv7, который реализован в некоторых ядрах Cortex-M.

1.8.2. Микропроцессор Cortex-M

Основные характеристики 32-битных ядер семейства Cortex-M следующие:

- 16 регистров ЦП общего назначения;
- 16-битные инструкции для оптимизации плотности кода;
- встроенный *контроллер приоритетных векторных прерываний* (nested vector interrupt controller, NVIC) с 8...16 уровнями приоритета;
- архитектура ARMv6-M (M0, M0+), ARMv7-M (M3, M4, M7) или ARMv8-M (M23, M33);
- дополнительный *блок защиты памяти* (memory protection unit, MPU) с 8 регионами;
- дополнительный механизм изоляции TEE (ARM TrustZone-M).

Общее адресное пространство памяти составляет 4 ГБ. Начало внутренней оперативной памяти обычно отображается по фиксированному адресу `0x20000000`. Отображение внутренней флеш-памяти, а также других периферийных устройств зависит от производителя чипа. Но самые высокие адреса размером 512 МБ (от `0xE0000000` до `0xFFFFFFFF`) зарезервированы для *блока управления системой* (system control block, SCB), представляющего собой набор параметров конфигурации и диагностики, к которым программное обеспечение может получить доступ в любое время для прямого взаимодействия с ядром.

Синхронная связь с периферийными устройствами и другими аппаратными компонентами может быть запущена через линии прерывания. Процессор может принимать и распознавать несколько различных цифровых входных сигналов и быстро реагировать на них, прерывая выполнение основной программы и временно переходя к определенному месту в памяти. Cortex-M поддерживает до 240 линий прерывания на высокопроизводительных ядрах семейства.

Вектор прерывания, расположенный в начале образа программы во флеш-памяти, содержит адреса подпрограмм прерывания, которые будут автоматически выполняться при определенных событиях. Благодаря NVIC линиям прерывания можно назначать приоритеты, так что когда прерывание с более высоким приоритетом происходит во время выполнения процедуры для более низкого прерывания, текущая процедура прерывания временно приостанавливается, чтобы позволить обслужить линию прерывания с более высоким приоритетом. Это обеспечивает минимальную задержку прерывания для критически важных событий системы.

Программное обеспечение микроконтроллера может работать в двух режимах: непривилегированном или привилегированном. ЦП имеет встроенную поддержку разделения привилегий между системным и прикладным программным обеспечением и даже предоставляет два разных регистра для двух независимых указателей стека. В главе 10 более подробно рассмотрено, как правильно реализовать разделение привилегий, а также как обеспечить разделение памяти при выполнении ненадежного кода. Технология разделения памяти, например, используется для сокрытия секретов, таких как закрытые ключи, от прямого доступа из внешнего мира. В главе 11 показано, как правильно реализовать разделение привилегий, а также как обеспечить разделение памяти внутри ОС при запуске кода приложения с другим уровнем доверия.

Ядро Cortex-M присутствует во многих микроконтроллерах от разных поставщиков микросхем. Программные инструменты одинаковы для всех платформ, но каждый вариант микроконтроллера имеет свою конфигурацию, которую необходимо учитывать при разработке системы. Доступны так называемые библиотеки конвергенции, помогающие абстрагироваться от деталей реализации конкретного производителя и улучшить переносимость между разными моделями и брендами. Производители предоставляют обучающие наборы (оценочные платы) и всю документацию, необходимую для начала работы и оценки на этапе проектирования, а также полезную для разработки прототипов на более позднем этапе. Некоторые из оценочных плат оснащены датчиками, мультимедийной электроникой или другими периферийными устройствами, расширяющими функциональные возможности микроконтроллера. Отдельные платы даже поставляются в комплекте с предварительно сконфигурированными сторонними библиотеками промежуточного программного обеспечения, включая коммуникационные стеки TCP/IP, библиотеки TLS и криптографии, простые файловые системы и другие вспомогательные компоненты, а также модули, которые можно быстро и легко добавить в программный проект.

1.9. Заключение

При составлении требований к встроенному программному обеспечению прежде всего вы должны иметь хорошее представление об аппаратной платформе и ее компонентах. В этой главе сделан краткий обзор архитектур

современных микроконтроллеров, показаны некоторые особенности встраиваемых устройств и сделали акцент на том, что разработчикам следует эффективно переосмыслить свой подход к удовлетворению требований и решению задач, в то же время принимая во внимание особенности и ограничения целевой платформы.

В следующей главе будут проанализированы инструменты и процедуры, обычно используемые при разработке встраиваемых систем, включая наборы инструментов командной строки и *интегрированные среды разработки* (integrated development environment, IDE). В ней будет показано, как организовать рабочий процесс и как эффективно предотвращать, находить и исправлять ошибки.

Глава 2

Рабочая среда и оптимизация рабочего процесса

Первый шаг на пути к успешному программному проекту – выбор правильных инструментов. Для разработки встраиваемых систем требуется набор аппаратных и программных инструментов, облегчающих жизнь разработчика и способных значительно повысить производительность и сократить общее время разработки. В этой главе представлено описание таких инструментов и даны советы по их использованию для улучшения рабочего процесса.

Первый раздел главы содержит обзор рабочего процесса программирования на «чистом» языке C и раскрывает изменения, необходимые для перехода к использованию среды разработки. Набор инструментов разработки GCC и его компоненты для создания приложений также представлены в первом разделе.

В последних двух разделах описаны стратегии взаимодействия с микроконтроллером, включая механизмы отладки и проверки программного обеспечения (ПО), работающего во встраиваемой системе.

В этой главе разобраны следующие темы:

- обзор рабочего процесса;
- текстовые редакторы и интегрированные среды разработки;
- набор инструментальных средств GCC;
- взаимодействие с целевым устройством;
- тестирование.

К концу главы вы узнаете, как создать оптимизированный рабочий процесс, следуя нескольким основным правилам, уделяя особое внимание подготовке тестов и разумному подходу к отладке.

2.1. Обзор рабочего процесса

Разработка программного обеспечения на C, как и на любом другом компилируемом языке, требует преобразования исходного кода в исполняемый формат для целевого процессора. Исходный код C в общем случае поддается переносу между различными архитектурами и средами выполнения. Разработчики встраиваемых систем используют набор инструментов для компиляции, компоновки, выполнения и отладки программного обеспечения, ориентированный на конкретный процессор (платформу).

Способ создания образа ПО для встраиваемой системы зависит от аналогичного набора инструментов, предназначенного для создания прошивки конечного устройства. В этом разделе дается обзор общего инструментария, необходимого для разработки программного обеспечения на C и создания программ, способных выполняться непосредственно на машине, на которой они были скомпилированы. Затем будет расширен рабочий процесс и добавим в него инструменты создания исполняемого кода для целевой платформы.

2.1.1. Компилятор C

Компилятор C – это инструмент, отвечающий за преобразование исходного кода в машинный код, который может быть интерпретирован конкретным процессором. Каждый компилятор может создавать машинный код только для одной среды, поскольку он переводит команды исходного кода в машинно зависимые инструкции и настроен на использование модели адресов и структуры регистров одной конкретной архитектуры. Собственный компилятор, включенный в большинство дистрибутивов GNU/Linux, представляет собой *коллекцию компиляторов GNU*, широко известную как GCC. Это открытая система компиляции, распространяемая под общедоступной лицензией GNU с 1987 года; с тех пор она успешно используется для построения UNIX-подобных систем. Встроенный GCC может компилировать код C в приложения и библиотеки, способные работать на той же архитектуре, что и машина, на которой работает компилятор.

Компилятор GCC принимает файлы исходного кода с расширением `.c` и создает объектные файлы с расширением `.o`, содержащие код программы и начальные значения переменных, преобразованные из текстового исходного кода в машинные инструкции. Компилятор можно настроить для выполнения дополнительных шагов оптимизации в конце компиляции, ориентированных на целевую платформу, и вставки отладочных данных для облегчения отладки на более позднем этапе.

Минимальная командная строка, используемая для компиляции исходного файла в объектный файл с помощью хост-компилятора, состоит только из опции `-c`, указывающей инструменту GCC скомпилировать исходный код в объектный файл с тем же именем:

```
$ gcc -c hello.c
```

В соответствии с этой командой GCC попытается скомпилировать исходный код C, содержащийся в файле `hello.c`, и преобразовать его в машинно зависимый код, который хранится во вновь созданном файле `hello.o`.

Для компиляции кода под конкретную целевую платформу требуется соответствующий набор инструментальных средств. Процесс генерации кода для другого целевого процессора называется *кросс-компиляцией*. *Кросс-компилятор* запускается на машине разработки (хосте) для создания машинно зависимого кода, который может выполняться на целевом процессоре.

Далее будет рассмотрен инструментарий на основе GCC, предназначенный для создания прошивки встраиваемой системы, и кратко описан синтаксис и характеристики компилятора GCC.

Первым шагом создания программы, состоящей из отдельных модулей, является компиляция всех исходных кодов в объектные файлы, чтобы компоненты, необходимые системе, были сгруппированы и собраны вместе на заключительном этапе, когда к работе приступает компоновщик.

2.1.2. Компоновщик

Компоновщик (linker) – это инструмент, который компонует исполняемый код и разрешает зависимости между объектными файлами, которые для него являются входными файлами.

Формат исполняемого файла по умолчанию, создаваемый компоновщиком, – это *формат исполняемых и компонуемых модулей* (executable and linkable format, ELF). Он является стандартным форматом по умолчанию для программ, объектов, общих библиотек и даже дампов ядра GDB во многих Unix и Unix-подобных системах. Формат был разработан для хранения программ на дисках и других носителях, поэтому операционная система хоста может выполнять код, загружая инструкции в ОЗУ и выделяя место для данных программы.

Исполняемые файлы разделены на разделы, которые могут быть сопоставлены с определенными областями памяти, необходимыми программе для выполнения. Файл ELF начинается с заголовка, содержащего указатель на различные разделы внутри самого файла, содержащего код программы и данные.

Компоновщик отображает содержимое областей, описывающих исполняемую программу, на разделы, имена которых обычно начинаются с символа точки (.). Для запуска исполняемого файла необходим следующий минимальный набор разделов:

- `.text`: содержит код программы, доступный только для чтения. Это исполняемые инструкции программы. Функции, скомпилированные в объектные файлы, размещаются компоновщиком в этом разделе, и программа всегда выполняет инструкции в этой области памяти;
- `.rodata`: содержит значения констант, которые нельзя изменить во время выполнения. Используется компилятором в качестве раздела по умолчанию для хранения констант, поскольку в нем не разрешено изменять сохраненные значения во время выполнения;

- `.data`: содержит значения всех инициализированных переменных программы, которые доступны в режиме чтения/записи во время выполнения. Это раздел, содержащий все переменные (статические или глобальные), которые были инициализированы в коде. Перед выполнением эта область обычно переназначается на доступное для записи место в ОЗУ, а содержимое ELF автоматически копируется во время инициализации программы, во время выполнения, перед запуском функции `main()`;
- `.bss`: это раздел, зарезервированный для неинициализированных данных, доступный в режиме чтения/записи во время выполнения. Он получил свое название от старой инструкции по сборке старого микрокода, разработанного для IBM 704 в 1950-х годах. Первоначально это была аббревиатура от Block Started by Symbol (BSS) – блока, используемого для резервирования фиксированного объема неинициализированной памяти. В контексте ELF раздел содержит все неинициализированные глобальные и статические символьные указатели, которые должны быть доступны в режиме чтения-записи во время выполнения. Поскольку значение не присвоено, файл ELF описывает только раздел в заголовке, но не дает для него никакого содержимого. Код инициализации должен гарантировать, что все переменные в этом разделе установлены в ноль перед выполнением функции `main()`.

При создании собственного программного обеспечения на хост-компьютере большая часть сложности этапа компоновки скрыта, но компоновщик по умолчанию настроен на размещение скомпилированных объектов в определенные разделы, которые впоследствии могут использоваться операционной системой для назначения соответствующих разделов в виртуальном адресном пространстве процесса при выполнении программы. Можно создать работающий исполняемый файл для хост-машины, просто вызвав `gcc`, на этот раз без параметра `-c`, предоставив список объектных файлов, которые должны быть скомпонованы для создания файла ELF. Параметр `-o` используется для указания имени выходного файла, которое в противном случае по умолчанию было бы `a.out`:

```
$ gcc -o helloworld hello.o world.o
```

Эта команда создает файл `helloworld`, который является исполняемым файлом ELF для хост-системы, используя два ранее скомпилированных объектных файла.

В случае встраиваемой системы все немного меняется, так как выполнение приложения без операционной системы подразумевает, что разделы должны быть сопоставлены с физическими областями в памяти во время компоновки. Чтобы компоновщик связал разделы с заранее известными физическими адресами, необходимо предоставить ему пользовательский файл сценария компоновщика, описывающий структуру памяти исполняемого приложения и содержащий, при необходимости, дополнительные разделы, которые могут потребоваться целевой системе.

Более подробное объяснение этапа компоновки представлено далее в разделе 2.3.3.

2.1.3. Инструмент автоматизации сборки Make

Существует несколько инструментов с открытым исходным кодом для автоматизации процесса сборки. Некоторые из них широко используются в различных средах разработки. Make – это стандартный инструмент UNIX для автоматизации последовательности действий, необходимых для создания требуемых двоичных образов из исходных кодов, проверки зависимостей для каждого компонента и выполнения шагов в правильном порядке. Make является частью многих UNIX-подобных систем и относится к группе инструментов POSIX. В дистрибутиве GNU/Linux он реализован как отдельный инструмент, являющийся частью проекта GNU. Далее для краткости в этой книге реализация GNU Make называется просто Make.

Make предназначен для запуска сборки по умолчанию простым вызовом команды make без аргументов из командной строки, при условии что в рабочем каталоге присутствует makefile – специальный файл инструкций, содержащий правила и рецепты для создания всех файлов, необходимых до тех пор, пока не будут сгенерированы ожидаемые выходные файлы. Существуют альтернативы с открытым исходным кодом, предлагающие аналогичные решения для автоматизации сборки, такие как CMake и SCons, но все примеры в этой книге построены с использованием Make, поскольку он предоставляет простую и вполне достаточную среду для управления системой сборки, и именно она стандартизирована POSIX.

Некоторые интегрированные среды разработки используют встроенные механизмы для координации шагов сборки или создания файлов сборки перед автоматическим вызовом Make, когда пользователь запрашивает сборку выходных файлов. Но редактирование make-файлов вручную дает полный контроль над промежуточными этапами создания финальных образов прошивки, где пользователь может настроить рецепты и правила, используемые для создания выходных файлов.

К версии, которую необходимо установить для кросс-компиляции кода под Cortex-M, нет особых требований, но необходимо позаботиться о некоторых дополнительных параметрах, таких как расположение исполняемых файлов инструментария или определенные флаги, необходимые компилятору при записи целей и директив в make-файле.

Одним из преимуществ использования процесса сборки является то, что целевые файлы могут иметь неявные зависимости от других промежуточных компонентов, которые автоматически разрешаются во время компиляции. Если все зависимости настроены правильно, make-файл гарантирует, что промежуточные шаги будут выполняться только тогда, когда это необходимо, сокращая время компиляции всего проекта, когда изменено лишь несколько исходных кодов или удалены отдельные объектные файлы.

Make-файлы имеют специальный синтаксис для описания правил. Каждое правило начинается с целевых файлов, ожидаемых в качестве выходных данных правила, двоеточия и списка предварительных условий, которые являются файлами, необходимыми для выполнения правила. Далее следует набор элементов рецепта, каждый из которых описывает действия, которые Make выполнит для создания желаемой цели:

```
target: [prerequisites]
 recipe
 recipe
 ...
```

По умолчанию Make выполнит первое правило, обнаруженное при анализе файла, если имя правила не указано в командной строке. Если какое-либо из предварительных условий недоступно, Make автоматически ищет правило в том же make-файле, которое может рекурсивно создать требуемый файл, пока цепочка требований не будет удовлетворена.

Make-файлы могут назначать пользовательскую строку текста внутренним переменным на время выполнения. Имена переменных можно назначать с помощью оператора = и ссылаться на них, добавляя к ним префикс $. Например, следующее присваивание используется для размещения имени двух объектных файлов в переменной OBJS:

```
OBJS = hello.o world.o
```

В табл. 2.1 перечислено несколько важных переменных, которые автоматически назначаются в правилах.

Таблица 2.1. Некоторые автоматические переменные, которые можно использовать в рецептах make-файлов

Переменная	Назначение
$(@)	Название цели для выполняющегося правила
$(^)	Список необходимых условий для этого правила (без повторов)
$(+)	Список необходимых условий для этого правила (с повторами, если есть)
$(<)	Первый элемент в списке необходимых условий

Эти переменные удобно использовать в строках действий рецепта. Например, рецепт создания ELF-файла helloworld из двух объектных файлов можно записать следующим образом:

```
helloworld: $(OBJS)
 gcc -o $(@) $(^)
```

Некоторые правила неявно определяются Make. Например, правило создания файлов hello.o и world.o из соответствующих исходных файлов может быть опущено, поскольку Make рассчитывает получить каждый из этих объектных файлов наиболее очевидным способом, то есть путем компиляции соответствующих исходных файлов C с тем же именем, если они есть. Это означает, что такой минималистичный make-файл уже способен скомпилировать два объекта из исходников и скомпоновать их вместе, используя набор опций по умолчанию для хост-системы.

Рецепт компоновки также может быть неявным, если исполняемый файл имеет то же имя, что и один из его обязательных объектов, за исключением

расширения .o. Если окончательный ELF-файл называется hello, то make-файл может состоять из единственной строки:

```
hello: world.o
```

Make автоматически разрешит зависимости hello.o и world.o, а затем скомпонует их, используя неявный рецепт компоновки, аналогичный тому, который использовался в явном виде.

Неявные правила используют предопределенные переменные, которые назначаются автоматически перед выполнением правил, но могут быть изменены в make-файле. Например, можно назначить другой компилятор по умолчанию, изменив переменную CC. В табл. 2.2 представлен краткий список наиболее важных переменных, которые можно использовать для изменения неявных правил и рецептов.

Таблица 2.2. Неявные предустановленные переменные, определяющие инструментарий и флаги по умолчанию

Переменная	Назначение	Значение по умолчанию
CC	Компилятор	cc
LD	Компоновщик	ld
CFLAGS	Флаги, передаваемые в компилятор на шаге компиляции исходных файлов	<пусто>
LDFLAGS	Флаги, передаваемые в компилятор на шаге компоновки	<пусто>

При компоновке приложения, выполняемого на «голом железе» для встраиваемых платформ, make-файл необходимо изменить соответствующим образом и, как показано далее в этой главе, требуется несколько флагов для правильной кросс-компиляции исходных кодов и указание компоновщику использовать желаемую структуру памяти для организации разделов. Кроме того, как правило, требуются дополнительные действия по обработке ELF-файла и его перевода в формат, который можно передать в целевую систему. Но синтаксис make-файла такой же, и приведенные здесь простые правила не слишком отличаются от тех, что использовались для сборки примера. Переменные по умолчанию тоже нуждаются в настройке, чтобы изменить поведение по умолчанию, если используются неявные правила.

Когда все зависимости в make-файле правильно настроены, Make гарантирует, что правила будут выполняться только тогда, когда цель старше, чем ее зависимости, тем самым сокращая время компиляции всего проекта в ситуациях, когда только несколько файлов исходных кодов изменены или одиночные объектные файлы были изменены либо удалены.

Make – очень мощный инструмент, и диапазон его возможностей выходит далеко за рамки тех немногих функций, которые используются для создания примеров в этой книге. Освоение способов автоматизации сборки помогает оптимизировать рабочий процесс. Синтаксис make-файлов включает в себя полезные функции, такие как условия, которые можно использовать для по-

лучения различных результатов путем вызова make-файла с использованием разных целей или переменных среды. Чтобы более детально изучить возможности Make, обратитесь к руководству GNU Make, доступному по адресу https://www.gnu.org/software/make/manual.

2.1.4. Отладчик

В хост-среде отладка приложения, работающего поверх операционной системы, выполняется путем запуска инструмента *отладчика* (debugger), который может подключаться к существующему процессу или создавать новый с помощью исполняемого файла ELF и его аргументов командной строки. Параметр отладки по умолчанию, предоставляемый пакетом GCC, называется GDB (аббревиатура от GNU Debugger). Хотя GDB является инструментом командной строки, было разработано несколько внешних интерфейсов для лучшей визуализации состояния выполнения, а некоторые интегрированные среды разработки предоставляют встроенные интерфейсы-обертки для взаимодействия с отладчиком при отслеживании отдельных выполняемых строк кода.

Ситуация немного меняется, когда отлаживаемое программное обеспечение работает на удаленной платформе. Версия GDB, распространяемая с набором инструментов и специфичная для целевой платформы, может быть запущена на машине разработки для подключения к сеансу удаленной отладки. Для сеанса отладки на удаленной целевой системе требуется промежуточный инструмент, сконфигурированный для преобразования команд GDB в фактические действия процессора и соответствующей аппаратной инфраструктуры для установки связи с ядром.

Некоторые встраиваемые платформы предоставляют аппаратные точки останова, которые используются для запуска системных исключений каждый раз при выполнении выбранных инструкций.

Позже в этой главе будет показано, как можно установить удаленный сеанс GDB с целевой системой, чтобы прервать выполнение в текущей точке, перейти к пошаговому выполнению кода, установить точки останова и наблюдения, а также проверить и изменить значения в памяти. Будет рассмотрено несколько команд GDB и дан краткий обзор некоторых функций, предоставляемых интерфейсом командной строки GDB и применяемых для эффективной отладки встроенных приложений.

Отладчик дает возможность увидеть, что делает программа во время выполнения, и облегчает поиск ошибок, непосредственно наблюдая за влиянием кода на память и регистры ЦП.

2.1.5. Цикл разработки встраиваемых систем

По сравнению с другими областями программирования, цикл разработки встраиваемых систем включает в себя несколько дополнительных шагов. Необходимо выполнить кросс-компиляцию исходного кода, создать образ

прошивки, а затем загрузить ее в целевое устройство; после этого нужно запустить тесты и, возможно, применить специальное оборудование на этапах измерения и проверки. Цикл разработки обычного приложения при использовании компилируемых языков показан на рис. 2.1.

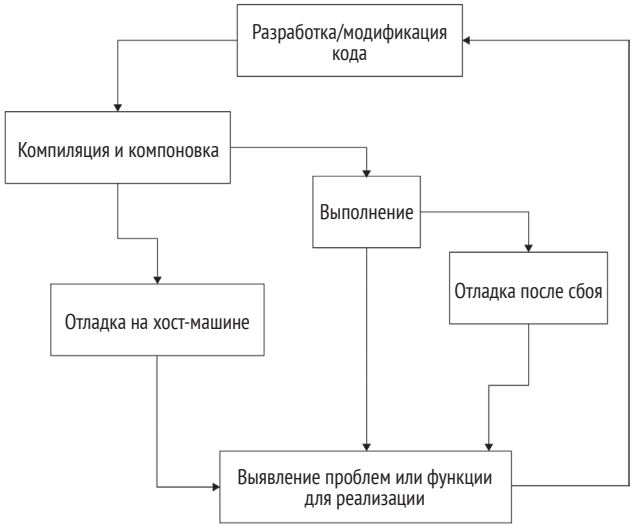

Рис. 2.1 ❖ Типичный цикл разработки приложения

Если платформа для разработки приложения совпадает с платформой его выполнения, то тестирование и отладку можно запустить сразу после компиляции. Благодаря этому часто бывает проще обнаружить проблемы. Это приводит к сокращению длительности типичного цикла разработки. Более того, если приложение аварийно завершает работу из-за ошибки, базовая операционная система может создать дамп ядра, который можно проанализировать с помощью отладчика позднее, восстановив содержимое виртуальной памяти и контекст регистров процессора непосредственно в момент, когда появляется ошибка.

В свою очередь, перехват критических ошибок во встраиваемой системе может быть осложнен потенциальным побочным эффектом повреждения памяти и регистров в связи с отсутствием виртуальных адресов и сегментации памяти, предоставляемых операционной системой. Даже если некоторые встраиваемые системы могут перехватывать ненормальные ситуации, вызывая диагностические прерывания, такие как аппаратный обработчик сбоев в Cortex-M, восстановление исходного состояния, вызвавшего ошибку, часто невозможно.

Кроме того, каждый раз, когда создается новое встраиваемое программное обеспечение, необходимо выполнить несколько трудоемких шагов, таких как перевод образа прошивки в определенный формат и загрузка образа в само устройство, что может занять от нескольких секунд до минуты, в зависимости от размера образа и скорости интерфейса, используемого для связи с целевой системой (рис. 2.2).

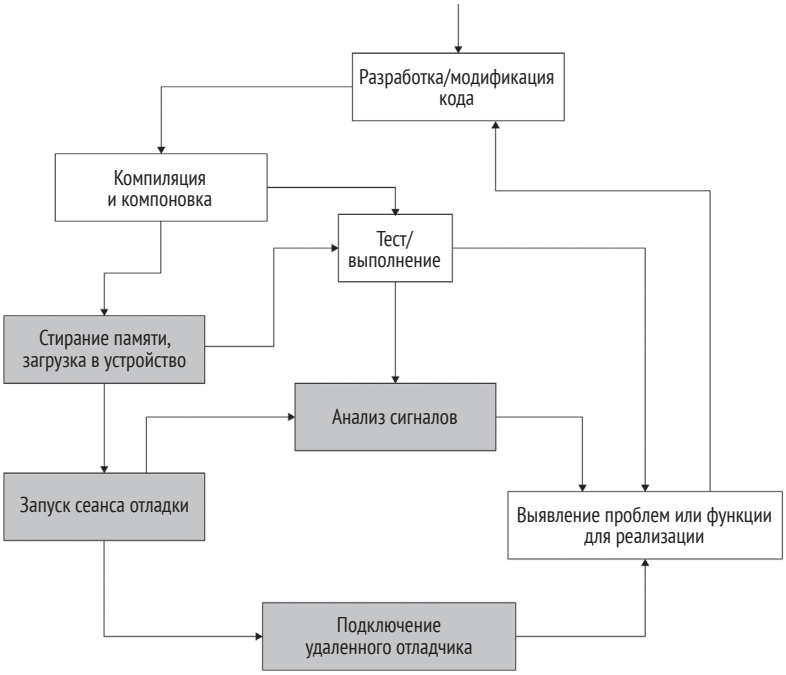

Рис. 2.2 ❖ Цикл разработки встраиваемых систем, включая дополнительные шаги, зависящие от среды

На некоторых этапах разработки, когда требуется несколько последовательных итераций для завершения реализации функции или обнаружения ошибки, время между компиляцией и тестированием программного обеспечения влияет на эффективность всего цикла разработки. Выполнение конкретных задач, реализованных в программном обеспечении, может быть проверено только с помощью анализа электрических сигналов или путем наблюдения за воздействием на периферийное устройство или удаленную систему. Анализ поведения электрических цепей встраиваемой системы требует подключения и настройки измерительного оборудования, что дополнительно удлиняет цикл разработки.

Разработка распределенной встраиваемой системы, состоящей из нескольких устройств с разными образами прошивки, может привести к повторению вышеупомянутых итераций для каждого из этих устройств. По возможности эти этапы следует исключить, используя один и тот же образ и разные параметры конфигурации на каждом устройстве, а также механизмы параллельного обновления прошивок. Такие протоколы, как JTAG, поддерживают загрузку образа программного обеспечения на несколько целевых устройств, использующих одну и ту же шину, что значительно сокращает время, необходимое для обновления прошивки, особенно в распределенных системах с большим количеством задействованных устройств.

Независимо от того, насколько сложным будет проект, в целом стоит потратить столько времени, сколько необходимо для оптимизации жизненного

цикла разработки программного обеспечения в начале, чтобы впоследствии повысить его эффективность. Ни один разработчик не любит слишком долго переключать внимание с процесса разработки кода на вспомогательные операции, и ему неприятно работать в неоптимальной среде, где пошаговое выполнение процесса занимает слишком много времени или требует взаимодействия с человеком.

Встраиваемое приложение можно разработать с нуля в обычном текстовом редакторе или создать новый проект в *интегрированной среде разработки* (integrated development environment, IDE).

2.2. Текстовый редактор или интегрированная среда?

Хотя, строго говоря, это вопрос предпочтений разработчика, в сообществе разработчиков встраиваемых систем до сих пор ведутся споры между теми, кто использует автономный текстовый редактор, и теми, кто предпочитает, чтобы все компоненты инструментария были интегрированы в один графический интерфейс.

Современные IDE обязательно содержат встроенные инструменты для следующих задач:

- управление компонентами проекта;
- быстрый доступ ко всем файлам для редактирования, а также расширения для загрузки программного обеспечения на плату;
- запуск сеанса отладки одним щелчком мыши.

Производители микроконтроллеров часто предоставляют свои наборы плат и компонентов для использования вместе с интегрированными средами разработки, которые упрощают доступ к расширенным функциям, специфичным для микроконтроллера, благодаря предварительно заданным настройкам и мастерам, облегчающим создание новых проектов. Большинство фирменных IDE от производителя чипов содержат виджеты для автоматической генерации кода настройки выводов для определенных микроконтроллеров, обычно с графическим интерфейсом. Некоторые из них даже предлагают симуляторы и инструменты для прогнозирования использования ресурсов во время выполнения, таких как динамическая память и энергопотребление.

Большинство этих инструментов основаны на предварительно настроенной версии Eclipse – популярной среды разработки для настольных ПК с открытым исходным кодом, изначально созданной как инструмент для разработки программного обеспечения Java, а затем получившей широкое распространение во многих других областях благодаря возможностям расширения и настройки интерфейса.

У использования IDE есть и недостатки. IDE, как правило, не содержат набор инструментов в собственном коде. Они всего лишь предоставляют интерфейс-обертку для взаимодействия с компилятором, компоновщиком, отладчиком

и другими инструментами. Для этого они должны хранить все флаги, параметры конфигурации, пути к включенным файлам и определенные во время компиляции символы в машиночитаемом файле конфигурации. Некоторым пользователям трудно получить доступ к этим параметрам, перемещаясь по нескольким меню графического интерфейса. Другие ключевые компоненты проекта, такие как скрипт компоновщика, также могут быть спрятаны глубоко под капотом, в некоторых случаях даже автоматически генерируются IDE и трудночитаемы. Но для большинства пользователей IDE эти недостатки компенсируются преимуществами разработки в интегрированной среде.

Есть одна последняя оговорка, которую необходимо учитывать. Рано или поздно проект будет собран и проведено его тестирование автоматически, как уже было сказано в разделе 2.1.3. Роботы – вообще ужасные пользователи IDE. Зато они могут построить и запустить любой тест, даже взаимодействующий с реальными целевыми устройствами, используя интерфейс командной строки. Команда разработчиков, использующая IDE для разработки встраиваемых систем, всегда должна рассматривать вероятность создания и тестирования любого программного обеспечения с помощью альтернативной стратегии через командную строку.

Хотя от сложности инструментария можно абстрагироваться с помощью графического интерфейса, полезно понимать функции набора приложений, спрятанных под капотом IDE. В оставшихся разделах этой главы представлен инструментарий GCC – самый популярный кросс-архитектурный набор компиляторов для многих 32-битных микроконтроллеров.

2.3. Инструментарий GCC

При использовании IDE происходит взаимодействие с единым пользовательским интерфейсом, но фактически инструментарий разработчика представляет собой набор автономных приложений, каждое из которых служит определенной цели.

GCC является одним из эталонных наборов инструментов для создания встраиваемых систем благодаря своей модульной структуре, позволяющей создавать средства разработки для различных архитектур. Благодаря своей модели с открытым исходным кодом и гибкости в создании на ее основе специализированных наборов инструментов GCC является одним из самых популярных инструментов разработки встраиваемых систем.

Создание программного обеспечения с использованием инструментов командной строки имеет несколько преимуществ, включая возможность автоматизации промежуточных шагов, которые будут собирать все модули из исходного кода в окончательный образ прошивки. Это особенно полезно, когда требуется запрограммировать несколько устройств подряд или автоматизировать сборку на сервере непрерывной интеграции.

ARM распространяет пакет GNU Arm Embedded Toolchain для всех самых популярных платформ разработки. Название набора инструментов начинается с трех букв, указывающих на целевую платформу. В случае GNU Arm

Embedded Toolchain префикс `arm-none-eabi` указывает на то, что серверная часть кросс-компилятора настроена на создание объектов для ARM без специальной поддержки API операционной системы и со встроенным ABI.

2.3.1. Кросс-компилятор

Кросс-компилятор, распространяемый в составе набора инструментов, представляет собой вариант GCC, в котором выполняемое на хост-машине приложение (бэкенд) настроено на создание объектных файлов, содержащих машинный код для другой архитектуры. Результатом компиляции является набор объектных файлов, содержащих символьные указатели, которые может интерпретировать только целевое устройство. Предоставляемый ARM вариант GCC `arm-none-eabi-gcc` может компилировать код C в машинные инструкции для нескольких различных целевых микроконтроллеров. Каждой архитектуре нужен свой собственный набор инструментов, который будет создавать исполняемые файлы для конкретных целевых устройств.

В бэкенде GCC для архитектуры ARM предусмотрен выбор конкретных целевых процессоров, который, в свою очередь, определяет выбор инструкций для ЦП и параметров оптимизации.

В табл. 2.3 перечислены некоторые специфичные для ARM параметры компилятора, доступные в бэкэнде GCC в виде флагов `-m`.

Таблица 2.3. Опции компилятора GCC ARM, зависящие от архитектуры

Опция	Описание
`-marm` `-mthub`	Выбор между набором инструкций ARM и Thumb
`-march=name`	Выбор архитектуры ISA в семействе (например, `armv7`)
`-mtune=name`	Выбор ЦП для оптимизации (например, `cortex-m3`)
`-mcpu=name`	Выбор архитектуры и ЦП для оптимизации (можно использовать вместо `-march` и `-mtune`)

Чтобы скомпилировать код, совместимый с базовым ядром ARM Cortex M4, каждый раз при вызове компилятора необходимо указывать параметры `-mthumb` и `-mcpu=cortex-m4`:

```
$ arm-none-eabi-gcc -c test.c -mthumb -mcpu=cortex-m4
```

Файл `test.o`, который является результатом этого шага компиляции, сильно отличается от файла, который можно скомпилировать из того же источника с помощью бэкенда `gcc`. Разница видна лучше, если вместо двух объектных файлов сравнивать промежуточный код сборки. Компилятор способен создавать промежуточные файлы кода сборки вместо скомпилированных и собранных объектов, когда он вызывается с параметром `-S`.

Как и в компиляторе бэкенда GCC, здесь доступны различные уровни оптимизации. В некоторых случаях имеет смысл использовать оптимизацию размера для создания меньших объектных файлов. Но желательно, чтобы во время разработки во флеш-память микроконтроллера помещался неоп-

тимизированный код. Это облегчает процедуру отладки, поскольку выполнение оптимизированного кода труднее отслеживать – компилятор может изменить порядок выполнения кода и скрыть содержание некоторых переменных. Параметры выбора уровня оптимизации, которые можно указать в командной строке, перечислены в табл. 2.4.

Таблица 2.4. Уровни оптимизации GCC

Опция	Результат
`-O0`	Не оптимизировать; оптимизация полностью отключена
`-O1`	Оптимизировать быстродействие
`-O2`	Более высокая оптимизация быстродействия
`-O3`	Максимальная оптимизация быстродействия
`-Os`	Оптимизация размера

Еще один параметр командной строки GCC, который часто используется при отладке и прототипировании, – это флаг `-g`, который указывает компилятору сохранять данные, связанные с отладкой, в финальном объекте, чтобы облегчить доступ к читаемым дескрипторам функций и переменных во время работы в отладчике.

Чтобы сообщить компилятору, что производится запуск приложения в автономной среде (т. е. на «голом железе» без ОС), используется параметр командной строки `-ffreestanding`. С точки зрения GCC, для автономной среды характерно возможное отсутствие стандартных библиотек на этапе компоновки, и, что наиболее важно, эта опция предупреждает компилятор о том, что ему не следует ожидать использования функции `main()` в качестве точки входа в программу или наличия какого-либо кода преамбулы до начала выполнения основной программы. Эта опция требуется при компиляции кода для встраиваемых платформ, так как она включает механизм загрузки, описанный в главе 4.

GCC поддерживает намного больше параметров командной строки, чем те, которые кратко представлены здесь. За более полным описанием доступных опций обратитесь к руководству GNU GCC по адресу https://gcc.gnu.org/onlinedocs/.

Чтобы интегрировать цепочку инструментов кросс-компиляции в автоматизированную сборку с помощью Make, необходимо внести несколько изменений в make-файл.

Если полагать, что набор инструментов правильно установлен на хосте разработки и доступен по указанному в системе пути выполнения, достаточно изменить команду компилятора по умолчанию с помощью переменной `CC` в make-файле:

```
CC=arm-none-eabi-gcc
```

Пользовательские параметры командной строки, необходимые для определения параметров компиляции, можно экспортировать через переменную `CFLAGS`:

```
CFLAGS=-mthumb -mcpu=cortex-m4 -ffreestanding
```

Использование переменных make-файла, таких как CC и CFLAGS, позволяет применять неявные правила, создание объектных файлов из исходных кодов C с тем же именем и пользовательскую конфигурацию компилятора.

2.3.2. Кто компилирует компиляторы?

Как правило, для конкретных хост-компьютеров и целевых микроконтроллеров можно скачать готовые исполняемые файлы набора инструментов GCC. Для компиляции кода микропроцессоров ARM Cortex-M в большинстве дистрибутивов GNU/Linux доступен инструментарий arm-none-eabi. Но в некоторых случаях приходится собирать набор инструментов из файлов исходного кода. Так бывает, например, когда компилятор для определенного микроконтроллера еще не существует или не поставляется в исполняемом формате для вашей любимой среды разработки. Процесс самостоятельной компиляции набора инструментов также полезен для лучшего понимания различных компонентов, входящих в его состав.

Crosstool-NG – проект с открытым исходным кодом, состоящий из набора скриптов, предназначенных для автоматического создания набора инструментов. Это приложение берет выбранную версию каждого компонента, а затем создает архив набора инструментов, который можно распространять в двоичной форме. Обычно в этом нет необходимости, хотя иногда это полезно, когда необходимо изменить исходные коды для определенного компонента, такого как, например, библиотеки C, которые интегрированы в набор инструментов. Создание новой конфигурации в crosstool-NG не составляет труда благодаря его конфигуратору menuconfig, основанному на ядре Linux. После установки crosstool-NG конфигуратор можно вызвать при помощи следующей команды:

```
$ ct-ng menuconfig
```

После создания конфигурации можно приступать к процессу сборки. Поскольку операция требует извлечения всех компонентов, их исправления и создания нового набора инструментов, получение всех компонентов может занять несколько минут, в зависимости от быстродействия хост-компьютера и скорости интернет-соединения. Процесс сборки можно запустить при помощи команды

```
$ ct-ng build
```

Для компиляции часто используемых наборов инструментов доступны предопределенные конфигурации (в основном для приложений, работающих под Linux). При компиляции инструментов для среды Linux есть несколько библиотек C на выбор. В случае, когда нужен набор инструментов с нуля, выбором по умолчанию является newlib. Несколько других библиотек предоставляют реализацию подмножества стандартной библиотеки C, например uClibc и musl. Библиотека newlib представляет собой небольшую кросс-платформенную библиотеку C, в основном предназначенную для

встраиваемых систем без ОС; она доступна по умолчанию во многих дистрибутивах GCC, включая кросс-компилятор `arm-none-eabi`, распространяемый ARM.

2.3.3. Компоновка исполняемого файла

Компоновка (linking) – это последний шаг в создании файла ELF. Кросс-компилятор GCC группирует все объектные файлы вместе и разрешает зависимости между символами. При передаче в командной строке параметра `-T <имя файла>` компоновщику предлагается заменить структуру памяти по умолчанию пользовательским сценарием, содержащимся в файле с указанным именем.

Сценарий компоновщика (linker script) представляет собой файл, содержащий описание разделов памяти в целевом устройстве, о которых необходимо знать заранее, чтобы компоновщик мог разместить символы в нужных разделах во флеш-памяти и проинструктировать программные компоненты об особых местах в областях отображения памяти, на которые можно ссылаться в коде. Файл сценария имеет расширение `.ld` и разработан на специальном языке.

Сценарий может взаимодействовать с кодом C, экспортируя символы, определенные в сценарии, и следуя указаниям, предоставленным в коде с использованием атрибутов GCC, связанных с символами. Ключевое слово `__attribute__` размещается перед определением символа, чтобы активировать специфичные для GCC нестандартные атрибуты для каждого символа.

Некоторые атрибуты GCC применяются для передачи компоновщику следующей информации:

- слабые символы, которые могут быть заменены символами с тем же именем;
- символы, которые должны храниться в определенном разделе файла ELF, заданном в скрипте компоновщика;
- неявно используемые символы, которые не позволяют компоновщику отбросить символ, потому что он нигде не упоминается в коде.

Атрибут `weak` используется для определения слабых символов, которые могут быть переопределены в любом другом месте кода другим определением с тем же именем. Рассмотрим, например, следующее определение:

```
void __attribute__(weak) my_procedure(int x) {/* ничего не делать */}
```

В этом случае процедура определена так, чтобы ничего не делать, но ее можно переопределить где-нибудь еще в кодовой базе, определив ее снова под тем же именем, но на этот раз без атрибута `weak`:

```
void my_procedure(int x) { y = x; }
```

Компоновщик гарантирует, что финальный исполняемый файл будет содержать ровно одну копию каждого определенного символа, то есть копии без атрибута, если он доступен. Этот механизм дает возможность иметь

в коде несколько различных реализаций одной и той же функциональности, которые можно выбирать, включая разные объектные файлы на этапе компоновки. Это особенно полезно при разработке кода, переносимого на разные целевые устройства, при сохранении тех же абстракций.

Помимо разделов по умолчанию, требуемых в описании ELF, могут быть добавлены пользовательские разделы для хранения определенных символов, таких как функции и переменные, по фиксированным адресам памяти. Это полезно при сохранении данных в начале страницы флеш-памяти, которые могут быть загружены в другое время, чем сама прошивка. В некоторых случаях так сохраняют индивидуальные настройки целевого устройства.

Использование специально указанного атрибута раздела GCC при определении символа гарантирует, что символ окажется в желаемой позиции финальной прошивки. Разделы могут иметь собственные имена, если в компоновщике существует запись, позволяющая их найти. Атрибут `section` можно добавить к определению символа следующим образом:

```
const uint8_t
 __attribute__((section(".keys")))
 private_key[KEY_SIZE] = {0};
```

В этом примере массив помещается в секцию `.keys`, что также требует отдельной записи в сценарии компоновщика.

Считается хорошей практикой, когда компоновщик отбрасывает неиспользуемые символы в финальном образе, особенно при применении сторонних библиотек, которые не полностью используются встроенным приложением. Это можно сделать в GCC с помощью сборщика мусора компоновщика, активируемого с помощью параметра командной строки `-gc-sections`. Если указан этот флаг, разделы, которые не используются в коде, автоматически отбрасываются, а неиспользуемые символы не включаются в финальный образ.

Чтобы компоновщик не отбрасывал символы, связанные с определенным разделом, для них следует указать атрибут `used`, который помечает символ как неявно используемый программой. В одном объявлении можно указать несколько атрибутов, разделенных запятыми, как показано ниже:

```
const uint8_t __attribute__((used,section(".keys")))
   private_key[KEY_SIZE] = {0};
```

В этом примере атрибуты указывают как на то, что массив `private_key` принадлежит к разделу `.keys`, так и на то, что сборщик мусора компоновщика не должен отбрасывать его, поскольку он помечен как используемый.

Простой скрипт компоновщика для встраиваемого целевого устройства определяет по крайней мере два раздела, отображаемых в `RAM` и `FLASH`, и экспортирует некоторые предопределенные символы, чтобы проинструктировать ассемблер инструментария об областях памяти. Встраиваемая система на «голом железе», построенная с помощью инструментария GNU, обычно начинается с раздела `MEMORY`, описывающего сопоставление двух разных областей в системе, например так:

```
MEMORY {
  FLASH(rx) : ORIGIN = 0x00000000, LENGTH=256k
  RAM(rwx) : ORIGIN = 0x20000000, LENGTH=64k
}
```

Здесь описаны две области памяти, используемые в системе. Первый блок размером 256 Кбайт отображается во `FLASH`, а флаги `r` и `x` указывают, что область доступна для операций чтения и выполнения соответственно. Вместе эти флаги образуют атрибут «только для чтения» для всей области и гарантируют, что там не будут размещены разделы с изменяемыми данными. С другой стороны, доступ к `RAM` можно получить напрямую в режиме записи, следовательно, переменные будут помещены в раздел внутри этой области. В этом конкретном примере `FLASH` отображается в начале адресного пространства, в то время как `RAM` отображается начиная с 512 МБ. Каждое целевое устройство имеет свое сопоставление адресного пространства и размер флеш-памяти/ОЗУ, что делает скрипт компоновщика аппаратно зависимым.

Как упоминалось ранее в данной главе, доступ к разделам ELF `.text` и `.rodata` возможен только для чтения, поэтому их можно безопасно хранить в области `FLASH`, поскольку они не будут изменены во время работы устройства. С другой стороны, и `.data`, и `.bss` должны быть отображены в оперативной памяти, чтобы обеспечить возможность их изменения.

В сценарий можно добавить дополнительные пользовательские разделы с указанием хранить их в определенном месте памяти. Сценарий компоновщика также может экспортировать символы, связанные с определенной позицией в памяти или с длиной разделов памяти с динамическим размером – так называемые внешние символы, доступные в исходном коде C.

Второй блок операторов в скрипте компоновщика называется `SECTIONS` и описывает размещение разделов в определенных позициях заданных областей памяти. Символ «точка» (.), когда он связан с переменной в скрипте, представляет текущую позицию в области, которая постепенно заполняется от более низких доступных адресов.

Для каждого раздела должна быть указана область, на которую он отображается. Следующий пример двоичного исполняемого файла (хотя и неполный) показывает, как с помощью сценария компоновщика могут быть развернуты различные разделы. Разделы `.text` и `.rodata` отображаются во флеш-памяти:

```
SECTIONS
{
    /* Раздел text (код и данные только для чтения) */
    .text :
    {
        . = ALIGN(4);
        _start_text = .;
        *(.text*) /* код */
        . = ALIGN(4);
        _end_text = .;
        *(.rodata*) /* данные только для чтения */
```

```
        . = ALIGN(4);
        _end_rodata = .;
} > FLASH
```

Изменяемые разделы отображаются в ОЗУ, и здесь нужно отметить два особых случая.

Ключевое слово AT применяется для указания компоновщику адреса загрузки, указывающего на область, где хранятся исходные значения переменных в .data, в то время как фактические адреса, используемые при выполнении, находятся в другой области памяти. Более подробно об адресе загрузки и виртуальном адресе для раздела .data говорится в главе 4.

Атрибут NOLOAD, используемый для раздела .bss, гарантирует, что в файле ELF для этого раздела не будут сохранены предопределенные значения. Неинициализированные глобальные и статические переменные отображаются компоновщиком в область оперативной памяти (RAM), которая выделяется компоновщиком:

```
_stored_data = .;
.data: AT(__stored_data)
{
    . = ALIGN(4);
    _start_data = .;
    *(.data*)
    . = ALIGN(4);
    _start_data = .;
} > RAM

.bss (NOLOAD):
{
    . = ALIGN(4);
    _start_bss = .;
    *(.bss*)
    . = ALIGN(4);
    _end_bss = .;
} > RAM
```

Альтернативный способ заставить компоновщик сохранить разделы в финальном исполняемом файле, избегая их удаления сборщиком мусора компоновщика, – это использование инструкции KEEP для пометки разделов. Это просто альтернатива описанному ранее механизму __attribute__((used)):

```
.keys :
{
    . = ALIGN(4);
    *(.keys*) = .;
    KEEP(*(.keys*));
} > FLASH
```

Полезно и желательно, чтобы компоновщик создавал файл .map вместе с финальным двоичным файлом. Это делается путем добавления параметра -Map=filename к шагу компоновки, например так:

```
$ arm-none-eabi-ld -o image.elf object1.o object2.o
-T linker_script.ld -Map=map_file.map
```

Файл `.map` содержит расположение и описание всех символов, сгруппированных по разделам. Это полезно для поиска определенного местоположения символов в образе, а также для проверки того, что нужные символы не были случайно отброшены из-за неправильной конфигурации.

Инструменты кросс-компиляции предоставляют стандартные библиотеки C для общих функций, таких как манипуляции со строками или объявления стандартных типов. По сути, это подмножество библиотечных вызовов, доступных в пространстве приложений операционной системы, включая стандартные функции ввода/вывода. Бэкенд-реализация этих функций часто остается за приложениями, поэтому вызов функции из библиотеки, требующей взаимодействия с оборудованием, например `printf`, подразумевает, что функция записи, обеспечивающая окончательную передачу в периферийное устройство, реализована вне этой библиотеки.

Реализация внутренней функции записи определяет, какой канал будет действовать как стандартный вывод для встроенного приложения. Компоновщик может автоматически разрешать зависимости от вызовов стандартной библиотеки, используя встроенную реализацию `newlib`. Чтобы исключить стандартные символы библиотеки C из процесса компоновки, можно добавить параметр `-nostdlib` к параметрам, передаваемым в GCC на этапе компоновки.

2.3.4. Преобразование двоичного формата

Несмотря на то что все скомпилированные символы представлены в двоичном формате, файл ELF имеет префикс заголовка, который содержит описание содержимого и указатели на позиции, с которых начинаются разделы в файле. Вся эта дополнительная информация не требуется для запуска на целевом устройстве, поэтому файл ELF, созданный компоновщиком, должен быть преобразован в простой двоичный файл. Инструмент под названием `objcopy` преобразовывает образы прошивок из одного стандартного формата в другие. Обычно выполняется преобразование ELF в простой двоичный образ без изменения символов. Чтобы конвертировать образ из ELF в двоичный формат, выполните следующую команду:

```
$ arm-none-eabi-objcopy -I elf -O binary image.elf image.bin
```

Здесь создается новый файл с именем `image.bin` из символов, содержащихся в исходном исполняемом файле ELF; полученный файл готов к загрузке в целевое устройство.

Даже если нет возможности напрямую загрузить образ прошивки в целевое устройство при помощи сторонних инструментов, можно загрузить его через отладчик и выгрузить на флеш-адрес. Исходный файл ELF также подходит в качестве объекта анализа для диагностических инструментов в инструментарии GNU, таких как `nm` и `readelf`, которые отображают символы

в каждом модуле с их типом и относительным адресом в двоичном образе. Более того, применяя инструмент `objdump` к финальному образу или даже к отдельным объектным файлам, можно получить некоторые сведения об образе, включая визуализацию всего ассемблерного кода, с помощью параметра `-d disassemble`:

```
$ arm-none-eabi-objcopy -I elf -O binary image.elf image.bin
```

Таким образом, рассмотрены все компоненты, необходимые для запуска, отладки и анализа скомпилированного программного обеспечения на целевом микроконтроллере. Чтобы выгрузить в него образ прошивки или запустить сеанс отладки, потребуются дополнительные специфические инструменты, описанные в следующем разделе.

2.4. Взаимодействие с целевым устройством

При разработке и отладке кода программы доступ к встраиваемым платформам обычно осуществляется через интерфейс JTAG или SWD. Через эти каналы связи можно загрузить программное обеспечение во флеш-память целевого устройства и получить доступ к встроенным функциям отладки. Некоторыми распространенными автономными адаптерами JTAG/SWD можно управлять через USB с хоста, в то время как специальные макетные платы для разработки и тестирования бывают оснащены дополнительным чипом для взаимодействия с каналом JTAG, который подключается к хосту через USB.

Мощным универсальным инструментом с открытым исходным кодом для доступа к функциям JTAG/SWD на целевом устройстве является Open On-Chip Debugger (OpenOCD). После настройки он создает локальные сокеты, которые можно использовать в качестве командной консоли и для взаимодействия с внешним интерфейсом отладчика. Некоторые макетные платы поставляются с дополнительными интерфейсами для связи с основным процессором. Например, макетные платы STMicroelectronics для Cortex-M редко поставляются без чипа с поддержкой протокола ST-Link, который обеспечивает прямой доступ к функциям отладки и управления флеш-памятью. Благодаря своей гибкой бэкенд-части OpenOCD может взаимодействовать с подобными устройствами, используя различные типы транспортных протоколов и физические интерфейсы, включая ST-Link. Поддерживается несколько различных плат, а файлы конфигурации можно найти с помощью функции поиска OpenOCD.

При запуске OpenOCD открывает два сокета локального TCP-сервера на заранее настроенных портах, предоставляя услуги связи с целевой платформой. Один сокет предоставляет интерактивную командную консоль, к которой можно получить доступ через Telnet, а другой – сервер GDB, используемый для удаленной отладки, как будет описано в следующем разделе.

OpenOCD поставляется вместе с двумя наборами файлов конфигурации, которые описывают целевой микроконтроллер и периферийные устройства

(в каталоге target/), а также интерфейс отладки, используемый для связи с ним через JTAG или SWD (в каталоге interface/). Третий набор конфигурационных файлов (в каталоге board/) содержит файлы конфигураций для широко известных систем, таких как макетные платы, оснащенные интерфейсным чипом.

Например, чтобы настроить OpenOCD под целевую макетную плату STM32F746-Discovery, можно использовать следующий файл конфигурации openocd.cfg:

```
telnet_port 4444
gdb_port 3333
source [find board/stm32f7discovery.cfg]
```

Файл конфигурации для конкретной платы, который был импортирован из openocd.cfg через директиву source, указывает OpenOCD использовать интерфейс ST-Link для связи с целевым устройством и устанавливает все параметры, специфичные для семейства микроконтроллеров STM32F7.

Два порта, указанных в основном файле конфигурации с помощью директив telnet_port и gdb_port, указывают OpenOCD открыть два TCP-сокета прослушивания.

К первому сокету, часто называемому консолью монитора, можно получить доступ, подключившись к локальному TCP-порту 4444 с помощью клиента Telnet из командной строки:

```
$ telnet localhost 4444
Open On-Chip Debugger
>
```

Последовательность директив OpenOCD для инициализации, очистки флеш-памяти и передачи образа начинается так:

```
> init
> halt
> flash probe 0
```

Выполнение останавливается в начале образа программного обеспечения. После команды probe происходит инициализация флеш-памяти, и OpenOCD распечатывает некоторую информацию, включая адрес, отображаемый для записи во флеш-память. В нашем примере с STM32F746 отображается следующая информация:

```
device id = 0x10016449
flash size = 1024kbytes
flash "stm32f2x" found at 0x08000000
```

Геометрию флеш-памяти можно получить с помощью следующей команды:

```
> flash info 0
```

Для STM32F746 будет получен вывод в консоль:

```
#0 : stm32f2x at 0x08000000, size 0x00100000, buswidth 0,
chipwidth 0
```

```
# 0: 0x00000000 (0x8000 32kB) not protected
# 1: 0x00008000 (0x8000 32kB) not protected
# 2: 0x00010000 (0x8000 32kB) not protected
# 3: 0x00018000 (0x8000 32kB) not protected
# 4: 0x00020000 (0x20000 128kB) not protected
# 5: 0x00040000 (0x40000 256kB) not protected
# 6: 0x00080000 (0x40000 256kB) not protected
# 7: 0x000c0000 (0x40000 256kB) not protected
STM32F7[4|5]x - Rev: Z
```

Эта флеш-память содержит восемь секторов. Если целевое устройство поддерживает такую функцию, флеш-память можно полностью стереть, выполнив следующую команду из консоли:

```
> flash erase_sector 0 0 7
```

После очистки флеш-памяти можно загрузить в нее скомпонованный образ прошивки в простом двоичном формате с помощью директивы `flash write_image`. Поскольку простой двоичный формат не содержит никакой информации об адресе назначения в отображаемом пространстве, начальный адрес во флеш-памяти должен быть указан в качестве последнего аргумента, как показано ниже:

```
> flash write_image /path/to/image.bin 0x08000000
```

Эти директивы могут быть добавлены к файлу `openocd.cfg` или к другим файлам конфигурации, чтобы автоматизировать все шаги, необходимые для определенного действия, такого как стирание флеш-памяти и загрузка обновленного образа.

Некоторые производители оборудования предлагают собственный набор инструментов для взаимодействия с устройствами. Устройства STMicroelectronics можно программировать с помощью утилит ST-Link, проекта с открытым исходным кодом, который включает в себя инструмент для прошивки (`st-flash`) и аналог сервера GDB (`st-util`). Некоторые платформы имеют встроенные загрузчики, которые принимают альтернативные форматы или процедуры передачи двоичных файлов. Типичным примером является протокол *обновления прошивки устройства* (device firmware upgrade, DFU), который представляет собой механизм развертывания прошивки на целевых устройствах через USB. Приложением на стороне хоста служит свободно распространяемый инструмент `dfu-util`.

Каждый инструмент, как универсальный, так и специальный, преследует одну и ту же цель – взаимодействие с устройством и предоставление интерфейса для отладки кода, хотя часто применяется интерфейс среды разработки.

Большинство IDE, предоставляемых производителями для работы с определенным семейством микроконтроллеров, предлагают собственные инструменты или поддерживают сторонние приложения для доступа к отображению флеш-памяти и управления выполнением программы на целевом устройстве. Хотя, с одной стороны, они довольно эффективно скрывают ненужную сложность операций и обеспечивают загрузку прошивки в один клик, с другой – вообще не предоставляют удобного интерфейса для про-

граммирования нескольких устройств одновременно или, по крайней мере, не позволяют делать это эффективно, когда речь идет о производстве больших партий, требующих загрузки исходной заводской прошивки.

Знание механизмов и процедур, выполняемых из интерфейса командной строки, позволяет понять, что происходит за кулисами каждый раз, когда новая прошивка загружается в устройство, и предвидеть проблемы, которые могут повлиять на процесс разработки на этом этапе.

2.4.1. Сеанс GDB

Независимо от стараний программиста или сложности проекта, большая часть времени разработки будет потрачена на то, чтобы убедиться, что программа работает правильно, или – что наиболее вероятно – разобраться, почему программа работает не так, как ожидалось. Это особенно актуально для только что созданного кода. Отладчик – самый мощный инструмент в наборе инструментов программиста, позволяющий напрямую взаимодействовать с процессором, задавать точки останова, пошагово управлять потоком выполнения инструкций и проверять значения регистров ЦП, локальных переменных и областей памяти. Навык работы с отладчиком означает экономию времени, потраченного на попытки выяснить, что происходит, и более эффективный поиск ошибок в коде.

Инструментарий `arm-none-eabi` предоставляет отладчик GDB, способный интерпретировать структуру памяти и регистров удаленного устройства, и к нему можно получить доступ с помощью тех же интерфейсов, что и к GDB хост-компьютера, при условии что серверная часть отладчика может взаимодействовать со встроенной платформой, используя OpenOCD или аналогичный хост-инструмент, обеспечивающий связь с устройством через протокол сервера GDB. Как было сказано ранее, OpenOCD можно настроить для предоставления интерфейса сервера GDB, который в рассматриваемой конфигурации доступен на порту 3333.

После запуска `arm-none-eabi-gdb` можно подключиться к инструменту с помощью команды GDB `target`. Во время работы OpenOCD к серверу GDB можно выполнить такие команды:

```
> target remote localhost:3333
```

Все команды GDB можно сокращать, поэтому часто используют краткий вариант команды:

```
> tar rem :3333
```

После подключения целевое устройство обычно останавливает выполнение прошивки, позволяя GDB получить информацию об инструкции, которая в данный момент выполняется, трассировке стека и значениях регистров ЦП.

С этого момента интерфейс отладчика можно использовать как обычно для пошагового выполнения кода, задания точек останова и наблюдения,

а также проверки и изменения регистров ЦП и доступных для записи областей памяти во время выполнения.

GDB можно использовать полностью из интерфейса командной строки, используя ярлыки и команды для запуска и остановки выполнения, а также для доступа к памяти и регистрам.

В табл. 2.5 перечислены некоторые команды GDB, доступные в сеансе отладки, и дано краткое объяснение их использования.

Таблица 2.5. Краткий перечень команд GDB, доступных в сеансе отладки

Команда	Описание
`file name`	Загружает все символы из ELF-файла в файловой системе хоста. Если ELF-файл был скомпилирован с опцией `GCC -g`, он будет содержать всю информацию, необходимую для отладки
`load`	Выгружает все загруженные на данный момент символы в целевое устройство. Применяется для записи новой прошивки в устройство во время сеанса отладки
`mon`	Доступные команды монитора в зависимости от платформы. Интерфейс монитора OpenOCD предоставляет такие команды, как `mon reset` и `mon init` для перезапуска ядра и начала выполнения программы

`break (b)`
`b function_name`
`b line_number`
`b file.c:line_num`
`b *address`
	Помещает точку останова в заданную позицию кода. Позиция может быть задана несколькими способами. Когда выполняется инструкция в точке останова, ядро ЦП временно останавливается, и управление передается отладчику

`watch (w) address`	Напоминает `break`, но вместо размещения точки останова в заданном месте кода наблюдает за переменной в памяти по заданному адресу (точка наблюдения) и прерывает выполнение при каждом изменении значения этой переменной. Применяется для отслеживания мест в коде, изменяющих определенное значение в памяти
`info b`	Предоставляет информацию о точках останова и точках наблюдения, установленных в рамках текущего сеанса отладки
`delete (d) n`	Удаляет точку останова `n`
`print (p) …`	Показывает значение переменной или выражения C, которое должно быть вычислено с использованием переменных и адресов памяти
`display …`	Аналогично `print`, но обновляет значение выражения всякий раз, когда управление возвращается к отладчику. Удобно для отслеживания изменений, происходящих при пошаговом выполнении кода
`next (n)`	Выполняет следующую инструкцию. Если следующим шагом идет вызов функции, автоматически помещает точку останова в место возврата и передает управление отладчику только после возврата из вызываемой функции
`step (s)`	Выполняет следующую инструкцию. Если следующим шагом идет вызов функции, автоматически помещает точку останова в место возврата и передает управление отладчику только после возврата из вызываемой функции

Таблица 2.5 (окончание)

Команда	Описание
`stepi (si)`	Аналогично `step`, но выполняет только одну инструкцию машинного кода, а не выражение целиком, в которое входит эта инструкция. Удобно для выполнения одиночных инструкций
`continue (c)`	Возобновляет выполнение кода на целевом устройстве с текущей позиции. Управление возвращается отладчику, если достигнута точка останова или в консоли нажата комбинация клавиш **Ctrl**+**C**
`set var=…` `set $cpu_reg=…`	Присваивает желаемое значение (или результат вычисления выражения) переменной в памяти или регистру ЦП, имя которого предваряет символ `$`
`backtrace (bt)`	Просмотр содержимого стека в обратном порядке, включая трассировку вызовов из текущего положения
`up` `down`	Перемещает указатель стека вызовов вверх или вниз. Для перемещения в контекст вызывающей функции применяется `up`, в контекст вызываемой функции – `down`

GDB – очень мощный и функциональный отладчик, и команды, показанные в этом разделе, представляют собой лишь малую часть его реальных возможностей. Очень полезно ознакомиться с другими функциями, предлагаемыми GDB, изучив его руководство, чтобы составить набор команд, который лучше всего соответствует вашим потребностям.

IDE часто предлагают отдельный графический режим для работы с сеансами отладки, который интегрирован с редактором и позволяет задавать точки останова, отслеживать переменные и исследовать содержимое областей памяти, пока система работает в режиме отладки.

2.5. Тестирование

Одной только отладки или даже простого анализа выходных данных часто бывает недостаточно для проверки поведения системы и выявления проблем и нежелательных эффектов в коде. Для проверки реализации отдельных компонентов, а также поведения всей системы в различных условиях могут использоваться разные подходы. Хотя в некоторых случаях результаты можно измерить непосредственно с хост-компьютера, в более конкретных случаях часто бывает трудно воспроизвести точный сценарий или получить необходимую информацию из выходных данных системы.

Внешние инструменты могут пригодиться, особенно при анализе коммуникационных интерфейсов и сетевых устройств в более сложной распределенной системе. В других случаях отдельные модули можно тестировать не по назначению, используя смоделированные или эмулированные среды для запуска небольших частей базы кода.

В этом разделе рассматриваются различные тесты, стратегии проверки и инструменты, чтобы предоставить решения для любого сценария.

2.5.1. Функциональные тесты

Подготовка набора тестов перед разработкой кода обычно считается оптимальной практикой в современном программировании. Первоочередная разработка тестов не только ускоряет этапы разработки, но и улучшает структуру рабочего процесса. Если с самого начала установить четкие и измеримые цели, будет труднее допустить концептуальные ошибки при разработке отдельных компонентов, а у разработчика будет стимул к более четкому разделению между модулями. В частности, у разработчика встраиваемых систем меньше возможностей проверить правильность поведения системы путем прямого взаимодействия с нею; поэтому *разработка через тестирование* (test-driven development, TDD) является предпочтительным подходом для проверки отдельных компонентов, а также функционального поведения всей встраиваемой системы, если ожидаемые результаты могут быть непосредственно измерены из хост-системы.

Следует учитывать, что тестирование часто вводит зависимости от конкретного оборудования, и иногда выходные данные встраиваемой системы могут быть проверены только с помощью специального оборудования или в очень уникальном и своеобразном сценарии использования. Во всех этих случаях обычная парадигма TDD менее применима, и максимальную пользу может принести модульная структура проекта, позволяющая тестировать как можно больше компонентов в синтетической среде, такой как эмуляторы или платформы модульного тестирования.

Разработка тестов часто включает в себя программирование хоста, чтобы он мог получать информацию о работающем целевом устройстве во время выполнения встроенного программного обеспечения или во время текущего сеанса отладки, когда код на целевом устройстве выполняется между точками останова. Прошивка устройства может предусматривать немедленный вывод тестовых данных через коммуникационный интерфейс, такой как последовательный порт UART. В свою очередь, поток тестовых данных может быть проанализирован хостом. Обычно удобнее писать инструменты тестирования для хоста, используя интерпретируемый язык программирования более высокого уровня, чтобы лучше организовать тестовые примеры и объединять синтаксический анализ результатов тестирования с использованием регулярных выражений. Python, Perl, Ruby и другие языки со схожими характеристиками обычно хорошо подходят для этой цели, в том числе благодаря наличию библиотек и компонентов, предназначенных для сбора и анализа результатов тестирования и взаимодействия с механизмами непрерывной интеграции. Хорошая организация тестовой и проверочной инфраструктуры больше всего способствует стабильности проекта, потому что ухудшение качества работы может быть обнаружено в нужный момент только в том случае, если все существующие тесты повторяются при каждой модификации. Постоянное выполнение всех тестовых примеров во время разработки не только повышает вероятность своевременного обнаружения нежелательных эффектов, но и упрощает рефакторинг компонентов на любой стадии жизни проекта.

Эффективность операций тестирования является ключевым моментом, поскольку программирование встраиваемых систем представляет собой итеративный процесс, в котором несколько шагов повторяются снова и снова, а подход, требуемый от разработчиков, гораздо более прогностический, чем реактивный.

2.5.2. Аппаратные инструменты

Если и существует инструмент, абсолютно незаменимый в практике разработчиков встраиваемого программного обеспечения, то это *логический анализатор* (logic analyzer). Изучая входные и выходные сигналы на выводах микроконтроллера, можно определить поведение сигналов, их синхронизацию и даже кодирование на уровне отдельных битов в протоколах обмена данными. Большинство логических анализаторов могут распознавать и декодировать последовательности символов, считывая логические уровни на линиях, что часто является наиболее эффективным способом проверки правильности реализации протоколов связи с периферийными устройствами и сетевыми конечными точками. Раньше логический анализатор представлял собой отдельный компьютер, но современные анализаторы часто реализованы в виде компактных электронных модулей, которые можно подключать к хост-компьютеру с помощью интерфейсов USB или Ethernet и использовать программное обеспечение на базе ПК для захвата и декодирования сигналов. Результатом этого процесса является полный дискретный анализ интересующих сигналов, которые оцифровываются с постоянной скоростью, а затем визуализируются на экране.

Хотя аналогичную задачу во многом могут выполнять осциллографы, их часто сложнее использовать, чем логические анализаторы при работе с дискретными сигналами. Тем не менее осциллограф является лучшим инструментом для анализа аналоговых сигналов, таких как аналоговый звук и связь по радиоканалу. В зависимости от задачи может быть удобнее использовать тот или иной инструмент, но в целом главное преимущество логического анализатора заключается в том, что он обеспечивает лучшее понимание дискретных сигналов. Логические анализаторы смешанных сигналов часто представляют собой хороший компромисс между гибкостью осциллографа и простотой и глубиной анализа дискретной логики сигналов.

Осциллографы и логические анализаторы часто применяются для отслеживания активности сигналов в определенном временном окне, что может быть сложно синхронизировать с работающим программным обеспечением. Вместо непрерывного захвата наблюдаемых сигналов начало захвата можно синхронизировать с физическим событием, таким как изменение значения цифрового сигнала в первый раз или превышение заданного порога аналоговым сигналом. Это делается путем настройки на приборе так называемого триггера запуска, который гарантирует, что захваченная информация охватывает только временной интервал, представляющий интерес для текущей диагностики.

2.5.3. Внешнее тестирование

Еще один эффективный способ ускорить разработку – максимально ограничить взаимодействие с реальным устройством. Это, конечно, не всегда возможно, особенно при разработке драйверов устройств, которые необходимо проверить на реальном оборудовании, но существуют инструменты и методологии для частичного тестирования программного обеспечения непосредственно на машине разработки.

Части кода, которые не зависят от ЦП, могут быть скомпилированы для архитектуры хост-машины и запущены напрямую, если их окружение правильно абстрагировано для имитации реальной среды. Программное обеспечение для тестирования может состоять из одной функции, и в этом случае модульный тест может быть создан специально для архитектуры разработки.

Модульные тесты (unit test) – это, как правило, небольшие приложения, которые проверяют поведение одного компонента, передавая ему известные входные данные и проверяя выходные данные. Для ОС Linux доступно несколько инструментов, помогающих в разработке модульных тестов. Библиотека `check` предоставляет интерфейс для определения модульных тестов путем разработки нескольких макросов препроцессора. В результате получаются небольшие автономные приложения, которые можно запускать каждый раз при изменении кода непосредственно на хост-компьютере. Те компоненты системы, от которых зависит тестируемая функция, абстрагируются с помощью *макетов объекта* (mock). Например, следующий код обнаруживает и отбрасывает специальную управляющую последовательность **Esc**+**C**, поступающую на вход из последовательного интерфейса, читая последовательный канал до тех пор, пока не будет возвращен символ `\0`:

```
int serial_parser(char *buffer, uint32_t max_len)
{
  int pos = 0;
  while (pos < max_len) {
    buffer[pos] = read_from_serial();
    if (buffer[pos] == (char)0)
      break;
    if (buffer[pos] == ESC) {
      buffer[++pos] = read_from_serial();
      if (buffer[pos] == 'c')
        pos = pos - 1;
        continue;
      }
      pos++;
    }
    return pos;
  }
```

Набор модульных тестов для проверки этой функции может выглядеть следующим образом:

```
START_TEST(test_plain) {
  const char test0[] = "hello world!";
```

```
  char buffer[40];
  set_mock_buffer(test0);
  fail_if(serial_parser(buffer, 40) != strlen(test0));
  fail_if(strcmp(test0,buffer) != 0);
}
END_TEST
```

Каждый тестовый пример может размещаться в своем блоке `START_TEST()`/`END_TEST` и предоставлять другую начальную конфигурацию:

```
START_TEST(test_escape) {
  const char test0[] = "hello world!";
  const char test1[] = "hello \033cworld!";
  char buffer[40];
  set_mock_buffer(test1);
  fail_if(serial_parser(buffer, 40) != strlen(test0));
  fail_if(strcmp(test0,buffer) != 0);
}
END_TEST

START_TEST(test_other) {
  const char test2[] = "hello \033dworld!";
  char buffer[40];
  set_mock_buffer(test2);
  fail_if(serial_parser(buffer, 40) != strlen(test2));
  fail_if(strcmp(test2,buffer) != 0);
}
END_TEST
```

Здесь первый тест `test_plain` проверяет, что строка без escape-символов будет правильно проанализирована. Второй тест проверяет, что escape-последовательность будет пропущена, а третий проверяет, что аналогичная escape-строка не испорчена выходным буфером.

Последовательная связь моделируется с помощью фиктивной функции, которая заменяет исходную функциональность `serial_read`, предоставляемую драйвером при выполнении кода на целевом устройстве. Это простой программный макет, передающий анализатору постоянный буфер, который можно повторно инициализировать с помощью вспомогательной функции `set_serial_buffer`. Код фиктивной функции выглядит так:

```
static int serial_pos = 0;
static char serial_buffer[40];

char read_from_serial(void) {
  return serial_buffer[serial_pos++];
}

void set_mock_buffer(const char *buf)
{
  serial_pos = 0;
  strncpy(serial_buffer, buf, 20);
}
```

Модульные тесты очень полезны для улучшения качества кода, но, конечно, достижение высокого охвата кода тестами требует большого количества времени и ресурсов. Функциональные тесты также можно запускать непосредственно в среде разработки, группируя функции в автономные модули и реализуя симуляторы, которые немного сложнее макетов для конкретных тестовых сценариев. В примере последовательного синтаксического анализатора можно было бы провести тестирование всей логики приложения поверх другого последовательного драйвера на хост-компьютере, который также может имитировать весь обмен по последовательной линии и взаимодействовать с другими компонентами в системы, такими как виртуальные терминалы и другие приложения, генерирующие входные последовательности.

При охвате большей части кода в рамках одного тестового случая увеличиваются сложность моделируемой среды и объем работы, необходимой для воспроизведения окружения встроенной системы на хост-компьютере. Тем не менее это хороший подход, особенно когда его можно использовать в качестве инструмента проверки на протяжении всего цикла разработки и даже интегрировать в автоматизированный процесс тестирования.

Иногда использование симулятора позволяет провести гораздо более полный набор тестов или может быть единственным приемлемым вариантом. Допустим, нам нужно провести тестирование встраиваемой системы, использующей GPS-приемник для позиционирования: тестирование логики приложения с отрицательными значениями широты невозможно, если находиться в Северном полушарии, поэтому использование симулятора данных, поступающих от приемника, будет наиболее доступным способом убедиться, что наше конечное устройство не перестанет работать при пересечении экватора.

2.5.4. Эмуляторы

Еще один действенный подход к запуску кода на машине разработки, гораздо менее инвазивный для нашей кодовой базы и ослабляющий особые требования к переносимости, – это эмуляция всей платформы на хост-компьютере. *Эмулятор* – это компьютерная программа, которая может воспроизводить функциональность всей системы, включая ее основной ЦП, память и набор периферийных устройств. Некоторые из современных гипервизоров виртуализации для ПК являются производными от QEMU, свободно распространяемого эмулятора программного обеспечения, способного виртуализировать целые системы, даже с архитектурой, отличной от архитектуры машины, на которой он работает. Поскольку он содержит полную реализацию набора инструкций для множества различных целей, QEMU может запускать образ прошивки, который был скомпилирован для целевого устройства, в отдельном процессе поверх операционной системы машины разработки. Одной из поддерживаемых целей, которая может запускать микрокод ARM Cortex-M3, является `lm3s6965evb`, старый микроконтроллер на базе Cortex-M, который

больше не рекомендуется производителем для новых разработок, но полностью поддерживается QEMU.

После создания двоичного образа с использованием `lm3s6965evb` в качестве цели и преобразования в исходный двоичный формат с помощью `objcopy` можно запустить полностью эмулируемую систему, вызвав QEMU следующим образом:

```
$ qemu-system-arm -M lm3s6965evb --kernel image.bin
```

Параметр `--kernel` предписывает эмулятору запускать образ при запуске, и, хотя это может показаться неправильным, он называется `kernel` (ядро), потому что QEMU широко используется для эмуляции автономных систем Linux для других синтетических целей. Точно так же удобный сеанс отладки можно запустить с помощью встроенного в QEMU сервера GDB с параметром `-gdb`, который может остановить эмулируемую систему, пока к ней не подключится клиент GDB:

```
$ qemu-system-arm -M lm3s6965evb --kernel image.bin -nographic
-S -gdb tcp::3333
```

Как и в случае с физическим целевым устройством, можно подключить `arm-none-eabi-gdb` к TCP-порту `3333` на локальном хосте и начать отладку образа программного обеспечения, как если бы он был загружен на реальное устройство.

Ограничение подхода с эмуляцией заключается в том, что QEMU можно использовать только для отладки общих функций, не требующих взаимодействия с реальным современным оборудованием. Тем не менее запуск QEMU с целевым процессором Cortex-M3 позволяет быстро ознакомиться с основными функциями Cortex-M, такими как управление памятью, обработка системных прерываний и режимы процессора, поскольку многие функции ЦП Cortex-M эмулируются точно.

Более точной эмуляции микроконтроллерных систем можно добиться с помощью Renode (https://renode.io). Renode – это настраиваемый эмулятор с открытым исходным кодом для множества различных микроконтроллеров и встраиваемых систем на базе ЦП. При помощи Renode можно эмулировать периферийные устройства, датчики, светодиоды и даже беспроводные и проводные интерфейсы для соединения нескольких эмулируемых систем и хост-сети.

Renode – это настольное приложение с консолью командной строки. При вызове из командной строки ему нужно предоставить один файл конфигурации с несколькими платформами и конфигурациями платы разработки, расположенными в каталоге `/scripts`. Это означает, что можно запустить, например, эмулятор платы STM32F4 Discovery, выполнив следующую команду:

```
$ renode /opt/renode/scripts/single-node/stm32f4_discovery.resc
```

Эта команда загрузит демонстрационную прошивку в эмулированную флеш-память микроконтроллера STM32F4 и перенаправит ввод-вывод одного из эмулируемых последовательных портов UART на консоль в новом

окне. Чтобы запустить демонстрацию, нужно ввести команду `start` в консоли Renode.

Пример скрипта поставляется с демообразом прошивки под управлением Contiki OS. Образ прошивки загружается скриптом через команду Renode:

```
sysbus LoadELF $bin
```

где `$bin` – это переменная, указывающая на путь (или URL-адрес) ELF-файла прошивки для загрузки в эмулируемую флеш-память. Этот параметр, а также порт анализатора UART и другие специальные команды, которые должны выполняться при запуске эмулятора, можно легко изменить, настроив файл сценария.

В Renode интегрирован сервер GDB, который можно запустить из консоли Renode или сценария перед запуском эмуляции, например с помощью следующей команды:

```
machine StartGdbServer 3333
```

В данном случае `3333` – это TCP-порт, который будет прослушивать сервер GDB, как и в других случаях с QEMU или отладчиком на физическом устройстве.

В отличие от QEMU, который представляет собой эмулятор базового уровня, Renode – это проект, созданный с целью оказания помощи разработчикам встраиваемых систем на протяжении всего жизненного цикла разработки. Возможность эмулировать различные полные платформы, создавая макеты для датчиков на нескольких архитектурах, включая RISC-V, делает его уникальным инструментом для быстрой автоматизации тестирования на нескольких объектах или тестирования систем даже при отсутствии физического оборудования.

И последнее, но не менее важное: благодаря собственному языку сценариев Renode идеально совмещается с системами автоматизации тестирования, где можно запускать, останавливать и возобновлять выполнение эмулируемого устройства, а также изменять конфигурацию всех устройств и периферийных устройств во время выполнения теста.

Подходы, предлагаемые для определения стратегий тестирования, учитывают различные сценарии. Здесь был представлен ряд возможных решений для проверки программного обеспечения, от лабораторного оборудования до тестов в смоделированных и эмулируемых средах, а на разработчика возлагается выбор наиболее подходящих средств тестирования в каждом конкретном случае.

2.6. Заключение

В этой главе представлены средства разработки встраиваемых систем. Рассмотрен подход, который помогает начать работу с инструментами и утилитами, необходимыми для связи с аппаратной платформой. Использование

подходящих инструментов может упростить разработку встраиваемых систем и сократить итерации рабочего процесса.

В следующей главе даны рекомендации по организации рабочего процесса при работе в больших командах. Основываясь на реальном опыте, предлагаются решения для разделения и организации задач, выполнения тестов, повторения этапов проектирования, а также разработки и реализации проекта встраиваемой системы.

Часть II

БАЗОВАЯ АРХИТЕКТУРА ВСТРАИВАЕМЫХ СИСТЕМ

В этой части книги продолжается знакомство с основными принципами разработки программного обеспечения встраиваемых систем, а затем шаг за шагом рассматриваем примеры кода, необходимого для реализации механизмов загрузки и управления памятью, с особым акцентом на безопасном использовании памяти.

Вторая часть состоит из следующих глав:

- главы 3 «Шаблоны архитектуры встраиваемых систем»;
- главы 4 «Процедура загрузки»;
- главы 5 «Управление памятью».

Глава 3

Шаблоны архитектуры встраиваемых систем

Запуск встроенного проекта с нуля означает постепенное продвижение к окончательному решению, прохождение всех этапов исследований и разработок с достижением синергетического эффекта от взаимодействия между всеми частями.

На протяжении всех этих этапов нужно соблюдать определенные требования к процессу разработки ПО. Чтобы получить наилучшие результаты без излишних трудозатрат и накладных расходов, вы должны изучить специальные инструменты и овладеть правильными стратегиями.

В этой главе описывается один из возможных вариантов использования инструментов управления проектом и шаблоны проектирования, основанные на реальном опыте авторов. Описание этого подхода может помочь вам понять динамику работы в команде, сосредоточенной на создании встраиваемого устройства или программного решения.

Здесь обсуждаются следующие темы:

- управление конфигурацией проекта;
- организация исходного кода;
- жизненный цикл проекта встраиваемой системы;
- соображения безопасности.

К концу главы вы ознакомитесь с шаблонами архитектуры, полезными для проектирования системы на основе спецификаций и ограничений платформы.

3.1. Управление конфигурацией проекта

При работе в команде координация и синхронизация рабочих процессов очень важны для достижения оптимальной продуктивности. Отслеживание и контроль жизненного цикла проекта упрощают процесс разработки, сокращая время простоя и расходы.

К наиболее важным инструментам, помогающим управлять жизненным циклом разработки, относятся:

- контроль версий;
- отслеживание выпусков ПО;
- проверка кода;
- непрерывная интеграция.

Эти четыре категории могут быть представлены в разных вариантах. Для синхронизации исходного кода между разработчиками обычно применяется *система контроля версий*. *Системы отслеживания выпусков* (issue tracking system, ITS) обычно состоят из веб-платформ, которые отслеживают действия и известные ошибки системы. *Проверка кода* (code review, код-ревью) может основываться на специальных веб-приложениях и применяться с помощью правил в системах контроля версий.

Инструменты непрерывной интеграции обеспечивают автоматическое выполнение задач сборки и тестирования по расписанию, периодически или при изменении кода, собирая результаты тестирования и уведомляя разработчиков о возможной регрессии приложения.

3.1.1. Контроль версий

Независимо от того, работаете ли вы в одиночку или в большой команде разработчиков, правильное отслеживание хода разработки играет важную роль. Инструменты контроля версий позволяют разработчикам в любой момент откатить неудачные эксперименты одним нажатием кнопки, а просмотр истории дает четкое представление о развитии проекта на всем протяжении цикла разработки.

Система контроля версий (version control system, VCS) способствует продуктивному взаимодействию членов команды, упрощая операции слияния кода. Самая последняя официальная версия кода называется *мастер-веткой* (master branch) или *основной веткой* (main branch), в зависимости от используемой системы контроля версий. VCS предлагают, среди прочего, детальный контроль доступа и атрибуцию авторства вплоть до каждого коммита.

Одной из самых современных и широко используемых систем контроля версий с открытым исходным кодом является Git. Первоначально созданный как VCS для ядра Linux, Git поддерживает ряд функций, но, что наиболее важно, предоставляет гибкий механизм, позволяющий быстро и надежно переключаться между различными версиями и ветками, а также облегчает интеграцию конфликтующих модификаций в код.

Примечание

При описании работы с VCS в этой книге применятся терминология Git.

Коммит (commit) – это действие системы контроля версий, которое приводит к созданию новой версии репозитория. Репозиторий отслеживает последовательность коммитов и изменений, внесенных в каждую версию в иерархической структуре:

- *ветка* (branch): линейная последовательность коммитов является веткой кода;
- *HEAD*: системный указатель на последнюю версию в ветке;
- *мастер-ветка* (master): Git обозначает основную ветку разработки меткой master. Ветка master является основным направлением разработки. Исправления ошибок и незначительные изменения могут быть переданы непосредственно в эту ветку;
- *функциональные ветки* (feature branch): они создаются для автономных задач в ходе текущих экспериментов, которые в конечном итоге будут объединены с мастер-веткой. Функциональные ветки, если ими не злоупотреблять, идеально подходят для работы над задачей в небольшой подгруппе и могут упростить процесс проверки кода, позволяя разработчикам одновременно работать над отдельными ветками и сосредоточить проверку выполненных задач в виде отдельных *запросов на слияние* (merge request).

Операция слияния (merge operation) представляет собой объединение двух версий из двух разных ветвей, которые потенциально могли разойтись и привести к конфликту в коде на протяжении дальнейшей разработки. Некоторые слияния тривиальны и автоматически разрешаются системой контроля версий, в то время как для других может потребоваться исправление вручную.

Использование осмысленных и подробных сообщений коммитов улучшает читаемость истории репозитория и помогает отслеживать регрессии позже. Для отслеживания выпущенных и распространяемых промежуточных версий можно использовать *теги* (tag).

3.1.2. Отслеживание деятельности

Использование ITS упрощает отслеживание деятельности и задач. Некоторые инструменты могут быть напрямую связаны с системой контроля версий, чтобы задачи можно было связать с конкретными коммитами в репозитории, и наоборот. В целом это хорошая идея, так как можно получить детальный обзор того, что было изменено в коде для выполнения конкретной задачи.

Прежде всего разбивка спецификаций на короткие действия облегчает подход к разработке. В идеале задачи должны быть как можно меньше по объему и поддаваться группировке по категориям. В дальнейшем можно расставлять приоритеты исходя из промежуточных целей и с учетом наличия конечного оборудования. Созданные задачи должны быть сгруппированы в промежуточные этапы, которые некоторые инструменты называют схемами или снимками, чтобы можно было измерить общий прогресс в достижении промежуточного результата на основе прогресса, достигнутого в отдельных задачах.

ITS можно использовать для отслеживания актуальных проблем в проекте. Отчет об ошибке должен быть достаточно подробным, чтобы другие разработчики могли понять симптомы и воспроизвести поведение, доказывающее

наличие дефекта в коде. В идеале испытатели (ранние пользователи), а затем и конечные пользователи должны иметь возможность добавлять новые проблемы в систему отслеживания, которая затем применяется для отслеживания хода общения с командой разработчиков. Проекты с открытым исходным кодом, основанные на разработках сообщества, должны предоставлять пользователям общедоступный интерфейс ITS.

Действия по исправлению ошибок обычно имеют более высокий приоритет, чем задачи разработки, за исключением нескольких случаев, например когда ошибка является результатом временной аппроксимации, выполненной промежуточным прототипом, который, как ожидается, будет исправлен в следующей итерации. Когда ошибка ухудшает поведение системы, работа которой была заранее доказана, она должна быть помечена как *регрессия*. Это важно, потому что регрессии обычно обрабатывают иначе, чем обычные ошибки, поскольку их можно отследить с точностью до коммита с помощью инструментов контроля версий.

Платформы *управления репозиторием* (repository control) предоставляют несколько инструментов, включая просмотр истории исходного кода и функции отслеживания проблем, описанные ранее. GitLab – это бесплатная платформа управления репозиторием с открытым исходным кодом, которую можно использовать в качестве самостоятельного решения. Проекты сообщества часто размещаются на открытых общественных платформах, таких как GitHub, цель которых – облегчить участие в разработке проектов с открытым исходным кодом и бесплатного ПО.

3.1.3. Проверка кода

Средства проверки кода, которые часто бывают интегрированы в ITS, облегчают совместную работу членов команды, поощряя критический анализ изменений, предлагаемых в кодовой базе, что помогает обнаружить потенциальные проблемы до того, как предлагаемые изменения попадут в мастер-ветку. В зависимости от требований проекта регулярная проверка кода может быть рекомендована или даже вменена в обязанность команде для повышения качества кода и раннего обнаружения ошибок путем проверки квалифицированным специалистом.

При правильной интеграции с VCS можно установить порог обязательных положительных отзывов от других членов команды, прежде чем коммит будет рекомендован для слияния с мастер-веткой. С помощью таких инструментов, как Gerrit, интегрированных с VCS, можно назначить проверку каждого отдельного коммита в основной ветке. В зависимости от размера вносимых изменений этот механизм может привести к некоторым ненужным накладным расходам, поэтому в большинстве случаев имеет смысл не проверять каждое мелкое изменение, а сгруппировать изменения, вносимые функциональной веткой в целом, чтобы облегчить проверку, когда она предлагается для слияния с мастер-веткой. Механизмы, основанные на запросах на слияние, дают рецензентам кода обзор изменений, внесенных в течение всей разработки предлагаемой модификации. В случае проектов с открытым

исходным кодом, которые принимают дополнения от сторонних разработчиков, проверки кода являются необходимым шагом для проверки изменений, поступающих от менее надежных участников. Проверка кода – самый мощный инструмент для предотвращения внедрения вредоносного кода, который может быть скрыт от обнаружения утилитами автоматического тестирования и анализа кода.

3.1.4. Непрерывная интеграция

Как упоминалось ранее, подход, основанный на тестировании, имеет решающее значение при разработке встраиваемых систем. Автоматизация тестов – лучший способ оперативно выявлять регрессии и дефекты в целом, пока идет разработка. Используя сервер автоматизации, такой как Jenkins, можно запланировать несколько действий или заданий, которые будут выполняться оперативно (например, при каждом коммите), периодически (например, каждый вторник в 01:00) или вручную по запросу пользователя. Вот несколько примеров заданий, которые можно автоматизировать для повышения эффективности работы над проектом встраиваемой системы:

- модульные тесты на машине разработки;
- тесты для проверки системы;
- функциональные тесты в смоделированной среде;
- функциональные тесты на физической целевой платформе;
- тесты на стабильность;
- статический анализ кода;
- создание документации;
- назначение тегов и номеров версий, сборка дистрибутивов.

Во время проектирования необходимо строго придерживаться заданного уровня качества и разработать под него соответствующие тесты. Покрытие кода модульным тестом можно измерить с помощью `gcov` при каждом выполнении теста. Некоторые проекты, предназначенные для жизненно важных приложений, могут потребовать очень высокого процента покрытия модульными тестами, но разработка полного набора тестов для сложной системы сильно влияет на общие усилия по программированию и значительно увеличивает стоимость разработки, поэтому в большинстве случаев рекомендуется найти правильный баланс между эффективностью и качеством.

При выполнении функциональных тестов нужен иной подход. Все функциональные возможности, реализованные в целевом устройстве, должны быть протестированы, включая подготовленные заранее тесты для определения показателей производительности и соответствия ожидаемым параметрам. Функциональные тесты следует запускать в среде, максимально приближенной к реальному сценарию использования, во всех тех случаях, когда невозможно воссоздать полный вариант использования в целевой системе и ее окружении.

3.2. Организация исходного кода

Кодовая база должна содержать весь исходный код, сторонние библиотеки, данные, сценарии и средства автоматизации, необходимые для создания окончательного образа. Рекомендуется хранить автономные библиотеки в отдельных каталогах, чтобы их можно было легко обновить до более новых версий, заменив подкаталог. Make-файлы и другие сценарии могут быть размещены в корневом каталоге проекта.

Код приложения должен быть коротким и синтетическим и обращаться к модулям, абстрагируя функции макросов. Функциональные модули должны описывать процесс, скрывая детали базовой реализации, например считывание данных с датчика, после того как они были должным образом обработаны. Стремление к небольшим, автономным и адекватно абстрагированным модулям также упрощает тестирование компонентов архитектуры. Сохранение большей части логики компонентов приложения отдельно от их аппаратной реализации улучшает переносимость между различными платформами и позволяет изменять периферийные устройства и интерфейсы целевого устройства даже на этапе разработки. Но чрезмерное абстрагирование влияет на затраты с точки зрения усилий по разработке и необходимых ресурсов, поэтому необходимо найти правильный баланс.

3.2.1. Аппаратная абстракция

Производители чипов выпускают различные средства макетирования и отладки для оценки микроконтроллеров и периферийных устройств, поэтому часть разработки программного обеспечения может выполняться на платформе прототипирования еще до начала проектирования конечного продукта.

Программное обеспечение, которое можно запустить на оценочной плате, обычно распространяется производителем чипа как эталонная реализация в виде исходного кода или проприетарных предварительно скомпилированных библиотек. Эти библиотеки могут быть настроены и адаптированы для целевого устройства, чтобы с самого начала выступать в качестве эталонной абстракции оборудования, а их настройки обновляются в соответствии с изменениями в конфигурации оборудования.

Как было сказано в главе 2, примеры кода в этой книге ориентированы на процессорное ядро Cortex-M. Поддержка аппаратных компонентов универсального микроконтроллера Cortex-M предоставляется в виде библиотеки под названием *Cortex Microcontroller Software Interface Standard* (CMSIS), распространяемой ARM в качестве базовой реализации. Производители чипов получают свои конкретные аппаратные абстракции, расширяя CMSIS. Приложение, использующее аппаратную абстракцию для конкретного целевого микроконтроллера, может получить доступ к периферийным устройствам через индивидуальные вызовы API и основные функции MCU через CMSIS.

Чтобы код был переносимым между разными микроконтроллерами одного семейства, драйверам может потребоваться дополнительный уровень абстракции поверх вызовов API конкретного поставщика. Если HAL реализует несколько целей, он может предоставить один и тот же API для доступа к общим функциям на нескольких платформах, скрывая аппаратную реализацию.

Назначение CMSIS и других свободно распространяемых версий ПО, таких как *libopencm3* и *unicore-mx*, состоит в том, чтобы объединить все общие абстракции Cortex-M и специфичный для поставщика код для наиболее распространенных производителей микросхем Cortex-M, маскируя разницу между платформозависимыми вызовами при управлении системой и периферийными устройствами.

Независимо от аппаратной абстракции, определенный код, необходимый на самой ранней стадии загрузки, полностью зависит от целевого устройства, для которого предназначено создаваемое ПО. Каждая платформа имеет свою собственную сегментацию адресного пространства, вектор прерывания и смещение регистра конфигурации. Это означает, что при работе с кодом, который должен быть переносимым между различными платформами, в make-файлах и сценариях автоматизации сборки должна быть предусмотрена возможность настройки для подключения правильного кода загрузки и конфигураций компоновщика.

Примеры, представленные в этой книге, не зависят от какой-либо конкретной аппаратной абстракции, поскольку они направлены на управление системными компонентами путем прямого взаимодействия с системными регистрами и реализации драйверов устройств для конкретной платформы, при этом основное внимание уделяется взаимодействию с аппаратным компонентом.

3.2.2. Промежуточный уровень

Некоторые из функций разрабатываемой системы могли быть реализованы ранее индивидуальным разработчиком, сообществом или предприятием. Решения могут быть универсальными, возможно, разработанными для другой платформы или даже заимствованными за пределами мира встраиваемых систем.

В любом случае всегда стоит начинать с поиска библиотек для любого преобразования данных, реализации протокола или модели подсистемы, которые, возможно, уже были разработаны и ждут интеграции в ваш проект.

Различные библиотеки с открытым исходным кодом и программные компоненты готовы к включению во встроенные проекты, что позволяет нам реализовать более широкий набор функций. Интеграция компонентов с открытым исходным кодом особенно полезна при разработке стандартных функций. Существует широкий выбор хорошо зарекомендовавших себя реализаций с открытым исходным кодом, предназначенных для встраиваемых устройств, которые можно легко интегрировать в свои проекты. Наиболее часто применяются следующие готовые решения:

- операционные системы реального времени;
- криптографические библиотеки;
- TCP/IP, 6LoWPAN и другие сетевые протоколы;
- библиотеки безопасности транспортного уровня (TLS);
- файловые системы;
- протоколы очереди сообщений IoT;
- парсеры.

Некоторые компоненты из этих категорий более подробно описаны далее в этой книге.

Наличие во встраиваемом устройстве операционной системы позволяет нам управлять областями памяти и выполнением потоков. В этом случае потоки выполняются независимо друг от друга, и можно даже реализовать разделение памяти между потоками и между запущенными потоками и ядром. Такой подход целесообразен при высокой сложности системы или при наличии в модулях известных блокирующих точек, которые нельзя перепроектировать. Другие библиотеки обычно требуют поддержки многопоточности, если используется операционная система; эту опцию можно включить во время компиляции.

Решение об интеграции сторонних библиотек должно основываться на измерении ресурсов, необходимых с точки зрения размера кода и используемой памяти для выполнения конкретных задач на целевой платформе. Поскольку вся прошивка распространяется в виде одного исполняемого файла, все лицензии компонентов должны быть совместимы, а интеграция не должна нарушать условия лицензии ни одного из ее отдельных компонентов.

3.2.3. Код приложения

Роль кода приложения заключается в координации работы всех задействованных модулей и управлении эвристикой системы. Качественный и хорошо спроектированный главный модуль позволяет нам иметь четкое представление обо всех макроблоках системы, о том, как они связаны друг с другом, и о времени выполнения различных компонентов.

Приложения без операционной системы построены вокруг бесконечного цикла функции `main`, которая отвечает за распределение процессорного времени между точками входа базовых библиотек и драйверов. Выполнение происходит последовательно, поэтому код не может быть приостановлен, кроме как с помощью обработчиков прерываний. По этой причине предполагается, что все функции и библиотеки, вызываемые из главного цикла, должны возвращаться как можно быстрее, потому что точки остановки, скрытые внутри других модулей, могут поставить под угрозу реактивность системы или даже заблокировать их навсегда с риском никогда не вернуться в исходное состояние внутри главного цикла. В идеале, в устройстве без операционной системы каждый компонент предназначен для взаимодействия с основным циклом с использованием парадигмы, управляемой событиями, при этом основной цикл постоянно ожидает событий и механизмов регистрации обратных вызовов для пробуждения приложения при определенных событиях.

Преимущества однопоточного подхода без операционной системы заключаются в том, что синхронизация между потоками не требуется, вся память доступна для любой функции в коде и нет необходимости реализовывать сложные механизмы, такие как переключение контекста и модели выполнения. Но некоторые базовые механизмы синхронизации все же могут потребоваться, когда прерывания включены, а поток выполнения может быть прерван внешними событиями в любой момент для выполнения определенного обработчика.

Если несколько задач должны выполняться поверх операционной системы, каждая задача должна быть максимально ограничена своим собственным модулем и явно экспортировать свою функцию запуска и общедоступные переменные как глобальные символы. В этом случае задачи могут уходить в режим «сна» и вызывать функции блокировки, которые должны реализовывать механизмы блокировки, специфичные для ОС.

Благодаря гибкости процессорного ядра Cortex-M в системе можно активировать различные степени разделения потоков и процессов.

ЦП предлагает несколько инструментов для облегчения разработки многопоточных систем с разделением задач, несколькими режимами выполнения, специфичными для ядра регистрами, разделением привилегий и методами сегментации памяти. Эти возможности позволяют архитекторам проектов определять сложные системы, больше ориентированные на приложения общего назначения, которые предлагают разделение привилегий и сегментацию памяти между процессами, а также более мелкие, простые и понятные системы, которым ничего этого не нужно, поскольку они предназначены для одной специфической цели.

Выбор модели выполнения, основанной на непривилегированных потоках, приводит к гораздо более сложной реализации изменений контекста в системе и может повлиять на задержку операций в реальном времени, поэтому в большинстве приложений реального времени по-прежнему желательно использовать однопоточные решения без операционной системы.

3.3. Соображения безопасности

Одним из наиболее важных аспектов, который следует учитывать при разработке новой системы, является безопасность. В зависимости от характеристик системы, требований и оценки рисков могут применяться различные подходы к безопасности. Функции безопасности часто представляют собой сочетание аппаратных и программных решений, направленных на обеспечение защиты от известных атак.

3.3.1. Устранение уязвимостей

Программные компоненты продолжают развиваться по мере появления новых функций, а дефекты исправляются по ходу дела. Некоторые дефекты,

обнаруженные и исправленные в более поздних версиях, могут повлиять на безопасность системы с устаревшим ПО, если не будут своевременно приняты соответствующие меры. Весьма безрассудно продолжать использовать устаревший код после того, как уязвимости в сторонних компонентах стали известны общественности.

Старые версии с известными уязвимостями, работающие в общедоступных сетях, имеют повышенную вероятность стать точкой приложения атаки в попытке повредить систему, получить контроль над выполнением программного обеспечения или украсть важные данные. Наилучший ответ готовится на самом раннем этапе проектирования системы и состоит в планировании удаленных обновлений в соответствии с требованиями безопасности.

При использовании сторонних библиотек уместно следить за разработкой их последних версий и тщательно проверять исправление ошибок, особенно когда они помечены как проблемы безопасности.

3.3.2. Применение криптографии

При необходимости следует использовать алгоритмы шифрования, например для шифрования данных, хранящихся локально или при передаче между двумя системами, аутентификации удаленного субъекта в сети и проверке того, что данные не были изменены и получены из надежного источника.

Хорошая криптография всегда основана на открытых, прозрачных стандартах, так что безопасность системы зависит исключительно от безопасности ключей, в соответствии с *принципом Керкхоффа*, сформулированным голландским криптографом Огюстом Керкхоффом в XIX веке, а не от секретного механизма, в ложной надежде на то, что его реализация никогда не будет раскрыта или не поддастся обратной разработке. Хотя это последнее соображение должно быть очевидным для тех, кто хоть немного разбирается в информационной безопасности, в прошлом во многих встраиваемых системах применялась «безопасность через незнание». Впрочем, зачастую это было следствием нехватки ресурсов для запуска хорошо зарекомендовавших себя криптографических примитивов на старых аппаратных архитектурах.

В настоящее время существуют криптографические библиотеки для встраиваемого оборудования, способные использовать новейшие стандартные алгоритмы, изначально разработанные для использования в ПК и серверах, в системах на основе микроконтроллеров, которые становятся все более мощными и пригодными для запуска математических примитивов (часто потребляющих значительные ресурсы ЦП). Полная криптографическая библиотека предлагает готовую к использованию реализацию, как правило, трех семейств алгоритмов:

- **асимметричная криптография** (RSA, ECC) основана на паре ключей, закрытом и открытом, связанных друг с другом. Помимо одностороннего шифрования, на этих алгоритмах основаны и другие механизмы, такие как аутентификация подписи и получение вторичных ключей, например, из двух пар ключей, для использования в качестве общего

секрета обеими конечными точками, взаимодействующими через ненадежную среду;

- **симметричная криптография** (AES, ChaCha20) чаще всего применяется для двунаправленного шифрования с использованием одного и того же предварительно согласованного общего секретного ключа в обоих направлениях;
- **алгоритмы хеширования** (SHA) обеспечивают вычисление дайджеста и часто используются для проверки того, что данные не были изменены.

Полный набор алгоритмов, оптимизированных для встраиваемых систем, предоставляется механизмом шифрования wolfCrypt, распространяемым как часть wolfSSL – профессионально поддерживаемой библиотеки с открытым исходным кодом, включающей протоколы безопасности транспортного уровня, которые будут подробно описаны в главе 9.

3.3.3. Аппаратная криптография

Аспекты безопасности необходимо учитывать с самого начала процесса проектирования, чтобы заранее выбрать программные и аппаратные компоненты, необходимые для реализации правильных механизмов защиты системы. Простое добавление криптографической библиотеки не гарантирует повышенный уровень безопасности в системе, если не выполнены остальные требования, что часто подразумевает использование определенных аппаратных компонентов.

Некоторым алгоритмам требуются случайные значения с высокой энтропией, которую трудно получить на микроконтроллерах без помощи специального оборудования, такого как *генераторы истинных случайных чисел* (true random number generator, TRNG).

Для криптографии на основе открытого ключа требуется так называемое *хранилище якорей доверия* (trust anchor storage) – особая область памяти, которую злоумышленник не может изменить во время выполнения. Обычно за выделение и обслуживание этой области отвечает контроллер флеш-памяти. Наконец, для хранения секретных ключей может потребоваться аппаратная поддержка безопасного хранилища, доступного только с помощью привилегированного кода или, в некоторых случаях, вообще недоступного из исполняемого кода и работающего только в сочетании с аппаратными криптографическими устройствами.

3.3.4. Запуск ненадежного кода

По мере того как встроенные системы становятся все более сложными, а объем памяти для кода увеличивается, нередки случаи, когда программные компоненты из нескольких источников интегрируются в один образ микропрограммы. Некоторые системы могут даже предоставлять комплект для

разработки программного обеспечения, который запускает пользовательский код, предоставленный пользователем.

Другие системы могут иметь интерфейс, позволяющий выполнять код удаленно. Во всех этих случаях уместно предусмотреть механизмы разделения для предотвращения случайного (или преднамеренного) доступа к областям памяти или периферийным устройствам, которые не должны быть доступны субъектам с более низкими правами.

Большинство микроконтроллеров предоставляют два уровня привилегий выполнения, а на некоторых платформах можно разделить адресуемое пространство памяти в соответствии с этими привилегиями посредством переключения контекста в ОС. Новые поколения микроконтроллеров поддерживают *доверенную среду выполнения* (trusted execution environment, TEE) для строгого соблюдения границ памяти в зависимости от уровня допуска выполняемого кода.

3.4. Жизненный цикл проекта встраиваемой системы

Современные фреймворки разработки предлагают разбивать работу на более мелкие этапы и отмечать вехи в ходе разработки проекта, одновременно сохраняя промежуточные рабочие версии. Каждая рабочая версия направлена на создание прототипа окончательной системы, а отсутствующие функции временно заменяются с помощью фиктивного кода.

Этот подход особенно эффективен при работе над проектами встраиваемых систем. В среде, где малейшая ошибка может быть фатальной для всей системы, работа над небольшими фрагментами действий, по одному за раз, является эффективным способом быстрого выявления ошибок и регрессий при работе с кодовой базой, при условии что механизм *непрерывной интеграции* (continuous integration, CI) был предусмотрен с первых шагов проекта.

Промежуточные версии необходимо сохранять и проверять как можно чаще, и по этой причине желательно создать прототип целевого устройства как можно раньше. Это необходимо учитывать при определении действий, приоритизации задач и распределении их по команде.

Как только основные шаги для достижения цели определены, нужно найти оптимальную последовательность создания рабочих прототипов устройства для промежуточных этапов. Зависимости между действиями по разработке кода и прототипов учитываются при расстановке приоритетов заданий.

Может случиться так, что новое понимание поведения системы и аппаратных ограничений может изменить представление об архитектуре системы, пока она находится в стадии разработки, поскольку возникли неожиданные проблемы. Изменение спецификаций проекта в ответ на измерения и оценки, выполненные на промежуточном прототипе, может потребовать серьезной переработки кода. Отбрасывание последовательных частей проекта для замены их новыми, улучшенными решениями обычно полезно для

качества проекта и может привести к повышению производительности на более поздних этапах. Этот процесс, известный как *рефакторинг*, не следует рассматривать как незапланированные затраты на разработку, если он направлен на улучшение архитектуры и поведения системы.

Наконец, процесс создания системного программного обеспечения включает в себя определение и разработку API для приложений, чтобы они взаимодействовали с системой желаемым образом. Встраиваемые системы в большинстве случаев предоставляют специальные API для доступа к системным ресурсам; но некоторые операционные системы и библиотеки могут предоставлять POSIX-подобные интерфейсы для доступа к функциям. В любом случае API является точкой входа для системных интерфейсов и должен быть разработан с учетом удобства использования и хорошо задокументирован.

3.4.1. Определение этапов проекта

При анализе спецификаций, определении необходимых этапов и назначении приоритетов может потребоваться учитывать несколько факторов. В качестве примера рассмотрим разработку устройства контроля качества воздуха с датчиком качества воздуха PM10, который каждый час собирает измерения во внутреннюю флеш-память, а затем один раз в сутки передает всю статистику на шлюз с помощью беспроводного приемопередатчика. Целевая система представляет собой специализированную плату на базе микроконтроллера Cortex-M, объема памяти которого достаточно для запуска окончательного программного обеспечения. Но проект аппаратной части устройства невозможно завершить до тех пор, пока не будут выполнены некоторые реальные измерения сигнала трансивера, передающего данные на шлюз.

Список шагов, которые необходимо выполнить для достижения конечной цели, вытекающий из этих условий, может выглядеть следующим образом.

1. Загрузить минимальную систему на целевую плату (пустой основной цикл).
2. Настроить последовательный порт 0 для связи с хост-компьютером и просмотра отладочной информации.
3. Настроить последовательный порт 1 для связи с датчиком.
4. Настроить таймер.
5. Разработать драйвер датчика PM10 (или адаптировать готовый драйвер).
6. Разработать приложение, которое просыпается каждый час по прерыванию таймера и считывает данные с датчика.
7. Разработать подмодуль работы с флеш-памятью для записи/чтения показаний датчика.
8. Настроить порт SPI для связи с чипом радиоканала.
9. Разработать драйвер радиоканала (или адаптировать готовый драйвер).
10. Интегрировать протокол для связи со шлюзом.
11. Сделать так, чтобы через каждые 24 измерения приложение отправляло ежедневный пакет данных на шлюз.

Примечание

Некоторые шаги могут зависеть от других шагов, поэтому существуют ограничения на порядок их выполнения. Некоторые зависимости можно обойти с помощью симуляторов или эмуляторов.

Например, можно внедрить протокол связи без работающего радиоканала только в том случае, если есть способ протестировать протокол с помощью эмуляторов радиоканала и шлюза. Сохранение модулей автономными и с минимальным набором вызовов API, доступных извне, упрощает отвязку отдельных модулей для их запуска и тестирования на разных архитектурах и в контролируемой среде перед их интеграцией в целевую систему.

3.4.2. Прототипирование

Анализ этапов проекта свидетельствует о том, что желательно отдать приоритет отладке радиоканала, чтобы дать возможность команде разработчиков оборудования поскорее приступить к разработке, поэтому для получения первого прототипа нужно сделать следующее:

1) загрузить минимальную систему на целевую плату (пустой цикл `main`);
2) настроить последовательный порт 0 для связи с хост-компьютером и просмотра отладочной информации;
3) настроить порт SPI для связи с чипом радиоканала;
4) разработать драйвер радиоканала (или адаптировать готовый драйвер);
5) настроить таймер;
6) разработать основное приложение для проверки радиоканала (отправка тестовых пакетов через равные промежутки времени).

Этот первый прототип уже похож на финальное устройство, даже если он еще не умеет общаться с датчиком. Параллельно можно запустить отладочное приложение на шлюзе для проверки, что сообщения получены и корректны.

Переходя к следующему определению прототипа, можно начать добавлять дополнительные функции. Для проверки работы протокола связи устройства со шлюзом не нужны реальные показания датчика, поскольку вместо этого можно использовать синтетические тестовые значения, которые воспроизводят определенное поведение. Это позволяет нам выполнять другие задачи, когда реальное оборудование недоступно.

Независимо от того, использует ли команда agile-разработку программного обеспечения или работает с другой методологией, быстрое прототипирование во встроенной среде разработки позволяет быстрее прояснять различные неопределенности, которые часто зависят от поведения оборудования и действий, которые необходимо предпринять в ПО.

Предоставление работоспособных промежуточных вариантов – обычная практика в командах разработчиков встраиваемых систем, которая напрямую вытекает из методологии agile-разработки. Гибкая разработка ПО предусматривает поставку работающих версий регулярно и в короткие про-

межутки времени. Как и в предыдущем примере, промежуточный прототип не обязательно должен реализовывать всю логику окончательного образа программного обеспечения; достаточно, чтобы его можно было использовать для проверки концепций, проведения измерений или демонстрации образцов работающей части системы.

3.4.3. Рефакторинг

Рефакторинг, который слишком часто считают лишь радикальным средством от сбоев, на самом деле является полезной стратегией, направленной на улучшение ПО до тех пор, пока система не приняла свою окончательную форму, а поддержка программных компонентов и периферийных устройств продолжает развиваться.

Рефакторинг работает лучше, если все тесты запускаются на старом коде. С одной стороны, модульные тесты необходимо адаптировать к новым сигнатурам функций при изменении внутреннего устройства модуля. С другой стороны, функциональные тесты для модуля, подвергаемого рефакторингу, не должны меняться, пока интерфейс отлаживаемого модуля по отношению к другим модулям остается прежним.

Компактные части кодовой базы многократно легче поддаются рефакторингу, чем большие, что дает нам еще одну причину делать каждый модуль небольшим и посвященным определенной функции в системе. Продвижение проекта через промежуточные прототипы подразумевает постоянные изменения в коде приложения, что требует меньше усилий, если подсистемы спроектированы так, чтобы быть независимыми друг от друга и от самого кода приложения.

3.4.4. API и документация

Общеизвестно утверждение, что о книге нельзя судить по обложке. Но о системе часто можно судить по ее API, который раскрывает многие аспекты внутренней реализации и системной архитектуры. Четкий, читаемый и простой для понимания API – одна из важнейших функций встраиваемой системы. Разработчики внешних приложений рассчитывают понять, как быстро получить доступ к функциям и максимально эффективно использовать вашу систему. API представляет собой контракт между системой и приложениями, и по этой причине он должен быть разработан заранее и как можно меньше подвергаться изменениям, если вообще меняться, на протяжении всего оставшегося цикла разработки.

Некоторые интерфейсы в API могут описывать сложные подсистемы и абстрагироваться от более сложных характеристик, поэтому всегда полезно предоставить качественную документацию, чтобы помочь разработчикам приложений использовать все возможности системы. Существуют разные способы предоставления документации вместе с кодом: либо рас-

пространение руководств пользователя в репозитории в виде отдельных файлов, либо включение пояснений различных интерфейсов непосредственно в код.

Количество комментариев в коде – еще не показатель качества. Комментарии, как правило, устаревают всякий раз, когда изменяется код, на который они ссылаются, а разработчик забывает обновить комментарий, чтобы он соответствовал новому поведению кода. Более того, не весь код нужно комментировать; хорошие привычки, такие как сохранение функций короткими и низкими по сложности или использование выразительных имен символов, в большинстве случаев сделают комментарии кода излишними, поскольку простой и качественный код может объяснить сам себя.

Есть исключения для строк кода, которые содержат сложные вычисления, сдвиг битов, сложные условия или дают скрытые эффекты, которые нелегко заметить при первом чтении кода. Некоторые части кода также могут потребовать описания в начале, например функции с несколькими возвращаемыми значениями и специальной обработкой ошибок. Операторы `switch/case`, не содержащие инструкции `break` между двумя `case`, всегда должны иметь комментарий, указывающий, что такое упущение является намеренным, а не ошибочным.

Комментарии также, возможно, должны объяснить, почему некоторые действия сгруппированы между двумя или более `case`. Добавление лишних комментариев, не дающих никакого ценного пояснения к коду, только усложняет чтение кода.

С другой стороны, описание поведения модуля в отдельном текстовом файле требует определенной самоотверженности, так как вся документация должна обновляться каждый раз, когда в код вносятся существенные изменения, а разработчикам приходится переключать внимание с разработки кода на его описание.

Обычно важной частью документации является описание упомянутого в разделе 3.4.4 «контракта» между системой и приложениями, перечисляющее и объясняющее функции и переменные, к которым приложения и другие задействованные компоненты могут обращаться во время выполнения. Поскольку эти объявления могут быть сгруппированы в файлах заголовков, можно описать весь контракт, добавив расширенные комментарии поверх объявления каждого экспортируемого символа.

Существуют программные инструменты, которые преобразуют комментарии кода в форматированную документацию. Популярным примером является Doxygen, бесплатный инструмент для создания документов с открытым исходным кодом, который анализирует комментарии, оформленные в соответствии с определенным синтаксисом по всей кодовой базе, и создает гипертекст, структурированные руководства в формате PDF и документацию во многих других форматах. Если документация находится непосредственно в кодовой базе, ее обновление меньше отвлекает разработчиков от основного процесса. Размещение генератора документации на сервере автоматизации рабочих процессов позволяет предоставить свежесгенерированную копию руководств для всех API при каждом коммите в master-ветке.

3.5. Заключение

В этой главе рассмотрены основные подходы и приемы, используемые при проектировании и управлении разработкой встраиваемых систем. Хотя возможно, что некоторые из описанных стратегий применимы не ко всем проектам, цель этой главы – побудить разработчиков встраиваемых систем искать улучшения рабочего процесса, которые могут привести к построению более эффективного и менее дорогого жизненного цикла разработки ПО. Также показаны возможность повышения безопасности путем добавления соответствующих процессов и компонентов, когда это требуется в зависимости от варианта использования.

В следующей главе описано, что происходит во время начальной загрузки встраиваемой системы и как подготовить загружаемое приложение, используя простой подход выполнения главного цикла на «голом железе».

Глава 4

Процедура загрузки

Теперь, когда вы познакомились с механизмами, инструментами и методами, необходимыми для разработки и отладки встраиваемого ПО, пришло время рассмотреть процедуры, необходимые для запуска программного обеспечения на целевом устройстве. Загрузка встраиваемой системы – это процесс, который часто требует знания конкретной системы и задействованных механизмов. Чтобы выяснить, каким требованиям должна соответствовать система для успешной загрузки исполняемых файлов из флеш-памяти, необходимо обратиться к технической документации микроконтроллера. В этой главе основное внимание будет уделено описанию процесса загрузки с акцентом на микроконтроллер Cortex-M, который используется в качестве базовой платформы. В частности, затронуты следующие темы:

- таблица векторов прерываний;
- структура памяти;
- сборка и запуск загрузочного кода;
- несколько этапов загрузки.

Данная глава завершает общий обзор принципов разработки встраиваемых систем, ПО которых основано на выполнении главного цикла.

4.1. Технические требования

Вы можете найти файлы кода для этой главы на GitHub по адресу https://github.com/PacktPublishing/Embedded-Systems-Architecture-Second-Edition/tree/main/Chapter4.

4.2. Таблица векторов прерываний

Таблица векторов прерываний (interrupt vector table), часто сокращенно IVT или просто IV, представляет собой массив указателей на функции, связанные с ЦП для обработки определенных *исключений* (exception), таких как сбои, запросы

системных служб от приложения и запросы на прерывание от периферийных устройств. IVT обычно располагается в начале двоичного образа и, таким образом, сохраняется, начиная с самого младшего адреса во флеш-памяти.

Запрос прерывания от аппаратного компонента или периферийного устройства заставит ЦП немедленно приостановить выполнение текущего потока инструкций и выполнить функцию в соответствующей позиции в векторе. По этой причине такие функции называются *подпрограммами обслуживания прерываний* (interrupt service routine, ISR). Исключения и сбои во время выполнения можно обрабатывать так же, как и аппаратные прерывания, поэтому специальные служебные процедуры связаны с внутренними триггерами ЦП через ту же таблицу.

Порядок ISR, перечисленных в векторе, и их точное положение зависят от архитектуры ЦП, модели микроконтроллера и поддерживаемых периферийных устройств. Каждая линия прерывания соответствует заранее определенному номеру прерывания и, в зависимости от особенностей микроконтроллера, может иметь приоритет.

В микроконтроллере Cortex-M первые 16 позиций в памяти зарезервированы для хранения указателей на системные обработчики, которые зависят от архитектуры и связаны с различными типами исключений, возникающих во время работы ЦП. Младший адрес используется для хранения начального значения указателя стека, а следующие 15 позиций отведены для системных служб и обработчиков ошибок. Но некоторые из этих позиций зарезервированы и не связаны с каким-либо событием. С помощью отдельных служебных процедур в ЦП Cortex-M можно обрабатывать следующие системные исключения:

- перезагрузка (reset);
- немаскируемое прерывание (non-maskable interrupt, NMI);
- отказ оборудования (hard fault);
- исключение памяти (memory exception);
- ошибка шины (bus fault);
- ошибка программы (usage fault);
- вызов супервайзера (supervisor call);
- событие монитора отладки (debug monitor event);
- вызов PendSV;
- такт системного таймера (system tick).

Порядок аппаратных прерываний, начиная с позиции 16, зависит от конфигурации микроконтроллера и, следовательно, от конкретной модели чипа, поскольку конфигурация прерываний относится к конкретным компонентам, интерфейсам и действиям внешней периферии.

Полный набор внешних обработчиков прерываний для чипов STM32F407 и LM3S можно найти в репозитории кода этой книги.

4.2.1. Код запуска

Чтобы загрузить работоспособную систему, нам нужно определить вектор прерывания и связать указатели с определенными функциями. В типичном

файле кода запуска для нашей базовой платформы вектор прерывания помещен в выделенный раздел с использованием атрибута `section` GCC. Поскольку раздел будет помещен в начало образа, нужно определить вектор прерывания, начиная с зарезервированного пространства для начального указателя стека, за которым следуют обработчики системных исключений.

Нули соответствуют позициям зарезервированных/неиспользуемых слотов:

```
__attribute__ ((section(".isr_vector")))
void (* const IV[])(void) =
{
  (void (*)(void))(END_STACK),
  isr_reset,
  isr_nmi,
  isr_hard_fault,
  isr_mem_fault,
  isr_bus_fault,
  isr_usage_fault,
  0, 0, 0, 0,
  isr_svc,
  isr_dbgmon,
  0,
  isr_pendsv,
  isr_systick,
```

Начиная с этой позиции, определены линии прерывания для внешних периферийных устройств следующим образом:

```
  isr_uart0,
  isr_ethernet,
  /* … и остальные внешние прерывания */
};
```

Код запуска также должен включать реализацию каждого символа, на который ссылается массив. Обработчик может быть определен как пустая процедура без аргументов в том же формате, что и сигнатура вектора прерывания:

```
void isr_bus_fault(void) {
  /* Ошибка шины! */
  while(1);
}
```

Обработчик прерывания неисправимой ошибки шины в этом примере никогда не возвращается и навсегда зависает в системе. Пустые обработчики прерываний могут быть связаны как с системными, так и с внешними прерываниями с помощью «слабых» символов[1], которые можно переопределить

[1] Каждый символ в программе либо «сильный», либо «слабый». Сильные: функции и инициализированные глобальные переменные. Слабые: неинициализированные глобальные переменные. – *Прим. перев.*

в модулях драйверов устройств, просто определив их снова в соответствующем разделе кода.

4.2.2. Обработчик сброса

При включении питания микроконтроллера выполнение программы начинается с *обработчика сброса* (reset handler). Это специальный ISR, который не возвращается, а выполняет инициализацию разделов `.data` и `.bss`, а затем вызывает точку входа приложения. Инициализация разделов `.data` и `.bss` заключается в копировании начального значения переменных из раздела `.data` во флеш-памяти в фактический раздел в ОЗУ, где доступ к переменным осуществляется во время выполнения, и заполнении раздела `.bss` в ОЗУ нулями, чтобы начальное значение статических символов было гарантированно равно нулю в соответствии с соглашением C.

Исходный и конечный адреса разделов `.data` и `.bss` в ОЗУ вычисляются компоновщиком при создании двоичного образа и экспортируются как указатели с помощью скрипта компоновщика. Реализация `isr_reset` может выглядеть следующим образом:

```
void isr_reset(void)
{
  unsigned int *src, *dst;
  src = (unsigned int *) &_stored_data;
  dst = (unsigned int *) &_start_data;
  while (dst != (unsigned int *)&_end_data) {
    *dst = *src;
    dst++;
    src++;
  }
  dst = &_start_bss;
  while (dst != (unsigned int *)&_end_bss) {
    *dst = 0;
    dst++;
  }
  main();
}
```

Когда переменные в разделах `.bss` и `.data` инициализированы, можно, наконец, вызвать функцию `main`, которая является точкой входа в приложение. Код приложения устроен таким образом, что функция `main` никогда не вернется, реализуя бесконечный цикл.

4.2.3. Размещение стека

В соответствии с *двоичным интерфейсом приложений* (application binary interface ABI) процессорного ядра, необходимо выделить место в памяти для стека выполнения. Это можно сделать разными способами, но обычно пред-

почтительнее отметить конец пространства стека в скрипте компоновщика и связать пространство стека с определенной областью в ОЗУ, не используемой каким-либо разделом.

Адрес, полученный через символ `END_STACK`, экспортируемый скриптом компоновщика, указывает на конец неиспользуемой области в оперативной памяти. Как упоминалось ранее, его значение должно храниться в начале таблицы векторов, в нашем случае по адресу 0, непосредственно перед IV. Адрес конца стека должен быть постоянным и не может быть рассчитан во время выполнения, поскольку содержимое IV хранится во флеш-памяти и, следовательно, не может быть изменено позже.

Правильное определение размера стека выполнения в памяти – деликатная задача, которая включает в себя оценку всей базы кода с учетом использования стека из локальных переменных и глубины трассировки вызовов в любой момент выполнения. Анализ всех факторов, связанных с использованием стека и устранением неполадок, будет частью более широкой темы, которой посвящена следующая глава. Наш простой код запуска, представленный выше, задает размер стека, который достаточно велик, чтобы содержать локальные переменные и стек вызовов функций, поскольку он отображается сценарием компоновщика как можно дальше от разделов `.bss` и `.data`. Остальные аспекты размещения стека рассматриваются в главе 5.

4.2.4. Обработчики отказов

События, связанные с отказом, инициируются ЦП в случае ошибок выполнения или нарушений политики. ЦП способен обнаруживать ряд ошибок выполнения, например следующие:

- попытка выполнить код за пределами областей памяти, помеченных как исполняемые;
- выборка данных или следующей инструкции для выполнения из недопустимого местоположения;
- недопустимое чтение или сохранение с использованием невыровненного адреса;
- деление на ноль;
- попытка доступа к отсутствующим функциям сопроцессора;
- попытка чтения/записи/выполнения вне областей памяти, разрешенных для текущего режима работы.

Некоторые базовые микроконтроллеры поддерживают разные типы исключений в зависимости от типа ошибки. Cortex-M3/M4 может различать ошибки шины, ошибки программы, нарушения доступа к памяти и общие ошибки, вызывая соответствующее исключение. В других, более мелких системах доступно меньше сведений о типе ошибки выполнения.

Очень часто сбой делает систему непригодной для использования или неспособной продолжать выполнение из-за повреждения значений регистров ЦП или стека. В некоторых случаях даже размещения точки останова внутри обработчика исключений недостаточно для обнаружения причины

проблемы, что усложняет отладку. Некоторые ЦП поддерживают расширенную информацию о причине сбоя, которая доступна через отображаемые в память регистры после возникновения исключения. В случае Cortex-M3/M4 эта информация доступна через *конфигурируемый регистр состояния сбоев* (configurable fault status register, CFSR), который отображается по адресу `0xE000ED28` на всех процессорах Cortex-M3/M4.

Нарушения ограничений памяти могут быть устранимыми, если соответствующий обработчик исключений реализует какую-то стратегию восстановления и может быть полезен для обнаружения и реагирования на ошибку во время выполнения, что особенно полезно в многопоточных средах, как будет показано более подробно в главе 9.

4.3. Схема памяти

Сценарий компоновщика, как вы уже знаете, содержит инструкции для компоновщика по сборке компонентов встраиваемой системы. В частности, в нем описываются разделы, отображаемые в памяти, и то, как они развертываются во флеш-памяти и ОЗУ целевого устройства, как в примере, представленном в главе 2.

В большинстве встраиваемых устройств и, в частности, на нашей базовой платформе раздел вывода `.text` в скрипте компоновщика, который содержит весь исполняемый код, должен также включать специальный раздел ввода, предназначенный для хранения IV в самом начале исполняемого образа.

Модифицируем скрипт компоновщика, добавив раздел `.isr_vector` в начало раздела вывода `.text` перед остальным кодом:

```
.text :
{
  *(.isr_vector)
  *(.text*)
  *(.rodata*)
} > FLASH
```

Определение во флеш-памяти области только для чтения, предназначенной для таблицы векторов, является единственным строгим требованием для правильной загрузки нашей системы, поскольку ЦП извлекает адрес функции `isr_reset` во время загрузки с адреса 0x04 в памяти.

Сразу после определения области текста и области только для чтения во флеш-памяти скрипт компоновщика должен экспортировать значение текущего адреса, которое является началом секции вывода `.data`, хранящейся во флеш-памяти. Этот раздел содержит начальные значения всех глобальных и статических переменных, которые были инициализированы в коде. В примере скрипта компоновщика начало раздела `.data` отмечено переменной скрипта компоновщика `_stored_data` следующим образом:

```
_stored_data = .;
```

Раздел данных в конечном итоге будет отображаться в ОЗУ, но его инициализация выполняется вручную в функции `isr_reset` путем копирования содержимого из флеш-памяти в фактическую область, назначенную разделу `.data` в ОЗУ. Сценарий компоновщика предоставляет механизм для разделения *адреса виртуальной памяти* (virtual memory address, VMA) и *адреса загрузочной памяти* (load memory address, LMA) для раздела с помощью ключевого слова `AT` в определении раздела. Если ключевое слово `AT` не указано, LMA по умолчанию устанавливается на тот же адрес, что и VMA. В нашем случае VMA секции `.data` находится в оперативной памяти и экспортируется с использованием указателя `_start_data`, который будет использовать `isr_vector` в качестве адреса назначения при копировании значений символов, сохраненных во флеш-памяти. Но LMA секции `.data` находится во флеш-памяти, поэтому установим адрес LMA на указатель `_stored_data` во флеш-памяти при определении секции вывода `.data`:

```
.data : AT (_stored_data)
{
  _start_data = .;
  *(.data*)
  . = ALIGN(4);
  _end_data = .;
} > RAM
```

Для `.bss` LMA отсутствует, так как в образе для этого раздела не хранятся данные. При включении раздела вывода `.bss` его VMA будет автоматически установлен в конец раздела вывода `.data`:

```
.bss :
{
  _start_bss = .;
  *(.bss*)
  . = ALIGN(4);
  _end_bss = .;
  _end = .;
} > RAM
```

Наконец, в этой схеме ожидается, что компоновщик предоставит начальное значение для стека выполнения. Использование самого старшего адреса в памяти является распространенным выбором для однопоточного приложения, хотя, как будет показано в следующей главе, это может вызвать проблемы в случае переполнения стека. Но для рассматриваемого примера это приемлемое решение, и поэтому нужно определить символ `END_STACK`, добавив следующую строку в сценарий компоновщика:

```
END_STACK = ORIGIN(RAM) + LENGTH(RAM);
```

Чтобы лучше понять, где каждый символ будет размещен в памяти, определения переменных могут быть добавлены в файл запуска в разных местах кода. Таким образом, можно проверить места хранения переменных в памяти при первом запуске исполняемого файла в отладчике.

Предположим, что у нас есть переменные, хранящиеся как в выходных разделах .data, так и в .bss, тогда структура памяти для примера кода запуска может выглядеть, как показано на рис. 4.1.

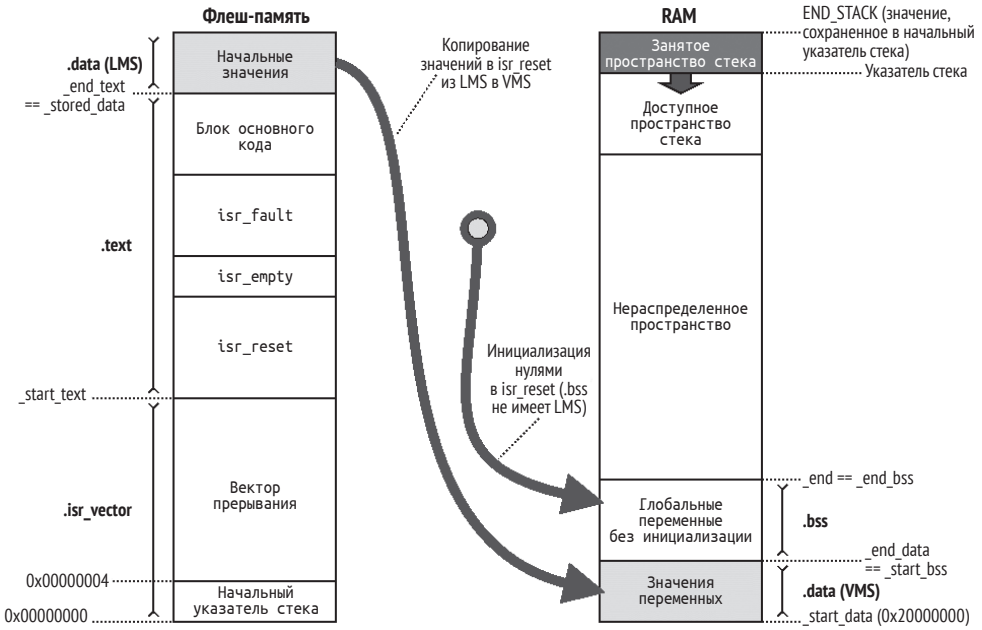

Рис. 4.1 ❖ Структура памяти в примере кода запуска

Когда происходит компоновка исполняемого файла, символы автоматически устанавливаются во время компиляции таким образом, чтобы указать начало и конец каждого раздела в памяти. В нашем случае переменным, указывающим начало и конец каждой секции, автоматически присваивается нужное значение в зависимости от размера секций, которые компоновщик будет включать при создании исполняемого файла. Поскольку размер каждой секции известен во время компиляции, компоновщик способен определить ситуации, когда секции .text и .data не помещаются во флеш-память. В таком случае в конце сборки генерируется ошибка компоновщика. Для проверки размера и местоположения каждого символа полезно создавать map-файл. В нашем примере кода загрузки раздел .text отображается в map-файле следующим образом:

```
.text 0x0000000000000000 0x168
0x0000000000000000 _start_text = .
*(.isr_vector)
.isr_vector 0x0000000000000000 0xf0 startup.o
0x0000000000000000 IV
*(.text*)
.text 0x00000000000000f0 0x78 startup.o
0x00000000000000f0 isr_reset
0x0000000000000134 isr_fault
```

```
0x000000000000013a isr_empty
0x0000000000000146 main
```

Аналогично можно найти границы каждой секции, экспортируемые скриптом компоновщика во время компиляции:

```
0x0000000000000000 _start_text = .
0x0000000000000168 _end_text = .
0x0000000020000000 _start_data = .
0x0000000020000004 _end_data = .
0x0000000020000004 _start_bss = .
0x0000000020000328 _end_bss = .
0x0000000020000328 _end = .
```

Раздел ввода `.rodata`, который в этом минималистичном примере пуст, отображается в области флеш-памяти между `.text` и LMA данных. Эта область зарезервирована для символов констант, потому что константы не должны отображаться в ОЗУ. При определении символов констант рекомендуется использовать модификатор `const` C, потому что ОЗУ часто является нашим самым ценным ресурсом, и в некоторых случаях даже сохранение нескольких байтов доступной для записи памяти путем перемещения символов констант во флеш-память может фактически спасти проект, так как флеш-память обычно намного больше, и ее использование можно легко определить во время компоновки.

4.4. Сборка и запуск загрузочного кода

Приведенный здесь пример является одним из самых простых исполняемых образов, которые можно запустить на целевом устройстве. Для сборки, компиляции и компоновки можно использовать простой make-файл, который автоматизирует все шаги и позволяет нам сосредоточиться на разработке программного обеспечения.

Когда образ прошивки готов, его можно загрузить в реальное устройство или, как вариант, запустить с помощью эмулятора.

4.4.1. Make-файл

Очень простой make-файл для сборки нашего запускаемого приложения описывает конечный образ (`image.bin`) и промежуточные шаги, необходимые для его сборки. Синтаксис make-файла, как правило, очень обширен, и рассмотрение всех функций, предоставляемых Make, выходит за рамки этой книги. Тем не менее знания нескольких базовых понятий будет достаточно, чтобы приступить к автоматизации процесса сборки.

В данном случае определить цели для нашего make-файла довольно просто. Исходный файл `startup.c`, содержащий векторы прерываний, некоторые обработчики исключений, а также основные и глобальные переменные, ко-

торые использовались в примере, можно скомпилировать в объектный файл `startup.o`. Компоновщик использует указания, представленные в сценарии компоновщика `target.ld`, для развертывания символов в правильных разделах, создавая исполняемый образ `.elf`.

Используем `objcopy` для преобразования исполняемого файла `.elf` в двоичный образ прошивки, который можно загрузить в целевое устройство или запустить с помощью QEMU (рис. 4.2).

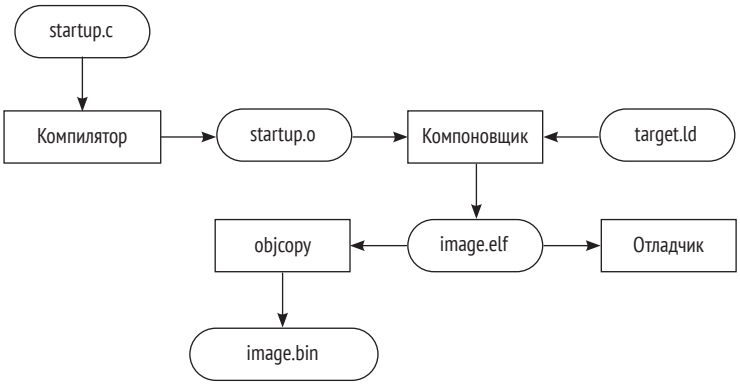

Рис. 4.2 ❖ Шаги сборки и зависимости

Make-файл должен содержать несколько переменных конфигурации для описания набора инструментов. Оператор присваивания (`=`) позволяет задавать значения переменных при вызове команды `make`. Некоторые из этих переменных неявно используются по умолчанию во время компиляции и компоновки. Общепринятой практикой является определение префикса инструментария с помощью переменной `CROSS_COMPILE` и использование ее в качестве префикса для инструментов, участвующих в процессе сборки:

```
CROSS_COMPILE=arm-none-eabi-
CC=$(CROSS_COMPILE)gcc
LD=$(CROSS_COMPILE)ld
OBJCOPY=$(CROSS_COMPILE)objcopy
```

Изменить кросс-компилятор по умолчанию для этого проекта можно, запустив `make` и присвоив другое значение переменной среды `CROSS_COMPILE`. Все имена инструментов имеют префикс расширения переменной `CROSS_COMPILE`, так что шаги сборки будут использовать компоненты из указанного инструментария. Точно так же можно определить наши флаги по умолчанию для компилятора и компоновщика:

```
CFLAGS=-mcpu=cortex-m3 -mthumb -g -ggdb -Wall -Wno-main
LDFLAGS=-T target.ld -gc-sections -nostdlib -Map=image.map
```

При вызове без аргументов Make создает первую цель, определенную в make-файле `image.bin`. Новую цель для `image.bin` можно определить следующим образом:

```
image.bin: image.elf
 $(OBJCOPY) -O binary $^ $@
```

Переменные `$@` и `$^` будут заменены в рецепте на цель и список зависимостей соответственно. Это означает, что в данном примере make-файл будет обрабатывать рецепт следующим образом:

```
arm-none-eabi-objcopy -O binary image.elf image.bin
```

Это команда, которая нам нужна для создания двоичного образа из исполняемого файла `.elf`.

Точно так же можно определить рецепт для `image.elf`, который является шагом компоновки, в зависимости от скомпилированного объектного файла `startup.o` и скрипта компоновщика:

```
image.elf: startup.o target.ld
 $(LD) $(LDFLAGS) startup.o -o $@
```

В данном случае не используется переменная `$^` для списка зависимостей, так как рецепт включает скрипт компоновщика в командную строку компоновщика с помощью `LDFLAGS`. Рецепт для шага компоновки будет расширен `main` следующим образом:

```
arm-none-eabi-ld -T target.ld -gc-sections -nostdlib
-Map=image.map startup.o -o image.elf
```

Использование `-nostdlib` гарантирует, что никакие стандартные библиотеки C, по умолчанию доступные в инструментарии, не будут автоматически связаны с проектом. Это гарантирует, что никакие символы не будут извлечены автоматически.

Последним шагом для разрешения зависимостей является компиляция исходного кода в объектный файл. Это делается в соответствии с неявным рецептом в make-файле, который в конечном итоге преобразуется в следующую команду при применении значений проекта по умолчанию:

```
arm-none-eabi-gcc -c -o startup.o startup.c -mcpu=cortex-m3
-mthumb -g -ggdb -Wall -Wno-main
```

Использование флага `-mcpu=cortex-m3` гарантирует, что созданный код совместим с устройствами на базе Cortex-M начиная с Cortex-M3. Фактически один и тот же двоичный файл может быть запущен на любом устройстве Cortex-M3, M4 или M7, и он остается универсальным до тех пор, пока не потребуется использовать какую-либо функцию, специфичную для ЦП, или определить обработчики аппаратных прерываний в порядке очереди, зависящей от конкретного микроконтроллера.

Определив цель `clean`, в любой момент времени можно начать с чистого листа, удалив промежуточные цели и окончательный образ и снова запустив `make`. Цель `clean` также часто включается в тот же самый make-файл. В нашем примере это выглядит следующим образом:

```
clean:
 rm -f image.bin image.elf *.o image.map
```

Цель `clean` обычно не имеет зависимостей. Запуск `make clean` удаляет все промежуточные и конечные целевые файлы, указанные в рецепте, оставляя исходники и скрипт компоновщика нетронутыми.

4.4.2. Запуск приложения

Готовый образ прошивки можно запустить на реальной цели или с помощью `qemu-system-arm`, как описано в главе 2. Поскольку приложение не будет выводить никакие данные во время работы на эмуляторе, чтобы больше узнать о фактическом поведении ПО, нам нужно подключить к нему отладчик. При запуске эмулятора `qemu-system-arm` необходимо вызывать с параметром остановки `-S`, чтобы он не запускал выполнение до тех пор, пока не будет подключен отладчик. Поскольку переменная `CFLAGS` на предыдущем шаге содержит параметр `-g`, все имена символов будут сохранены в исполняемом файле `.elf`, чтобы отладчик мог отслеживать выполнение кода построчно, устанавливая точки останова и просматривая значения переменных.

Пошаговое выполнение процедур и сравнение адресов и значений с файлом `.map` помогает понять, что происходит и как меняется контекст на протяжении всей последовательности загрузки.

4.5. Загрузка в несколько этапов

Загрузка целевого устройства с применением специального *загрузчика* (bootloader) полезна в нескольких случаях. В реальной жизни возможность обновлять работающее программное обеспечение на удаленных устройствах означает, что разработчики могут исправлять ошибки и добавлять новые функции после развертывания первой версии встраиваемой системы.

Эта возможность удобна при обслуживании, когда ошибка обнаруживается в полевых условиях или когда программное обеспечение должно быть переработано, чтобы адаптироваться к новым требованиям пользователя. Загрузчики могут поддерживать автоматическое удаленное обновление прошивки и другие полезные функции, например:

- загрузка образа приложения из внешнего хранилища;
- проверка целостности образа приложения перед загрузкой;
- механизмы аварийного завершения в случае повреждения приложения.

Несколько загрузчиков могут быть объединены в цепочку для выполнения поэтапной последовательности загрузки. Это позволяет нам иметь отдельные образы программного обеспечения для нескольких этапов загрузки, которые можно загружать во флеш-память независимо друг от друга. Загрузчик первого этапа, если он присутствует, обычно очень прост и используется для простого выбора точки входа следующего этапа. Но в некоторых случаях на ранних стадиях лучше использовать несколько более сложные конструкции для реализации механизмов обновления программного обеспечения или других функций. Предлагаемый здесь пример показывает разделение между

двумя этапами загрузки, достигнутое с использованием функциональных возможностей, доступных во многих процессорах Cortex-M. Единственная цель этого простого загрузчика – инициализировать систему для загрузки приложения на следующем этапе.

4.5.1. Загрузчик

Загрузчик первого этапа запускается как обычное автономное приложение. Его вектор прерывания должен располагаться в начале флеш-памяти, а обработчик сброса инициализирует связанные разделы памяти `.data` и `.bss`, как при обычной одноэтапной загрузке. Раздел в начале прошивки должен быть зарезервирован для секций `.text` и `.data` загрузчика. Поэтому сценарий компоновщика для загрузчика будет включать только начало флеш-памяти, а у приложения будет смещение того же размера.

Загрузчик и приложение, по сути, будут встроены в два отдельных двоичных файла. Таким образом, два сценария компоновщика могут иметь одинаковые имена разделов и отличаться только описанием раздела `FLASH` в памяти компоновщика. Тем не менее метод, предлагаемый далее, использует лишь одну из возможных конфигураций: для более сложной системы может потребоваться экспорт полной геометрии с использованием начальных адресов и размеров всех разделов.

Если нужно зарезервировать 4 Кбайта для раздела загрузчика, можно жестко задать область `FLASH` в скрипте компоновщика загрузчика следующим образом:

```
FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x00001000
```

Точно так же сценарий компоновщика приложения имеет смещение в источнике, точно соответствующее размеру загрузчика, так что раздел `.text` приложения всегда начинается с адреса `0x1000`. С точки зрения приложения вся область `FLASH` начинается с адреса `0x00001000`:

```
FLASH (rx) : ORIGIN = 0x00001000, LENGTH = 0x0003F000
```

Структура флеш-памяти для этого примера показана на рис. 4.3.

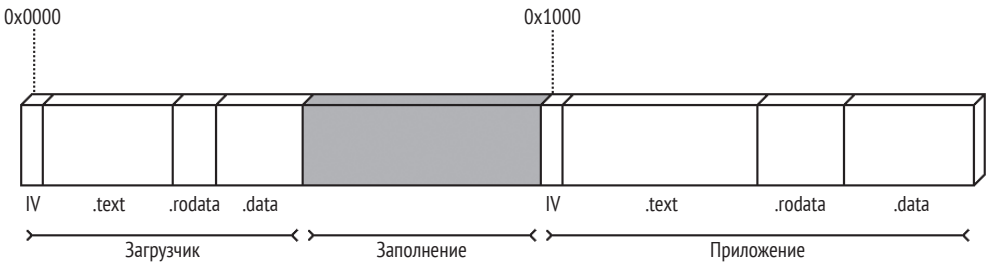

Рис. 4.3 ❖ Схема содержимого флеш-памяти с указанием разделов как загрузчика, так и приложения

Загрузчик и приложение запускают отдельный код и могут определять свой собственный вектор на основе обработчика, который будет использоваться на соответствующем этапе. Простейший пример работающего загрузчика можно реализовать, явно указав адрес приложения и перейдя к точке входа, являющейся обработчиком вектора `reset` и хранящейся со смещением 4 внутри таблицы векторов.

Приложение может применять собственную структуру памяти. При запуске оно сможет инициализировать новые разделы `.data` и `.bss` в соответствии с новой геометрией и даже определить новый начальный указатель стека и IV. Загрузчик может получить эти два указателя, прочитав первые два слова IV, хранящиеся по адресу `0x1000`:

```
uint32_t app_end_stack = (*((uint32_t *)(APP_OFFSET)));
void (* app_entry)(void);
app_entry = (void *)(*((uint32_t *)(APP_OFFSET + 4)));
```

Прежде чем перейти к точке входа приложения, нужно сбросить указатель основного стека выполнения на конечный адрес стека. Поскольку MSP в архитектуре ARMv7-M является регистром ЦП специального назначения, он может быть записан только с помощью специальной инструкции ассемблера (msr, move special from register). Следующий код встроен в загрузчик для установки правильного указателя стека приложения на значение, хранящееся во флеш-памяти в начале образа приложения:

```
asm volatile("msr msp, %0" ::"r"(app_end_stack));
```

В Cortex-M3 и других, более мощных 32-разрядных процессорах Cortex-M в области системного блока управления присутствует управляющий регистр, который можно использовать, чтобы указать смещение таблицы векторов во время выполнения. Это *регистр смещения таблицы векторов* (vector table offset register, VTOR), расположенный по адресу `0xE000ED08`. Запись смещения в этот регистр означает, что с этого момента новая таблица IV на своем месте, и обработчики прерываний, определенные в приложении, будут выполняться при исключениях:

```
uint32_t * VTOR = (uint32_t *)0xE000ED08;
*VTOR = (uint32_t *)(APP_OFFSET);
```

Когда этот механизм недоступен, как в микроконтроллерах Cortex-M0, у которых нет VTOR, приложение по-прежнему будет использовать вектор прерывания совместно с загрузчиком после его запуска. Чтобы обеспечить другой набор обработчиков прерываний, соответствующие указатели функций можно разместить в другой области флеш-памяти, а при поступлении каждого прерывания загрузчик будет проверять, было ли запущено приложение, и, если это так, вызывать соответствующий обработчик из таблицы в пространстве приложения.

При размещении указателей на обработчики прерываний и другие процедуры обработки исключений важно учитывать, что исключение может возникнуть в любой момент выполнения кода, особенно если загрузчик включил

периферийные устройства или активировал таймеры в ЦП. Чтобы предотвратить непредсказуемые переходы к процедуре прерывания, рекомендуется отключить все прерывания на время обновления указателей.

В наборе инструкций процессора предусмотрена возможность временной маскировки всех прерываний. При работе с глобально отключенными прерываниями выполнение основного кода не может быть прервано никаким исключением, кроме NMI. В Cortex-M прерывания можно временно отключить с помощью директивы ассемблера `cpsid i`:

```
asm volatile ("cpsid i");
```

Чтобы снова включить прерывания, используется директива `cpsie i`:

```
asm volatile ("cpsie i");
```

Запуск кода с отключенными прерываниями следует выполнять столько раз, сколько это необходимо, а не только в особых случаях, когда другие решения недоступны, поскольку это влияет на задержку всей системы. В рассмотренном нами примере полное отключение прерываний используется для обеспечения того, чтобы никакие служебные процедуры не вызывались во время перемещения таблицы IV.

Последнее действие, выполняемое загрузчиком за свою недолгую жизнь, – это прямой переход к обработчику `reset` в IV приложения. Поскольку из этого вызова не будет возврата, а совершенно новое пространство стека было только что выделено, требуется принудительно выполнить безусловный переход, устанавливая значение регистра счетчика программ ЦП для начала выполнения с адреса `app_entry`, на который указывает `isr_reset`:

```
asm volatile("mov pc, %0" :: "r"(app_entry));
```

В рассматриваемом примере возврат из этого вызова никогда не произойдет, так как было заменено значение указателя стека выполнения. Это совместимо с поведением обработчика `reset`, который, в свою очередь, перейдет к функции `main` в приложении.

4.5.2. Сборка образа

Поскольку два исполняемых файла будут встроены в отдельные файлы `.elf`, придется использовать специальные средства для объединения содержимого двух разделов в один образ прошивки для загрузки в целевое устройство или для запуска в эмуляторе. Раздел загрузчика можно заполнить нулями до нужного размера, используя параметр `--pad-to` утилиты `objcopy` при преобразовании исполняемого файла `.elf` в двоичный образ. Износ флеш-памяти при записи можно уменьшить, используя для заполнения пустых ячеек значение `0xFF`, для чего нужно передать параметр `--gap- fill=0xFF`. Полученный образ `bootloader.bin` будет иметь размер ровно `4096` байт, так что остается присоединить к нему образ приложения. Для создания образа, содержащего два раздела, нужно выполнить следующие команды:

```
$ arm-none-eabi-objcopy -O binary --pad-to=4096 --gap-fill=0xFF
bootloader.elf bootloader.bin
$ arm-none-eabi-objcopy -O binary app.elf app.bin
$ cat bootloader.bin app.bin > image.bin
```

Если взглянуть на содержимое файла `image.bin` в шестнадцатеричном редакторе, можно определить конец загрузчика в первом разделе по нулям, используемым `objdump` в качестве заполнителя, и найти код приложения, начинающийся с адреса `0x1000`.

Выровняв смещение приложения по началу физической страницы во флеш-памяти, можно даже загрузить два изображения отдельными шагами, что позволит вам, например, обновить код приложения, оставив нетронутым раздел загрузчика.

4.5.3. Отладка системы с поэтапным загрузчиком

Разделение загрузки между двумя или более этапами подразумевает, что символы двух исполняемых файлов связаны с разными файлами `.elf`. Отладка с использованием обоих наборов символов по-прежнему возможна, но символы из обоих файлов `.elf` должны быть загружены в отладчик в два этапа. Когда отладчик выполняется с использованием символов из загрузчика (путем добавления файла `bootloader.elf` в качестве аргумента или с помощью команды `file` из командной строки GDB), символы загрузчика загружаются в таблицу символов для сеанса отладки. Чтобы добавить символы из файла `.elf` приложения, можно подключить соответствующий файл на более позднем этапе, используя директиву `add-symbol-file`.

Директива `add-symbol-file`, в отличие от `file`, обеспечивает загрузку символов второго исполняемого файла без перезаписи ранее загруженных и позволяет указать адрес, с которого начинается секция `.text`. В системе, рассматриваемой в этом примере, нет конфликта между двумя наборами символов, так как два раздела не имеют общей области флеш-памяти. Отладчик может продолжить выполнение в обычном режиме и по-прежнему иметь все доступные символы после того, как загрузчик перейдет к точке входа приложения:

```
> add-symbol-file app.elf
add symbol table from file "app.elf"(y or n) y
Reading symbols from app.elf...done.
```

Совместное использование одних и тех же имен для разделов и символов между двумя исполняемыми файлами является законным, поскольку эти два исполняемых файла автономны и не связаны друг с другом. Отладчик знает о повторяющихся именах, когда происходит обращение к символу по его имени во время отладки. Например, если поместить точку останова на `main` и правильно загрузим символы из обоих исполняемых файлов, точка останова будет установлена в обоих местах:

```
> b main
Breakpoint 1 at 0x14e: main. (2 locations)
> info b
Num Type Disp Enb Address What
1 breakpoint keep y <MULTIPLE>
1.1 y 0x0000014e in main at startup_bl.c:53
1.2 y 0x00001158 in main at startup.c:53
```

Отдельные этапы загрузки полностью изолированы друг от друга и не имеют общего исполняемого кода. По этой причине на разных этапах загрузки может работать программное обеспечение, распространяемое с разными лицензиями, даже если они несовместимы между собой. Как показано в примере, два образа ПО могут использовать одни и те же имена символов, и это не приведет к возникновению конфликтов, поскольку они работают в двух разных системах.

Но в некоторых случаях несколько этапов загрузки могут иметь общие функции, которые было бы логично реализовать с использованием одной и той же библиотеки. К сожалению, нет простого способа получить доступ к символам библиотеки из отдельных образов программного обеспечения. Механизм, описанный в следующем примере, обеспечивает доступ к совместно используемым библиотекам двух этапов путем сохранения во флеш-памяти символов, необходимых только один раз.

4.5.4. Общие библиотеки

Предположим, что имеется небольшая библиотека, предоставляющая утилиты общего назначения или драйверы устройств, которые используются как загрузчиком, так и приложением. Даже если библиотека занимает небольшой объем памяти, желательно не иметь дубликатов определений одних и тех же функций во флеш-памяти. Вместо этого библиотека может быть размещена компоновщиком в специальном разделе загрузчика и использоваться на более позднем этапе. В предыдущем примере двухэтапной загрузки можно безопасно поместить указатели функций API в массив, начинающийся с адреса `0x400`, который находится за концом вектора прерывания. В реальном проекте смещение должно быть достаточно большим, чтобы расположить библиотеку после таблицы векторов в памяти. Раздел `.utils` размещается в скрипте компоновщика между таблицей векторов и началом `.text` в загрузчике:

```
.text :
{
  _start_text = .;
  KEEP(*(.isr_vector))
  . = 0x400;
  KEEP(*(.utils))
  *(.text*)
  *(.rodata*)
  . = ALIGN(4);
```

```
  _end_text = .;
} > FLASH
```

Фактические определения функций могут быть помещены в другой исходный файл и скомпонованы в загрузчике. На самом деле в разделе `.utils` находится таблица, содержащая указатели на фактические адреса функций внутри раздела `.text` загрузчика:

```
__attribute__((section(".utils"),used))
static void *utils_interface[4] = {
(void *)(utils_open),
(void *)(utils_write),
(void *)(utils_read),
(void *)(utils_close)
};
```

Структура загрузчика теперь имеет дополнительный раздел `.utils`, размещенный по адресу `0x400` и содержащий таблицу с указателями на библиотечные функции, которые предназначены для использования на других этапах (рис. 4.4).

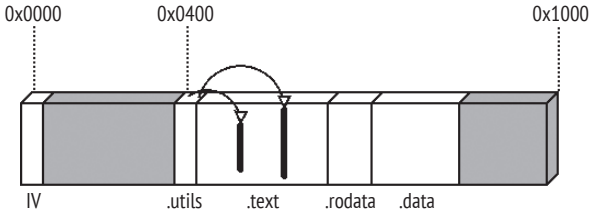

Рис. 4.4 ❖ Структура загрузчика с разделом `.utils`

Приложение ожидает найти таблицу функций по заданному адресу:

```
static void **utils_interface = (void**)(0x00000400);
```

Адреса отдельных функций, которые были сохранены в загрузчике, теперь доступны, но нет информации о сигнатуре этих функций. По этой причине приложение может правильно обращаться к API только в том случае, если указатели преобразованы в соответствии с ожидаемой сигнатурой функции. Затем можно предоставить встроенный адаптер, чтобы код приложения мог напрямую обращаться к функции:

```
static inline int utils_read(void *buf, int size) {
  int (*do_read)(void*, int) = (int (*)(void*,int))
    (utils_interface[2]);
  return do_read(buf, size);
}
```

В этом случае контракт неявно распределяется между двумя модулями, и во время компиляции соответствие между сигнатурами функций не проверяется, равно как и достоверность указателя функции, хранящегося во флеш-

памяти. С другой стороны, это правильный подход, позволяющий избежать дублирования двоичного кода, и он может быть эффективным способом сократить занимаемый объем флеш-памяти за счет совместного использования символов в разных контекстах.

4.5.5. Удаленное обновление прошивки

Одной из причин включения загрузчика во встраиваемую систему часто является предоставление механизма для удаленного обновления работающего приложения. Как упоминалось в предыдущей главе, надежный механизм обновления часто является критическим требованием для управления уязвимостями. В многофункциональных встраиваемых системах под управлением Linux загрузчики часто оснащены собственными стеками TCP/IP, драйверами сетевых устройств и специфичными для протокола реализациями для автономной передачи обновлений ядра и файловой системы. На небольших встраиваемых системах часто бывает удобно возложить эту задачу на приложение, которое в большинстве случаев уже использует аналогичные каналы связи для других функциональных целей. После того как новая прошивка загружена и сохранена в любой энергонезависимой памяти (например, в разделе в конце флеш-памяти), загрузчик может реализовать механизм установки полученного обновления путем перезаписи предыдущей прошивки в раздел приложений.

4.5.6. Безопасная загрузка

Во многих проектах применяется механизм предотвращения запуска несанкционированного или измененного встроенного ПО, которое могло быть намеренно скомпрометировано злоумышленником при попытке получить контроль над системой. Это задача для безопасных загрузчиков, которые используют криптографические методы для проверки подлинности подписи, рассчитанной по содержимому прошивки. Безопасные загрузчики, реализующие такие механизмы, полагаются на защищенную область памяти для хранения открытого ключа и требуют использования манифеста, который должен быть прикреплен к файлу образа микропрограммы. Манифест содержит подпись, созданную владельцем закрытого ключа, связанного с открытым ключом, хранящимся на устройстве. Криптографическая проверка подписи – очень эффективный метод предотвращения несанкционированных обновлений встроенного ПО как удаленно, так и в результате физического подключения.

Разработка безопасного загрузчика с нуля требует значительного объема работы. Несколько проектов с открытым исходным кодом предоставляют механизм для подписания и проверки образов прошивок с использованием криптографических алгоритмов. WolfBoot – это безопасный загрузчик, обеспечивающий проверку целостности и подлинности текущей прошивки и кандидатов на установку обновлений. Он обеспечивает отказоустойчивый

механизм для замены содержимого двух разделов встроенного ПО во время установки обновления, чтобы иметь резервную копию на случай сбоя выполнения недавно обновленного образа. Загрузчик поставляется с инструментами для создания подписи и прикрепления манифеста к файлу, который будет передан на устройство, а также с широким набором настраиваемых параметров, шифров и функций.

4.6. Заключение

Понимание процедуры загрузки является ключевым шагом на пути к разработке встраиваемой системы. Была показана загрузка со входом прямо в приложение на «голом железе», а также структуры, участвующие в многоступенчатой загрузке системы, такие как отдельные сценарии компоновщика с разными точками входа, перемещение IV через регистры ЦП и использование общего кода на разных этапах загрузки.

В следующей главе будут рассмотрены механизмы и способы управления памятью, которые представляют собой наиболее важный фактор, требующий особого внимания при разработке безопасных и надежных встраиваемых систем.

Глава 5

Управление памятью

Работа с памятью – одна из самых важных задач программиста встраиваемых систем и, безусловно, самая важная, которую необходимо учитывать на каждом этапе разработки системы. В этой главе рассказывается о моделях, обычно используемых для управления памятью во встраиваемых системах, геометрии и отображении памяти, а также о том, как избежать проблем, которые могут поставить под угрозу стабильность и безопасность программного обеспечения, работающего на целевом устройстве.

Эта глава состоит из четырех разделов:

- отображение памяти;
- стек выполнения;
- управление кучей;
- блок защиты памяти.

К концу данной главы вы будете иметь глубокие знания о том, как обращаться с памятью во встраиваемой системе.

5.1. Технические требования

Файлы кода для этой главы можно скачать на GitHub по адресу https://github.com/PacktPublishing/Embedded-Systems-Architecture-Second-Edition/tree/main/Chapter5/memory.

5.2. Отображение памяти

При работе с памятью из прикладного ПО удобно использовать ряд абстракций. В современных операционных системах на персональных компьютерах каждый процесс может обращаться к своему собственному пространству памяти, которое также можно перемещать, переназначая блоки памяти на

адреса виртуальной памяти. Более того, возможно динамическое выделение памяти через пулы виртуальной памяти, предоставляемые ядром. Встраиваемые устройства не располагают такими механизмами, поскольку нет возможности назначать виртуальные адреса блокам физической памяти. Во всех контекстах и режимах работы ко всем символам можно получить доступ, только указав физические адреса.

Как было показано в предыдущей главе, загрузка и выполнение встраиваемого приложения на «голом железе» требуют определения разделов во время компиляции в назначенных областях доступного адресного пространства с помощью сценария компоновщика. Чтобы правильно настроить разделы памяти встраиваемой системы, важно проанализировать свойства различных областей и методы, которые можно использовать для организации областей памяти и управления ими.

5.2.1. Модель памяти и адресное пространство

Общее количество доступных адресов зависит от размера указателей памяти. 32-разрядные машины могут ссылаться на непрерывное пространство памяти размером 4 ГБ, которое сегментировано для размещения всех отображаемых в памяти устройств в системе. Как правило, адресное пространство разбивают на следующие сегменты:

- внутренняя оперативная память;
- флеш-память;
- регистры управления системой;
- внутренние компоненты микроконтроллера;
- внешняя периферийная шина;
- дополнительная внешняя оперативная память.

Каждая область памяти имеет фиксированный физический адрес, который может зависеть от характеристик платформы. Все местоположения жестко закодированы, а некоторые из них зависят от платформы.

В ARM Cortex-M все адресуемое пространство разделено на шесть макрообластей. В зависимости от своего назначения они имеют разные разрешения, так что есть области памяти, которые могут быть доступны только для операций чтения во время выполнения или к которым невозможен прямой доступ. Эти ограничения реализованы аппаратно, но могут быть настроены во время выполнения на микроконтроллерах, которые имеют модуль MPU (рис. 5.1).

Как правило, в этих областях отображаются только небольшие участки, соответствующие физическим компонентам. Попытка доступа к памяти, которая не привязана к какому-либо оборудованию, вызывает исключение в ЦП. Планируя разработку ПО для определенной платформы, важно знать расположение и размеры разделов памяти, соответствующих архитектуре микроконтроллера, чтобы правильно описать геометрию доступного адресного пространства в сценарии компоновщика и в исходном коде.

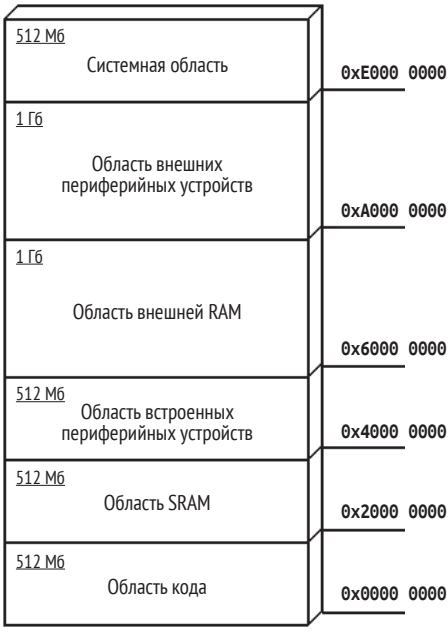

Рис. 5.1 ❖ Адресное пространство ARM Cortex-M

5.2.2. Область исполняемого кода

Нижние 512 МБ адресного пространства в микроконтроллере Cortex-M зарезервированы для исполняемого кода. Целевые устройства, поддерживающие XIP, всегда отображают флеш-память в пределах этой области, и память обычно недоступна для записи во время выполнения кода. В наших предыдущих примерах разделы `.text` и `.rodata` отображаются в пределах этого региона, поскольку они остаются постоянными во время выполнения кода программы. Кроме того, в эту область помещаются начальные значения всех ненулевых определенных символов, и их необходимо явно скопировать и повторно сопоставить с доступным для записи сегментом, чтобы иметь возможность изменять их значение во время выполнения. Как вы уже знаете, таблица векторов прерываний (IVT) обычно располагается в начале отображаемой секции. В области кода могут отображаться несколько банков флеш-памяти. Области, связанные с физическими устройствами, должны быть известны заранее и зависят от конструкции оборудования. Одни микроконтроллеры сопоставляют область кода с адресом `0x00000000`, в то время как другие выбирают иной начальный адрес (например, `0x10000000` или `0x08000000`). Флеш-память STM32F4 отображается по адресу `0x08000000` и предоставляет псевдоним, чтобы к той же памяти можно было получить доступ во время выполнения, начиная с адреса `0x00000000`.

Примечание

Когда адрес флеш-памяти начинается с 0, указатели NULL могут быть разыменованы и будут указывать на начало области кода, которая обычно доступна для чтения. Хотя это технически нарушает стандарт C, во встроенном коде C в таких случаях принято читать с адреса 0x00000000 – например, для чтения начального указателя стека в IVT на ARM.

5.2.3. Области оперативной памяти

Банки внутренней оперативной памяти (ОЗУ) сопоставляются с адресами во втором блоке размером 512 Мбайт, начиная с адреса 0x20000000. Банки внешней памяти могут быть отображены в любом месте в области 1 ГБ, начиная с адреса 0x60000000. В зависимости от геометрии внутренней SRAM внутри микроконтроллера Cortex-M или размещения банков внешней памяти реально доступные области памяти могут отображаться в несмежных разных частях памяти в пределах допустимого диапазона. Управление памятью должно учитывать разрывы в физическом отображении и обращаться к каждому разделу отдельно. MPU STM32F407, например, имеет два несмежных отображаемых блока внутренней SRAM:

- 128 Кбайт SRAM по адресу 0x20000000 (в двух смежных блоках по 112 и 16 Кбайт);
- отдельный банк памяти Core-Coupled Memory (CCM) объемом 64 Кбайта, отображаемый по адресу 0x10000000.

Этот второй банк памяти тесно связан с ЦП и оптимизирован для критичных ко времени операций, что обеспечивает доступ с нулевым ожиданием со стороны ЦП.

В этом случае можно ссылаться на два блока как на две отдельные области в сценарии компоновщика:

```
flash (rx) : ORIGIN = 0x08000000, LENGTH = 256K
SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 128K
CCMSRAM(rwx) : ORIGIN = 0x10000000, LENGTH = 64K
```

Хотя область ОЗУ предназначена для данных, она обычно сохраняет разрешения на выполнение, поэтому участки кода можно загрузить в ОЗУ и выполнить в штатном режиме. Выполнение кода в оперативной памяти расширяет гибкость системы, позволяя нам обрабатывать участки кода перед их загрузкой в память. Двоичные файлы, которые не предназначены для непосредственного выполнения (in place), могут храниться на любом устройстве и в других форматах, даже с использованием алгоритмов сжатия или шифрования. С другой стороны, возможность использования разделов в ОЗУ для хранения исполняемого кода отнимает у системы драгоценную оперативную память. Все преимущества и недостатки должны быть тщательно продуманы заранее при проектировании системы, особенно с точки зрения реальных требований к оперативной памяти, исходящих от приложения.

5.2.4. Области доступа к периферийным устройствам

Область размером 512 Мбайт, следующая за внутренней областью ОЗУ, начиная с адреса `0x40000000`, зарезервирована для периферийных устройств, которые обычно встроены в микроконтроллер. В свою очередь, область размером 1 Гбайт, начинающаяся с адреса `0xA0000000`, используется для отображения микросхем внешней памяти и других устройств, которые могут быть отображены в памяти в адресном пространстве микроконтроллера, но не являются частью чипа. Для правильного доступа к периферийным устройствам конфигурация внутренних компонентов микроконтроллера и адреса отображаемых в памяти устройств должны быть известны заранее. В этой области невозможно размещение исполняемого кода.

5.2.5. Системная область

Самые старшие 512 Мбайт памяти Cortex-M зарезервированы для доступа к конфигурации системы и частным блокам управления. Эта область содержит регистры управления системой, которые используются для программирования процессора, и регистры управления периферийными устройствами, используемые для настройки устройств и периферийных модулей. Выполнение кода в этой области не разрешено, и область доступна, только когда процессор работает на привилегированном уровне, как более подробно объяснено в главе 10.

Доступ к аппаратным регистрам путем разыменования их общеизвестных адресов полезен для настройки и получения их значений во время выполнения. Но компилятор не видит разницы между назначением переменной, отображенной в ОЗУ, и регистром конфигурации в системном блоке управления. По этой причине компилятор часто считает хорошей идеей оптимизировать код, изменив порядок транзакций памяти, что на самом деле может привести к непредсказуемым последствиям, когда следующая операция зависит от правильного завершения предыдущей операции с памятью. По этой причине требуется особая осторожность при доступе к регистрам конфигурации, чтобы убедиться, что текущая операция с памятью завершена до того, как будет начата следующая.

5.2.6. Порядок транзакций памяти

В процессорах ARM система управления памятью не гарантирует, что транзакции памяти будут выполняться в том же порядке, что и инструкции, которые их генерируют. Порядок транзакций с памятью можно изменить, чтобы приспособиться к характеристикам оборудования, таким как состояние ожидания, необходимое для доступа к базовой физической памяти,

или механизм опережающего прогнозирования ветвлений, реализованный на уровне микрокода. В то время как микроконтроллеры Cortex-M гарантируют строгий порядок транзакций с участием периферийных устройств и системных областей, во всех других случаях код должен быть соответствующим образом выстроен и снабжен правильными барьерами памяти, чтобы гарантировать, что предыдущие транзакции памяти были завершены до начала следующей инструкции. Набор инструкций Cortex-M включает в себя три вида барьеров:

- барьер памяти данных (data memory barrier, DMB);
- барьер синхронизации данных (data synchronization barrier, DSB);
- барьер синхронизации инструкций (instruction synchronization barrier, ISB).

DSB – это *мягкий* барьер, вызываемый для обеспечения выполнения всех ожидающих транзакций до того, как произойдет следующая транзакция памяти. DSB используется для фактической приостановки выполнения до тех пор, пока не будут выполнены все ожидающие транзакции. Кроме того, ISB также очищает конвейер ЦП и обеспечивает повторную выборку всех новых инструкций после транзакций в памяти, тем самым предотвращая любые побочные эффекты, вызванные устаревшим содержимым памяти. Существует ряд случаев, когда требуется использование барьера:

- после обновления VTOR, чтобы изменить адрес IV;
- после обновления отображения памяти;
- во время выполнения кода, который изменяет сам себя.

5.3. Стек выполнения

Как было показано в предыдущей главе, приложение на «голом железе» начинает выполняться с пустой областью стека. Стек выполнения растет в обратном направлении, от старшего адреса, предоставленного при загрузке, к более низким адресам каждый раз, когда сохраняется новый элемент. Стек постоянно отслеживает цепочку вызовов функций, сохраняя точку ветвления при каждом вызове функции, но он также служит временным хранилищем во время выполнения функций. Переменные в пределах локальной области действия каждой функции хранятся внутри стека во время выполнения функции. По этой причине контроль за использованием стека является одной из наиболее важных задач при разработке встраиваемой системы.

Программирование встраиваемых систем требует, чтобы программист всегда был в курсе использования стека во время кодирования. Размещение в стеке больших объектов, таких как буферы канала обмена данными или длинные строки, как правило, не является хорошей идеей, учитывая, что место для стека всегда очень ограничено. Компилятору можно дать указание выдавать предупреждение каждый раз, когда пространство стека, необходимое для одной функции, превышает определенный порог, как, например, в этом коде:

```
void function(void)
{
   char buffer[200];
   read_serial_buffer(buffer);
}
```

При компиляции с параметром GCC `-Wstack-usage=100` будет выдано следующее предупреждение:

```
main.c: In function 'function':
main.c:15:6: warning: stack usage is 208 bytes [-Wstack-usage=]
```

Его можно перехватить во время компиляции.

Хотя этот механизм полезен для выявления чрезмерного использования локального стека, он неэффективен для выявления всех потенциальных переполнений стека в коде, поскольку вызовы функций могут быть вложенными, а их использование стека суммируется. Наша функция использует 208 байт стека всякий раз, когда она вызывается. Сюда входят 200 байт для размещения локальной переменной буфера в стеке и 8 дополнительных байт для хранения двух указателей: источника вызова в разделе кода, который хранится как точка возврата, и указателя кадра, который содержит старое расположение указателя стека до вызова.

По замыслу стек увеличивается каждый раз, когда вызывается функция, и снова сжимается, когда происходит возврат из функции. В данном случае особенно сложно оценить использование стека рекурсивными функциями. По этой причине следует по возможности избегать использования рекурсии в коде или сводить ее к минимуму и держать под строгим контролем, зная, что область памяти, зарезервированная для стека в целевом устройстве, невелика (рис. 5.2).

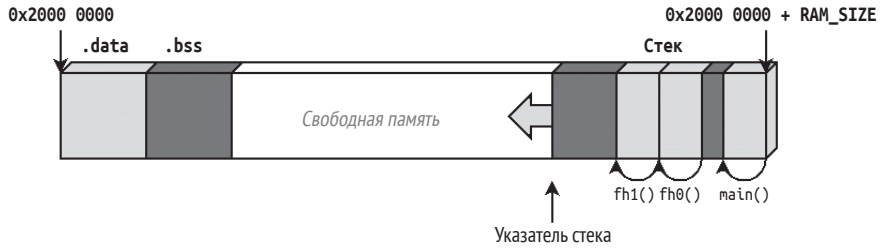

Рис. 5.2 ❖ При вызове функции указатель стека перемещается вниз для хранения указателей кадров и локальных переменных

5.3.1. Размещение стека

Начальный указатель на область стека можно выбрать при загрузке, задав нужный адрес памяти в первом слове IVT, что соответствует началу загружаемого во флеш-память двоичного образа.

Этот указатель может быть установлен во время компиляции различными способами. В простом примере из главы 4 показано, как можно назначить определенную область для стека или использовать символы, экспортированные из сценария компоновщика.

Использование сценария компоновщика для описания областей и сегментов памяти облегчает перенос кода на аналогичные платформы.

Поскольку STM32F407 предоставляет дополнительный банк памяти объемом 64 Кбайт по адресу `0x10000000`, можно зарезервировать нижние 16 Кбайт для стека выполнения, а остальные оставить в отдельном разделе для последующего использования. Сценарий компоновщика должен определить область сверху в блоке `MEMORY`:

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 1M
  SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 128K
  CCRAM(rwx) : ORIGIN = 0x10000000, LENGTH = 64K
}
```

Два символа теперь можно экспортировать в конец файла, присвоив постоянные, предварительно определенные значения:

```
_stack_size = 16 * 1024;
_stack_end = ORIGIN(CCRAM) + _stack_size;
```

Приложение может обращаться к значениям `_stack_size` и `_stack_end` как к обычным символам C. `_stack_end` размещается по адресу `0` при инициализации таблицы векторов, чтобы указать самый высокий адрес стека:

```
__attribute__ ((section(".isr_vector")))
void (* const IV[])(void) =
{
  (void (*)(void))(&_end_stack),
  isr_reset, // Reset
  isr_fault, // NMI
  isr_fault, // HardFault
  /* остальные подпрограммы обработки прерываний */
```

По возможности рекомендуется выделять под стек отдельную область памяти, как в этом случае. К сожалению, это возможно не на всех платформах.

Большинство встраиваемых устройств с отображением физической памяти предлагают единую непрерывную область отображения для всей оперативной памяти. Обычная стратегия, используемая для организации памяти в этих случаях, заключается в размещении указателя начала стека по наивысшему доступному адресу в конце отображаемой памяти. Таким образом, стек может свободно расти сверху вниз, в то время как приложение все еще может использовать память для выделения динамических объектов с самого нижнего адреса, который не используется никаким другим разделом. Хотя этот механизм считается наиболее эффективным, создавая иллюзию того, что можно использовать каждый байт доступной оперативной памяти, он

опасен, поскольку две области, растущие в противоположных направлениях, могут столкнуться, что приведет к непредсказуемым результатам.

5.3.2. Переполнение стека

Основная проблема с размером и размещением стека заключается в том, что очень сложно, если вообще возможно, восстановить работоспособность системы после возникновения переполнения стека в однопоточном приложении на «голом железе». Если стек автономно расположен в своей собственной физической области, такой как отдельный банк памяти, и если его нижняя граница не пересекается с областью отображения какого-либо устройства, переполнение стека вызовет исключение фатальной ошибки выполнения, которое может быть перехвачено, чтобы, например, остановить и корректно перезагрузить устройство.

Но если, например, соседняя память используется для иных целей, указатель стека может вторгнуться в соседний сегмент с большим риском повреждения данных в этой области памяти, что грозит катастрофическими последствиями, включая даже инъекцию вредоносного кода и выполнение произвольного кода атаки на целевое устройство. Наилучшая стратегия обычно состоит в выделении достаточного пространства стека при загрузке, максимально возможной изоляции стека от других разделов памяти и проверке использования стека во время выполнения. Настройка стека для использования наименьших доступных адресов в ОЗУ гарантирует, что переполнение стека приведет к фатальной ошибке выполнения, а не к вторжению в соседнюю область памяти. Впрочем, самый классический подход для системы на «голом железе» с одной непрерывной областью ОЗУ заключается в том, чтобы поместить начальный указатель стека на наивысший доступный адрес и смещать его в обратном направлении к более низким адресам. Сценарий компоновщика экспортирует самый верхний отображаемый адрес в качестве начального указателя стека:

```
_end_stack = ORIGIN(RAM) + LENGTH(RAM);
```

Доступная память между концом раздела `.bss` и самым нижним адресом в стеке может использоваться приложением для динамического распределения, и в то же время стек может расти в противоположном направлении. Это эффективный способ использовать всю доступную память, потому что стек не требует нижней границы, но он безопасен только до тех пор, пока общий объем памяти, используемой с обеих сторон, помещается в обозначенные области. Если разделам разрешено динамически увеличиваться в сторону более высоких адресов, всегда существует вероятность столкновения при встречном движении (рис. 5.3).

Конфликты между двумя смежными областями памяти – очень распространенные и опасные события во встраиваемых системах с одной непрерывной областью памяти. Решение, предложенное далее в этой главе, в разделе 5.5, может быть использовано для разделения памяти на два логических

блока путем вставки третьего недоступного блока посередине, а также для выявления и перехвата таких случаев.

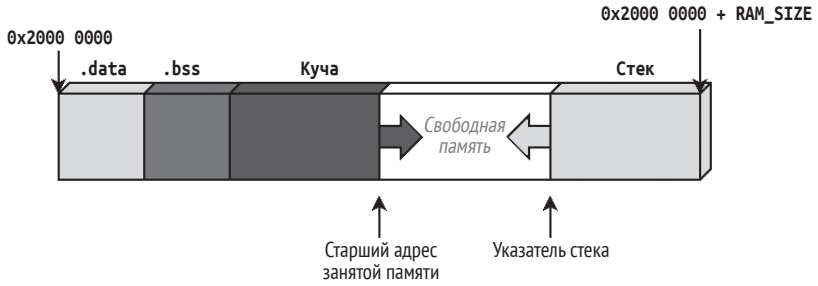

Рис. 5.3 ❖ Рост динамически выделяемой памяти (кучи) и стека в противоположных направлениях

5.3.3. Закрашивание стека

Эффективный способ измерения объема пространства стека, необходимого во время выполнения, состоит в заполнении предполагаемого пространства стека известным паттерном. Этот механизм, неофициально называемый *закрашиванием стека* (stack painting), показывает максимальное расширение стека выполнения в любое время. Запустив программное обеспечение с закрашенным стеком, через некоторое время можно измерить объем используемого стека, ища последний нетронутый шаблон закрашивания и предполагая, что указатель стека перемещался во время выполнения, но никогда не пересекал эту точку.

Можно выполнить закрашивание стека вручную в обработчике сброса во время инициализации памяти. Для этого нам нужно назначить область для закрашивания. В данном случае это будут последние 8 Кбайт памяти до `_end_stack`. Напомним, что при манипуляциях со стеком в функции `reset_handler` не следует использовать локальные переменные. Функция `reset_handler` сохранит значение текущего указателя стека в глобальной переменной `sp`:

```
static unsigned int sp;
```

В обработчик перед вызовом `main()` можно добавить следующий раздел:

```
asm volatile("mrs %0, msp" : "=r"(sp));
dst = ((unsigned int *)(&_end_stack)) - (8192 / sizeof(unsigned
int)); ;
while (dst < sp) {
  *dst = 0xDEADC0DE;
  dst++;
}
```

Первая ассемблерная инструкция используется для сохранения текущего значения указателя стека в переменной `sp`, гарантируя, что закрашивание

остановится после того, как область будет закрашена, но только до последнего неиспользуемого адреса в стеке (рис. 5.4).

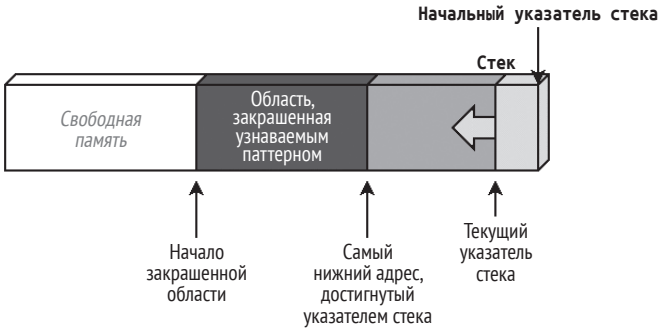

Рис. 5.4 ❖ Закрашивание области стека узнаваемым паттерном помогает оценить использование памяти стека в прототипе устройства

Текущее использование стека можно периодически проверять во время выполнения – например, в цикле `main` – для обнаружения области, окрашенной узнаваемым паттерном. Области, в которых сохранилась окраска, никогда не использовались стеком выполнения и указывают на объем доступного стека.

Этот механизм можно использовать для проверки объема стека, необходимого приложению для комфортной работы. В соответствии с проектом эта информация может быть использована позже для установки безопасного нижнего предела сегмента, который можно использовать для стека. Но закрашивание стека не всегда эффективно, поскольку дает измерение стека, используемого во время выполнения в штатном режиме, но может упускать из виду крайние и нестандартные случаи, когда использование стека может быть больше. Расширение охвата тестами при одновременном наблюдении за закрашиванием стека в конце каждого теста помогает выделить соответствующий объем пространства стека на этапе разработки.

5.4. Управление динамическим выделением памяти

Критически важные для безопасности встраиваемые системы часто проектируются без использования какого-либо динамического выделения памяти. Хотя это может показаться чрезмерной перестраховкой, такой подход сводит к минимуму влияние наиболее распространенных ошибок программирования в коде приложения, которые могут привести к катастрофическим последствиям для работающей системы.

С другой стороны, динамическое выделение памяти – мощный инструмент, поскольку он дает полный контроль над временем жизни и разме-

ром блоков памяти. Многие сторонние библиотеки, разработанные для встраиваемых устройств, предполагают наличие существующей реализации динамического выделения памяти. Динамическая память управляется через структуру так называемой *кучи* (heap) в памяти, путем отслеживания состояния и размера для каждого выделения, увеличения указателя на следующую область свободной памяти и повторного использования блоков, которые были освобождены, если обрабатываются новые запросы на выделение.

Стандартный программный интерфейс для динамического выделения памяти состоит из двух основных функций:

```
void *malloc(size_t size);
void free(void *ptr);
```

Эти сигнатуры функций определены стандартом ANSI-C и обычно встречаются в операционных системах. Они позволяют нам запрашивать новую область памяти заданного размера и освобождать ранее выделенную область, на которую ссылается определенный указатель. Более полное управление динамическим выделением поддерживает дополнительный вызов `realloc`, который позволяет нам изменять размер ранее выделенной области памяти либо на месте, либо путем ее перемещения в новый сегмент, достаточно большой, чтобы содержать объект заданного размера:

```
void *realloc(void *ptr, size_t size);
```

Хотя `realloc` обычно не используется в большинстве реализаций встраиваемых систем, в некоторых случаях может быть полезно изменить размер объектов в памяти.

В зависимости от реализации управление памятью может быть более или менее эффективным при объединении освободившихся смежных блоков для создания более крупных доступных сегментов без необходимости выделения нового пространства. Операционные системы реального времени обычно предлагают распределители с различным управлением динамическим выделением. Например, FreeRTOS предоставляет пять различных переносимых менеджеров выделения на выбор.

Если выбрано решение, допускающее динамическое распределение, важно разработать его с учетом нескольких важных факторов:

- геометрия динамически распределяемой области (кучи) и соседних областей;
- верхняя граница раздела, выделенного для кучи, если она используется совместно со стеком, для предотвращения коллизий кучи и стека;
- стратегия, применяемая в случае нехватки памяти для удовлетворения запросов на новые выделения;
- способы борьбы с фрагментацией памяти и минимизации накладных расходов на неиспользуемые блоки;
- использование отдельных пулов для разделения памяти, используемой конкретными объектами и модулями;
- распределение единого пула памяти по несмежным областям.

Когда на целевом устройстве нет доступного распределителя, например если ведется разработка приложения с нуля для «голого железа», нам может потребоваться реализовать распределитель, отвечающий характеристикам проекта. Это можно сделать либо с нуля, предоставив пользовательскую реализацию функций `malloc/free`, либо используя реализации, предоставляемые используемой библиотекой C. Первый подход дает полный контроль над фрагментацией, областями памяти и пулами, которые будут использоваться для реализации динамического выделения, в то время как второй скрывает большую часть работы, но при этом позволяет настраивать область памяти (как правило, непрерывную) и границы. В следующих двух разделах эти две возможные стратегии будут рассмотрены более подробно.

5.4.1. Пользовательская реализация

В отличие от серверов и персональных компьютеров, где выделение памяти осуществляется с использованием страниц определенного размера, во встраиваемых системах с «голым железом» куча обычно представляет собой одну или несколько смежных областей физической памяти, которые можно разделить внутри на произвольные блоки. Построение динамического выделения памяти на основе интерфейса `malloc/free` состоит в отслеживании запрошенных выделений в памяти. Обычно это делается путем прикрепления небольшого заголовка перед каждым выделением для отслеживания состояния и размера выделенного раздела, который можно использовать в функции `free` для проверки выделенного блока и его доступности для следующего выделения. Базовая реализация динамической памяти, начиная с первого доступного адреса после конца раздела `.bss`, может представлять каждый блок в памяти с помощью преамбулы, такой как следующая:

```
struct malloc_block {
  unsigned int signature;
  unsigned int size;
};
```

Для идентификации допустимых блоков и различения блоков, которые все еще используются, от блоков, которые уже были освобождены, могут быть назначены две разные подписи:

```
#define SIGNATURE_IN_USE (0xAAC0FFEE)
#define SIGNATURE_FREED (0xFEEDFACE)
#define NULL (((void *)0))
```

Функция `malloc` должна отслеживать наивысший адрес в куче. В этом примере статическая переменная используется для обозначения текущего конца кучи. Она устанавливается на начальный адрес и будет увеличиваться каждый раз, когда выделяется новый блок:

```
void *malloc(unsigned int size)
{
```

```
  static unsigned int *end_heap = 0;
  struct malloc_block *blk;
  char *ret = NULL;
  if (!end_heap) {
    end_heap = &_start_heap;
}
```

Следующие две строки гарантируют, что запрошенный блок выровнен по 32 битам для оптимизации доступа к `malloc_block`:

```
if (((size >>2) << 2) != size)
  size = ((size >> 2) + 1) << 2;
```

Затем функция `malloc` сначала ищет в куче раздел памяти, который был ранее освобожден:

```
blk = (struct malloc_block *)&_start_heap;
while (blk < end_heap) {
  if ((blk->signature == SIGNATURE_FREED) &&
       (blk->size >= size)) {
    blk->signature = SIGNATURE_IN_USE;
    ret = ((char *)blk) + sizeof(struct malloc_block);
    return ret;
  }
  blk = ((char *)blk) + sizeof(struct malloc_block) +
      blk->size;
}
```

Если доступный слот не найден или если ни один из них не является достаточно большим для запрошенного выделения, память выделяется в конце стека, и указатель соответствующим образом обновляется:

```
  blk = (struct malloc_block *)end_heap;
  blk->signature = SIGNATURE_IN_USE;
  blk->size = size;
  ret = ((char *)end_heap) + sizeof(struct malloc_block);
  end_heap = ret + size;
  return ret;
}
```

В обоих случаях возвращаемый адрес скрывает стоящую за ним управляющую структуру `malloc_block`. Переменная `end_heap` всегда указывает на конец последнего блока, выделенного в куче, но это не показатель используемой памяти, так как за это время могут быть освобождены промежуточные блоки. Этот пример функции `free`, демонстрирующий очень простой случай, выполняет только базовые проверки блока, который необходимо освободить, и устанавливает подпись, указывающую, что блок больше не используется:

```
void free(void *ptr)
{
  struct malloc_block *blk = (struct malloc_block *)
        (((char *)ptr)-sizeof(struct malloc_block));
```

```
  if (!ptr)
    return;
  if (blk->signature != SIGNATURE_IN_USE)
    return;
  blk->signature = SIGNATURE_FREED;
}
```

Этот пример очень упрощен и объясняет основные принципы динамического выделения памяти без учета всех ограничений реальной жизни. На самом деле выделение и освобождение объектов разных размеров может привести к фрагментации. Чтобы свести к минимуму влияние этого явления с точки зрения использования памяти и неиспользуемого пространства между активными выделениями, функция `free` должна, как минимум, реализовать какой-то механизм для объединения смежных областей, которые больше не используются. Кроме того, в предыдущем примере `malloc` предполагается, что область кучи не имеет верхней границы, не выполняется никакой проверки нового местоположения указателя `end_heap` и не определяется стратегия, когда нет доступной памяти для выделения.

Хотя наборы инструментов и библиотеки часто предоставляют реализацию `malloc` и `free` по умолчанию, реализация пользовательских механизмов динамического выделения памяти на основе кучи по-прежнему имеет смысл в тех случаях, когда доступные реализации не соответствуют требованиям, например если необходимо управлять отдельными пулами памяти или объединять отдельные разделы физической памяти, чтобы использовать их в одном пуле.

Проблемы фрагментации не могут быть полностью решены в системах с отображением физической памяти, потому что невозможно перемещать ранее выделенные блоки для оптимизации доступного пространства. Но эту проблему можно смягчить, контролируя количество выделений, максимально повторно используя выделенные блоки и избегая частых вызовов `malloc`/`free`, особенно для запроса блоков разного размера.

Использование динамической памяти, независимо от реализации, создает ряд проблем с безопасностью, и его следует избегать во всех жизненно важных системах и вообще там, где это не требуется. Более простые специализированные встраиваемые системы могут быть спроектированы таким образом, чтобы полностью избежать использования динамического распределения памяти. В этих случаях можно ограничиться простым интерфейсом `malloc`, реализующим постоянное выделение памяти во время запуска.

5.4.2. Использование библиотеки newlib

Наборы инструментов содержат различные утилиты, которые часто включают в себя механизмы динамического выделения памяти. Инструменты на основе GCC содержат сокращенный набор стандартных вызовов C, обычно во встроенной стандартной библиотеке C. Популярным элементом, часто включаемым во встроенный инструментарий ARM-GCC, является `newlib`.

Обеспечивая реализацию многих стандартных вызовов, newlib остается максимально гибкой, позволяя настраивать операции, связанные с оборудованием. Библиотека newlib может быть интегрирована как в однопоточные приложения на «голом железе», так и в операционную систему реального времени при условии реализации необходимых системных вызовов.

В случае использования malloc для newlib требуется реализация функции sbrk. Ожидается, что эта функция будет перемещать указатель кучи вперед каждый раз, когда для нового выделения памяти требуется расширение пространства кучи, и возвращать старое значение кучи функции malloc, чтобы завершать выделение памяти каждый раз, когда существующий, ранее освобожденный и повторно используемый блок не найден в пуле (рис. 5.5).

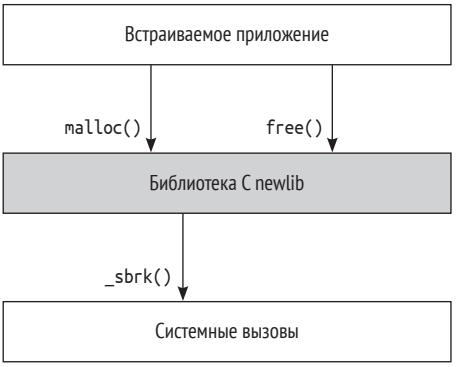

Рис. 5.5 ❖ Библиотека newlib реализует malloc и free и опирается на существующую реализацию _sbrk

Возможная реализация функции _sbrk показана в следующем примере кода:

```
void * _sbrk(unsigned int incr)
{
  static unsigned char *heap = NULL;
  void *old_heap = heap;
  if (((incr & 0x03) != incr)
    incr = ((incr >> 2) + 1) << 2;
  if (old_heap == NULL)
    old_heap = heap = (unsigned char *)&_start_heap;
  heap += incr;
  return old_heap;
}
```

Если код компонуется без флага -nostdlib, функции malloc и free, если они вызываются в любом месте кода, будут автоматически найдены в библиотеке newlib, встроенной в набор инструментов, и включены в окончательный двоичный файл. В этом случае отсутствие определения символа _sbrk приведет к ошибке компоновки.

5.4.3. Ограничение кучи

Во всех рассмотренных до сих пор функциях распределения памяти программное обеспечение не налагало ограничений на объем памяти, зарезервированной для кучи. В то время как переполнение стека часто трудно предотвратить и очень трудно восстановиться после него, исчерпание доступной памяти кучи зачастую может быть изящно обработано приложением (например путем отмены или отсрочки операции, требующей выделения памяти). В более сложных многопоточных системах операционная система может активно реагировать на нехватку памяти, завершая несущественные процессы, чтобы освободить память для новых выделений. Некоторые продвинутые системы, использующие механизмы подкачки страниц, такие как Linux, могут реализовывать перераспределение доступной памяти. Этот механизм гарантирует, что при выделении памяти никогда не произойдет сбой, а `malloc` никогда не вернет `NULL`, чтобы указать на сбой.

Вместо этого потребляющие память процессы в системе могут быть в любое время остановлены потоком ядра, чтобы освободить место для новых выделений из других менее ресурсоемких процессов. Во встраиваемой системе, особенно если нет многопоточности, лучший выбор – заставить распределитель возвращать `NULL`, когда в куче не осталось физического места, чтобы система могла продолжать работать, а приложение могло восстановиться, распознав ситуацию нехватки памяти. Раздел в памяти, выделенный для кучи, можно ограничить, экспортировав адрес ее верхней границы в скрипт компоновщика, как показано здесь:

```
_heap_end = ORIGIN(RAM) + LENGTH(RAM);
```

Код системы для реализации `malloc` из библиотеки `newlib` может учитывать новую верхнюю границу в функции `_sbrk()`:

```
void * _sbrk(unsigned int incr) {
  static unsigned char *heap = NULL;
  void *old_heap = heap;
  if (((incr & 0x03) != incr)
    incr = ((incr >> 2) + 1) << 2;
  if (old_heap == NULL)
    old_heap = heap = (unsigned char *)&_start_heap;
  if ((heap + incr) >= &_end_heap)
    return (void *)(-1);
  else
    heap += incr;
  return old_heap;
}
```

Специальное значение `(void *)(-1)`, возвращаемое `sbrk` в случае нехватки памяти для выделения кучи, указывает вызывающей функции `malloc`, что для выполнения запрошенного выделения недостаточно места. Затем `malloc` вернет `NULL` вызывающему процессу.

В этом случае очень важно, чтобы вызывающий процесс всегда проверял возвращаемое значение при каждом вызове `malloc()` и чтобы логика приложения могла правильно определить, что системе не хватает памяти, и отреагировать, пытаясь исправить это.

5.4.4. Несколько пулов памяти

В некоторых системах полезно хранить отдельные разделы в виде блоков динамической памяти, каждый из которых предназначен для определенной функции в системе. Механизмы динамического выделения памяти с использованием отдельных пулов могут быть реализованы по разным причинам, например для обеспечения того, чтобы определенные модули или подсистемы не использовали больше памяти, чем объем, назначенный им во время компиляции, или для того, чтобы выделения с одинаковым размером могли повторно использовать один и тот же объем физического пространства в памяти, снижая влияние фрагментации или даже выделяя предопределенную фиксированную область в памяти для операций прямого доступа к памяти периферийными или сетевыми устройствами. Разделы для разных пулов можно разграничить, как обычно, экспортируя символы в сценарий компоновщика. В следующем примере пространство в памяти предварительно выделяется для двух пулов по 8 и 4 Кбайта соответственно, расположенных в конце раздела `.bss` в ОЗУ:

```
PROVIDE(_start_pool0 = _end_bss);
PROVIDE(_end_pool0 = _start_pool0 + 8KB);
PROVIDE(_start_pool1 = _end_pool0);
PROVIDE(_end_pool1 = _start_pool1 + 4KB);
```

Необходимо определить пользовательскую функцию распределения, поскольку интерфейс `malloc` не поддерживает выбор пула, но функции можно сделать общими для обоих пулов. Глобальная структура может быть заполнена значениями, экспортируемыми компоновщиком:

```
struct memory_pool {
  void *start;
  void *end;
  void *cur;
};
static struct memory_pool mem_pool[2] = {
  {
    .start = &_start_pool0;
    .end = &_end_pool0;
  },
  {
    .start = &_start_pool1;
    .end = &_end_pool1;
  },
};
```

Функция должна принимать дополнительный аргумент для указания пула. Затем выполняется выделение по прежнему алгоритму, только меняются текущий указатель и границы выбранного пула. В этой версии ошибки нехватки памяти обнаруживаются перед перемещением текущего значения кучи вперед, возвращая NULL для уведомления вызывающей функции:

```
void *mempool_alloc(int pool, unsigned int size)
{
  struct malloc_block *blk;
  struct memory_pool *mp;
  char *ret = NULL;
  if (pool != 0 && pool != 1)
    return NULL;
  mp = mem_pool[pool];
  if (!mp->cur)
    mp->cur = mp->start;
  if (((size >>2) << 2) != size)
    size = ((size >> 2) + 1) << 2;
  blk = (struct malloc_block *)mp->start;
  while (blk < mp->cur) {
  if ((blk->signature == SIGNATURE_FREED) &&
      (blk->size >= size)) {
    blk->signature = SIGNATURE_IN_USE;
    ret = ((char *)blk) + sizeof(struct malloc_block);
    return ret;
  }
  blk = ((char *)blk) + sizeof(struct malloc_block) +
        blk->size;
  }
  blk = (struct malloc_block *)mp->cur;
  if (mp->cur + size >= mp->end)
    return NULL;
  blk->signature = SIGNATURE_IN_USE;
  blk->size = size;
  ret = ((char *)mp->cur) + sizeof(struct malloc_block);
  mp->cur = ret + size;
  return ret;
}
```

Этот механизм тоже не учитывает фрагментацию памяти, поэтому функция mempool_free может иметь ту же реализацию, что и free для упрощенного malloc, поскольку единственное, что необходимо сделать, – это пометить освобождаемые блоки как неиспользуемые.

В более полных случаях, когда free или отдельная подпрограмма сборщика мусора заботится о слиянии смежных освобожденных блоков, может потребоваться отслеживать освобожденные блоки в каждом пуле, в списке или в другой структуре данных, которую можно просмотреть для проверки возможности слияния.

5.4.5. Распространенные ошибки использования динамической памяти

Использование динамического выделения памяти считается небезопасным в некоторых средах, поскольку широко известно, что оно является источником неприятных ошибок, которые в целом являются критическими и очень трудными для выявления и исправления. Динамическое размещение сегментов иногда трудно отследить, особенно когда размер и сложность кода увеличиваются и вдобавок имеется много динамически выделяемых структур данных. Это серьезная проблема даже для многопоточных сред, где можно реализовать резервные механизмы, такие как завершение некорректно работающего приложения, но она становится критической для однопоточных встраиваемых систем, где такие ошибки часто фатальны. Наиболее распространенные типы ошибок при программировании с динамическим выделением памяти:

- разыменование указателя NULL;
- двойной вызов `free`;
- использование памяти после вызова `free`;
- сбой вызова `free`, что приводит к утечке памяти.

Некоторых из них можно избежать, соблюдая несколько простых правил. Функция `malloc` возвращает значение, которое всегда следует проверять перед использованием указателя. Это особенно важно в средах, где ресурсы ограничены, и распределитель может возвращать указатели `NULL`, указывающие на отсутствие доступной памяти для выделения. Предпочтительный подход заключается в наличии стратегии, которой нужно следовать, когда необходимая память недоступна. В любом случае перед попыткой разыменования все динамические указатели должны быть проверены, чтобы убедиться, что они не указывают на значение `NULL`.

Освобождение указателей `NULL` – допустимая операция, которая должна быть идентифицирована при вызове функции `free`. Если указатель имеет значение `NULL`, никаких действий (кроме проверки в начале функции) не выполняется и вызов игнорируется.

Также можно проверить, не освобождалась ли память до этого. В функции `free` реализована простая проверка сигнатуры структуры `malloc_block` в памяти. Можно было бы добавить сообщение журнала или даже точку останова для отладки причин второго вызова функции `free`:

```
if (blk->signature != SIGNATURE_IN_USE) {
  /* Обнаружен второй вызов free! */
  asm("BKPT #0") ;
  return;
}
```

К сожалению, этот механизм может работать только в некоторых случаях. На самом деле если блок, который был ранее освобожден, снова назначается распределителем, будет невозможно обнаружить дальнейшее использова-

ние его исходной ссылки, а второе освобождение приведет к потере второй ссылки. По той же причине трудно диагностировать ошибки использования памяти после освобождения, так как невозможно обнаружить, что к освобожденному блоку памяти снова обращались. Освобожденные блоки можно закрасить узнаваемым паттерном, так что если содержимое блока изменится после вызова `free`, то следующий вызов `malloc` для этого блока сможет обнаружить изменение. Но это опять же не гарантирует обнаружения всех случаев и работает только для доступа на запись к освобожденному указателю; этот прием не позволяет идентифицировать все случаи обращения к освобожденной памяти для чтения.

Утечки памяти легко обнаружить по факту сбоя системы, но иногда трудно локализовать причину. При ограниченных ресурсах часто бывает так, что при отсутствии *своевременного* освобождения памяти очень быстро расходуется вся доступная куча. Несмотря на то что существуют методы, используемые для отслеживания выделений памяти, часто бывает достаточно заглянуть в программу с помощью отладчика и найти повторяющиеся выделения одинакового размера, чтобы отследить ошибку, вызывающую утечку памяти.

Таким образом, из-за своей катастрофической и скрытной природы ошибки динамической памяти являются одной из самых больших проблем во встраиваемых системах. Разработка продуманного безопасного кода приложения часто требует меньше сил и времени, чем поиск ошибок памяти на системном уровне. Тщательный анализ времени жизни каждого выделенного объекта и максимально понятная и удобочитаемая логика могут предотвратить большинство проблем, связанных с обработкой указателей, и сэкономить много времени, которое в противном случае было бы потрачено на отладку.

5.5. Блок защиты памяти

В системе без сопоставления виртуальных адресов сложнее создать разделение между разделами, к которым может обращаться программное обеспечение во время выполнения. *Блок защиты памяти* (memory protection unit, MPU) является дополнительным компонентом многих микроконтроллеров на базе ARM. MPU применяется для разделения областей в памяти путем установки локальных разрешений и атрибутов. Этот механизм применяется, например, для предотвращения доступа к памяти, когда ЦП работает в пользовательском режиме, или запрета выполнения кода из доступных для записи мест в ОЗУ. Когда MPU включен, он применяет правила, вызывая прерывание исключения памяти, если эти правила нарушаются.

Хотя MPU обычно используют операционные системы для создания разделения стека процессов и обеспечения привилегированного доступа к системной памяти, он может быть полезен в ряде других случаев, включая приложения на «голом железе».

5.5.1. Регистры конфигурации MPU

В Cortex-M часть блока управления, связанная с конфигурацией MPU, расположена в блоке управления системой, начиная с адреса `0xE000ED90`. Для доступа к MPU предназначены пять регистров:

- **регистр типа MPU** (смещение `0x00`) содержит информацию о доступности системы MPU и количестве поддерживаемых областей. Этот регистр также доступен в системах без MPU, чтобы указать, что функциональность не поддерживается;
- **регистр управления MPU** (смещение `0x04`) используется для активации системы MPU и включения фонового сопоставления по умолчанию для всех областей, которые не отображаются явно в MPU. Если фоновое сопоставление не включено, доступ к несопоставленным областям запрещен;
- **регистр номера региона MPU** (смещение `RNR 0x08`) используется для выбора региона для настройки;
- **регистр базового адреса региона MPU** (смещение `RBAR 0x0C`) позволяет изменить базовый адрес выбранного региона;
- **регистр атрибутов и размера области MPU** (смещение `RASR 0x10`) определяет разрешения, атрибуты и размер выбранной области.

5.5.2. Программирование MPU

MPU микроконтроллеров Cortex-M поддерживает до восьми различных программируемых областей. Функция, которая включает MPU и настраивает все регионы, может быть вызвана в начале программы. Регистры MPU отображаются в библиотеках HAL, но в качестве примера будет определена собственная версия и получен к ним прямой доступ:

```
#define MPU_BASE 0xE000ED90
#define MPU_TYPE (*(volatile uint32_t *)(MPU_BASE + 0x00))
#define MPU_CTRL (*(volatile uint32_t *)(MPU_BASE + 0x04))
#define MPU_RNR (*(volatile uint32_t *)(MPU_BASE + 0x08))
#define MPU_RBAR (*(volatile uint32_t *)(MPU_BASE + 0x0c))
#define MPU_RASR (*(volatile uint32_t *)(MPU_BASE + 0x10))
```

В данном примере используются следующие определения значений битовых полей для установки правильных атрибутов в `RASR`:

```
#define RASR_ENABLED (1)
#define RASR_RW (1 << 24)
#define RASR_RDONLY (5 << 24)
#define RASR_NOACCESS (0 << 24)
#define RASR_SCB (7 << 16)
#define RASR_SB (5 << 16)
#define RASR_NOEXEC (1 << 28)
```

Возможные значения размера, которые должны помещаться в поле размера RASR в битах 1:5, кодируются следующим образом:

```
#define MPUSIZE_1K (0x09 << 1)
#define MPUSIZE_2K (0x0a << 1)
#define MPUSIZE_4K (0x0b << 1)
#define MPUSIZE_8K (0x0c << 1)
#define MPUSIZE_16K (0x0d << 1)
#define MPUSIZE_32K (0x0e << 1)
#define MPUSIZE_64K (0x0f << 1)
#define MPUSIZE_128K (0x10 << 1)
#define MPUSIZE_256K (0x11 << 1)
#define MPUSIZE_512K (0x12 << 1)
#define MPUSIZE_1M (0x13 << 1)
#define MPUSIZE_2M (0x14 << 1)
#define MPUSIZE_4M (0x15 << 1)
#define MPUSIZE_8M (0x16 << 1)
#define MPUSIZE_16M (0x17 << 1)
#define MPUSIZE_32M (0x18 << 1)
#define MPUSIZE_64M (0x19 << 1)
#define MPUSIZE_128M (0x1a << 1)
#define MPUSIZE_256M (0x1b << 1)
#define MPUSIZE_512M (0x1c << 1)
#define MPUSIZE_1G (0x1d << 1)
#define MPUSIZE_2G (0x1e << 1)
#define MPUSIZE_4G (0x1f << 1)
```

Первое, что нужно сделать, когда происходит вход в функцию mpu_enable, – это убедиться, что функциональность MPU доступна на нашем целевом устройстве, проверив регистр MPU_TYPE:

```
int mpu_enable(void)
{
  volatile uint32_t type;
  volatile uint32_t start;
  volatile uint32_t attr;
  type = MPU_TYPE;
  if (type == 0) {
    /* MPU отсутствует! */
    return -1;
}
```

Перед настройкой MPU необходимо убедиться, что он отключен, пока производится изменение базовых адресов и атрибутов каждого региона:

```
MPU_CTRL = 0;
```

Область флеш-памяти, содержащая исполняемый код, может быть помечена как доступная только для чтения область 0. Значения атрибутов RASR следующие:

```
start = 0;
attr = RASR_ENABLED | MPUSIZE_256K | RASR_SCB |
```

```
    RASR_RDONLY;
mpu_set_region(0, start, attr);
```

Вся область ОЗУ может отображаться как доступная для чтения-записи. Если нам не нужно выполнять код из ОЗУ, можно установить бит *запрета выполнения* (execute-never, XN) в атрибутах области. В этом случае ОЗУ отображается как область 1:

```
start = 0x20000000;
attr = RASR_ENABLED | MPUSIZE_64K | RASR_SCB | RASR_RW
    | RASR_NOEXEC;
mpu_set_region(1, start, attr);
```

Поскольку отображение памяти обрабатывается в том же порядке, что и номера областей памяти, можно использовать область 2 для создания исключения в области 1. Области с более высокими номерами имеют приоритет над областями с меньшими номерами, поэтому исключения могут быть созданы в рамках существующего сопоставления с меньшим числом.

Область 2 используется для определения защитной области как нижней границы для стека, растущего в обратном направлении; назначение этой области заключается в перехвате переполнения стека. Если в какой-то момент программа попытается получить доступ к защитной области, она вызовет исключение, и операция завершится неудачно. В данном случае защитная область занимает 1 Кбайт внизу стека. У нее нет разрешений на доступ, настроенных в ее атрибутах. MPU гарантирует, что область недоступна во время выполнения:

```
start = (uint32_t)(&_end_stack) - (STACK_SIZE + 1024);
attr = RASR_ENABLED | MPUSIZE_1K | RASR_SCB |
RASR_NOACCESS | RASR_NOEXEC;
mpu_set_region(2, start, attr);
```

Системная область описана как доступная для чтения и записи, неисполняемая и некешируемая, чтобы программа по-прежнему могла обращаться к системным регистрам после повторной активации MPU. Для этого используется область 3:

```
start = 0xE0000000;
attr = RASR_ENABLED | MPUSIZE_256M | RASR_SB
RASR_RW | RASR_NOEXEC;
mpu_set_region(3, start, attr);
```

В качестве последнего шага нужно снова включить MPU. Блок MPU позволяет нам определить *фоновую область*, установив разрешения по умолчанию для тех областей, которые не упомянуты в конфигурациях активного региона в явном виде. В нашем случае, согласно политике фоновой области, отсутствие определения приводит к запрету доступа ко всем областям, которые явно не сопоставлены:

```
  MPU_CTRL = 1;
  return 0;
}
```

Вспомогательная функция, которая устанавливает начальный адрес и атрибуты для областей памяти, выглядит следующим образом:

```
static void mpu_set_region(int region, uint32_t start, uint32_t
attr)
{
  MPU_RNR = region;
  MPU_RBAR = start;
  MPU_RNR = region;
  MPU_RASR = attr;
}
```

Значение, используемое для установки атрибутов и размеров в `MPU_RASR` в этом примере, определяется в соответствии со структурой самого регистра. `MPU_RASR` – это битовый регистр, содержащий следующие поля:

- **бит 0**: разрешение/запрет области;
- **биты 1:5**: размер раздела (см. специальные значения для назначения этому полю);
- **биты 16:18**: указывают, является ли память буферизуемой, кешируемой и совместно используемой соответственно. Устройства и системные регистры всегда должны быть помечены как некешируемые, чтобы гарантировать строгий порядок транзакций, как объяснялось в начале этой главы;
- **биты 24:26**: права доступа (чтение/запись), разделенные для режима пользователя и супервизора;
- **бит 28**: запрет выполнения (флаг `XN`).

Теперь можно разработать программу, переполняющую стек, и увидеть в отладчике разницу между случаями, когда функция `mpu_enable` вызывается и когда нет. Если MPU имеется на целевом устройстве, он может перехватывать переполнение стека, вызывая исключение в CPU (рис. 5.6).

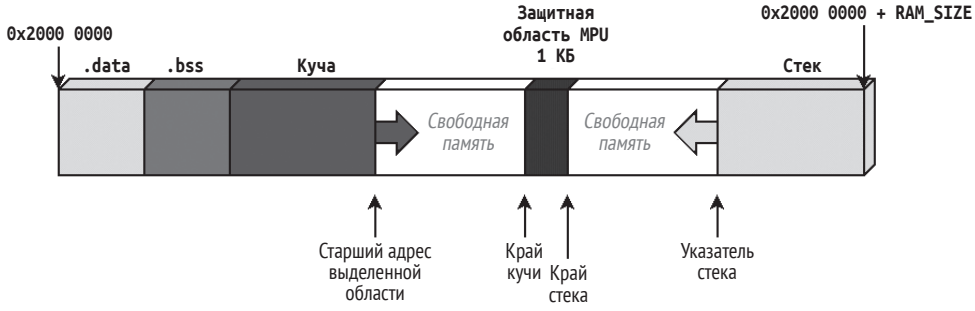

Рис. 5.6 ❖ Защитная область помечена как недоступная в MPU для предотвращения переполнения стека

Конфигурация, которая была использована для MPU в рассмотренном примере, очень строгая, она не разрешает доступ никуда, кроме областей, представляющих флеш-память и ОЗУ. Дополнительная защитная область размером 1 Кбайт гарантирует, что переполнение стека будет обнаружено

во время выполнения. Эта конфигурация фактически вводит искусственное разделение между двумя областями, выделенными для кучи и стека, в физически непрерывном пространстве путем введения блока, который реплицирует недоступные блоки между ними. Хотя само по себе выделение памяти, заходящей за край кучи, не вызовет переполнения стека, любая попытка доступа к памяти в защитной области вызовет ошибку памяти.

В реальных приложениях конфигурация MPU может быть более сложной и даже изменять свои значения во время выполнения. Например, в главе 10 объясняется, как можно использовать MPU для изоляции адресных пространств потоков в операционной системе реального времени.

5.6. Заключение

Управление памятью во встроенной системе является источником наиболее критических ошибок, и по этой причине необходимо уделять особое внимание разработке и внедрению правильных решений для используемой платформы и приложений. Стек выполнения должен быть тщательно размещен, измерен и разграничен, когда это возможно.

Системы, не использующие динамическое распределение памяти, более безопасны, но для встраиваемых систем высокой сложности методы динамического распределения могут быть полезны и даже необходимы. Программисты должны помнить, что ошибки в обработке памяти могут быть критическими для системы и их очень трудно обнаружить, поэтому требуется особая осторожность, когда код обрабатывает динамически выделяемые указатели.

MPU является очень важным и удобным инструментом для обеспечения прав доступа и атрибутов областей памяти, и его можно использовать для нескольких целей. В показанном выше примере реализован механизм на основе MPU, устанавливающий физическую границу для указателя стека.

В следующей главе будут рассмотрены другие распространенные компоненты, входящие в состав современных микроконтроллеров. Вы узнаете, как обращаться с настройками тактовой частоты, приоритетами прерываний, связью через порты ввода-вывода общего назначения и другими дополнительными функциями микроконтроллера.

Часть III

АППАРАТНЫЕ МОДУЛИ И ИНТЕРФЕЙС СВЯЗИ

В этой части книги рассмотрено программирование драйверов устройства и интерфейсов обмена данными. Она охватывает взаимодействие встраиваемой системы с внешним миром, вплоть до построения распределенных сетей через TCP/IP и повышения безопасности устройств IoT.

Эта часть состоит из следующих глав:

- главы 6 «Периферийные устройства общего назначения»;
- главы 7 «Интерфейсы локальной шины»;
- главы 8 «Энергопотребление и энергосбережение»;
- главы 9 «Распределенные системы и архитектура IoT».

Глава 6

Периферийные устройства общего назначения

Современные микроконтроллеры оснащены встроенными периферийными модулями, которые помогают создавать стабильные и надежные встраиваемые системы. Как только система запущена и начала работу, можно получить программный доступ к памяти и периферийным устройствам, а также настроить основные аппаратные функции. Затем необходимо инициализировать все компоненты системы путем активации соответствующих периферийных устройств через системные регистры, установки правильных частот для тактовых линий, а также настройки и активации прерываний. В этой главе описан интерфейс, предоставляемый микроконтроллером для доступа к встроенным периферийным устройствам и некоторым основным системным функциям. Основное внимание сосредоточено на следующих темах:

- контроллер прерываний;
- системное время;
- таймеры общего назначения;
- универсальный ввод/вывод (GPIO);
- сторожевой таймер.

Хотя эти периферийные устройства часто доступны через библиотеки аппаратной поддержки, реализованные и распространяемые производителями микросхем, наш подход основан на полном понимании принципов работы аппаратных компонентов и значения всех задействованных регистров. Это понимание будет достигнуто за счет настройки и использования функций микроконтроллера напрямую через интерфейс, предоставляемый аппаратной логикой.

При разработке драйверов для конкретной платформы необходимо изучить интерфейс, предоставляемый микроконтроллером для доступа к периферийным устройствам и функциям ЦП. В показанных далее примерах в качестве тестового устройства используется популярный микроконтроллер

STM32F4. Проверка возможной реализации на тестовой платформе позволяет нам лучше понять, как работать с более сложными целевыми устройствами, предоставляющими аналогичные функции, опираясь на документацию, предоставленную производителем чипа.

6.1. Технические требования

Вы можете скачать файлы кода для этой главы на GitHub по адресу https://github.com/PacktPublishing/Embedded-Systems-Architecture-Second-Edition/tree/main/Chapter6.

6.1.1. Побитовые операции

В примерах кода этой главы широко используются побитовые операции для проверки, установки и очистки отдельных битов в больших регистрах (в большинстве случаев 32-битных). Вы уже должны быть знакомы с побитовыми логическими операциями в C.

В примерах обычно используются следующие операции:

- **установка N-го бита в регистре R с помощью присваивания** `R |= (1 << N)`: новое значение регистра R будет содержать результат побитовой операции ИЛИ между его исходным значением и битовой маской, содержащей все нули, кроме бита в позиции, которую нужно установить в единицу;
- **очистка (сброс) N-го бита в регистре R с помощью присваивания** `R &= ~(1 << N)`: новое значение регистра является результатом побитовой операции И между исходным значением и битовой маской, содержащей все единицы, за исключением нулевого значения бита в позиции, которую нужно очистить;
- **проверка того, установлен или сброшен N-й бит регистра R, через условие** `(R & (1 << N) == (1 << N))`: возвращает `true`, только если установлен N-й бит регистра.

6.2. Контроллер прерываний

Точность систем реального времени повысилась благодаря быстрому развитию современных встроенных систем, в частности благодаря появлению контроллеров прерываний со сложной логикой. Назначение различных приоритетов линиям прерываний гарантирует меньшую задержку прерывания для источников с более высоким приоритетом и заставляет систему быстрее реагировать на события с приоритетом. Но прерывания могут возникать в любое время, в том числе в ходе выполнения другой программы обслужи-

вания прерываний. В этом случае контроллер прерываний объединяет обработчики прерываний в цепочку, а порядок выполнения зависит от уровней приоритета, назначенных источнику прерывания.

Одной из причин популярности семейства микропроцессоров Cortex-M среди разработчиков встраиваемых приложений реального времени, возможно, является конструкция их программируемого контроллера реального времени, а именно Nested Vector Interrupt Controller, или сокращенно NVIC. Он поддерживает до 240 источников прерываний, которые могут быть сгруппированы по 256 уровням приоритета, в зависимости от битов, зарезервированных для хранения приоритета в логике микропроцессора. Эти свойства делают его очень гибким, поскольку приоритеты можно изменить во время работы системы, что обеспечивает максимальную свободу выбора для программиста. Как вы уже знаете, NVIC подключается к таблице векторов, расположенной в начале области кода. Всякий раз, когда происходит прерывание, текущее состояние выполняющегося приложения автоматически помещается в стек процессором, и выполняется служебная процедура, связанная со строкой прерывания.

В системах, не имеющих механизма приоритета прерывания, реализуется обработка прерываний в обратном порядке. В этих случаях цепочка прерываний подразумевает, что контекст восстанавливается в конце выполнения первой служебной процедуры в очереди, а затем снова сохраняется при входе в следующую. NVIC реализует механизм последовательного сцепления для выполнения вложенных прерываний. Если одно или несколько прерываний происходят во время выполнения другой служебной процедуры, операция извлечения, обычно выполняемая в конце прерывания для восстановления контекста из стека, будет отменена, и вместо этого контроллер извлечет местоположение второго обработчика в векторе прерываний, гарантируя, что он будет выполнен сразу после первого. Из-за увеличения скорости операций сохранения и восстановления стека, реализуемых аппаратно, задержка прерывания значительно снижается во всех тех случаях, когда прерывания объединены в цепочку. Благодаря своей аппаратной реализации NVIC позволяет нам изменять параметры во время работы системы и может перетасовывать порядок выполнения процедур обслуживания прерываний, связанных с ожидающими сигналами, в соответствии с уровнями приоритета. Более того, одно и то же прерывание не может выполняться дважды в одной и той же цепочке обработчиков, что может быть вызвано изменением приоритетов в других обработчиках. Это обеспечивается логикой NVIC, которая гарантирует отсутствие петель в цепочке.

6.2.1. Настройка прерываний от периферийных устройств

Каждая линия прерывания может быть включена и отключена с помощью регистров NVIC в областях `NVIC_ISER` и `NVIC_ICER`, расположенных по адресам `0xE000E100` и `0xE000E180` соответственно. Если целевое устройство поддержи-

вает более 32 внешних прерываний, массивы 32-битных регистров отображаются в одних и тех же местах. Каждый бит в регистрах используется для активации определенной линии прерывания, связанной с позицией бита в этом конкретном регистре. Например, на микроконтроллере STM32F4, чтобы активировать линию прерывания контроллера последовательного периферийного интерфейса (SPI) `SPI1`, которая связана с номером 35, во втором регистре в области `NVIC_ISER` должен быть установлен четвертый бит.

Стандартная функция NVIC для включения прерывания активирует флаг, соответствующий номеру прерывания NVIC для источника, в ассоциированном регистре `NVIC_ISER`:

```
#define NVIC_ISER_BASE (0xE000E100)

static inline void nvic_irq_enable(uint8_t n)
{
  int i = n / 32;
  volatile uint32_t *nvic_iser =
    ((volatile uint32_t *)(NVIC_ISER_BASE + 4 * i));
  *nvic_iser |= (1 << (n % 32));
}
```

Точно так же, чтобы отключить прерывание, функция `nvic_irq_disable` активирует соответствующий бит в регистре очистки прерывания `NVIC_ICER`:

```
#define NVIC_ICER_BASE (0xE000E180)
static inline void nvic_irq_disable(uint8_t n)
{
  int i = n / 32;
  volatile uint32_t *nvic_icer =
    ((volatile uint32_t *)(NVIC_ICER_BASE + 4 * i));
  *nvic_icer |= (1 << (n % 32));
}
```

Приоритеты прерывания отображаются в массив 8-битных регистров, каждый из которых содержит значение приоритета для соответствующей строки прерывания, начиная с адреса `0xE000E400`, чтобы к ним можно было получить независимый доступ для изменения приоритета во время выполнения:

```
#define NVIC_IPRI_BASE (0xE000E400)
static inline void nvic_irq_setprio(uint8_t n,
    uint8_t prio)
{
  volatile uint8_t *nvic_ipri = ((volatile uint8_t *)
    (NVIC_IPRI_BASE + n));
  *nvic_ipri = prio;
}
```

Эти функции пригодятся для маршрутизации и приоритизации линий прерываний всякий раз, когда периферийное устройство использует прерывания.

6.3. Системное время

Наличие хронометража является основным требованием практически для любой встраиваемой системы. Микроконтроллер можно запрограммировать на запуск прерывания через равные промежутки времени, что обычно используется для равномерного приращения показаний системных часов. Для этого при запуске необходимо выполнить несколько шагов настройки, чтобы иметь стабильное прерывание по системному такту. Многие процессоры позволяют выбирать различную частоту тактовых импульсов при использовании единственного тактового генератора. Импульсы тактового генератора, который может быть внутренним или внешним по отношению к ЦП, используются для получения основных тактовых импульсов процессора. Логика настройки тактовой частоты, встроенная в ЦП, реализована с помощью контура *фазовой автоподстройки частоты* (phase-locked loop, PLL), который умножает входные тактовые сигналы от внешнего стабильного источника и формирует желаемые частоты, используемые ЦП и встроенными периферийными устройствами.

6.3.1. Настройка состояний ожидания флеш-памяти

Если код инициализации запускается из флеш-памяти, может потребоваться настройка интервала ожидания для флеш-памяти в соответствии с выбранной тактовой частотой. Если микропроцессор работает на высоких частотах, ему может потребоваться несколько интервалов ожидания между двумя последовательными операциями доступа к памяти с возможностью *выполнения прямых операций* над ячейками (execute-in-place, XIP). Несоблюдение правильных интервалов ожидания, соответствующих соотношению между скоростью ЦП и временем доступа к флеш-памяти, скорее всего, приведет к серьезной ошибке. Расположение регистров конфигурации флеш-памяти зависит от платформы; обычно они находятся в области, отведенной внутренним периферийным модулям. В STM32F407 регистры конфигурации флеш-памяти отображаются начиная с адреса `0x40023800`. *Регистр управления доступом* (access control register, ACR), к которому нам нужно получить доступ для настройки состояний ожидания, расположен в начале области:

```
#define FLASH_BASE (0x40023C00)
#define FLASH_ACR (*(volatile uint32_t *)(FLASH_BASE +
   0x00))
```

Младшие три бита в регистре `FLASH_ACR` используются для установки количества состояний ожидания. Согласно техническому описанию STM32F407, идеальное количество состояний ожидания для доступа к флеш-памяти при работе системы на частоте 168 МГц равно 5. Одновременно можно включить кеш данных и инструкций, активировав биты 10 и 9 соответственно:

```
void flash_set_waitstates(void) {
  FLASH_ACR = 5 | (1 << 10) | (1 << 9);
}
```

После того как состояния ожидания установлены, можно безопасно запускать код из флеш-памяти после установки частоты ЦП на более высокую скорость, поэтому можно приступить к фактической настройке модуля тактовых импульсов и их распределения по периферийным устройствам.

6.3.2. Настройка источника тактовых импульсов

Настройка источника тактовых импульсов в микроконтроллерах Cortex-M происходит через регистры *сброса и управления тактированием* (reset and clock control, RCC), расположенные по определенному адресу во внутренней периферийной области. Конфигурация RCC зависит от производителя чипа, поскольку она зависит от логики PLL, реализованной в микроконтроллере. Регистры описаны в документации микроконтроллера, и часто производитель чипа предоставляет пример исходного кода, демонстрирующий, как правильно настроить тактирование на микроконтроллере. На нашем тестовом устройстве STM32F407, при использовании внешнего генератора опорных тактовых импульсов с частотой 8 МГц, следующая процедура настраивает системный источник тактовых импульсов с частотой 168 МГц и гарантирует, что тактовые импульсы также распределяются по каждой периферийной шине. Следующий код инициализирует PLL требуемым значением и обеспечивает тактирование ядра ЦП с нужной частотой. Эта процедура распространена среди многих микроконтроллеров STM Cortex-M, и значения для настройки PLL можно получить из документации на микросхему или рассчитать с помощью программных средств, предоставляемых STMicroelectronics.

Примеры кода, представленные далее в этой главе, будут использовать ориентированный на конкретную систему модуль, экспортирующий функции, необходимые для настройки тактирования и установки задержки флеш-памяти. А сейчас проанализируем две возможные реализации конфигурации PLL на двух разных микроконтроллерах Cortex-M.

Чтобы получить доступ к конфигурации PLL в STM32F407-Discovery, сначала нужно определить некоторые макросы быстрого доступа к адресам регистров, доступных в RCC:

```
#define RCC_BASE (0x40023800)
#define RCC_CR (*(volatile uint32_t *)(RCC_BASE + 0x00))
#define RCC_PLLCFGR (*(volatile uint32_t *)(RCC_BASE +
     0x04))
#define RCC_CFGR (*(volatile uint32_t *)(RCC_BASE + 0x08))
#define RCC_CR (*(volatile uint32_t *)(RCC_BASE + 0x00))
```

Для лучшей читаемости и в расчете на возможное сопровождение кода в будущем нужно также определять мнемонику, связанную с однобитными значениями в соответствующих регистрах:

```
#define RCC_CR_PLLRDY (1 << 25)
#define RCC_CR_PLLON (1 << 24)
#define RCC_CR_HSERDY (1 << 17)
#define RCC_CR_HSEON (1 << 16)
#define RCC_CR_HSIRDY (1 << 1)
#define RCC_CR_HSION (1 << 0)

#define RCC_CFGR_SW_HSI 0x0
#define RCC_CFGR_SW_HSE 0x1
#define RCC_CFGR_SW_PLL 0x2
#define RCC_PLLCFGR_PLLSRC (1 << 22)

#define RCC_PRESCALER_DIV_NONE 0
#define RCC_PRESCALER_DIV_2 8
#define RCC_PRESCALER_DIV_4 9
```

Определим значения констант для конкретной платформы, используемые для настройки PLL:

```
#define CPU_FREQ (168000000)
#define PLL_FULL_MASK (0x7F037FFF)
#define PLLM 8
#define PLLN 336
#define PLLP 2
#define PLLQ 7
#define PLLR 0
```

Для краткости определим один дополнительный макрос, вызывающий ассемблерную инструкцию DMB, поскольку она будет использоваться в коде, чтобы гарантировать, что любая ожидающая передача из памяти в регистры конфигурации будет завершена до выполнения следующего оператора:

```
#define DMB() asm volatile ("dmb");
```

Следующая функция реализует последовательность инициализации PLL, чтобы установить правильную частоту процессора. Сначала она включает внутренний высокочастотный генератор и ждет, пока он будет готов, опрашивая CR:

```
void rcc_config(void)
{
  uint32_t reg32;
  RCC_CR |= RCC_CR_HSION;
  DMB();
  while ((RCC_CR & RCC_CR_HSIRDY) == 0)
    ;
```

Затем она выбирает внутренний генератор в качестве временного источника опорных тактовых импульсов:

```
reg32 = RCC_CFGR;
reg32 &= ~((1 << 1) | (1 << 0));
```

```
RCC_CFGR = (reg32 | RCC_CFGR_SW_HSI);
DMB();
```

Далее аналогичным образом активируется внешний опорный генератор:

```
RCC_CR |= RCC_CR_HSEON;
DMB();
while ((RCC_CR & RCC_CR_HSERDY) == 0)
  ;
```

На этом устройстве тактовые импульсы могут распределяться на все периферийные устройства через три системные шины. С помощью умножителей частота каждой шины может быть увеличена в два или четыре раза. Для целевого устройства установим тактовую частоту для HPRE, PPRE1 и PPRE2 на уровне 168, 84 и 46 МГц соответственно:

```
reg32 = RCC_CFGR;
reg32 &= ~0xF0;
RCC_CFGR = (reg32 | (RCC_PRESCALER_DIV_NONE << 4));
DMB();
reg32 = RCC_CFGR;
reg32 &= ~0x1C00;
RCC_CFGR = (reg32 | (RCC_PRESCALER_DIV_2 << 10));
DMB();
reg32 = RCC_CFGR;
reg32 &= ~0x07 << 13;
RCC_CFGR = (reg32 | (RCC_PRESCALER_DIV_4 << 13));
DMB();
```

В регистр конфигурации PLL записываем параметры для масштабирования частоты внешнего опорного генератора до нужного значения:

```
reg32 = RCC_PLLCFGR;
reg32 &= ~PLL_FULL_MASK;
RCC_PLLCFGR = reg32 | RCC_PLLCFGR_PLLSRC | PLLM |
  (PLLN << 6) | (((PLLP >> 1) - 1) << 16) |
  (PLLQ << 24);
DMB();
```

Затем активируется модуль PLL, и выполнение приостанавливается до тех пор, пока его выход не станет стабильным:

```
RCC_CR |= RCC_CR_PLLON;
DMB();
while ((RCC_CR & RCC_CR_PLLRDY) == 0);
```

Теперь можно выбрать PLL в качестве окончательного источника системных тактовых импульсов:

```
reg32 = RCC_CFGR;
reg32 &= ~((1 << 1) | (1 << 0));
RCC_CFGR = (reg32 | RCC_CFGR_SW_PLL);
DMB();
```

```
while ((RCC_CFGR & ((1 << 1) | (1 << 0))) !=
   RCC_CFGR_SW_PLL);
```

Внутренний опорный генератор больше не используется и может быть отключен. Управление возвращается вызывающей стороне, и на этом настройка тактирования системы завершена.

Как упоминалось ранее, процедура инициализации модуля тактирования строго зависит от конфигурации PLL в микроконтроллере. Чтобы правильно инициализировать источники тактовых импульсов, необходимых для работы ЦП и периферийных устройств на желаемых частотах, настоятельно рекомендуется тщательно изучить техническое описание микроконтроллера, предоставленное производителем чипа. В качестве второго примера можно проверить, как Quick EMUlator (QEMU) способен эмулировать поведение микроконтроллера LM3S6965. Эмулятор предоставляет виртуальный источник тактовых импульсов, который можно настроить с помощью той же процедуры инициализации, которая описана в документации производителя. На нашей платформе для настройки тактирования используются два регистра, называемых `RCC` и `RCC2`:

```
#define RCC (*(volatile uint32_t*))(0x400FE060)
#define RCC2 (*(volatile uint32_t*))(0x400FE070)
```

Чтобы сбросить регистры `RCC` в известное состояние, значение сброса должно быть записано в эти регистры при загрузке:

```
#define RCC_RESET (0x078E3AD1)
#define RCC2_RESET (0x07802810)
```

Этот микроконтроллер использует немаскированное системное прерывание (raw interrupt), чтобы уведомить, что PLL настроен на запрошенную частоту. Состояние прерывания можно проверить, прочитав бит 6 в регистре Raw Interrupt Status (RIS):

```
#define RIS (*(volatile uint32_t*))(0x400FE050)
#define PLL_LRIS (1 << 6)
```

Процедура настройки тактирования в этом случае начинается со сброса регистров `RCC` и установки соответствующих значений для настройки PLL. Модуль PLL настроен на генерацию тактовой частоты 400 МГц от источника генератора 8 МГц:

```
void rcc_config(void)
{
  RCC = RCC_RESET;
  RCC2 = RCC2_RESET;
  DMB();
  RCC = RCC_SYSDIV_50MHZ | RCC_PWMDIV_64 |
    RCC_XTAL_8MHZ_400MHZ | RCC_USEPWMDIV;
```

Результирующая частота 50 МГц для тактирования ядра ЦП получается из этой основной тактовой частоты 400 МГц с использованием системного делителя. Частота таковых импульсов сначала делится на 2, а затем на 4:

```
RCC2 = RCC2_SYSDIV2_4;
DMB();
```

Внешние генераторы включены:

```
RCC &= ~RCC_OFF;
RCC2 &= ~RCC2_OFF;
```

Делитель системных тактовых импульсов также включен. В то же время установка *бита обхода* (bypass bit) означает, что генератор используется как источник системных тактовых импульсов в обход PLL:

```
RCC |= RCC_BYPASS | RCC_USESYSDIV;
DMB();
```

Выполнение задерживается до тех пор, пока PLL не войдет в стабильное состояние и не зафиксируется на желаемой частоте:

```
while ((RIS & PLL_LRIS) == 0) ;
```

Теперь достаточно отключить биты обхода в регистрах RCC, чтобы начать использовать выход PLL для тактирования системы:

```
  RCC &= ~RCC_BYPASS;
  RCC2 &= ~RCC2_BYPASS;
}
```

6.3.3. Распределение тактовых импульсов

Когда становятся доступны импульсы тактирования шины, логику RCC можно запрограммировать для распределения тактовых импульсов на отдельные периферийные устройства. Для этого RCC содержит побитовые регистры источника синхронизации периферийных устройств. Установка соответствующего бита в одном из регистров включает тактирование для каждого отображаемого периферийного устройства в микроконтроллере. Каждый регистр может управлять тактированием 32 периферийных устройств.

Порядок периферийных устройств и, следовательно, соответствующий регистр и бит строго зависят от конкретных микроконтроллеров. STM32F4 имеет три регистра, предназначенных для этой цели. Например, чтобы включить тактирование для внутреннего сторожевого таймера, достаточно установить бит номер 9 в регистре по адресу 0x40021001c:

```
#define APB1_CLOCK_ER (*(uint32_t *)(0x4002001c))
#define WDG_APB1_CLOCK_ER_VAL (1 << 9)

APB1_CLOCK_ER |= WDG_APB1_CLOCK_ER_VAL;
```

Отключение тактирования неиспользуемого периферийного устройства экономит электроэнергию; следовательно, если целевое устройство поддерживает управляемое тактирование, в нем можно реализовать оптимизацию

и точную настройку энергопотребления, отключая отдельные периферийные устройства во время выполнения встроенного приложения.

6.3.4. Включение SysTick

После стабилизации частоты тактовых импульсов процессора можно настроить основной таймер в системе – SysTick. Поскольку специальный системный таймер присутствует не на всех версиях Cortex-M, иногда приходится использовать обычный вспомогательный таймер для отслеживания системного времени. Но в большинстве случаев прерывание SysTick можно включить, обратившись к регистрам конфигурации, которые находятся в блоке управления системой в области конфигурации системы. Во всех микроконтроллерах Cortex-M, которые имеют системный таймер, блок конфигурации располагается начиная с адреса `0xE000E010` и содержит четыре регистра:

- регистр управления/состояния (`SYSTICK_CSR`) со смещением `0`;
- регистр значения для перезагрузки (`SYSTICK_RVR`) со смещением `4`;
- регистр текущего значения (`SYSTICK_CVR`) со смещением `8`;
- регистр калибровки (`SYSTICK_CALIB`) со смещением `12`.

SysTick работает как таймер обратного отсчета. Он содержит 24-битное значение, которое уменьшается с каждым тактом ЦП. Таймер перезагружает одно и то же значение каждый раз, когда достигает `0`, и запускает прерывание SysTick, если он настроен на это.

В качестве ярлыка для доступа к регистрам SysTick определим их расположение:

```
#define SYSTICK_BASE (0xE000E010)
#define SYSTICK_CSR (*(volatile uint32_t *)(SYSTICK_BASE + 0x00))
#define SYSTICK_RVR (*(volatile uint32_t *)(SYSTICK_BASE + 0x04))
#define SYSTICK_CVR (*(volatile uint32_t *)(SYSTICK_BASE + 0x08))
#define SYSTICK_CALIB (*(volatile uint32_t *)(SYSTICK_BASE + 0x0C))
```

Поскольку частота ЦП известна в герцах, можно определить интервал системных тактов (прерываний), установив значение в *регистре значений перезагрузки* (reload value register, RVR). Например, чтобы получить интервал в 1 мс между двумя последовательными системными тактами, нужно поделить частоту на 1000 и вычесть 1, чтобы учесть значение таймера с отсчетом от нуля, гарантируя, что следующее прерывание произойдет после того, как счетчик уменьшится ровно N раз, что соответствует обратному отсчету от N – 1 до нуля. Также можно установить начальное значение таймера на 0, чтобы первое прерывание срабатывало сразу после включения обратного отсчета. После настройки интервала SysTick можно включить, настроив регистр управления/статуса. Значение трех младших битов CSR следующее:

- **бит 0**: включает обратный отсчет. После того как этот бит установлен, счетчик таймера SysTick автоматически уменьшается с каждым тактовым интервалом ЦП;

- **бит 1**: разрешает прерывание. Если этот бит установлен, когда счетчик достигает 0, будет сгенерировано прерывание SysTick;
- **бит 2**: выбор источника тактирования. Если этот бит сброшен, в качестве источника используется внешний опорный тактовый генератор. Тактовая частота ЦП используется в качестве источника, когда этот бит установлен.

Поскольку цель – определить собственный обработчик прерывания SysTick, поэтому тоже нужно установить бит 1. Поскольку тактовая частота ЦП и интервал следования системных тактов уже настроены правильно, нужно также установить бит 2. Последняя строка нашей подпрограммы `systick_enable` активирует три бита в CSR:

```
void systick_enable(void) {
  SYSTICK_RVR = ((CPU_FREQ / 1000) - 1);
  SYSTICK_CVR = 0;
  SYSTICK_CSR = (1 << 0) | (1 << 1) | (1 << 2);
}
```

Настроенный нами системный таймер идентичен тому, который используется операционными системами реального времени (RTOS) для инициирования переключения процессов. В нашем случае может быть полезно использовать непрерывные системные «настенные часы», просто измеряющие время, прошедшее с момента настройки часов. Простейшая подпрограмма обслуживания прерывания системного таймера может выглядеть примерно так:

```
volatile unsigned int jiffies = 0;
void isr_systick(void)
{
  ++jiffies;
}
```

Этой простой функции и связанной с ней глобальной переменной достаточно для прозрачного отслеживания времени на протяжении работы приложения. На самом деле прерывание по каждому системному такту происходит независимо от любых других процессов, строго через равные промежутки времени, и значение переменной `jiffies` увеличивается в обработчике прерывания без изменения потока основного приложения. Фактически каждый раз, когда значение счетчика системных тактов достигает 0, выполнение основного приложения приостанавливается, и очень быстро выполняется короткая подпрограмма обработки прерывания. После возврата из `isr_systick` поток основного приложения возобновляется, восстанавливая контекст выполнения, сохраненный в памяти за мгновение до обработки прерывания.

Причина, по которой переменная системного таймера должна быть определена и объявлена везде как `volatile`, заключается в том, что ее значение должно изменяться во время выполнения приложения независимо от поведения системы, возможно, предсказанного компилятором для локального контекста выполнения. Ключевое слово `volatile` в этом случае гарантирует, что компилятор вынужден создавать код, который проверяет значение

переменной каждый раз, когда она создается, путем запрета использования оптимизаций, основанных на ложном предположении, что переменная не изменяется локальным кодом.

Вот пример главной функции `main`, которая использует подготовленные нами ранее функции для загрузки системы, настройки тактирования и включения SysTick:

```
void main(void) {
  flash_set_waitstates();
  clock_config();
  systick_enable();
  while(1) {
    WFI();
  }
}
```

Здесь определен ярлык для инструкции ассемблера WFI (сокращение от wait for interrupt – ожидание прерывания). Он используется в основном приложении для удержания ЦП в неактивном состоянии до тех пор, пока не произойдет следующее прерывание:

```
#define WFI() asm volatile ("wfi")
```

Чтобы убедиться, что SysTick действительно работает, программу можно запустить с подключенным отладчиком и через некоторое время остановить. Если механизм системных тактов настроен правильно, переменная `jiffies` всегда должна отображать время в миллисекундах, прошедшее с момента загрузки.

6.4. Таймеры общего назначения

Наличие таймера SysTick не является обязательным для младших микроконтроллеров семейства. Некоторые устройства могут не иметь системного таймера, но все они предоставляют тот или иной интерфейс для программирования нескольких таймеров общего назначения, чтобы программа могла выполнять операции, зависящие от времени. Таймеры в целом очень гибки и просты в настройке и, как правило, способны запускать прерывания через регулярные промежутки времени. STM32F4 поддерживает до 17 таймеров, каждый из которых имеет разные характеристики. Таймеры обычно независимы друг от друга, так как каждый из них имеет собственную линию прерывания и отдельный сигнал управления тактированием. Например, на STM32F4 необходимо выполнить определенные шаги, необходимые для включения источника тактового сигнала и линии прерывания для таймера 2. Логика таймера основана на счетчике, значение которого увеличивается или уменьшается на каждом такте. Логика, реализованная на нашей базовой платформе, очень гибкая и поддерживает несколько функций, включая возможность внешнего тактирования через вывод микроконтроллера, воз-

можность конкатенации таймеров и даже внутреннюю реализацию таймера, которую можно запрограммировать. Таймер можно настроить на прямой или обратный отсчет, а также запускать события прерывания при различных значениях внутреннего счетчика. Таймеры могут быть однократными или непрерывными.

Абстракцию интерфейса таймера обычно можно найти в библиотеках поддержки, предоставляемых поставщиком микросхем, или в других библиотеках с открытым исходным кодом. Но чтобы лучше понять внутреннюю логику микроконтроллера, приведенный здесь пример кода снова напрямую связывается с периферийными устройствами с использованием регистров конфигурации.

В этом примере в основном используются настройки по умолчанию для таймера общего назначения на STM32F407. По умолчанию счетчик увеличивается на каждом такте до значения автоматической перезагрузки и постоянно генерирует прерывания при переполнении. Можно настроить предварительный делитель частоты тактовых импульсов для увеличения длительности интервалов между прерываниями. Чтобы начать генерировать прерывания, возникающие с постоянным заданным интервалом, необходимо получить доступ к нескольким регистрам:

- **регистры управления 1** и **2** (CR1 и CR2);
- **регистр прямого доступа к памяти** (DMA) / **регистр разрешения прерываний** (DIER);
- **регистр статуса** (SR);
- **счетчик предварительного делителя** (PSC);
- **регистр автообновления** (ARR).

Как правило, смещения для этих регистров одинаковы для всех таймеров, так что их адреса можно вычислить из базового адреса с помощью макроса. В данном случае определяется только регистр используемого таймера:

```
#define TIM2_BASE (0x40000000)
#define TIM2_CR1 (*(volatile uint32_t *)(TIM2_BASE + 0x00))
#define TIM2_DIER (*(volatile uint32_t *)(TIM2_BASE +
    0x0c))
#define TIM2_SR (*(volatile uint32_t *)(TIM2_BASE + 0x10))
#define TIM2_PSC (*(volatile uint32_t *)(TIM2_BASE + 0x28))
#define TIM2_ARR (*(volatile uint32_t *)(TIM2_BASE + 0x2c))
```

Кроме того, для лучшей читаемости кода определим некоторые соответствующие битовые позиции в регистрах, которые нужно настроить:

```
#define TIM_DIER_UIE (1 << 0)
#define TIM_SR_UIF (1 << 0)
#define TIM_CR1_CLOCK_ENABLE (1 << 0)
#define TIM_CR1_UPD_RS (1 << 2)
```

Прежде всего определим процедуру обслуживания прерывания. Интерфейс таймера требует, чтобы был очищен один флаг в регистре состояния, чтобы подтвердить прерывание. В этом простом случае все, что нужно делать, – это увеличивать локальную переменную, чтобы убедиться, что тай-

мер выполняется, проверив значение переменной в отладчике. Переменная `timer2_ticks` помечена как `volatile`, чтобы компилятор не оптимизировал ее, поскольку она никогда не используется в коде:

```
void isr_tim2(void)
{
  static volatile uint32_t timer2_ticks = 0;
  TIM2_SR &= ~TIM_SR_UIF;
  timer2_ticks++;
}
```

Процедура обслуживания прерывания должна быть связана с событием прерывания путем включения указателя на функцию в нужном месте внутри вектора прерывания, определенного в `startup.c`:

```
isr_tim2 , // TIM2_IRQ 28
```

Если таймер подключен к другой ветви дерева тактирования, как в этом случае, нам необходимо учитывать дополнительный коэффициент масштабирования между тактовой линией таймера и фактической тактовой частотой ЦП при вычислении значений для предварительного делителя и порога перезагрузки. Таймер 2 на STM32F407 подключен к шине Advanced Peripheral Bus (APB), которая работает на половине тактовой частоты процессора.

Применяемая нами инициализация является примером функции, которая автоматически вычисляет значения `TIM2_PSC` и `TIM2_ARR` и инициализирует таймер на основе заданного интервала, выраженного в миллисекундах. Переменная `clock` должна содержать частоту источника тактовых импульсов для таймера, которая может отличаться от частоты процессора.

Следующие определения специфичны для базовой платформы, отображая адрес для настройки способа подачи тактового сигнала и номер прерывания устройства, которое нужно использовать:

```
#define APB1_CLOCK_ER (*(volatile uint32_t *)(0x40023840))
#define APB1_CLOCK_RST (*(volatile uint32_t *)
     (0x40023820))
#define TIM2_APB1_CLOCK_ER_VAL (1 << 0)
#define NVIC_TIM2_IRQN (28)
```

А вот функция, которую нужно вызвать из `main`, чтобы включить непрерывно работающий таймер, генерирующий прерывания с желаемым интервалом:

```
int timer_init(uint32_t clock, uint32_t interval_ms)
{
  uint32_t val = 0;
  uint32_t psc = 1;
  uint32_t err = 0;
  clock = (clock / 1000) * interval_ms;
  while (psc < 65535) {
    val = clock / psc;
```

```
    err = clock % psc;
    if ((val < 65535) && (err == 0)) {
      val--;
      break;
    }
    val = 0;
    psc++;
  }
  if (val == 0)
    return -1;
  nvic_irq_enable(NVIC_TIM2_IRQN);
  nvic_irq_setprio(NVIC_TIM2_IRQN, 0);
  APB1_CLOCK_RST |= TIM2_APB1_CLOCK_ER_VAL;
  DMB();
  TIM2_PSC = psc;
  TIM2_ARR = val;
  TIM2_CR1 |= TIM_CR1_CLOCK_ENABLE;
  TIM2_DIER |= TIM_DIER_UIE;
  DMB();
  return 0;
}
```

Представленный здесь пример – лишь одно из возможных применений системных таймеров. На нашей базовой платформе таймеры можно использовать для различных целей, таких как измерение интервалов между импульсами, синхронизация друг с другом или периодическая активация сигналов с заданной частотой следования и длительностью рабочего цикла. Применение для последней из перечисленных целей будет объяснено в разделе 6.5.3 далее в этой главе. Для более детального знакомства с другими способами применения универсальных таймеров на целевом устройстве, пожалуйста, обратитесь к справочному руководству используемого микроконтроллера. Теперь, когда наша система сконфигурирована и готова к работе, и вы научились управлять системным временем и генерировать синхронные события, наконец пришло время представить наши первые периферийные устройства, чтобы начать общаться с внешним миром. В следующем разделе рассмотрены линии GPIO в их многочисленных конфигурациях, которые позволяют управлять напряжением или измерять его на отдельных выводах микроконтроллера.

6.5. Линии ввода/вывода общего назначения (GPIO)

Большинство выводов микросхемы микроконтроллера представляют собой настраиваемые линии ввода/вывода (иногда их называют *портами* ввода/вывода). Каждый вывод может быть сконфигурирован для выдачи логических уровней напряжения в качестве цифрового выхода или для считывания

логического состояния в качестве цифрового входа. Но некоторые выводы микроконтроллера могут быть связаны с альтернативными функциями, такими как аналоговый вход, последовательный интерфейс или выходной импульс от таймера. Выводы могут иметь несколько возможных конфигураций, но в каждый момент времени активна только одна из них. Контроллер GPIO предоставляет конфигурацию всех выводов и управляет ассоциацией контактов с периферийными модулями микроконтроллера, когда используются альтернативные функции.

6.5.1. Конфигурация выводов

В зависимости от логики контроллера GPIO выводы могут активироваться все вместе, по отдельности или группами. Чтобы реализовать драйвер для настройки выводов и их использования по мере необходимости, можно обратиться к техническому описанию микроконтроллера или к любому примеру реализации, предоставленному поставщиком чипа.

В случае STM32F4 контакты GPIO разделены на группы. Каждая группа подключена к отдельной линии тактирования, поэтому для использования выводов, связанных с группой, должны быть включены тактовые импульсы на соответствующей линии. Следующий код соединит источник тактового сигнала через контроллер GPIO с группой `D`:

```
#define AHB1_CLOCK_ER (*(volatile uint32_t *)(0x40023840))
#define GPIOD_AHB1_CLOCK_ER (1 << 3)
AHB1_CLOCK_ER |= GPIOD_AHB1_CLOCK_ER;
```

Регистры конфигурации, связанные с контроллерами GPIO, также сопоставляются с определенными адресами в области периферийных устройств. В случае контроллера `GPIOD` базовый адрес равен `0x40020C00`. В микроконтроллерах STM32F4 имеется 10 различных регистров для настройки и использования каждой группы цифровых входов/выходов. Поскольку группы состоят не более чем из 16 выводов, некоторые регистры могут использовать 2 бита на вывод:

- регистр режима (смещение `0` в адресном пространстве) выбирает режим (между цифровым входом, цифровым выходом, альтернативной функцией или аналоговым входом), используя 2 бита на контакт;
- регистр типа выхода (смещение `4`) выбирает логику управления выходным сигналом (двухтактный или открытый сток);
- регистр выходной скорости (смещение `8`) выбирает выходную скорость;
- регистр выбора нагрузки (смещение `12`) включает или отключает внутренний притягивающий или подтягивающий резистор;
- входные данные порта (смещение `16`), используется для чтения состояния цифрового входа;
- выходные данные порта (смещение `20`), содержит текущее значение цифрового выхода;
- установка/сброс бита порта (смещение `24`), используется для задания высокого или низкого уровня цифрового выходного сигнала;

- блокировка конфигурации порта (смещение 28);
- регистр младшего бита альтернативной функции (смещение 32), 4 бита на вывод, выводы 0–7;
- регистр старшего бита альтернативной функции (смещение 36), 4 бита на вывод, выводы 8–15.

Вывод должен быть настроен перед использованием, а маршрутизация тактового сигнала настроена для передачи тактовых импульсов на группу. Конфигурации, доступные с этим контроллером GPIO, удобнее рассматривать на конкретных примерах.

6.5.2. Цифровой выход

Включение цифрового выхода возможно путем настройки режима вывода в битах регистра режима, соответствующих данному выводу. Чтобы иметь возможность управлять логическим уровнем на выводе D13, который также подключен к светодиоду на демонстрационной плате нашей базовой платформы, нам необходимо получить доступ к следующим регистрам:

```
#define GPIOD_BASE 0x40020c00
#define GPIOD_MODE (*(volatile uint32_t *)(GPIOD_BASE + 0x00))
#define GPIOD_OTYPE (*(volatile uint32_t *)(GPIOD_BASE + 0x04))
#define GPIOD_PUPD (*(volatile uint32_t *)(GPIOD_BASE + 0x0c))
#define GPIOD_ODR (*(volatile uint32_t *)(GPIOD_BASE + 0x14))
#define GPIOD_BSRR (*(volatile uint32_t *)(GPIOD_BASE + 0x18))
```

В следующих примерах для изменения назначения контактов используются коммутирующие функции. Здесь показаны два регистра, содержащих настройки коммутирующих функций:

```
#define GPIOD_AFL (*(volatile uint32_t *)(GPIOD_BASE + 0x20))
#define GPIOD_AFH (*(volatile uint32_t *)(GPIOD_BASE + 0x24))
```

Следующие простые функции предназначены для управления состоянием вывода D15, подключенного к синему светодиоду на отладочной плате STM32F4. Основная программа должна вызвать `led_setup` перед вызовом любой другой функции, чтобы настроить контакт как выход и активировать встроенный нагрузочный резистор:

```
#define LED_PIN (15)
void led_setup(void)
{
  uint32_t mode_reg;
```

Сначала настраиваем подачу тактовых импульсов для контроллера `GPIOD`:

```
AHB1_CLOCK_ER |= GPIOD_AHB1_CLOCK_ER;
```

Теперь изменяем содержимое регистра режима, чтобы настроить вывод GPIO D15 как цифровой выход. Операция выполняется в два этапа. Сначала

стирается любое предыдущее значение, установленное в 2 битах, соответствующих положению режима вывода в регистре:

```
GPIOD_MODE &= ~ (0x03 << (LED_PIN * 2));
```

В той же позиции записываем значение 1, означающее, что вывод теперь сконфигурирован как цифровой выход:

```
GPIOD_MODE |= 1 << (LED_PIN * 2);
```

Чтобы включить притягивающие и подтягивающие внутренние резисторы, нужно сделать то же самое. Значение, которое необходимо установить в этом случае, равно 2, что делается следующими инструкциями:

```
  GPIOD_PUPD &= ~(0x03 << (LED_PIN * 2));
  GPIOD_PUPD |= 0x02 << (LED_PIN * 2);
}
```

После вызова функции `setup` приложение и обработчики прерываний могут вызывать экспортированные функции для установки высокого или низкого значения вывода, воздействуя на регистр установки/сброса битов:

```
void led_on(void)
{
  GPIOD_BSRR |= 1 << LED_PIN;
}
```

Старшая половина `BSRR` используется для сброса выводов. Запись 1 в бит регистра сброса переводит логический уровень вывода в низкий уровень:

```
void led_off(void)
{
  GPIOD_BSRR |= 1 << (LED_PIN + 16);
}
```

Определим удобную функцию для переключения состояния светодиода с включенного на выключенное и наоборот:

```
void led_toggle(void)
{
  if ((GPIOD_ODR & (1 << LED_PIN)) == (1 << LED_PIN))
    led_off();
  else
    led_on();
}
```

Используя таймер, настроенный в предыдущем разделе, можно запустить небольшую программу, которая мигает синим светодиодом на плате STM32F407-Discovery. Функцию `led_toggle` можно вызвать из сервисной процедуры `timer`, реализованной в предыдущем разделе:

```
void isr_tim2(void)
{
```

```
  TIM2_SR &= ~TIM_SR_UIF;
  led_toggle();
}
```

В основной программе перед запуском таймера необходимо инициализировать драйвер светодиода:

```
void main(void) {
  flash_set_waitstates();
  clock_config();
  led_setup();
  timer_init(CPU_FREQ, 1, 1000);
  while(1)
  WFI();
}
```

Цикл `main` этой программы пуст. Действие `led_toggle` вызывается каждую секунду, чтобы мигать светодиодом.

6.5.3. Широтно-импульсная модуляция

Широтно-импульсная модуляция (ШИМ, PWM) широко применяется для управления различными типами приводов, кодирования сообщений в сигналы с разной длительностью импульса и, как правило, для генерации импульсов с фиксированной частотой и переменной скважностью на цифровых выходах для различных целей.

Интерфейс таймера позволяет назначать контакты для вывода ШИМ-сигнала. В нашем базовом микроконтроллере четыре выходных канала сравнения могут быть связаны с таймерами общего назначения, а выводы, подключенные к каналам `OC`, могут быть настроены для автоматического вывода сигнала ШИМ. На плате STM32F407-Discovery контакт PD15 синего светодиода, использованный в предыдущем примере для демонстрации функциональности цифрового выхода, связан с `OC4`, которым может управлять таймер 4. Согласно документации на микросхему, выбор альтернативной функции `2` для этого вывода напрямую соединяет его с `OC4`.

На рис. 6.1 показана конфигурация контактов для использования альтернативной функции 2 с подключением к выходу таймера.

Вывод инициализируется и настраивается для использования альтернативной конфигурации вместо простого цифрового выхода путем очистки битов регистра `MODE` и записи значения `2`:

```
GPIOD_MODE &= ~ (0x03 << (LED_PIN * 2));
GPIOD_MODE |= (2 << (LED_PIN * 2));
```

Выводы с 0 по 7 в этой группе GPIO используют по 4 бита каждый в регистре `AFL` контроллера `GPIOD`. Старшие выводы (в диапазоне 8–15) используют по 4 бита в регистре `AFH`. Когда требуется альтернативный режим, нужный номер альтернативной функции программируется в 4 битах, связанных с контактом 15, поэтому в данном случае используется регистр `AFH`:

```
uint32_t value;
if (LED_PIN < 8) {
  value = GPIOD_AFL & (~(0xf << (LED_PIN * 4)));
  GPIOD_AFL = value | (0x2 << (LED_PIN * 4));
} else {
  value = GPIOD_AFH & (~(0xf << ((LED_PIN - 8) * 4)));
  GPIOD_AFH = value |(0x2 << ((LED_PIN - 8) * 4));
}
```

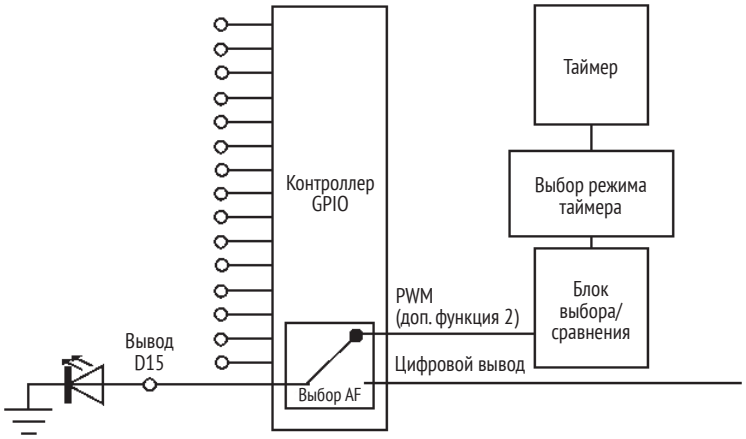

Рис. 6.1 ❖ Настройка вывода D15 для использования альтернативной функции 2 подключает его к выходу таймера

Функция `pwm_led_init()`, которую можно вызвать из основной программы для настройки вывода PD15 светодиода, будет выглядеть следующим образом:

```
void led_pwm_setup(void)
{
  AHB1_CLOCK_ER |= GPIOD_AHB1_CLOCK_ER;
  GPIOD_MODE &= ~ (0x03 << (LED_PIN * 2));
  GPIOD_MODE |= (2 << (LED_PIN * 2));
  GPIOD_OSPD &= ~(0x03 << (LED_PIN * 2));
  GPIOD_OSPD |= (0x03 << (LED_PIN * 2));
  GPIOD_PUPD &= ~(0x03 << (LED_PIN * 2));
  GPIOD_PUPD |= (0x02 << (LED_PIN * 2));
  GPIOD_AFH &= ~(0xf << ((LED_PIN - 8) * 4));
  GPIOD_AFH |= (0x2 << ((LED_PIN - 8) * 4));
}
```

Функция, настраивающая таймер для генерации ШИМ, аналогична той, которая используется в простом таймере для генерации прерывания в примере с цифровым выходом, за исключением того, что настройка таймера для вывода ШИМ включает изменение значения четырех дополнительных регистров:

- регистр разрешения захвата/сравнения (CCER);
- регистры 1 и 2 режима захвата/сравнения (CCMR1 и CCMR2);
- конфигурация канала захвата 4 (CC4).

Функция, которая используется в примере для настройки ШИМ с заданным рабочим циклом, имеет следующую сигнатуру:

```
int pwm_init(uint32_t clock, uint32_t dutycycle)
{
```

По-прежнему необходимо включить тактовые импульсы для таймера 4:

```
APB1_CLOCK_RST &= ~TIM4_APB1_CLOCK_ER_VAL;
APB1_CLOCK_ER |= TIM4_APB1_CLOCK_ER_VAL;
```

И таймер, и его выходные каналы сравнения временно отключены, чтобы начать настройку с чистого листа:

```
TIM4_CCER &= ~TIM_CCER_CC4_ENABLE;
TIM4_CR1 = 0;
TIM4_PSC = 0;
```

В этом примере задана фиксированная частота ШИМ 100 кГц, установив значение автоматической перезагрузки на `1/100000` входного тактового сигнала и запретив использование предварительного делителя:

```
uint32_t val = clock / 100000;
```

Рабочий цикл рассчитывается в соответствии со значением, которое передается в качестве второго параметра функции `pwm_init()` и выражается в процентах. Для расчета соответствующего порогового уровня используется простая формула, так что, например, значение 80 означает, что вывод ШИМ будет активен в течение 4/5 времени (80 % цикла). Полученное значение уменьшается на единицу, но только если оно не равно нулю, чтобы избежать отрицательного переполнения:

```
lvl = (val * threshold) / 100;
if (lvl != 0)
lvl--;
```

Затем выполняется запись в регистр значения компаратора `CCR4` и регистр значения автоматической перезагрузки `ARR`. Кроме того, в этом случае значение `ARR` также уменьшается на 1, поскольку счетчик отсчитывает от нуля:

```
TIM4_ARR = val - 1;
TIM4_CCR4 = lvl;
```

Чтобы правильно настроить сигнал ШИМ на этой платформе, сначала нужно должным образом очистить части регистра `CCMR1`, которые нужно настроить. К ним относятся выбор захвата и настройка режима:

```
TIM4_CCMR1 &= ~(0x03 << 0);
TIM4_CCMR1 &= ~(0x07 << 4);
```

Выбранный режим PWM1 является лишь одной из возможных альтернативных конфигураций, основанных на таймере захвата/сравнения. Для включения режима устанавливаем значение PWM1 в CCMR2, предварительно очистив соответствующие биты регистров:

```
TIM4_CCMR1 &= ~(0x03 << 0);
TIM4_CCMR1 &= ~(0x07 << 4);
TIM4_CCMR1 |= TIM_CCMR1_OC1M_PWM1;
TIM4_CCMR2 &= ~(0x03 << 8);
TIM4_CCMR2 &= ~(0x07 << 12);
TIM4_CCMR2 |= TIM_CCMR2_OC4M_PWM1;
```

Включим выходной компаратор OC4. Затем таймер настраивается на автоматическую перезагрузку сохраненного значения каждый раз, когда счетчик переполняется:

```
  TIM4_CCMR2 |= TIM_CCMR2_OC4M_PWM1;
  TIM4_CCER |= TIM_CCER_CC4_ENABLE;
  TIM4_CR1 |= TIM_CR1_CLOCK_ENABLE | TIM_CR1_ARPE;
}
```

Использование ШИМ для управления напряжением, подаваемым на светодиод, изменяет его яркость в соответствии с длительностью рабочего цикла. Следующий пример кода уменьшает видимую яркость светодиода до 50 % по сравнению с яркостью светодиода, питаемого от выхода постоянного напряжения в предыдущем примере с цифровым выходом:

```
void main(void) {
  flash_set_waitstates();
  clock_config();
  led_pwm_setup();
  pwm_init(CPU_FREQ, 50);
  while(1)
    WFI();
}
```

Влияние ШИМ на яркость светодиода можно продемонстрировать более наглядно, динамически изменяя рабочий цикл. Например, можно настроить второй таймер для генерации прерывания каждые 50 мс. В обработчике прерывания коэффициент заполнения циклически изменяется в диапазоне 0–80 % и обратно за 16 шагов. На первых 8 шагах рабочий цикл увеличивается на 10 % при каждом прерывании, от 0 до 80 %, а на последних 8 шагах он уменьшается с той же скоростью, возвращая рабочий цикл обратно к 0:

```
void isr_tim2(void) {
  static uint32_t tim2_ticks = 0;
  TIM2_SR &= ~TIM_SR_UIF;
  if (tim2_ticks > 16)
    tim2_ticks = 0;
  if (tim2_ticks > 8)
    pwm_init(master_clock, 10 * (16 - tim2_ticks));
  else
```

```
    pwm_init(master_clock, 10 * tim2_ticks);
  tim2_ticks++;
}
```

Если настроить таймер 2 в основной программе для генерации прерывания с постоянным интервалом, как в предыдущих примерах, можно увидеть, как яркость светодиода циклично пульсирует, плавно увеличиваясь и уменьшаясь.

В этом случае таймер 2 инициализируется основной программой, и связанный с ним обработчик прерываний 20 раз в секунду обновляет настройки таймера 4:

```
void main(void) {
  flash_set_waitstates();
  clock_config();
  led_pwm_setup();
  pwm_init(CPU_FREQ, 0);
  timer_init(CPU_FREQ, 1, 50);
  while(1)
    WFI();
}
```

6.5.4. Цифровой вход

Вывод GPIO, настроенный в режиме цифрового входа, определяет *логический* уровень приложенного к нему напряжения. Логическое значение всех входных контактов контроллера GPIO можно считать из *регистра входных данных* (input data register, IDR). На базовой плате контакт `A0` подключен к пользовательской кнопке, поэтому состояние кнопки можно в любое время прочитать из приложения.

Контроллер `GPIOA` можно включить, подав на него тактовые импульсы:

```
#define AHB1_CLOCK_ER (*(volatile uint32_t *)(0x40023830))
#define GPIOA_AHB1_CLOCK_ER (1 << 0)
```

Сам контроллер отображается по адресу `0x40020000`:

```
#define GPIOA_BASE 0x40020000
#define GPIOA_MODE (*(volatile uint32_t *)(GPIOA_BASE + 0x00))
#define GPIOA_IDR (*(volatile uint32_t *)(GPIOA_BASE + 0x10))
```

Чтобы настроить контакт микроконтроллера как вход, достаточно включить режим `0`, очистив два бита режима для контакта `0`:

```
#define BUTTON_PIN (0)
void button_setup(void)
{
  AHB1_CLOCK_ER |= GPIOA_AHB1_CLOCK_ER;
  GPIOA_MODE &= ~ (0x03 << (BUTTON_PIN * 2));
}
```

Теперь приложение может в любой момент проверить состояние кнопки, прочитав младший бит IDR. При нажатии кнопки опорное напряжение подключается к выводу, а значение бита, соответствующего выводу, изменяется с `0` на `1`:

```
int button_is_pressed(void)
{
  return (GPIOA_IDR & (1 << BUTTON_PIN)) >> BUTTON_PIN;
}
```

6.5.5. Ввод, управляемый прерыванием

Необходимость постоянно считывать состояние вывода путем постоянного опроса IDR во многих случаях неудобна, когда приложение должно реагировать на изменения состояния в непредсказуемые моменты времени. Микроконтроллеры обычно обладают механизмом для подключения цифровых входов к линиям прерывания, чтобы приложение могло реагировать в реальном времени на события, связанные с вводом, поскольку в момент изменения состояния ввода выполнение автоматически прерывается для выполнения соответствующей служебной процедуры.

В нашем базовом микроконтроллере контакт A0 может быть подключен к *контроллеру внешних прерываний* (external interrupt, EXTI) по событиям. EXTI имеет триггеры обнаружения фронта сигнала, которые можно связать с линиями прерывания. Номер контакта в группе GPIO определяет номер прерывания EXTI, связанного с ним, поэтому процедура обработки прерывания EXTI 0 может быть подключена к контакту 0 любой группы GPIO, если это необходимо (рис. 6.2).

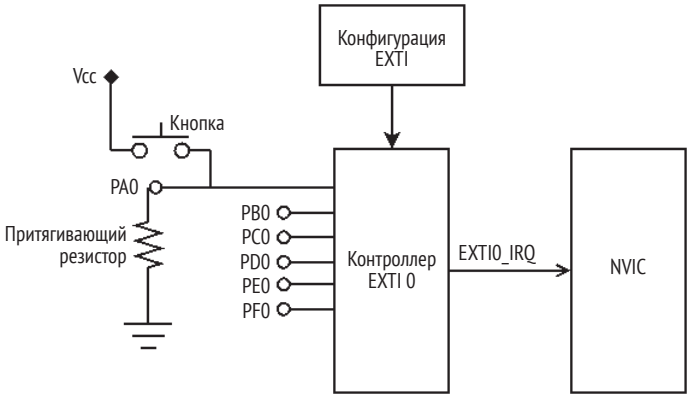

Рис. 6.2 ❖ Контроллер EXTI0, связывающий триггеры обнаружения фронта с пользовательской кнопкой, подключенной к PA0

Для связывания PA0 с EXTI 0 необходимо изменить регистр конфигурации EXTI, чтобы задать номер группы GPIO в битах, связанных с EXTI 0.

В STM32F4 регистры конфигурации EXTI (EXTI_CR) расположены по адресу 0x40013808. Каждый регистр используется для настройки контроллера прерываний, связанного с линией EXTI. Младшие четыре бита первого регистра относятся к линии EXTI 0. Группа A GPIO обозначается числом 0, поэтому нам нужно убедиться, что соответствующие биты в первом регистре EXTI_CR сброшены. Цель следующего примера – продемонстрировать, как включить прерывание EXTI 0 и связать его с выводом A0, поэтому для доступа к первому регистру EXTI_CR для настройки группы GPIO A предоставляются следующие определения:

```
#define EXTI_CR_BASE (0x40013808)
#define EXTI_CR0 (*(volatile uint32_t *)(EXTI_CR_BASE + 0x00))
#define EXTI_CR_EXTI0_MASK (0x0F)
```

Прерывание EXTI0 подключено к линии NVIC номер 6, поэтому нужно добавить это определение для настройки NVIC:

```
#define NVIC_EXTI0_IRQN (6)
```

Контроллер EXTI в микроконтроллерах STM32F4 расположен по адресу 0x40013C00 и содержит следующие регистры:

- *регистр маски прерывания* (interrupt mask register, IMR) со смещением 0. Устанавливает/сбрасывает соответствующий бит, чтобы включить/выключить прерывание для каждой из линий EXTI;
- *регистр маски события* (event mask register, EMR) со смещением 4. Устанавливает/сбрасывает соответствующий бит, чтобы включить/выключить триггер события для соответствующей линии EXTI;
- *регистр выбора триггера восходящего фронта* (rising trigger select register, RTSR) со смещением 8. Устанавливает соответствующий бит для генерации событий и прерываний, когда уровень на соответствующем цифровом входе переключается с 0 на 1;
- *регистр выбора триггера спадающего фронта* (falling trigger select register, FTSR) со смещением 12. Устанавливает соответствующий бит для генерации событий и прерываний, когда уровень на соответствующем цифровом входе переключается с 1 обратно на 0;
- *регистр разрешения программных прерываний* (software interrupt enable register, SWIER) со смещением 16. При установке бита в этом регистре будет немедленно сгенерировано соответствующее событие прерывания и выполнена процедура обслуживания. Этот механизм можно использовать для реализации пользовательских программных прерываний;
- *регистр отложенных прерываний* (pending interrupt register, PR) со смещением 20. Чтобы очистить отложенное прерывание, процедура обслуживания должна установить бит, соответствующий линии EXTI, иначе прерывание останется отложенным. Новая служебная процедура будет запускаться до тех пор, пока бит PR для линии EXTI не будет сброшен.

Для удобства можно определить регистры следующим образом:

```
#define EXTI_BASE (0x40013C00)
#define EXTI_IMR (*(volatile uint32_t *)(EXTI_BASE + 0x00))
```

```
#define EXTI_EMR (*(volatile uint32_t *)(EXTI_BASE + 0x04))
#define EXTI_RTSR (*(volatile uint32_t *)(EXTI_BASE + 0x08))
#define EXTI_FTSR (*(volatile uint32_t *)(EXTI_BASE + 0x0c))
#define EXTI_SWIER (*(volatile uint32_t *)(EXTI_BASE + 0x10))
#define EXTI_PR (*(volatile uint32_t *)(EXTI_BASE + 0x14))
```

Процедура включения прерывания по переднему фронту PA0, связанному с нажатием кнопки, будет выглядеть так:

```
void button_setup(void)
{
  AHB1_CLOCK_ER |= GPIOA_AHB1_CLOCK_ER;
  GPIOA_MODE &= ~ (0x03 << (BUTTON_PIN * 2));
  EXTI_CR0 &= ~EXTI_CR_EXTI0_MASK;
  nvic_irq_enable(NVIC_EXTI0_IRQN);
  EXTI_IMR |= 1 << BUTTON_PIN;
  EXTI_EMR |= 1 << BUTTON_PIN;
  EXTI_RTSR |= 1 << BUTTON_PIN;
}
```

В этом коде установлены соответствующие биты ISR, IMR и RTSR, и в NVIC разрешено прерывание. Вместо того чтобы запрашивать изменение значения цифрового входа, теперь можно определить процедуру обслуживания прерывания, которая будет вызываться при каждом нажатии кнопки:

```
volatile uint32_t button_presses = 0;
void isr_exti0(void)
{
  EXTI_PR |= 1 << BUTTON_PIN;
  button_presses++;
}
```

В этом простом примере ожидается, что счетчик `button_presses` будет увеличиваться на единицу при каждом нажатии кнопки. В реальной жизни нажатия кнопок, основанных на механическом контакте (например, на STM32F407-Discovery), сложно обрабатывать с помощью этого механизма. Замыкание механических контактов не происходит мгновенно. Одно нажатие физической кнопки может вызвать прерывание по событию нарастающего фронта несколько раз на протяжении переходной фазы смыкания контактов. Это явление, известное как эффект *дребезга контактов*, можно смягчить с помощью специальных методов устранения дребезга, которые здесь не обсуждаются.

6.5.6. Аналоговый вход

Некоторые выводы имеют возможность динамически измерять приложенное напряжение и присваивать измеренному значению дискретное число, используя *аналого-цифровой преобразователь* сигнала (АЦП, ADC). Это очень полезно для получения данных от различных датчиков, способных пере-

давать информацию в виде выходного напряжения или просто с помощью переменного резистора.

Конфигурация подсистемы АЦП может существенно различаться на разных платформах. АЦП на современных микроконтроллерах предлагают широкий выбор вариантов конфигурации. Наш базовый микроконтроллер оснащен 3 отдельными контроллерами АЦП, которые совместно используют 16 входных каналов, каждый из которых имеет разрядность 12 бит. Доступно несколько функций, таких как передача полученных данных по прямому доступу к памяти и отслеживание сигналов между двумя пороговыми значениями сторожевого таймера.

Контроллеры АЦП обычно предназначены для автоматической выборки входных значений несколько раз в секунду и обеспечивают стабильные результаты, которые доступны немедленно. Случай, который рассматривается, проще и состоит из одноразовой операции чтения для одного преобразования.

Чтобы связать конкретный вывод с контроллером, нужно предварительно убедиться, что этот вывод действительно поддерживает функцию АЦП, и сконфигурировать его как аналоговый вход. Затем можно будет считать полученный в результате преобразования аналоговый сигнал. В следующем примере контакт B1 используется как аналоговый вход и может быть подключен к контроллеру `ADB1` через канал `9`. Для конфигурации контроллера ADB1 определим следующие константы и регистры:

```
#define APB2_CLOCK_ER (*(volatile uint32_t *)(0x40023844))
#define ADC1_APB2_CLOCK_ER_VAL (1 << 8)
#define ADC1_BASE (0x40012000)
#define ADC1_SR (*(volatile uint32_t *)(ADC1_BASE + 0x00))
#define ADC1_CR1 (*(volatile uint32_t *)(ADC1_BASE + 0x04))
#define ADC1_CR2 (*(volatile uint32_t *)(ADC1_BASE + 0x08))
#define ADC1_SMPR1 (*(volatile uint32_t *)(ADC1_BASE + 0x0c))
#define ADC1_SMPR2 (*(volatile uint32_t *)(ADC1_BASE + 0x10))
#define ADC1_SQR3 (*(volatile uint32_t *)(ADC1_BASE + 0x34))
#define ADC1_DR (*(volatile uint32_t *)(ADC1_BASE + 0x4c))
#define ADC_CR1_SCAN (1 << 8)
#define ADC_CR2_EN (1 << 0)
#define ADC_CR2_CONT (1 << 1)
#define ADC_CR2_SWSTART (1 << 30)
#define ADC_SR_EOC (1 << 1)
#define ADC_SMPR_SMP_480CYC (0x7)
```

Теперь дадим определения для обычной настройки GPIO, на этот раз сопоставленные для GPIOB:

```
#define AHB1_CLOCK_ER (*(volatile uint32_t *)(0x40023830))
#define GPIOB_AHB1_CLOCK_ER (1 << 1)
#define GPIOB_BASE (0x40020400)
#define GPIOB_MODE (*(volatile uint32_t *)(GPIOB_BASE + 0x00))
#define ADC_PIN (1)
#define ADC_PIN_CHANNEL (9)
```

Три АЦП совместно используют несколько регистров для общих настроек, таких как коэффициент предварительного делителя тактового сигнала, поэтому все они будут работать на одной частоте. Коэффициент деления для АЦП должен быть таким, чтобы тактовая частота находилась в пределах рабочего диапазона преобразователя, рекомендованного техническим описанием, – в нашей платформе общий предварительный делитель вдвое снижает частоту тактового сигнала APB2. Общие регистры конфигурации АЦП начинаются с порта `0x40012300`:

```
#define ADC_COM_BASE (0x40012300)
#define ADC_COM_CCR (*(volatile uint32_t *)(ADC_COM_BASE + 0x04))
```

На основе этих определений уже можно разработать функцию инициализации. Для этого нужно включить тактирование как для контроллера АЦП, так и для группы GPIO:

```
int adc_init(void)
{
  APB2_CLOCK_ER |= ADC1_APB2_CLOCK_ER_VAL;
  AHB1_CLOCK_ER |= GPIOB_AHB1_CLOCK_ER;
```

PB1 устанавливается в режим аналогового ввода, соответствующий значению `3` в регистре режима:

```
GPIOB_MODE |= 0x03 << (ADC_PIN * 2);
```

`ADC1` временно отключается для настройки желаемой конфигурации. В регистр предварительного делителя записано значение `0`, что означает деление на 2 входного тактового сигнала. Это гарантирует, что частота, подаваемая на контроллер АЦП, находится в пределах его рабочего диапазона. Режим сканирования отключен, как и непрерывный режим, поскольку в данном примере не используются эти функции:

```
ADC1_CR2 &= ~(ADC_CR2_EN);
ADC_COM_CCR &= ~(0x03 << 16);
ADC1_CR1 &= ~(ADC_CR1_SCAN);
ADC1_CR2 &= ~(ADC_CR2_CONT);
```

Частоту дискретизации можно установить с помощью двух регистров `SMPR1` и `SMPR2`, в зависимости от используемого канала. Каждый регистр представляет частоту дискретизации для одного канала, используя 3 бита на регистр, поэтому каналы с `0` по `9` настраиваются с помощью `SMPR1`, а все остальные – с помощью `SMPR2`. Каналу для PB1 присвоен номер `9`, поэтому используется регистр `SMPR1`, но стоит заметить, что существует общий механизм установки частоты дискретизации на любом канале:

```
if (ADC_PIN_CHANNEL > 9) {
  uint32_t val = ADC1_SMPR2;
  val = ADC_SMPR_SMP_480CYC << ((ADC_PIN_CHANNEL - 10) * 3);
  ADC1_SMPR2 = val;
} else {
```

```
  uint32_t val = ADC1_SMPR1;
  val = ADC_SMPR_SMP_480CYC << (ADC_PIN_CHANNEL * 3);
  ADC1_SMPR1 = val;
}
```

Наконец, канал активируется согласно последовательности преобразования контроллера АЦП с использованием *регистров последовательности* (sequence register, SQR). Механизмы преобразования предусматривают, что несколько каналов могут быть добавлены к одной и той же последовательности на контроллере путем заполнения регистров в обратном порядке, от SQR3 до SQR1. Каждый исходный канал представлен пятью битами, поэтому каждый регистр содержит до шести источников, за исключением SQR1, который хранит пять и резервирует старшие биты для указания длины стека, хранящегося в регистрах, минус один. В нашем случае нет необходимости настраивать поле «длина минус один», так как оно будет равно нулю для одного источника в SQR1:

```
ADC1_SQR3 |= (ADC_PIN_CHANNEL);
```

Снова включим аналоговый преобразователь ADC1 установкой бита разрешения в управляющем регистре CR2 и успешно возвращаемся из функции инициализации:

```
  ADC1_CR2 |= ADC_CR2_EN;
  return 0;
}
```

После того как АЦП был инициализирован и сконфигурирован для преобразования аналогового сигнала на PB1, аналого-цифровое преобразование может быть запущено в любое время. Простая функция блокирующего чтения инициирует преобразование, ждет успешного начала преобразования, а затем ждет завершения преобразования, просматривая бит окончания преобразования (end of conversion, EOC) в регистре состояния:

```
int adc_read(void)
{
  ADC1_CR2 |= ADC_CR2_SWSTART;
  while (ADC1_CR2 & ADC_CR2_SWSTART)
    ;
  while ((ADC1_SR & ADC_SR_EOC) == 0)
    ;
```

Когда преобразование завершено, соответствующее дискретное значение доступно в младших 12 битах регистра данных и может быть возвращено вызывающей стороне:

```
  return (int)(ADC1_DR);
}
```

И так, было показано, как общаться с внешним миром с помощью GPIO. Тот же самый интерфейс настройки и управления GPIO пригодится в следующей

главе для настройки более сложных интерфейсов локальной шины с использованием альтернативных функций для связанных линий GPIO.

В следующем разделе представлен сторожевой таймер, последний из общих системных компонентов, рассматриваемых в этой главе. Сторожевой таймер обычно имеется в различных микроконтроллерах и обеспечивает удобную процедуру аварийного восстановления всякий раз, когда по какой-либо причине система зависает и не возобновляет свою нормальную работу.

6.6. Сторожевой таймер

Общей чертой многих микроконтроллеров является наличие *сторожевого таймера* (watchdog timer). Сторожевой таймер гарантирует, что система не застрянет в бесконечном цикле или любой другой блокирующей ситуации в коде. Это особенно полезно в приложениях на «голом железе», которые полагаются на цикл, управляемый событиями, где требуется, чтобы вызовы не блокировались и возвращались в основной цикл событий в течение разрешенного периода времени.

Сторожевой таймер следует рассматривать как самое последнее средство для восстановления зависшей системы путем принудительной перезагрузки независимо от текущего состояния выполнения кода в ЦП.

Наша базовая платформа оснащена одним независимым сторожевым таймером с 12-разрядным счетчиком, подобным счетчику универсальных таймеров, и настраиваемым предварительным делителем тактовой частоты. Но значения предварительного делителя сторожевого таймера кратны 2 и имеют диапазон от 4 (значение 0) до 256 (значение 6).

Тактовые импульсы подаются от низкочастотного источника через независимую ветвь распределения тактовой частоты. По этой причине на сторожевой таймер не влияет настройка остальной системы маршрутизации тактовых импульсов.

Область конфигурации сторожевого таймера отображается в адресной области периферийных устройств и состоит из четырех регистров:

- ключевой регистр (смещение `0`), используемый для запуска трех операций разблокировки, запуска и сброса путем записи предопределенных значений в младшие 16 бит;
- регистр предварительного делителя (смещение `4`) для настройки коэффициента предварительного делителя счетчика;
- регистр перезагрузки (смещение `8`), содержащий значение перезагрузки счетчика;
- регистр состояния (смещение `12`), предоставляющий флаги состояния для синхронизации операций настройки.

На регистры можно ссылаться с помощью макросов быстрого доступа:

```
#define IWDG_BASE (0x40003000)
#define IWDG_KR (*(volatile uint32_t *)(IWDG_BASE + 0x00))
```

```
#define IWDG_PR (*(volatile uint32_t *)(IWDG_BASE + 0x04))
#define IWDG_RLR (*(volatile uint32_t *)(IWDG_BASE + 0x08))
#define IWDG_SR (*(volatile uint32_t *)(IWDG_BASE + 0x0c))
```

Через ключевой регистр можно запустить следующие операции:

```
#define IWDG_KR_RESET 0x0000AAAA
#define IWDG_KR_UNLOCK 0x00005555
#define IWDG_KR_START 0x0000CCCC
```

В статусе предусмотрены два значащих бита состояния, и их необходимо проверить, чтобы убедиться, что сторожевой таймер не занят, прежде чем разблокировать таймер и установить значение для предварительного масштабирования и перезагрузки:

```
#define IWDG_SR_RVU (1 << 1)
#define IWDG_SR_PVU (1 << 0)
```

Функция инициализации для настройки и запуска сторожевого таймера может выглядеть следующим образом:

```
int iwdt_init(uint32_t interval_ms)
{
  uint32_t pre = 0;
  uint32_t counter;
```

В следующей строке входное значение в миллисекундах задает рабочую частоту сторожевого таймера, которая составляет 32 кГц:

```
counter = interval_ms << 5;
```

Но минимальный коэффициент предварительного делителя равен 4, поэтому значение следует снова разделить. Затем ищется минимальное значение предварительного делителя, дающее нам значение счетчика, которое умещается в доступные 12 разрядов, уменьшая вдвое значение счетчика и увеличивая коэффициент предварительного делителя до тех пор, пока счетчик не будет соответствующим образом масштабирован:

```
  counter >>= 2;
  while (counter > 0xFFF) {
  pre++;
  counter >>= 1;
}
```

Следующие проверки гарантируют, что заданный интервал срабатывания сторожевого таймера не приведет к обнулению счетчика или значению, слишком большому для доступного коэффициента масштабирования:

```
if (counter == 0)
  counter = 1;
if (pre > 6)
  return -1;
```

На этом инициализация регистров завершена, но устройство требует от нас инициировать запись операцией разблокировки, и только после проверки доступности регистров для записи:

```
while(IWDG_SR & IWDG_SR_PR_BUSY);
IWDG_KR = IWDG_KR_UNLOCK;
IWDG_PR = pre;
while (IWDG_SR & IWDG_SR_RLR_BUSY);
IWDG_KR = IWDG_KR_UNLOCK;
IWDG_RLR = counter;
```

Запуск сторожевого таймера состоит из записи команды `START` в ключевой регистр, чтобы инициировать операцию запуска:

```
  IWDG_KR = IWDG_KR_START;
  return 0;
}
```

После запуска сторожевой таймер не может быть остановлен и будет работать вечно, уменьшая счетчик до тех пор, пока он не достигнет нуля, и перезагружая систему.

Единственный способ предотвратить перезагрузку системы – это сбросить счетчик сторожевого таймера вручную. Драйвер сторожевого таймера должен экспортировать функцию, позволяющую приложению сбрасывать счетчик, например в конце каждой итерации цикла `main()`. Вот пример:

```
void iwdt_reset(void)
{
  IWDG_KR = IWDG_KR_RESET;
}
```

В качестве простого теста для драйвера сторожевого таймера в `main()` можно инициализировать 2-секундный счетчик сторожевого таймера:

```
void main(void) {
  flash_set_waitstates();
  clock_config();
  button_setup();
  iwdt_init(2000);
  while(1)
  WFI();
}
```

Сторожевой таймер сбрасывается при нажатии кнопки в процедуре обслуживания прерывания кнопки GPIO:

```
void isr_exti0(void)
{
  EXTI_PR |= (1 << BUTTON_PIN);
  iwdt_reset();
}
```

В этом тестовом примере система перезагрузится, если пользователь не нажмет кнопку в течение 2 секунд подряд, поэтому единственный способ сохранить работоспособность системы – многократно нажимать кнопку.

6.7. Заключение

Подсистема тактирования, таймеры и линии ввода-вывода – это периферийные устройства общего назначения, показанные в этой главе и применяемые практически в каждом микроконтроллере. Хотя детали реализации, такие как имена и размещение регистров, различаются для разных целевых устройств, общие принципы работы совпадают для большинства встраиваемых платформ, а периферийные устройства общего назначения служат компонентами для создания самых основных системных функций, а также обеспечивают средства взаимодействия с датчиками и исполнительными устройствами.

В следующей главе рассказано про каналы последовательной связи, поддерживаемые большинством микропроцессоров в качестве интерфейсов связи с другими устройствами, находящимися поблизости от встраиваемой системы.

Глава 7

Интерфейсы локальной шины

Связь между встраиваемой системой и другими системами или устройствами, находящимися поблизости, обеспечивается несколькими протоколами. Большинство микроконтроллеров, разработанных для встраиваемых систем, поддерживают наиболее распространенные интерфейсы последовательного канала связи. Некоторые протоколы настолько популярны, что стали стандартом для проводной связи между микроконтроллерами и для обмена данными с внешними электронными устройствами, такими как датчики, приводы, дисплеи, беспроводные приемопередатчики и многие другие периферийные устройства. В этой главе описана работа последовательных протоколов, особое внимание уделяется реализации системного программного обеспечения на примерах, работающих на базовой платформе. В частности, в данной главе рассмотрены следующие темы:

- принцип работы последовательного канала;
- асинхронная последовательная шина на базе UART;
- шина SPI;
- шина I^2C.

К концу главы вы узнаете, как интегрировать распространенные протоколы последовательной связи во встраиваемую систему.

7.1. Технические требования

Вы можете скачать файлы кода для этой главы на GitHub по адресу https://github.com/PacktPublishing/Embedded-Systems-Architecture-Second-Edition/tree/main/Chapter7.

7.2. Принцип работы последовательного канала

Все протоколы, которые будут рассмотрены в этой главе, управляют доступом к последовательной шине, состоящей из одного или нескольких проводов и передающей информацию в виде электрических сигналов высокого и низкого уровня строго заданной продолжительности. Протоколы отличаются способом передачи и приема информации по линиям шины данных. Чтобы передать байт, приемопередатчик кодирует его как битовую последовательность, которая синхронизируется при помощи тактовых импульсов. Логические значения бита интерпретируются приемником, считывающим его значение на шине по определенному фронту тактового импульса.

Для каждого протокола определена полярность тактовых импульсов и порядок следования битов при передаче байта – от старшего к младшему или наоборот. Например, система, передающая символ ASCII *D* по последовательной линии, управляемой нарастающим фронтом тактовых импульсов, начиная со старшего бита, будет генерировать сигнал, показанный на рис. 7.1.

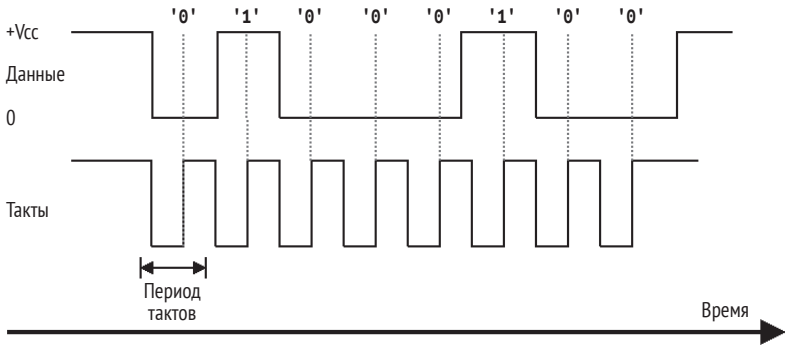

Рис. 7.1 ❖ Логические уровни шины с привязкой к нарастающему фронту тактовых импульсов и первым старшим битом приемник интерпретирует в значение байта `0x44`

Далее рассмотрим характеристики интерфейсов последовательной связи, реализованных в соответствии с различными стандартами. В частности, будут показаны способы синхронизации тактов между двумя конечными точками обмена данными; сигналы, которые определяются каждым протоколом для доступа к физической среде; и, наконец, детали реализации для программирования доступа к периферийным устройствам, которые могут различаться на разных платформах.

7.2.1. Синхронизация тактов и символов

Чтобы принимающая сторона могла понять сообщение, такты приемника и передатчика должны быть синхронизированы. Синхронизация может быть

неявной, например при установке одинаковой скорости передачи данных для чтения и записи на шине, или может быть достигнута в явном виде за счет совместного использования линии тактирования. Последовательные протоколы, которые не предусматривают использование общей линии тактовых импульсов, называются *асинхронными*.

Во всех остальных протоколах синхронизация символов должна быть явной. Поскольку, как правило, информацию отправляют и получают в виде байтов, начало каждой 8-битной последовательности следует помечать либо с помощью специальных последовательностей преамбулы на линии данных, либо путем включения и выключения тактового сигнала в нужный момент. Стратегия синхронизации символов определяется по-разному для каждого протокола.

7.2.2. Физические линии шины

Количество физических линий (проводов), необходимых для установления двунаправленной связи, также зависит от конкретного протокола. Поскольку по одному проводу можно передать только 1 бит информации в одном направлении за раз, для достижения полнодуплексной связи приемопередатчик должен быть подключен к двум разным проводам для передачи и приема данных. Если протокол поддерживает полудуплексную связь, он должен предоставлять надежный механизм для регулирования доступа к среде и переключения между приемом и передачей данных по одному и тому же проводу.

Важное примечание

Две конечные точки должны иметь общий источник опорного напряжения. Обычно это общий провод, т. е. «земля». Это означает, что кроме сигнальных линий может понадобиться еще один провод для соединения линий «земли» разных устройств, если они не имеют такого соединения.

Устройства, подключенные к шине, в зависимости от протокола могут либо использовать одинаковую реализацию и действовать как одноранговые узлы, либо исполнять разные роли при обмене данными, например одно из устройств может быть ведущим и генерировать сигнал выбора ведомого устройства, с которым будет установлена связь.

Последовательный протокол может предусматривать связь между более чем двумя устройствами на одной шине. Конфликтов на шине избегают за счет использования дополнительных линий выбора ведомых устройств, по одному на каждое ведомое устройство, использующее ту же шину, или путем назначения логических адресов каждой конечной точке и включения адреса получателя для связи в преамбулу каждой передачи. Основываясь на этой классификации, обзор подходов, реализованных в наиболее популярных последовательных протоколах для встраиваемых систем, можно свести в табл. 7.1.

Таблица 7.1. Краткий обзор подходов, реализованных в наиболее популярных последовательных протоколах для встраиваемых систем

Протокол	Кол-во линий	Тактирование	Синхронизация символов	Порядок битов	Режим связи
На основе UART	2 (RX/TX)	Асинхронное	Настраиваемые биты старт/стоп и четности	LSB-MSB	«Один к одному» или «один ко многим», но текущий обмен только с одной точкой
SPI	3 (данные ведущий–ведомый, данные ведомый–ведущий, такты)	Общее по линии CLK	По фронту тактового импульса	MSB-LSB или LSB-MSB	Один ведущий, несколько ведомых, которых выбирают по отдельными линиям
I^2C	2 (данные и такты)	Общее по линии SCL	По фронту тактового импульса с возможностью затягивания такта	MSB-LSB	«Один ко многим», ведущий или ведомый, с адресной логикой и динамическим выбором ведущего
USB	2 (полудуплексная дифференциальная пара D+/D–)	Синхронизация по особой последовательности в начале передачи	CRC/конец пакета	LSB-MSB	Один хост и несколько устройств
CAN	2 (полудуплексная дифференциальная пара CAN-Hi/CAN-Low)	Асинхронная	Стартовый бит, CRC и конец последовательности	MSB-LSB	«Один ко многим», допускается несколько ведущих с арбитражем между ними
1-Wire	1	Синхронизация по фронту сигнала	Выбор ведомого перед каждым байтом CRC	LSB-MSB	Ведущий–ведомый

В этой главе рассмотрены только первые три протокола, поскольку они наиболее широко используются для связи со встроенными периферийными устройствами.

7.2.3. Программирование периферийных устройств

Несколько периферийных модулей, реализующих описанные выше протоколы, обычно интегрированы в микроконтроллеры. Соответствующая последовательная шина подключается к определенным выводам микроконтроллеров напрямую или через блок выбора функции вывода. Периферийные модули могут быть активированы подачей тактовых импульсов и управляются через регистры конфигурации, отображенные в периферийной области в пространстве памяти. Выводы, соединенные с последовательными шинами, также должны быть настроены для реализации соответствующей функции, а задействованные линии прерывания должны быть настроены для обработки в таблице векторов.

Некоторые микроконтроллеры, включая нашу базовую платформу, поддерживают *прямой доступ к памяти* (direct memory access, DMA) для ускорения операций обмена данными между периферийным устройством и физической оперативной памятью. Во многих случаях эта функция полезна для ускорения обработки данных связи и повышения скорости отклика системы. Контроллер прямого доступа к памяти можно запрограммировать так, чтобы он инициировал операцию передачи и вызывал прерывание после ее завершения.

Интерфейс управления функциями, относящимися к каждому протоколу, специфичен для функций, предоставляемых периферийным устройством. В следующих разделах анализируются интерфейсы, предоставляемые периферийными устройствами UART, SPI и I^2C, и в качестве примеров одной из возможных реализаций аналогичных драйверов устройств приводятся образцы кода, адаптированные к эталонной платформе.

7.3. Асинхронная последовательная шина на основе UART

Исторически используемый для многих различных целей благодаря простоте своей асинхронной природы, UART восходит к истокам вычислительной техники и до сих пор является очень популярным стандартом, используемым во многих системах. Персональные компьютеры до начала 2000-х годов имели как минимум один последовательный порт RS-232, реализованный с помощью контроллера UART и приемопередатчиков, позволяющих работать при более высоких напряжениях. В настоящее время последователь-

ные порты UART практически полностью заменил USB, но хост-компьютеры по-прежнему могут получать доступ к последовательным шинам внешних устройств с помощью периферийных устройств USB-UART. Микроконтроллеры имеют одну или несколько пар контактов, которые можно связать с внутренним контроллером UART и подключить к последовательной шине для организации двунаправленного, асинхронного, полнодуплексного канала связи с устройством, подключенным к той же шине.

7.3.1. Описание протокола

Как упоминалось ранее, асинхронная последовательная связь основана на неявной синхронизации скорости передачи данных между передатчиком и приемником, чтобы гарантировать правильную обработку данных на принимающей стороне. При достаточно высоком быстродействии периферийного модуля скорость асинхронного последовательного канала может быть увеличена до нескольких мегабит в секунду.

Стратегия синхронизации символов основана на обнаружении начала передачи каждого отдельного байта. Когда ни одно устройство не передает, шина находится в состоянии ожидания.

Чтобы инициировать передачу, приемопередатчик переводит линию TX в низкий логический уровень на время длительностью не менее половины периода выборки битов в зависимости от скорости передачи. Биты, составляющие передаваемый байт, затем преобразуются в логические значения 0 или 1, которые удерживаются на линии TX в течение времени, соответствующего длительности бита при заданной скорости передачи. После того как приемник распознает начало передачи, далее следуют биты символа в определенном порядке, от младшего бита до самого старшего.

Количество битов, составляющих символ, также настраивается. Длина данных по умолчанию, равная 8 битам, позволяет преобразовать каждый символ в байт. В конце данных можно настроить необязательный бит четности для подсчета количества активных битов, что является очень упрощенной формой избыточной проверки. Бит четности, если он используется, можно настроить для указания того, является ли количество битов 1 в символе четным или нечетным. При возврате в состояние ожидания необходимо использовать 1 или 2 стоповых бита для обозначения конца символа.

Стоповый бит передается путем подтягивания сигнала к высокому уровню на протяжении всей передачи бита, отмечая конец текущего символа и заставляя приемник инициировать прием следующего. Одиночный стоповый бит является наиболее часто используемым значением по умолчанию; использование стоп-бита полуторной длительности или двух стоповых битов обеспечивает более длительный интервал между символами, который был полезен в прошлом для связи с более медленным оборудованием, но редко используется сегодня.

Обе конечные точки должны быть одинаково настроены до начала сеанса связи. Контроллеры последовательного канала обычно не поддерживают динамическое определение скорости передачи символов или каких-либо на-

строек от устройства, подключенного к другому концу линии, и по этой причине единственный способ успешно установить последовательную связь – использовать одинаковые заранее известные настройки. В основном это следующие параметры:

- скорость передачи данных в битах в секунду;
- количество битов данных в каждом символе (обычно 8);
- значение бита четности, если он присутствует (`O` – нечетный, `E` – четный, `N` отсутствует);
- количество стоповых битов.

Кроме того, отправитель должен быть настроен на отправку 1, 1.5 или 2 стоповых бит в конце каждой передачи. В прошлом более широко использовались стоповые биты длительностью 1.5 и 2 для синхронизации связи с древними электромеханическими устройствами. В настоящее время биты проверки четности и стоповые биты длиннее 1 не нужны для связи с использованием современных приемопередатчиков и используются редко.

Эти настройки часто записывают в сокращенном виде наподобие 115200-8-N-1 или 38400-8-O-2, что означает, соответственно, последовательный канал 115.2 Кбит/с с 8 битами данных на символ, без бита четности и 1 стоповым битом, и канал 38 400 Кбит/с с теми же битами данных, проверкой на нечетность и 2 стоповыми битами.

7.3.2. Программирование контроллера

Отладочные платы обычно имеют несколько UART, и наша плата с микроконтроллером STM32F407 не является исключением. Согласно техническому описанию, встроенный периферийный модуль `UART3` может быть связан с контактами PD8 (TX) и PD9 (RX), которые будут использоваться в этом примере. Код, необходимый для включения тактирования группы `D` GPIO и настройки выводов `8` и `9` в альтернативный режим с альтернативной функцией `7`, выглядит следующим образом:

```
#define AHB1_CLOCK_ER (*(volatile uint32_t *)(0x40023830))
#define GPIOD_AHB1_CLOCK_ER (1 << 3)
#define GPIOD_BASE 0x40020c00
#define GPIOD_MODE (*(volatile uint32_t *)(GPIOD_BASE + 0x00))
#define GPIOD_AFL (*(volatile uint32_t *)(GPIOD_BASE + 0x20))
#define GPIOD_AFH (*(volatile uint32_t *)(GPIOD_BASE + 0x24))
#define GPIO_MODE_AF (2)
#define UART3_PIN_AF (7)
#define UART3_RX_PIN (9)
#define UART3_TX_PIN (8)
static void uart3_pins_setup(void)
{
  uint32_t reg;
AHB1_CLOCK_ER |= GPIOD_AHB1_CLOCK_ER;

reg = GPIOD_MODE & ~ (0x03 << (UART3_RX_PIN * 2));
```

```
GPIOD_MODE = reg | (2 << (UART3_RX_PIN * 2));
reg = GPIOD_MODE & ~ (0x03 << (UART3_TX_PIN * 2));
GPIOD_MODE = reg | (2 << (UART3_TX_PIN * 2));

reg = GPIOD_AFH & ~(0xf << ((UART3_TX_PIN - 8) * 4));
GPIOD_AFH = reg | (UART3_PIN_AF << ((UART3_TX_PIN - 8) * 4));
reg = GPIOD_AFH & ~(0xf << ((UART3_RX_PIN - 8) * 4));
GPIOD_AFH = reg | (UART3_PIN_AF << ((UART3_RX_PIN - 8) * 4));
}
```

Устройство имеет собственный бит включения тактовых импульсов в регистре APB1_CLOCK_ER в позиции 18:

```
#define APB1_CLOCK_ER (*(volatile uint32_t *)(0x40023840))
#define UART3_APB1_CLOCK_ER_VAL (1 << 18)
```

Доступ к каждому контроллеру UART можно получить с помощью регистров, отображаемых в области периферийных устройств, с фиксированным смещением от базового адреса контроллера UART:

- регистр состояния UART (SR): регистр только для чтения, содержащий флаги состояния со смещением 0;
- регистр UART_Data (DR): регистр данных для чтения/записи со смещением 4;
- регистр скорости передачи данных UART (BRR): устанавливает делитель тактовой частоты для получения желаемой скорости передачи данных со смещением 8;
- регистры конфигурации UART (CRx): один или несколько регистров UART_CRx со смещением 12 для установки параметров последовательного порта, включения прерываний и прямого доступа к памяти, а также включения и выключения приемопередатчика.

В этом примере кода определены макросы быстрого доступа для доступа к следующим регистрам UART3:

```
#define UART3 (0x40004800)
#define UART3_SR (*(volatile uint32_t *)(UART3))
#define UART3_DR (*(volatile uint32_t *)(UART3 + 0x04))
#define UART3_BRR (*(volatile uint32_t *)(UART3 + 0x08))
#define UART3_CR1 (*(volatile uint32_t *)(UART3 + 0x0c))
#define UART3_CR2 (*(volatile uint32_t *)(UART3 + 0x10))
```

Определим позиции в соответствующих битовых полях:

```
#define UART_CR1_UART_ENABLE (1 << 13)
#define UART_CR1_SYMBOL_LEN (1 << 12)
#define UART_CR1_PARITY_ENABLED (1 << 10)
#define UART_CR1_PARITY_ODD (1 << 9)
#define UART_CR1_TX_ENABLE (1 << 3)
#define UART_CR1_RX_ENABLE (1 << 2)
#define UART_CR2_STOPBITS (3 << 12)
#define UART_SR_TX_EMPTY (1 << 7)
```

В начале функции инициализации может быть вызвана вспомогательная функция `uart3_pins_setup` для назначения вывода. Функция инициализации принимает аргументы для установки скорости передачи данных, бита четности и стоповых битов на порту `UART3`:

```
int uart3_setup(uint32_t bitrate, uint8_t data,
char parity, uint8_t stop)
{
  uart3_pins_setup();
```

Включение устройства:

```
APB1_CLOCK_ER |= UART3_APB1_CLOCK_ER_VAL;
```

В регистре конфигурации `CR1` установлен бит включения передатчика:

```
UART3_CR1 |= UART_CR1_TX_ENABLE;
```

`UART_BRR` содержит настройку делителя тактовой частоты для желаемой скорости передачи данных:

```
UART3_BRR = CLOCK_SPEED / bitrate;
```

Наша функция также принимает символ, указывающий нужную четность. Варианты O и E обозначают нечетность или четность соответственно. Любой другой символ оставит использование четности отключенным:

```
/* По умолчанию: без бита четности */
UART3_CR1 &= ~(UART_CR1_PARITY_ENABLED |
    UART_CR1_PARITY_ODD);
 switch (parity) {
     case 'O':
         UART3_CR1 |= UART_CR1_PARITY_ODD;
         /* включение четности */
     case 'E':
         UART3_CR1 |= UART_CR1_PARITY_ENABLED;
         break;
}
```

Количество стоповых битов устанавливается в соответствии с параметром. Конфигурация сохраняется с использованием 2 бит регистра, где значения `0` и `2` означают один и два стоповых бита соответственно:

```
reg = UART3_CR2 & ~UART_CR2_STOPBITS;
if (stop > 1)
  UART3_CR2 = reg | (2 << 12);
```

На этом настройка завершена. Теперь можно включить UART, чтобы инициировать передачу:

```
  UART3_CR1 |= UART_CR1_UART_ENABLE;
  return 0;
}
```

Последовательные данные теперь можно отправлять на вывод PD8, просто копируя по одному байту в регистр UART_DR.

7.3.3. Hello world!

Одной из наиболее полезных функций при разработке встраиваемой системы является использование одного из доступных портов UART для вывода отладочных сообщений и другой информации, полученной во время выполнения. Сообщения могут быть прочитаны на хост-компьютере с помощью преобразователя UART в USB (рис. 7.2).

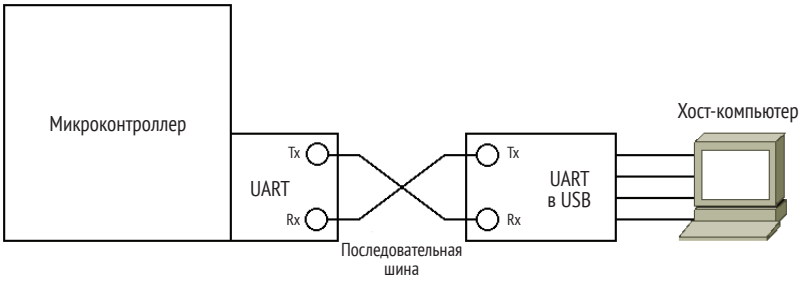

Рис. 7.2 ❖ Хост-компьютер подключен к последовательному порту целевой платформы с помощью преобразователя

Логика UART содержит буферы типа FIFO[1] на прием и передачу. Передающий FIFO пополняется путем записи в регистр UART_DR. При выводе данных на линию UART TX в режиме опроса перед записью очередного символа необходимо выполнить проверку, что FIFO пуст, поскольку в FIFO одновременно помещается не более одного символа. Когда буфер FIFO пуст, устройство устанавливает в 1 бит, связанный с флагом TX_FIFO_EMPTY в UART3_SR. Следующая функция показывает, как передать всю строку символов, полученную в качестве аргумента, ожидая, пока FIFO не опустеет после каждого байта:

```
void uart3_write(const char *text)
{
  const char *p = text;
  int i;
  volatile uint32_t reg;
  while(*p) {
    do {
      reg = UART3_SR;
    } while ((reg & UART_SR_TX_EMPTY) == 0);
    UART3_DR = *p;
    p++;
  }
}
```

[1] First in – first out, первым вошел – первым вышел.

В главном цикле `main()` эту функцию можно вызвать с предварительно отформатированной строкой, оканчивающейся `NULL`:

```
#include "system.h"
#include "uart.h"
void main(void) {
  flash_set_waitstates();
  clock_config();
  uart3_setup(115200, 8, 'N', 1);
  uart3_write("Hello World!\r\n");
  while(1)
    WFI();
}
```

Если к последовательной шине подключен хост-компьютер, то можно вывести на его дисплей текст Hello World! с помощью любой программы терминала последовательного порта, например minicom.

Записав при помощи логического анализатора сигнал на выводе PD8, используемом в качестве `UART_TX` встраиваемого устройства, и настроив правильные параметры декодирования, можно получить лучшее представление о том, как последовательный поток анализируется на принимающей стороне. Логический анализатор может показать, как происходит выборка битов данных после каждого стартового импульса, и выделить символ ASCII, связанный с байтом. Инструменты логического анализатора обычно способны декодировать биты, захваченные на линии, и отображать каждый байт в формате ASCII. Это быстрый и точный способ убедиться, что настройки последовательного канала совпадают, интервал между последовательными битами соответствует выбранной скорости передачи данных, а сигнал на линии соответствует данным, как показано на рис. 7.3. Здесь показано, как устройство отправляет строку `"Hello"` с помощью функции `uart3_write`.

Рис. 7.3 ❖ Снимок экрана логического анализатора, показывающий первые 5 байт, отправленных на хост с использованием UART3

7.3.4. Функция printf библиотеки newlib

Запись предварительно отформатированных строк в регистр – не самый идеальный API для доступа к последовательному порту и передачи отладочных сообщений. Разработчики приложений, безусловно, предпочли бы использовать стандартную функцию C `printf`. Когда набор инструментов включает в себя реализацию стандартной библиотеки C, это обычно дает нам возможность подключить стандартный вывод основной программы к последовательному интерфейсу. К счастью, инструментарий, который используется для

работы с базовой платформой, позволяет ссылаться на функции библиотеки newlib. Подобно тому, как это было сделано в главе 5 для функций malloc и free из newlib, предоставляется внутренняя функция с именем _write(), которая перенаправляет вывод из строки, отформатированной всеми вызовами printf(). Реализованная ниже функция _write будет получать все строки, предварительно отформатированные функцией printf():

```
int _write(void *r, uint8_t *text, int len)
{
  char *p = (char *)text;
  int i;
  volatile uint32_t reg;
  text[len - 1] = 0;
  while(*p) {
    do {
       reg = UART3_SR;
    } while ((reg & UART_SR_TX_EMPTY) == 0);
    UART3_DR = *p;
    p++;
  }
  return len;
}
```

В этом случае связывание с newlib позволяет нам использовать printf для создания сообщений, включая парсинг аргументов, как показано в следующем примере функции main():

```
#include <stdio.h>
#include "system.h"
#include "uart.h"

void main(void) {
  char name[] = "World";
  flash_set_waitstates();
  clock_config();
  uart3_setup(115200, 8, 'N', 1);

  printf("Hello %s!\r\n", name);
  while(1)
    WFI();
```

Этот второй пример выдаст тот же результат, что и первый, но на этот раз с использованием функции printf из newlib.

7.3.5. Получение данных

Чтобы включить приемник на том же UART, функция инициализации также должна включить приемник с помощью соответствующего переключателя в регистре UART_CR1:

```
UART3_CR1 |= UART_CR1_TX_ENABLE | UART_CR1_RX_ENABLE;
```

После этой инструкции приемная часть приемопередатчика также будет включена. Для чтения данных в режиме опроса с блокировкой до получения символа можно использовать следующую функцию, которая вернет значение прочитанного байта:

```
char uart3_read(void)
{
  char c;
  volatile uint32_t reg;
  do {
    reg = UART3_SR;
  } while ((reg & UART_SR_RX_NOTEMPTY) == 0);
  c = (char)(UART3_DR & 0xff);
  return c;
}
```

С помощью этой функции можно, например, вывести на консоль каждый символ, полученный от хоста:

```
void main(void) {
  char c[2];
  flash_set_waitstates();
  clock_config();
  uart3_setup(115200, 8, 'N', 1);
  uart3_write("Hello World!\r\n");
  while(1) {
    c[0] = uart3_read();
    c[1] = 0;
    uart3_write(c);
    uart3_write("\r\n");
  }
}
```

7.3.6. Ввод/вывод с использованием прерываний

Примеры в этом разделе основаны на опросе состояния UART путем постоянной проверки флагов `UART_SR`. Операция записи в буфер сопровождается состоянием занятости, которое может длиться несколько миллисекунд, в зависимости от длины строки. Хуже того, показанная выше функция чтения вращается в замкнутом цикле до тех пор, пока из внешнего мира не поступят данные для чтения, а это означает, что вся система зависает до тех пор, пока не будут получены новые данные. В однопоточной встроенной системе возврат к основному циклу с наименьшей возможной задержкой важен для поддержания стабильно быстрого отклика системы.

Правильный способ обслуживания связи UART без блокировки – использовать линию прерывания, связанную с модулем UART, для запуска действий на основе полученного события. UART можно настроить для подачи сигнала прерывания при возникновении нескольких типов событий. Как было пока-

зано в предыдущих примерах, для управления операциями ввода и вывода нас интересуют, в частности, два конкретных события:

- пустое событие TX FIFO, позволяющее передать следующие данные;
- непустое событие RX FIFO, сигнализирующее о наличии вновь полученных данных.

Прерывание по этим двум событиям можно разрешить, установив соответствующие биты в UART_CR1. Определим две вспомогательные функции для независимого включения и выключения прерываний:

```
#define UART_CR1_TXEIE (1 << 7)
#define UART_CR1_RXNEIE (1 << 5)

static void uart3_tx_interrupt_onoff(int enable)
{
  if (enable)
    UART3_CR1 |= UART_CR1_TXEIE;
  else
    UART3_CR1 &= ~UART_CR1_TXEIE;
}

static void uart3_rx_interrupt_onoff(int enable)
{
  if (enable)
    UART3_CR1 |= UART_CR1_RXNEIE;
  else
    UART3_CR1 &= ~UART_CR1_RXNEIE;
}
```

С событиями прерывания можно связать служебную процедуру, а затем проверить флаги в UART_SR для определения причины прерывания:

```
void isr_uart3(void)
{
  volatile uint32_t reg;
  reg = UART3_SR;
  if (reg & UART_SR_RX_NOTEMPTY) {
      /* Получение нового байта */
  }

  if ((reg & UART_SR_TX_EMPTY)
  {
    /* окончание ожидания передачи */
  }
}
```

Реализация процедуры прерывания зависит от конкретной системы. Операционная система реального времени может решить мультиплексировать доступ к последовательному порту для нескольких потоков и будить потоки, ожидающие доступа к ресурсу. В однопоточном приложении возможно добавление промежуточных системных буферов для обеспечения неблокирующих вызовов, возвращающихся сразу после копирования данных из буфе-

ра приемника или в буфер передатчика. Использование соответствующих структур, таких как циклические буферы, для реализации системных очередей ввода и вывода, обеспечивает оптимальное использование выделенной памяти.

7.4. Шина SPI

Шина SPI (serial peripherial interface, последовательный периферийный интерфейс) работает по принципу ведущий–ведомый. Как следует из названия, интерфейс изначально был разработан для управления периферийными устройствами. Каждый сеанс связи инициируется ведущим устройством на шине, а периферийные устройства обычно играют роль ведомых. Благодаря полнодуплексной конфигурации канала и синхронному тактированию SPI может работать намного быстрее, чем асинхронная связь, из-за высокой устойчивости к расхождению тактовых частот между системами, использующими одну шину. SPI широко используется в качестве коммуникационного протокола для различных устройств из-за его простой логики и гибкости, обусловленной тем фактом, что ведомое устройство не нужно предварительно настраивать для связи с заранее заданной скоростью, которая соответствует скорости ведущего устройства. Несколько периферийных устройств могут использовать одну и ту же шину, если определены стратегии доступа к среде. Как правило, ведущее устройство использует отдельные линии GPIO для выбора ведомого.

7.4.1. Описание протокола

Конфигурация приемопередатчика SPI очень гибкая. Обычно приемопередатчик микроконтроллера может реализовать обе роли – ведущего и ведомого. Несколько параметров шины должны быть известны заранее и совместно использоваться ведущим устройством и всеми ведомыми устройствами на одной шине:

- полярность тактов: какой фронт тактового импульса является активным – восходящий или нисходящий;
- фаза тактового импульса: какой уровень на линии тактов соответствует состоянию ожидания – высокий или низкий;
- длина пакета данных: любое значение от 4 до 16 бит;
- порядок битов: как начинается передача – со старшего или младшего бита.

Поскольку тактовые импульсы на шину отправляет ведущее устройство, SPI не имеет какой-то определенной скорости работы, хотя использование слишком высокой скорости подходит не для всех периферийных устройств и микроконтроллеров.

Связь SPI с ведомым устройством отключена до тех пор, пока ведущее устройство не инициирует транзакцию. В начале каждой транзакции веду-

щий выбирает ведомого, активируя соответствующую линию выбора ведомого (рис. 7.4).

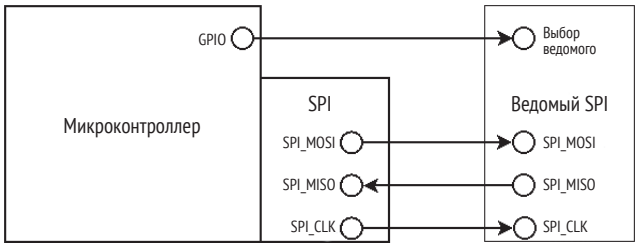

Рис. 7.4 ❖ Для выбора определенного ведомого устройства на шине можно использовать специальную линию, подключенную к GPIO

Чтобы начать сеанс связи, ведущий должен подать тактовые импульсы на линию CLK и отправить ведомому последовательность команд по линии MOSI. После появления тактовых импульсов ведомое устройство может немедленно начать передачу байтов в обратном направлении, используя линию MISO.

Даже если ведущий завершил передачу, он должен полностью соблюдать протокол, реализованный ведомым устройством, и позволить ему закончить ответ, сохраняя тактовые импульсы на время транзакции. Ведомому отводится заранее назначенное количество байтовых слотов для связи с ведущим.

Если нужно сохранить тактовые импульсы на шине, даже когда нет данных для передачи ведомому, ведущий может продолжать отправлять через линию MOSI фиктивные байты, которые игнорирует ведомый. Тем временем ведомому устройству разрешено отправлять данные по линии MISO, если ведущее устройство продолжает поддерживать тактирование шины. В отличие от UART, в модели связи ведущий–ведомый, реализованной в SPI, ведомые никогда не могут спонтанно инициировать сеанс связи, поскольку ведущий является единственным устройством на шине, которому разрешено передавать тактовые импульсы. Каждая транзакция SPI является автономной, и в конце сеанса выбор ведомого устройства отменяется путем отключения соответствующего сигнала выбора ведомого.

7.4.2. Программирование приемопередатчика

На отладочной плате базовой платформы к шине `SPI1` подключен акселерометр, работающий как ведомое устройство, поэтому можно изучить реализацию ведущего акселерометра на микроконтроллере, настроив приемопередатчик и выполнив двунаправленную транзакцию к периферийному устройству.

Регистры конфигурации шины `SPI1` отображаются в области периферийных устройств:

```
#define SPI1 (0x40013000)

#define SPI1_CR1 (*(volatile uint32_t *)(SPI1))
#define SPI1_CR2 (*(volatile uint32_t *)(SPI1 + 0x04))
#define SPI1_SR (*(volatile uint32_t *)(SPI1 + 0x08))
#define SPI1_DR (*(volatile uint32_t *)(SPI1 + 0x0c))
```

Периферийное устройство содержит в общей сложности четыре регистра:

- два регистра битовых полей конфигурации;
- один регистр состояния;
- один двунаправленный регистр данных.

Очевидно, что интерфейс приемопередатчика SPI аналогичен интерфейсу UART, так как настройка параметров связи происходит через регистры `SPI_CRx`, состояние FIFO можно контролировать, просматривая `SPI_SR`, а `SPI_DR` можно использовать для чтения и записи данных в последовательную шину.

Регистр конфигурации `CR1` содержит следующую информацию:

- фаза тактовых импульсов, 0 или 1 (бит 0);
- полярность тактовых импульсов (бит 1);
- флаг режима ведущего SPI (бит 2);
- коэффициент делителя скорости передачи данных (биты 3–5);
- флаг включения SPI (бит 6);
- другие параметры конфигурации, такие как длина слова, первый бит (LSB/MSB) и остальные флаги, которые не будут использоваться в этом примере, поскольку для данных параметров надо оставить значения по умолчанию.

Регистр конфигурации `CR2` содержит флаги для включения событий прерывания и передачи DMA (прямой доступ к памяти), а также флаг *включения вывода выбора ведомого устройства* (slave select output enable, SSOE), который понадобится нам в следующем примере.

Регистр состояния `SPI1_SR` подобен регистру состояния UART в предыдущем разделе, так как он тоже содержит флаги для определения того, пуст ли FIFO передачи, и когда FIFO на принимающей стороне не пуст, для регулирования фаз передачи.

Определим биты, которые соответствуют флагам, задействованным в следующем примере:

```
#define SPI_CR1_MASTER (1 << 2)
#define SPI_CR1_SPI_EN (1 << 6)
#define SPI_CR2_SSOE (1 << 2)
#define SPI_SR_RX_NOTEMPTY (1 << 0)
#define SPI_SR_TX_EMPTY (1 << 1)
```

RCC управляет тактами и линиями сброса приемопередатчика `SPI1`, подключенного к шине `APB2`:

```
#define APB2_CLOCK_ER (*(volatile uint32_t *)(0x40023844))
#define APB2_CLOCK_RST (*(volatile uint32_t *)(0x40023824))
#define SPI1_APB2_CLOCK_ER_VAL (1 << 12)
```

Приемопередатчик можно сбросить, отправив импульс сброса с RCC:

```
static void spi1_reset(void)
{
  APB2_CLOCK_RST |= SPI1_APB2_CLOCK_ER_VAL;
  APB2_CLOCK_RST &= ~SPI1_APB2_CLOCK_ER_VAL;
}
```

Выводы микроконтроллера PA5, PA6 и PA7 можно связать с приемопередатчиком SPI1, назначив им соответствующую альтернативную функцию:

```
#define SPI1_PIN_AF 5
#define SPI1_CLOCK_PIN 5
#define SPI1_MOSI_PIN 6
#define SPI1_MISO_PIN 7

static void spi1_pins_setup(void)
{
  uint32_t reg;
  AHB1_CLOCK_ER |= GPIOA_AHB1_CLOCK_ER;
  reg = GPIOA_MODE & ~(0x03 << (SPI1_CLOCK_PIN * 2));
  reg &= ~(0x03 << (SPI1_MOSI_PIN));
  reg &= ~(0x03 << (SPI1_MISO_PIN));
  reg |= (2 << (SPI1_CLOCK_PIN * 2));
  reg |= (2 << (SPI1_MOSI_PIN * 2)) | (2 << (SPI1_MISO_PIN *2))
  GPIOA_MODE = reg;
  reg = GPIOA_AFL & ~(0xf << ((SPI1_CLOCK_PIN) * 4));
  reg &= ~(0xf << ((SPI1_MOSI_PIN) * 4));
  reg &= ~(0xf << ((SPI1_MISO_PIN) * 4));
  reg |= SPI1_PIN_AF << ((SPI1_CLOCK_PIN) * 4);
  reg |= SPI1_PIN_AF << ((SPI1_MOSI_PIN) * 4);
  reg |= SPI1_PIN_AF << ((SPI1_MISO_PIN) * 4);
  GPIOA_AFL = reg;
}
```

Дополнительный вывод микроконтроллера, подключенный к линии выбора микросхемы акселерометра, – это PE3, который настроен как выход с подтягивающим внутренним резистором. Логика этого вывода имеет активный низкий уровень, поэтому логический ноль включит микросхему акселерометра:

```
#define SLAVE_PIN 3
static void slave_pin_setup(void)
{
  uint32_t reg;
  AHB1_CLOCK_ER |= GPIOE_AHB1_CLOCK_ER;
  reg = GPIOE_MODE & ~(0x03 << (SLAVE_PIN * 2));
  GPIOE_MODE = reg | (1 << (SLAVE_PIN * 2));
  reg = GPIOE_PUPD & ~(0x03 << (SLAVE_PIN * 2));
  GPIOE_PUPD = reg | (0x01 << (SLAVE_PIN * 2));
  reg = GPIOE_OSPD & ~(0x03 << (SLAVE_PIN * 2));
  GPIOE_OSPD = reg | (0x03 << (SLAVE_PIN * 2));
}
```

Инициализация приемопередатчика начинается с настройки четырех задействованных выводов. Затем включается тактирование модуля SPI, и приемопередатчик получает импульс сброса через RCC:

```
void spi1_setup(int polarity, int phase)
{
  spi1_pins_setup();
  slave_pin_setup();
  APB2_CLOCK_ER |= SPI1_APB2_CLOCK_ER_VAL;
  spi1_reset();
```

Параметры по умолчанию (начало передачи с MSB, 8-битная длина слова) остаются без изменений. Коэффициент масштабирования скорости передачи данных этого контроллера выражается в степени 2, начиная с 2, что соответствует значению битового поля 0, и удваивается при каждом приращении. Универсальный драйвер должен вычислить правильный коэффициент масштабирования в соответствии с желаемой тактовой частотой. В этом простом случае применяется жестко заданный масштабный коэффициент 64, соответствующий значению 5 в битовом поле.

Запишем нужные значения в SPI1_CR1:

```
SPI1_CR1 = SPI_CR1_MASTER | (5 << 3) | (polarity << 1) |
(phase << 0);
```

Установим бит, соответствующий флагу SSOE в SPI1_CR2, и включаем приемопередатчик:

```
  SPI1_CR2 |= SPI_CR2_SSOE;
  SPI1_CR1 |= SPI_CR1_SPI_EN;
}
```

Теперь можно начинать операции чтения и записи, поскольку ведущий и ведомый SPI-контроллеры готовы к выполнению транзакций.

7.4.3. Транзакции по шине SPI

Функции чтения и записи представляют две разные фазы транзакции SPI. Большинство ведомых устройств SPI способны обмениваться данными с использованием полнодуплексного механизма, так что передача байтов происходит в обоих направлениях, пока от ведущего устройства поступают тактовые импульсы. В течение каждого интервала обмена данные передаются в обоих направлениях независимо по линиям MISO и MOSI.

Во многих ведомых устройствах реализована распространенная стратегия, согласно которой ведущее устройство получает доступ к чтению и записи регистров ведомого устройства по шине SPI при помощи хорошо задокументированных команд.

Плата STM32F407DISCOVERY имеет акселерометр, подключенный к шине SPI1, который реагирует на специальные команды, обращаясь к определенным регистрам в памяти устройства для чтения или записи. В этих случаях

операции чтения и записи выполняются последовательно: в течение первого интервала ведущий передает дескриптор команды, затем в последующие интервалы передаются собственно байты в том или ином направлении в соответствии с командой.

Описанный ниже пример операции состоит из чтения регистра WHOAMI в акселерометре с использованием дескриптора команды 0x8F. Периферийное устройство должно ответить одним байтом, содержащим значение 0x3B, которое идентифицирует устройство и показывает, что связь SPI работает правильно. Но во время передачи командного байта ведомому устройству еще нечего передавать, поэтому результат первой операции чтения можно отбросить. Точно так же после отправки команды ведущему больше нечего сообщить ведомому, поэтому он выводит значение 0xFF на линию MOSI, одновременно считывая байт, переданный ведомым устройством по линии MISO.

Для успешного чтения одного байта на этом конкретном устройстве необходимо выполнить следующие шаги:

1) включить ведомое устройство подачей низкого уровня на линию выбора ведомого устройства;
2) отправить ведомому байт, содержащий код операции чтения одного байта;
3) отправить один фиктивный байт, в то время как ведомое устройство передает ответ, используя тактовые импульсы на шине;
4) считать значение, переданное ведомым устройством в течение второго интервала;
5) выключить ведомое устройство подачей высокого уровня на линию выбора ведомого устройства.

Для этого определим блокирующие функции чтения и записи следующим образом:

```
uint8_t spi1_read(void)
{
  volatile uint32_t reg;
  do {
    reg = SPI1_SR;
  } while ((reg & SPI_SR_RX_NOTEMPTY) == 0);
  return (uint8_t)SPI1_DR;
}

void spi1_write(const char byte)
{
  int i;
  volatile uint32_t reg;
  SPI1_DR = byte;
  do {
    reg = SPI1_SR;
  } while ((reg & SPI_SR_TX_EMPTY) == 0);
}
```

Операция чтения ожидает, пока в SPI1_SR не будет включен флаг RX_NOTEMPTY, прежде чем передавать содержимое регистра данных. Функция передачи,

наоборот, сначала отправляет значение байта для передачи в регистр данных, а затем опрашивает конец операции, ожидая появления флага `TX_EMPTY`.

Теперь эти две операции можно объединить. Ведущий должен в явном виде отправить всего 2 байта данных, поэтому наше основное приложение может запросить содержимое регистра идентификации акселерометра, выполнив следующие действия:

```
slave_on();
spi1_write(0x8F);
b = spi1_read();
spi1_write(0xFF);
b = spi1_read();
slave_off();
```

Вот что происходит на шине:

- во время первой записи по линии MOSI отправляется команда `0x8F`;
- значение, прочитанное с помощью первого вызова функции `spi1_read`, представляет собой фиктивный байт, который ведомое устройство поместило в MISO при прослушивании входящей команды. Полученное значение в данном конкретном случае не имеет смысла – поэтому оно отбрасывается;
- вторая запись помещает фиктивный байт в линию MOSI, так как ведущему больше нечего передавать. Но это заставляет его генерировать такты для второго байта, который необходим ведомому устройству для ответа на команду;
- второй вызов функции чтения обрабатывает ответ, переданный по линии MISO во время записи фиктивного байта от ведущего. Значение, полученное в этой второй транзакции, является действительным ответом от ведомого устройства, согласно описанию команды в документации.

Анализируя последовательную транзакцию с помощью логического анализатора, можно четко различить две фазы и чередующееся релевантное содержимое – сначала на MOSI для передачи команды, а затем на MISO для получения ответа (рис. 7.5).

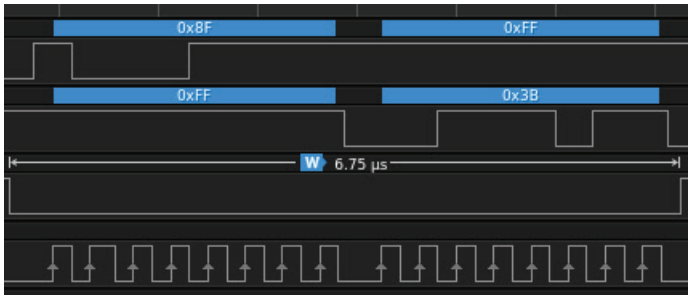

Рис. 7.5 ❖ Двунаправленная транзакция SPI, содержащая запрос от ведущего и ответ от ведомого (сверху вниз: `SPI1_MISO`, `SPI1_MOSI`, `SLAVE_SELECT` и `SPI1_CLOCK`)

Использование блокирующих операций с циклом занятости – очень плохая практика. Причина, по которой они здесь используются, состоит в том, чтобы продемонстрировать примитивные операции, необходимые для успешного выполнения двунаправленных транзакций SPI. В реальной встроенной системе всегда рекомендуется использовать передачу на основе прерывания, чтобы гарантировать, что ЦП не будет зациклен в ожидании завершения передачи. Контроллеры SPI выдают сигналы прерывания по изменению состояния буферов FIFO контроллера, чтобы синхронизировать транзакцию SPI с действиями, необходимыми при передаче данных в любом направлении.

7.4.4. Передача данных по шине SPI на основе прерываний

Интерфейс для управления прерываниями приемопередатчика SPI на самом деле очень похож на интерфейс UART, показанный раньше. Для правильной реализации неблокирующих транзакций их необходимо разделить на фазы чтения и записи, чтобы события смогли запускать связанные действия.

Установка следующих двух битов в регистре `SPI1_CR2` активирует триггер прерывания при пустом FIFO для передачи и непустом FIFO для приема соответственно:

```
#define SPI_CR2_TXEIE (1 << 7)
#define SPI_CR2_RXNEIE (1 << 6)
```

Соответствующая программа обработки прерывания по-прежнему может просматривать значения в `SPI1_SR`, чтобы перевести транзакцию в следующую фазу:

```
void isr_spi1(void)
{
  volatile uint32_t reg;
  reg = SPI1_SR;
  if (reg & SPI_SR_RX_NOTEMPTY) {
    /* Конец передачи: новые данные доступны на MISO*/
  }
  if ((reg & SPI_SR_TX_EMPTY)
  {
    /* Конец передачи: буфер TX FIFO пуст*/
  }
}
```

Реализацию верхней половины прерывания оставлена на усмотрение читателей, поскольку она зависит от API, который система должна реализовать, характера транзакций и их влияния на скорость отклика системы. Надо заметить, что редкие высокоскоростные SPI-транзакции могут быть короткими и разбросанными по времени, поэтому даже реализация блокирующих операций оказывает меньшее влияние на задержку системы.

7.5. Шина I²C

Третий протокол последовательной связи, представленный в этой главе, – I²C. С точки зрения коммуникационной стратегии этот протокол имеет некоторое сходство с SPI. Но скорость передачи данных по умолчанию для шины I²C намного ниже, поскольку протокол отдает приоритет низкому энергопотреблению по сравнению с пропускной способностью.

Одна и та же двухпроводная шина может связывать несколько участников, как ведущих, так и ведомых, и нет необходимости в дополнительных сигналах для физического выбора ведомых устройств, участвующих в транзакции, так как им назначены фиксированные логические адреса (рис. 7.6).

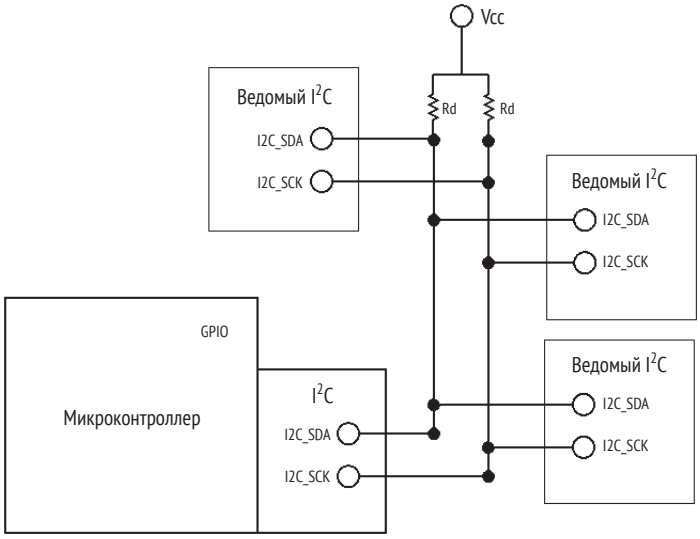

Рис. 7.6 ❖ Шина I²C с тремя ведомыми устройствами и внешними подтягивающими резисторами

Один провод передает тактовые импульсы, генерируемые ведущим устройством, а другой используется в качестве двунаправленного канала синхронных данных. Это возможно благодаря уникальному механизму арбитража канала, который основан на конструкции приемопередатчиков и может очень четко обрабатывать присутствие нескольких ведущих на одной шине.

Два сигнала должны быть подключены к напряжению высокого уровня шины (обычно 3.3 В) с помощью подтягивающих резисторов. При передаче контроллеры никогда не повышают уровень сигнала принудительно, а вместо этого позволяют ему свободно подниматься до значения по умолчанию, установленного подтягивающими резисторами. Как следствие логический нулевой уровень всегда является доминирующим; если какое-либо из устройств, подключенных к шине, устанавливает ноль, притягивая линию вниз, все устройства будут считывать уровень на линии низким, независимо

от того, сколько других отправителей хотели бы выставить уровень логической 1 на шине. Это позволяет одновременно использовать шину нескольким приемопередатчикам, а операции передачи можно координировать, инициируя новые транзакции только тогда, когда шина становится доступной. В этом разделе будет дано введение в протокол, чтобы познакомиться с программными инструментами для управления периферийными устройствами контроллера I^2C. Дополнительную информацию о связи по шине I^2C и соответствующую документацию можно найти по адресу https://www.i2c-bus.org/.

7.5.1. Описание протокола

Синхронизация между ведущим и ведомым достигается с помощью распознаваемых состояний START и STOP, которые определяют начало и конец транзакции соответственно. Изначально шина находится в состоянии ожидания, и оба сигнала находятся на высоком логическом уровне, когда все участники простаивают.

Состояние `START` является единственным случаем, когда ведущий устанавливает низкий уровень на линии SDA до SCL. Это специальное состояние сообщает ведомым устройствам и другим ведущим устройствам на шине, что транзакция инициирована (и шина занята). Состояние `STOP` определяется переходом SDA от низкого уровня к высокому, в то время как SCL остается высоким. После состояния `STOP` шина снова простаивает, и следующий сеанс связи возможен только в том случае, если возникает новое состояние `START`.

Итак, ведущий реализует состояние `START`, переводя сначала SDA, а затем SCL в низкий уровень. Кадр состоит из девяти тактов. После восходящего фронта каждого тактового импульса уровень SDA не меняется до тех пор, пока тактовый импульс снова не станет низким. Это позволяет нам передавать 1 кадр из 8 байт в первые 8 нарастающих фронтов тактовых импульсов. Во время последнего тактового импульса ведущий не управляет линией SDA, которая удерживается в высоком уровне подтягивающим резистором. Любой приемник, который хочет подтвердить прием кадра, может задать низкий уровень сигнала. Это состояние девятого тактового импульса называется ACK (от acknowledgement – подтверждение приема). Если ни одно принимающее устройство не подтверждает кадр, SDA остается высоким, и отправитель понимает, что кадр не достиг предполагаемого получателя (рис. 7.7).

Транзакция состоит из двух или более кадров и всегда инициируется ведущим устройством. Первый кадр каждой транзакции называется *адресным кадром* и содержит адрес и режим следующей операции. Все последующие кадры в транзакции являются кадрами данных, содержащими по 1 байту каждый. Ведущий определяет, сколько кадров составляет транзакция и направление передачи данных, сохраняя транзакцию активной в течение желаемого количества кадров, прежде чем применить состояние `STOP`.

Ведомые устройства имеют фиксированные 7-битные адреса, по которым с ними можно связаться по шине. Ведомое устройство, обнаружившее состояние `START` на шине, должно прослушивать кадр адреса и сравнивать его со своим адресом. Если адрес совпадает, кадр адреса должен быть под-

твержден путем перевода линии SDA в низкий уровень во время девятого тактового импульса при передаче кадра.

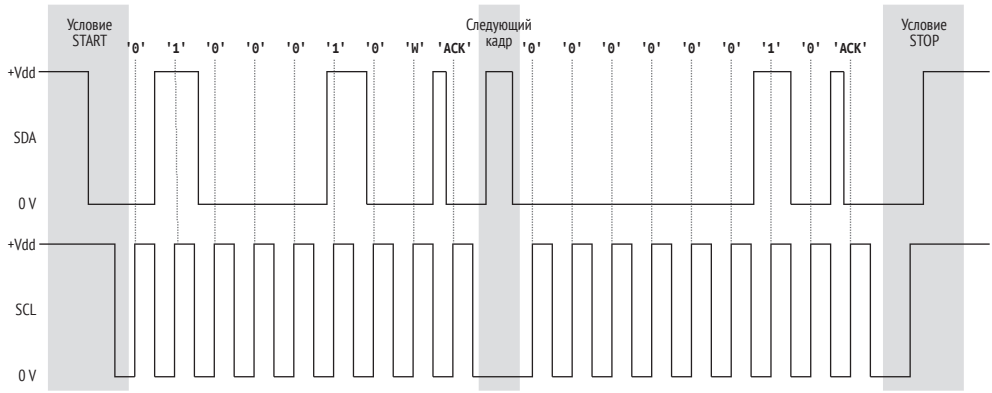

Рис. 7.7 ❖ Однобайтовая транзакция I^2C на шине с правильными состояниями START и STOP и флагом ACK, установленным получателем

Данные всегда передаются с начальным старшим битом (MSB), а формат адресного кадра показан на рис. 7.8.

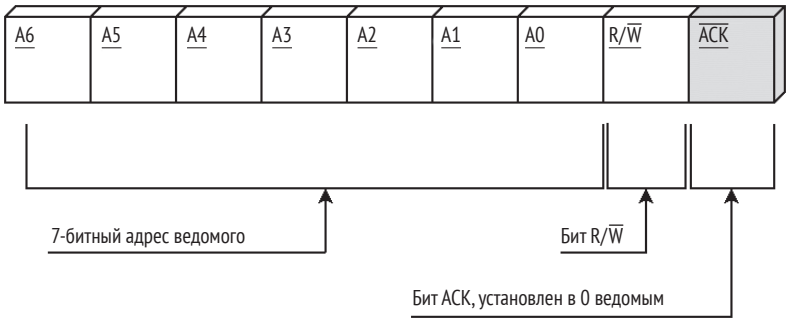

Рис. 7.8 ❖ Формат адресного кадра, содержащего 7-битный адрес получателя и флаг $R/\overline{W}$

Бит $R/\overline{W}$ устанавливается ведущим и указывает направление транзакции. Значение 0 указывает на операцию записи, а значение 1 – на операцию чтения. В зависимости от значения этого бита байты данных, следующие за транзакциями, передаются либо к ведомому устройству (операция записи), либо от выбранного ведомого устройства к ведущему устройству (операция чтения). В операции чтения направление передачи бита ACK также инвертируется, и предполагается, что ведущий подтверждает каждый кадр, полученный от ведомого в рамках транзакции. Ведущий может принять решение прервать передачу в любое время, не переводя ACK в последнем кадре в низкий уровень, а затем выполнив состояние `STOP`.

Транзакция продолжается сразу после передачи кадра адреса, и данные могут быть переданы с использованием последующих однобайтных кадров данных, которые могут быть подтверждены получателем. Если значение бита R/$\overline{W}$ в адресном кадре установлено на 0, ведущий намеревается инициировать операцию записи. Как только ведомое устройство подтвердило адресный фрейм, распознав себя в качестве адресата, оно готово к приему данных и подтверждает фреймы данных до тех пор, пока ведущее устройство не отправит состояние `STOP`.

Согласно протоколу I^2C, если состояние `START` повторяется в конце транзакции вместо установки состояния `STOP`, то новая транзакция может быть запущена сразу же, без перевода шины в состояние ожидания. Повторение состояния `START` гарантирует, что две или более транзакций могут выполняться на одной и той же шине без перерывов, например не позволяя другому ведущему устройству вклиниться между ними.

Менее популярный формат предусматривает 10-битные адреса для ведомых устройств. 10-битные адреса представляют собой расширение стандарта, представленное позднее и обеспечивающее совместимость с 7-битными адресуемыми устройствами на той же шине. Адрес передается с использованием двух последовательных кадров, и первые 5 бит (A6-A2) в первом кадре устанавливаются на 11110, чтобы показать, что это 10-битный адрес. Согласно спецификации протокола, адреса, начинающиеся с 0000 или 1111, зарезервированы и не должны использоваться ведомыми устройствами. В 10-битном формате 2 старших бита содержатся в разрядах A1 и A0 первого кадра, а второй кадр содержит оставшиеся 8 бит. Бит R/$\overline{W}$ сохраняет свою позицию в первом кадре. Этот механизм адресации не очень распространен, так как его поддерживает лишь небольшое количество ведомых устройств.

7.5.2. Затягивание тактов

До сих пор ведущее устройство было единственным, кто управляет сигналом SCL во время транзакций по шине I^2C. Это верно всегда, за исключением случаев, когда ведомое устройство еще не готово передать запрошенные данные от ведущего. В этом конкретном случае ведомое устройство может временно отложить транзакцию, удерживая линию синхронизации на низком уровне, что приводит к приостановке транзакции. Эта операция называется *затягиванием такта* (clock stretching). Ведущее устройство признает свою неспособность генерировать тактовый сигнал, поскольку перевод линии SCL в свободное состояние со стороны ведущего не приводит к появлению на этой линии высокого логического уровня. Ведущее устройство будет пытаться перевести сигнал SCL в его естественное высокое состояние до тех пор, пока запрошенные данные, наконец, не будут доступны на ведомом устройстве, что в конечном итоге снимает удержание линии.

Передача теперь может возобновиться после удержания в течение неопределенного времени, и ведущий по-прежнему должен выдать девять тактовых импульсов для завершения передачи. Поскольку в рамках этой транзакции больше кадров не ожидается, ведущий в конце не устанавливает бит ACK на

низкий уровень и вместо этого выполняет состояние STOP, чтобы правильно завершить транзакцию (рис. 7.9).

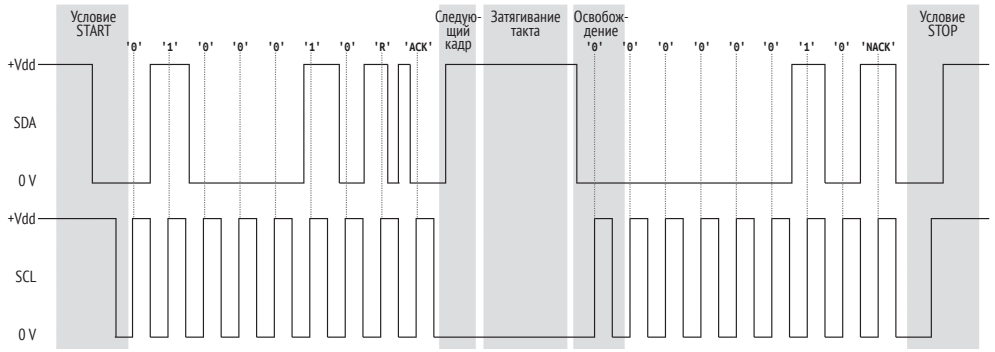

Рис. 7.9 ❖ Транзакция чтения I²C с ответным кадром, задержанным ведомым устройством с использованием метода затягивания тактов

Несмотря на то что не все устройства поддерживают затягивание тактов, этот механизм полезен для завершения транзакций, когда запрошенные данные немного запаздывают. Затягивание тактов – это уникальная особенность I²C, которая делает его очень универсальным протоколом для связи с датчиками и другими периферийными устройствами. Эта возможность очень важна для связи с более медленными устройствами, которые не могут сразу предоставить данные для своевременного завершения транзакции. Желательно, чтобы эта функция корректно поддерживалась ведущим устройством, предназначенным для связи с универсальными ведомыми устройствами I²C. Чтобы обеспечить затягивание такта, ведомое устройство должно иметь аппаратную конфигурацию, которая позволяет ему удерживать линию SCL на низком логическом уровне, пока оно снова не будет готово. Это означает, что в данном конкретном случае линия SCL является двунаправленной, и ведомое устройство должно быть спроектировано так, чтобы иметь возможность управлять логическим уровнем на линии.

7.5.3. Несколько ведущих на одной шине

Протокол I²C предусматривает механизм для обнаружения и реагирования на присутствие нескольких ведущих на шине, который, опять же, основан на поведении линии SDA.

Прежде чем инициировать какую-либо связь, ведущий проверяет доступность шины, проверяя линии SDA и SCL. Конфигурация состояния START сама по себе исключает большинство конфликтов. Одновременный запуск кадра двумя ведущими исключается всякий раз, когда на линии SDA обнаруживается низкий уровень в начальное время отсрочки между двумя фронтами. Но этот механизм не идеально исключает доступ двух ведущих I²C к каналу,

потому что конфликты все еще возможны из-за задержки распространения сигнала по проводу.

Два ведущих устройства, которые одновременно инициируют транзакцию, непрерывно сравнивают состояние линии после передачи каждого бита. В случае двух ведущих, идеально синхронизированных для двух разных передач, первый бит с другим значением на двух источниках будет замечен только ведущим, передающим значение 1, потому что ожидаемое значение не отражается в фактическом состоянии линии. Этот ведущий немедленно прерывает транзакцию, и передатчик может определить ошибку как конфликт в сети, поскольку в данном контексте арбитраж линии был проигран в пользу другого ведущего. Между тем другой ведущий ничего не заметит, как и ведомые, потому что транзакция будет продолжаться, несмотря на молчаливую конкуренцию за линию шины.

7.5.4. Программирование контроллера

Микроконтроллеры могут иметь на борту один или несколько контроллеров I^2C, которые можно привязать к определенным контактам, используя альтернативные функции. Чтобы включить шину `I2C1` на базовой плате, нужно включить тактирование и запустить процедуру инициализации, обращаясь к регистру управления, данных и состояния, отображенному в области памяти периферийных устройств:

```
#define APB1_CLOCK_ER (*(volatile uint32_t *)(0x40023840))
#define APB1_CLOCK_RST (*(volatile uint32_t *)(0x40023820))
#define I2C1_APB1_CLOCK_ER_VAL (1 << 21)
```

Контроллер `I2C1` на STM32F407 связан с контактами PB6 и PB9, когда они настроены на альтернативную функцию `AF 4`:

```
#define I2C1_PIN_AF 4
#define I2C1_SCL 6
#define I2C1_SDA 9
#define GPIO_MODE_AF (2)

static void i2c1_pins_setup(void)
{
  uint32_t reg;
  AHB1_CLOCK_ER |= GPIOB_AHB1_CLOCK_ER;
  /* Режим = AF */
  reg = GPIOB_MODE & ~(0x03 << (I2C1_SCL * 2));
  reg &= ~(0x03 << (I2C1_SDA * 2));
  GPIOB_MODE = reg | (2 << (I2C1_SCL * 2)) |
      (2 << (I2C_SDA * 2));
  /* Альтернативная функция: */
  reg = GPIOB_AFL & ~(0xf << ((I2C1_SCL) * 4));
  GPIOB_AFL = reg | (I2C1_PIN_AF << ((I2C1_SCL - 8) * 4));
  reg = GPIOB_AFH & ~(0xf << ((I2C1_SDA - 8) * 4));
  GPIOB_AFH = reg | (I2C1_PIN_AF << ((I2C1_SDA - 8) * 4));
}
```

Функция инициализации обращается к регистрам конфигурации контроллера I^2C, отображенным в области периферийных устройств. После настройки контактов и запуска RCC скорость приемопередатчика калибруется с использованием частоты тактового сигнала шины APB1 в МГц. Когда тактовые импульсы откалиброваны, приемопередатчик включается установкой бита в регистре CR1. Использованные в следующем примере параметры настраивают тактирование шины на частоту 400 кГц. Хотя настройка протокола по умолчанию предусматривает тактовую частоту 100 кГц, опция 400 кГц была добавлена позже и теперь поддерживается многими современными устройствами:

```
#define I2C1 (0x40005400)
#define APB1_SPEED_IN_MHZ (42)
#define I2C1_CR1 (*(volatile uint32_t *)(I2C1))
#define I2C1_CR2 (*(volatile uint32_t *)(I2C1 + 0x04))
#define I2C1_OAR1 (*(volatile uint32_t *)(I2C1 + 0x08))
#define I2C1_OAR2 (*(volatile uint32_t *)(I2C1 + 0x0c))
#define I2C1_DR (*(volatile uint32_t *)(I2C1 + 0x10))
#define I2C1_SR1 (*(volatile uint32_t *)(I2C1 + 0x14))
#define I2C1_SR2 (*(volatile uint32_t *)(I2C1 + 0x18))
#define I2C1_CCR (*(volatile uint32_t *)(I2C1 + 0x1c))
#define I2C1_TRISE (*(volatile uint32_t *)(I2C1 + 0x20))
#define I2C_CR2_FREQ_MASK (0x3ff)
#define I2C_CCR_MASK (0xfff)
#define I2C_TRISE_MASK (0x3f)
#define I2C_CR1_ENABLE (1 << 0)

void i2c1_setup(void)
{
  uint32_t reg;
  i2c1_pins_setup();
  APB1_CLOCK_ER |= I2C1_APB1_CLOCK_ER_VAL;
  I2C1_CR1 &= ~I2C_CR1_ENABLE;
  i2c1_reset();
  reg = I2C1_CR2 & ~(I2C_CR2_FREQ_MASK);
  I2C1_CR2 = reg | APB1_SPEED_IN_MHZ;
  reg = I2C1_CCR & ~(I2C_CCR_MASK);
  I2C1_CCR = reg | (APB1_SPEED_IN_MHZ * 5);
  reg = I2C1_TRISE & ~(I2C_TRISE_MASK);
  I2C1_TRISE = reg | APB1_SPEED_IN_MHZ + 1;
  I2C1_CR1 |= I2C_CR1_ENABLE;
}
```

С этого момента контроллер готов к настройке и использованию в режиме ведущего или ведомого. Данные можно читать и записывать с помощью I2C1_DR так же, как это было в случае SPI и UART. Основное отличие здесь заключается в том, что для ведущего устройства I^2C состояния START и STOP необходимо инициировать вручную путем установки соответствующих значений в регистре I2C1_CR1. Для этой цели предназначены следующие функции:

```
static void i2c1_send_start(void)
{
  volatile uint32_t sr1;
```

```
  I2C1_CR1 |= I2C_CR1_START;
  do {
    sr1 = I2C1_SR1;
  } while ((sr1 & I2C_SR1_START) == 0);
}
static void i2c1_send_stop(void)
{
  I2C1_CR1 |= I2C_CR1_STOP;
}
```

В конце каждого состояния шина должна быть проверена на наличие возможных ошибок или аномальных событий. Комбинация флагов в `I2C1_CR1` и `I2C1_CR2` должна отражать ожидаемое состояние для продолжения транзакции, или она должна быть корректно прервана в случае тайм-аутов либо неисправимых ошибок.

Из-за сложности, вызванной большим количеством событий, возможных во время настройки транзакции, необходимо реализовать полный конечный автомат, который отслеживает фазы передачи, чтобы использовать приемопередатчик в режиме ведущего.

В качестве демонстрации основных принципов работы с приемопередатчиком ниже представлен пример кода последовательного взаимодействия с шиной, но реальный сценарий потребует от нас отслеживания состояния каждой транзакции и реагирования на множество ситуаций, возможных в рамках комбинации флагов, содержащихся в `I2C1_SR1` и `I2C1_SR2`. Эта последовательность операций инициирует транзакцию к ведомому I^2C с адресом `0x42`, и если ведомое устройство отвечает, оно отправляет 2 байта со значениями `0x00` и `0x01` соответственно. Единственная цель этого примера – показать взаимодействие с трансивером, и в нем не предусмотрено исправление ни одной из возможных ошибок. В начале транзакции обнулим флаги, связанные с состоянием ACK или `STOP`, и включим приемопередатчик, используя младший бит в `CR1`:

```
void i2c1_test_sequence(void)
{
  volatile uint32_t sr1, sr2;
  const uint8_t address = 0x42;
  I2C1_CR1 &= ~(I2C_CR1_ENABLE | I2C_CR1_STOP | I2C_CR1_ACK);
  I2C1_CR1 |= I2C_CR1_ENABLE;
```

Чтобы гарантировать, что никакой другой ведущий не занимает шину, процедура зависает до тех пор, пока в приемопередатчике не будет снят флаг занятости:

```
do {
  sr2 = I2C1_SR2;
} while ((sr2 & I2C_SR2_BUSY) != 0);
```

Состояние `START` отправляется с использованием функции, определенной ранее, которая также будет ждать, пока на шине не появится такое же состояние `START`:

```
i2c1_send_start();
```

Адрес получателя устанавливается равным старшим 7 битам байта, который мы собираемся передать. Младший бит сброшен, что указывает на операцию записи. Чтобы продолжить после успешной передачи адреса, подтвержденного принимающим ведомым устройством, в I2C1_SR2 должны быть установлены два флага, указывающие на то, что был выбран режим ведущего, а шина все еще занята:

```
I2C1_DR = (address << 1);
do {
  sr2 = I2C1_SR2;
} while ((sr2 & (I2C_SR2_BUSY | I2C_SR2_MASTER)) !=
        (I2C_SR2_BUSY | I2C_SR2_MASTER));
```

Обмен данными с ведомым теперь инициирован, и можно передать 2 байта данных. Событие EMPTY буфера передачи указывает на окончание передачи каждого байта в кадре транзакции:

```
I2C1_DR = (0x00);
do {
  sr1 = I2C1_SR1;
} while ((sr1 & I2C_SR1_TX_EMPTY) != 0);
I2C1_DR = (0x01);
do {
  sr1 = I2C1_SR1;
} while ((sr1 & I2C_SR1_TX_EMPTY) != 0);
```

Наконец, устанавливается состояние STOP, и транзакция завершается:

```
  i2c1_send_stop();
}
```

7.5.5. Обработка прерываний

Интерфейс событий контроллера I^2C на базовом целевом устройстве достаточно сложен, чтобы обеспечить два отдельных обработчика прерываний для каждого приемопередатчика. Предлагаемая реализация универсального ведущего I^2C включает правильную настройку прерывания и определение всех комбинаций состояний и событий. Контроллер I^2C можно настроить так, чтобы прерывания ассоциировались со всеми соответствующими событиями, происходящими на шине, что позволяет точно настраивать конкретные сценарии и более или менее полно реализовывать протокол I^2C.

7.6. Заключение

Эта глава содержит информацию, которой достаточно, чтобы начать программировать системную поддержку наиболее популярных интерфейсов связи по локальной шине, доступных на встраиваемых устройствах. Наличие

доступа к периферийным устройствам и другим микроконтроллерам, расположенным поблизости, является одним из типичных требований встраиваемых систем, взаимодействующих с датчиками, исполнительными механизмами и другими устройствами.

Существует несколько реализаций, обеспечивающих более высокий уровень абстракции для рассмотренных здесь приемопередатчиков. Протоколы последовательной связи, описанные в этой главе, а именно UART, SPI и I^2C, обычно доступны через драйверы, входящие в комплект поддержки платы, и их не нужно заново разрабатывать с нуля. Эта глава специально сосредоточена на изучении поведения компонентов с точки зрения реализации на низком уровне, чтобы лучше понять интерфейс, предоставляемый производителем оборудования, и, возможно, предложить инструменты для разработки новых способов доступа к интерфейсам, адаптированных или оптимизированных для конкретной платформы или сценария, а также дать понимание выбора, доступного для некоторых параметров протоколов.

В следующей главе описаны механизмы, используемые для снижения энергопотребления встраиваемых систем, путем использования функций низкого и сверхнизкого энергопотребления, присутствующих в современных встраиваемых устройствах.

Глава 8

Управление питанием и энергосбережение

Энергоэффективность всегда была одним из ведущих факторов на рынке микроконтроллеров. С начала 2000-х годов 16-разрядные RISC-микроконтроллеры для обработки сигналов, такие как MSP430, специально разрабатываются для использования в системах с низким энергопотреблением, и до сих пор они лидируют среди архитектур со сверхнизким энергопотреблением в области встраиваемых систем.

Тем не менее за последние несколько лет удалось значительно снизить размеры и энергопотребление более продвинутых 32-разрядных RISC-микроконтроллеров, обладающих богатым набором функций и способных работать с операционными системами реального времени, и они успешно вышли на рынок систем с низким и сверхнизким энергопотреблением. Системы и устройства с батарейным питанием, основанные на методах сбора энергии от внешних источников (energy-harvesting), становятся все более распространенными во многих отраслях. Беспроводная связь с низким энергопотреблением в настоящее время применяется в ряде подключенных платформ, поэтому все большее число систем IoT разрабатывают с учетом низкого и сверхнизкого энергопотребления.

В зависимости от архитектуры микроконтроллеры предлагают различные стратегии для снижения энергопотребления во время работы и реализации состояний с низким энергопотреблением, которые потребляют очень мало энергии при активации.

Снижение энергопотребления встраиваемой системы часто является сложной задачей. Любые внутренние модули микроконтроллера могут потреблять энергию, если их не отключить должным образом. Генерация высокочастотных тактовых импульсов – одна из самых энергозатратных операций, поэтому тактовый сигнал ЦП и шины следует включать только тогда, когда они используются.

Поиск идеальной стратегии энергосбережения зависит от допустимых компромиссов между производительностью и энергосбережением. Микроконтроллеры, предназначенные для приложений со сверхнизким энергопотреблением, способны замедлять частоту процессора и даже переходить

в различные варианты состояния гибернации, когда все тактовые генераторы останавливаются, а внешние периферийные устройства отключаются для максимальной экономии энергии.

При использовании соответствующих методов профилирования энергопотребления и реализации стратегий сверхнизкого энергопотребления устройства с батарейным питанием могут работать в течение нескольких лет, прежде чем потребуется замена источника питания. Используя альтернативные источники энергии, такие как солнечные батареи, преобразователи тепла или другие способы извлечения энергии из окружающей среды, хорошо спроектированная встраиваемая система может работать бесконечно долго, пока это позволяют внешние условия.

Усовершенствованные микропроцессоры, работающие на очень высоких скоростях, как правило, не предназначены для эффективной оптимизации энергопотребления, что делает микроконтроллеры младшего уровня с низким энергопотреблением, такие как Cortex-M, столь популярными во всех тех встраиваемых системах, где малое энергопотребление является одним из основных требований.

В этой главе рассмотрены несколько ключевых приемов проектирования встраиваемых систем с низким и сверхнизким энергопотреблением. В качестве примеров оптимизации энергопотребления на реальных объектах демонстрируются экономичные версии микроконтроллера Cortex-M. Глава разделена на четыре раздела:

- конфигурация системы;
- режимы работы с низким энергопотреблением;
- измерение потребляемой мощности;
- разработка встроенных приложений с низким энергопотреблением.

К концу данной главы вы научитесь управлять различными конфигурациями с низким энергопотреблением микроконтроллера и периферийных устройств.

8.1. Технические требования

Файлы кода для этой главы доступны на GitHub по адресу https://github.com/PacktPublishing/Embedded-Systems-Architecture-Second-Edition/tree/main/Chapter8.

8.2. Конфигурация системы

Система, в спецификацию которой включены ограничения по энергопотреблению, должна быть спроектирована таким образом, чтобы соответствовать требованиям во всех ее аспектах, включая аппаратное обеспечение, программное обеспечение и механическую конструкцию. При выборе компонентов и периферийных устройств необходимо учитывать их энергетиче-

ские характеристики. Внешние периферийные устройства часто являются наиболее энергоемкими компонентами, поэтому их питание должно прерываться микроконтроллером, когда они не используются.

В этом разделе описываются лучшие методики, касающиеся конфигурации периферийных устройств, настроек тактирования системы и управления напряжением, а также их влияние на энергопотребление.

8.2.1. Аппаратная часть системы

В маломощных встраиваемых системах конструкция аппаратного обеспечения должна предусматривать возможность включения и выключения питания периферийных устройств с помощью вывода GPIO. Это лучше сделать с помощью линии, которая по умолчанию имеет низкий уровень, поддерживаемый с помощью пассивных компонентов, когда GPIO не управляется микроконтроллером. Для управления питанием, подаваемым на внешние периферийные устройства, часто используются МОП-транзисторы, к затвору которых подключен вывод GPIO.

Даже когда периферийные устройства отключены путем разрыва линии источника питания, небольшие токи могут протекать через другие подключенные к ним электрические цепи, такие как последовательная шина или другие управляющие сигналы. Разработчик оборудования должен обнаруживать и идентифицировать эти утечки на ранних стадиях прототипирования, чтобы свести к минимуму потери энергии таким образом.

Кроме того, если стратегия энергосбережения предусматривает возможность перевода микропроцессора в режим глубокого сна, логика входных сигналов должна быть адаптирована для обеспечения правильных событий пробуждения и возобновления нормальной работы. Сигналы, которые не формируются в спящем режиме, должны сохранять определенное логическое значение, обеспечиваемое пассивными компонентами.

8.2.2. Управление тактированием

Внутренние периферийные устройства и интерфейсы, которые не используются, также должны оставаться выключенными. Если платформа поддерживает внутреннюю маршрутизацию тактовых импульсов, то для управления энергопотреблением внутренних модулей и интерфейсов можно включать и отключать подачу тактовых импульсов. Каждая работающая линия передачи тактовых импульсов сама по себе увеличивает энергопотребление. Кроме того, чем выше коэффициент умножения, применяемый для генерации тактового сигнала ЦП из медленного генератора, тем выше мощность, потребляемая ФАПЧ. Умножитель тактовой частоты на основе ФАПЧ является одним из наиболее энергоемких компонентов системы. Мощность, потребляемая ЦП, также прямо пропорциональна его тактовой частоте. Многие ЦП допускают работу с пониженной тактовой частотой, предлагая ряд компромиссов между производительностью и энергосбережением. Соответственно, ФАПЧ

обычно можно переконфигурировать во время выполнения приложения для адаптации к различным профилям. Но каждое изменение системной тактовой частоты требует перенастройки всех делителей для всех используемых таймеров и периферийных устройств.

На тестовой платформе, с которой выполняются тесты в этой книге, можно перенастроить частоту ЦП во время выполнения, чтобы сэкономить значительное количество энергии, когда системе не требуется вычислительная мощность. Для этого нужно внести изменения в функцию в `system.c`, которую раньше использовали для настройки системной тактовой частоты во всех примерах, чтобы разрешить выбор двух разных рабочих частот. В режиме высокой производительности система работает на максимальной частоте 168 МГц. Если аргумент флага энергосбережения не равен нулю, тактовый генератор настроен на работу на частоте 48 МГц для экономии энергии:

```
void clock_pll_on(int powersave)
{
  uint32_t reg32, plln, pllm, pllq,
      pllp, pllr, hpre, ppre1, ppre2,
      flash_waitstates;
  if (powersave) {
    cpu_freq = 48000000;
    pllm = 8;
    plln = 96;
    pllp = 2;
    pllq = 2;
    pllr = 0;
    hpre = RCC_PRESCALER_DIV_NONE;
    ppre1 = RCC_PRESCALER_DIV_4;
    ppre2 = RCC_PRESCALER_DIV_2;
    flash_waitstates = 5;
  } else {
    cpu_freq = 168000000;
    pllm = 8;
    plln = 336;
    pllp = 2;
    pllq = 7;
    pllr = 0;
    hpre = RCC_PRESCALER_DIV_NONE;
    ppre1 = RCC_PRESCALER_DIV_4;
    ppre2 = RCC_PRESCALER_DIV_2;
    flash_waitstates = 3;
  }
```

Здесь также было изменено количество состояний ожидания для операции `flash`, поскольку, согласно документации STM32F407, на частоте 48 МГц флеш-памяти требуется всего три состояния ожидания:

```
flash_set_waitstates(flash_waitstates);
```

Процедура настройки системной тактовой частоты обычная. Сначала включаем HSI и выбираем его в качестве временного источника тактирова-

ния. После этого включается внешний генератор с частотой 8 МГц, и от него можно питать ФАПЧ:

```
RCC_CR |= RCC_CR_HSION;
DMB();
while ((RCC_CR & RCC_CR_HSIRDY) == 0) {};

reg32 = RCC_CFGR;
reg32 &= ~((1 << 1) | (1 << 0));
RCC_CFGR = (reg32 | RCC_CFGR_SW_HSI);
DMB();

RCC_CR |= RCC_CR_HSEON;
DMB();
while ((RCC_CR & RCC_CR_HSERDY) == 0)
   ;
```

Настраиваем параметры делителей и умножителей тактовой частоты для выбранного режима в регистре конфигурации ФАПЧ, а затем включаем ФАПЧ:

```
  reg32 = RCC_CFGR;
  reg32 &= ~(0xF0);
  RCC_CFGR = (reg32 | (hpre << 4));
  DMB();
  reg32 = RCC_CFGR;
  reg32 &= ~(0x1C00);
  RCC_CFGR = (reg32 | (ppre1 << 10));
  DMB();
  reg32 = RCC_CFGR;
  reg32 &= ~(0x07 << 13);
  RCC_CFGR = (reg32 | (ppre2 << 13));
  DMB();

  reg32 = RCC_PLLCFGR;
  reg32 &= ~(PLL_FULL_MASK);
  RCC_PLLCFGR = reg32 | RCC_PLLCFGR_PLLSRC | pllm |
  (plln << 6) | (((pllp >> 1) - 1) << 16) | (pllq << 24);
}
```

Изменение частоты ЦП и системной тактовой частоты означает, что все периферийные устройства, использующие системные тактовые импульсы, должны быть перенастроены. Если в системе есть работающий таймер или приложение использует какое-либо устройство, тактируемое от системы, необходимо позаботиться о том, чтобы процесс перенастройки прошел без ущерба для работоспособности системы.

Работа системы на более низкой скорости дает другие преимущества, такие как возможность уменьшить количество состояний ожидания, необходимых для доступа к флеш-памяти, и включить дополнительные функции с низким энергопотреблением, которые доступны только тогда, когда система не работает на полной скорости.

Встраиваемые платформы обычно оснащают низкочастотными тактовыми генераторами, работающими на частоте в несколько килогерц, которые можно использовать в качестве источников тактовых импульсов для устройств хронометража, таких как сторожевые устройства и *часы реального времени* (real-time clock, RTC). Внешние или внутренние низкочастотные генераторы могут продолжать работу в режимах с низким энергопотреблением и применяться для реализации стратегий пробуждения.

8.2.3. Управление напряжением

Микроконтроллеры имеют относительно широкий диапазон рабочих напряжений. Но низкое напряжение питания делает невозможной работу ЦП на полной скорости, а флеш-памяти могут потребоваться дополнительные состояния ожидания из-за физических свойств оборудования. Тем не менее в некоторых случаях логика, устойчивая к низкому напряжению, способствует снижению энергопотребления.

Часто применяются внутренние управляемые регуляторы напряжения, которые можно настроить для получения более низкого напряжения сигналов ядра, чтобы достичь компромисса между энергопотреблением и производительностью, когда ЦП не работает на максимальной частоте.

Важным аспектом, которым часто пренебрегают, является мощность, потребляемая триггерами Шмитта в логике цифрового входа. Когда выводы GPIO настроены как цифровой вход, но не переведены в стабильное логическое состояние с помощью внешних пассивных компонентов, напряжение на них может колебаться вокруг среднего значения из-за электромагнитных полей в окружающей среде. Это приводит к хаотичному срабатыванию входных триггеров, что влечет за собой потерю энергии при каждом изменении логического состояния.

8.3. Режимы работы с низким энергопотреблением

Микроконтроллеры могут работать в разных режимах энергопотребления, переходя от полного быстродействия к полной гибернации. Правильное понимание режимов микроконтроллера с низким энергопотреблением имеет основополагающее значение для проектирования систем с улучшенными профилями энергопотребления. Каждая архитектура обеспечивает определенные конфигурации питания, при которых ЦП или другие шины и периферийные устройства отключены, а также определяет соответствующие механизмы, используемые системным программным обеспечением для входа и выхода из режимов пониженного энергопотребления.

В микроконтроллере на базе ARM различные режимы энергопотребления можно кратко описать так:

- **нормальный рабочий режим** (normal operation mode): активные компоненты выбираются путем подачи таковых импульсов, система тактирования работает на нужной частоте;
- **режим сна** (sleep mode): тактирование ЦП временно приостанавливается, но все периферийные устройства продолжают работать в обычном режиме. Поскольку ЦП не работает, в этом режиме экономится заметное, хотя и не очень большое, количество энергии. Выполнение может быть возобновлено после получения запроса на прерывание;
- **режим остановки** (stop mode): тактирование ЦП и шины отключено. Все периферийные устройства, питаемые от микроконтроллера, выключены. Внутреннее ОЗУ и регистры ЦП сохраняют текущие значения, поскольку основной источник питания остается включенным. Энергопотребление значительно снижено, но систему по-прежнему можно разбудить и возобновить выполнение с помощью внешнего прерывания или события. Этот режим также часто не очень удачно называют режимом глубокого сна (deep sleep mode), хотя на самом деле это лишь один из двух доступных режимов глубокого сна;
- **режим ожидания** (standby mode): все источники питающего напряжения выключены, содержимое ОЗУ и регистров теряется. Небольшое количество энергии, порядка нескольких микроватт, может потребоваться для поддержания работоспособности схемы пробуждения во время фазы ожидания. В этом случае пробуждение возможно только при нескольких определенных условиях, таких как часы реального времени с внешним питанием или аппаратное событие пробуждения (например, нажатие кнопки). Когда система выходит из режима ожидания, выполняется обычная процедура загрузки, и выполнение возобновляется с процедуры сброса.

Микрокод ARMv7 предоставляет две инструкции для перехода в режимы работы с низким энергопотреблением:

- **ожидание прерывания** (wait for interrupt, WFI);
- **ожидание события** (wait for event, WFE).

Эти инструкции могут быть выполнены в любое время в обычном рабочем режиме. WFI переводит систему в режим пониженного энергопотребления до тех пор, пока не будет получен следующий запрос на прерывание, в то время как WFE работает немного иначе. Только несколько событий в системе, включая внешние прерывания, могут инициировать смену режима энергопотребления. Обычные запросы на прерывание не вернут систему обратно в нормальный рабочий режим, если она находится в спящем режиме или режиме остановки, который был введен с помощью WFE.

Режим пониженного энергопотребления, который вводится при вызове, зависит от настроек, хранящихся в *регистре управления системой* (system control register, SCR), который на Cortex-M находится в области конфигурации системы по адресу `0xE000ED10`. SCR содержит только три значимых 1-битных флага:

- SLEEPONEXIT (бит 1): если этот флаг установлен, система перейдет в режим пониженного энергопотребления в конце выполнения следующего обработчика прерывания;
- SLEEPDEEP (бит 2): определяет, какой режим вводится при вызове инструкций WFI или WFE либо при возврате из прерывания с активным SLEEPONEXIT. Если этот флаг сброшен, выбран спящий режим. Если флаг установлен, при входе в режим пониженного энергопотребления система будет переведена в режим остановки или ожидания, в зависимости от конфигурации регистров управления питанием;
- SEVONPEND (бит 4): когда этот флаг установлен, любое прерывание, ожидающее обработки в режиме пониженного энергопотребления, вызовет событие пробуждения, независимо от того, был ли введен спящий режим или режим останова с помощью инструкции WFI или WFE.

Обратите внимание, что бит 0, бит 3 и биты 5–31 зарезервированы (должны иметь значение 0).

8.3.1. Конфигурация глубокого сна

Для выбора между режимами остановки и ожидания и для настройки определенных параметров, связанных с режимами глубокого сна, наша эталонная платформа предоставляет контроллер питания, отображаемый в области внутренних периферийных устройств по адресу 0x40007000. Контроллер состоит из двух регистров:

- PWR_CR (регистр управления) со смещением 0;
- PWR_SCR (регистр состояния и управления) со смещением 4.

В этих двух регистрах можно настроить следующие параметры:

- **управление выходом регулятора напряжения** (regulator voltage-scaling output selection, VOS), задается с помощью бита 14 PWR_CR. Когда он установлен, контроллер экономит дополнительную энергию в нормальном рабочем режиме, настраивая внутренний регулятор на создание немного более низкого напряжения для логики ядра ЦП. Эта функция доступна только в том случае, если целевое устройство не работает на максимальной частоте;
- **отключение флеш-памяти в режиме глубокого сна** (flash power down in deep sleep, FPDS), задается битом 9 PWR_CR. Если он установлен при переходе в один из режимов глубокого сна, флеш-память будет полностью отключена, пока система находится в спящем режиме. Это приводит к умеренной экономии энергии, но также влияет на время пробуждения;
- **отключение питания в глубоком спящем режиме** (power down in deep sleep, PDDS), устанавливается через PWR_CR, бит 1. Этот бит определяет, какой именно вариант применяется, когда ЦП переходит в режим глубокого сна. Если бит сброшен, выбирается режим остановки (stop mode). Если бит установлен, система переходит в режим ожидания (standby mode);

- **глубокий сон с низким энергопотреблением** (low-power deep sleep, LPDS), устанавливается с помощью бита 0 PWR_CR. Этот бит действует только в режиме остановки. Если он установлен, потребление энергии немного снижается в режиме глубокого сна за счет включения *режима пониженной нагрузки* (under-drive mode) во внутреннем регуляторе напряжения. На логику ядра подается минимальное напряжение, позволяющее сохранить содержимое памяти и регистров. Эта функция доступна только в том случае, если система не работает на полной скорости;
- **включение вывода пробуждения** (enable wake-up pin, EWUP), задается битом 4 PWR_CSR. Этот флаг определяет, может ли контакт пробуждения использоваться как обычный GPIO или он зарезервирован для обнаружения сигнала пробуждения в режиме ожидания. С этой функцией на нашей базовой платформе связан вывод PA0.

Флаг пробуждения (wake-up flag, WUF) автоматически устанавливается аппаратно при выходе из спящего режима или режима глубокого сна и может быть прочитан через бит 0 регистра PWR_CSR. Запись 1 в бит 2 регистра PWR_CR очищает флаг пробуждения (clear the wake-up flag, CWUF).

Микроконтроллер STM32F407 позволяет получить доступ к регистрам, связанным с режимами пониженного энергопотребления и конфигурацией, с помощью следующих макросов:

```
#define SCB_SCR (*(volatile uint32_t *)(0xE000ED10))
#define SCB_SCR_SEVONPEND (1 << 4)
#define SCB_SCR_SLEEPDEEP (1 << 2)
#define SCB_SCR_SLEEPONEXIT (1 << 1)

#define POW_BASE (0x40007000)
#define POW_CR (*(volatile uint32_t *)(POW_BASE + 0x00))
#define POW_SCR (*(volatile uint32_t *)(POW_BASE + 0x04))

#define POW_CR_VOS (1 << 14)
#define POW_CR_FPDS (1 << 9)
#define POW_CR_CWUF (1 << 2)
#define POW_CR_PDDS (1 << 1)
#define POW_CR_LPDS (1 << 0)
#define POW_SCR_WUF (1 << 0)
#define POW_SCR_EWUP (1 << 4)
```

Для активации режимов с низким энергопотреблением и генерации спонтанных событий нужно определить макросы, содержащие одиночные встроенные ассемблерные инструкции, следующим образом:

```
#define WFI() asm volatile ("wfi")
#define WFE() asm volatile ("wfe")
```

Если переход в спящий режим осуществляется через WFI, система приостанавливает выполнение до следующего прерывания. Вместо этого переход в спящий режим с помощью WFE гарантирует, что только выбранные события

могут снова разбудить систему. События различных типов, происходящие в системе, можно активировать для пробуждения из WFE.

При входе в WFE все прерывания, активные в NVIC, по-прежнему будут считаться событиями, тем самым активируя пробуждение из WFE. Прерывания можно временно отфильтровать, отключив соответствующую линию IRQ в NVIC. Если прерывание фильтруется таким образом с помощью NVIC, оно остается в состоянии ожидания и обрабатывается, как только система возвращается в нормальный рабочий режим.

8.3.2. Режим остановки

Спящий режим включается по умолчанию каждый раз, когда вызываются инструкции WFI или WFE, пока SCB_SCR_SLEEPDEEP остается выключенным. Другие режимы с низким энергопотреблением можно включить, установив флаг SLEEPDEEP. Чтобы войти в один из доступных режимов глубокого сна, перед вызовом WFI или WFE необходимо настроить регистры SCB_SCR и POW. В зависимости от конфигурации система переходит в один из двух режимов глубокого сна: остановки или ожидания.

В следующем примере непрерывно работающий таймер с частотой 1 Гц переключает светодиод 10 раз перед погружением системы в режим глубокого сна с использованием WFE. Цикл main остается в спящем режиме между прерываниями таймера, используя WFI:

```
void main(void) {
  int sleep = 0;
  pll_on(0);
  button_setup();
  led_setup();
  timer_init(CPU_FREQ, 1, 1000);
  while(1) {
    if (timer_elapsed) {
      WFE(); /* consume timer event */
      led_toggle();
      timer_elapsed = 0;
    }
    if (tim2_ticks > 10) {
      sleep = 1;
      tim2_ticks = 0;
    }
    if (sleep) {
      enter_lowpower_mode();
      WFE();
      sleep = 0;
      exit_lowpower_mode();
    } else
      WFI();
  }
}
```

Подпрограмма обслуживания прерывания для таймера увеличивает счетчик tim2_ticks на 1 и устанавливает флаг timer_elapsed, который заставит цикл main переключать светодиод и потреблять событие, сгенерированное таймером:

```
void isr_tim2(void) {
  nvic_irq_clear(NVIC_TIM2_IRQN);
  TIM2_SR &= ~TIM_SR_UIF;
  tim2_ticks++;
  timer_elapsed++;
}
```

Процедура enter_lowpower_mode отвечает за установку необходимых значений в системном блоке управления и в регистрах управления питанием в зависимости от желаемого режима пониженного энергопотребления и соответствующую настройку всех оптимизаций.

Процедура enter_lowpower_mode выполняет следующие действия:

1) выключает светодиод;
2) устанавливает значения в SCB_SCR и регистре питания для настройки режима пониженного энергопотребления, который будет введен по инструкции WFE;
3) выбирает одну дополнительную оптимизацию мощности.

Процедура реализована в следующем коде:

```
void enter_lowpower_mode(void)
{
  uint32_t scr = 0;
  led_off();
  scr = SCB_SCR;
  scr &= ~SCB_SCR_SEVONPEND;
  scr |= SCB_SCR_SLEEPDEEP;
  scr &= ~SCB_SCR_SLEEPONEXIT;
  SCB_SCR = scr;
  POW_CR |= POW_CR_CWUF | POW_CR_FPDS | POW_CR_LPDS;
}
```

В данном случае режим остановки настраивается на максимально возможное снижение энергопотребления за счет активации настроек регулятора напряжения с низким энергопотреблением (через POW_CR_LPDS) и отключения флеш-памяти (через POW_CR_FPDS).

Теперь вход в режим пониженного энергопотребления осуществляется через вызов WFE(). Чтобы иметь возможность разбудить систему, нужно настроить событие EXTI, которое связано с нажатием пользователем кнопки на плате. Для этого нужно настроить EXTI0 на срабатывание по нарастающему фронту, поскольку при нажатии кнопки вывод PA0 меняет свой логический уровень с 0 на 1.

Поскольку нас не особо интересует само прерывание, нам достаточно сделать так, чтобы флаг для генерации запроса на прерывание был сброшен в EXTI. Контроллер событий в любом случае обеспечит генерацию события,

потому что флаг, относящийся к входному контакту, принудительно установлен в регистре EXTI_EMR.

Начальная конфигурация для события кнопки пользователя выглядит следующим образом:

```
void button_setup(void)
{
  uint32_t reg;
  AHB1_CLOCK_ER |= GPIOA_AHB1_CLOCK_ER;
  APB2_CLOCK_ER |= SYSCFG_APB2_CLOCK_ER;
  GPIOA_MODE &= ~ (0x03 << (BUTTON_PIN * 2));
  EXTI_CR0 &= ~EXTI_CR_EXTI0_MASK;
  EXTI_IMR &= ~0x7FFFFF;
  reg = EXTI_EMR & ~0x7FFFFF;
  EXTI_EMR = reg | (1 << BUTTON_PIN);
  reg = EXTI_RTSR & ~0x7FFFFF;
  EXTI_RTSR = reg | (1 << BUTTON_PIN);
  EXTI_FTSR &= ~0x7FFFFF;
}
```

Для кнопки не настроены прерывания, так как одного события достаточно, чтобы разбудить плату в режиме остановки.

При входе в режим остановки блок ФАПЧ будет отключен, а HSI будет автоматически выбран в качестве источника тактового сигнала, когда система вернется в нормальный рабочий режим. Чтобы восстановить конфигурацию тактирования, необходимо выполнить несколько шагов сразу после выхода из режима остановки:

1) сбросить флаг SCB_SCR_SLEEPDEEP, чтобы следующий вызов WFI или WFE не приводил к другому переключению в режим остановки;
2) сбросить флаг пробуждения в регистре POW_CR, установленный аппаратно в конце режима остановки;
3) заново сконфигурировать ФАПЧ, поскольку тактовый генератор снова запущен;
4) включить светодиод;
5) снова разрешить прерывание TIM2, чтобы таймер восстановил свою функциональность в нормальном рабочем режиме:

```
void exit_lowpower_mode(void)
{
  SCB_SCR &= ~SCB_SCR_SLEEPDEEP;
  POW_CR |= POW_CR_CWUF | POW_CR_CSBF;
  clock_pll_on(0);
  timer_init(cpu_freq, 1, 1000);
  led_on();
}
```

Режим глубокого сна значительно снижает энергопотребление, и это идеальная ситуация, когда система должна поддерживать текущее рабочее состояние, но может быть заморожена на более длительный период.

8.3.3. Режим ожидания

В режиме ожидания система может перейти в режим сверхнизкого энергопотребления, потребляя всего несколько микроампер и ожидая повторной инициализации внешним событием. Для входа в режим ожидания необходимо установить флаг SCB_SCR_PDDS перед вызовом WFI или WFE. Пока система находится в режиме ожидания, все источники питающего напряжения выключены, за исключением низкоскоростных генераторов, которые используются для тактирования независимого сторожевого таймера и часов реального времени.

Процедура входа в режим ожидания немного отличается от той, которая используется для входа в режим остановки. Флаг SCB_SCR_PDDS установлен для выбора режима ожидания в качестве варианта глубокого сна. Флаг SCB_SCR_LPDS в этом случае не активируется, потому что он не действует в режиме ожидания:

```
void enter_lowpower_mode(void)
{
  uint32_t scr = 0;
  led_off();
  scr = SCB_SCR;
  scr &= ~SCB_SCR_SEVONPEND;
  scr |= SCB_SCR_SLEEPDEEP;
  scr &= ~SCB_SCR_SLEEPONEXIT;
  SCB_SCR = scr;
  POW_CR |= POW_CR_CWUF | POW_CR_FPDS | POW_CR_PDDS;
  POW_SCR |= POW_CR_CSBF;
}
```

В этом случае настраивать событие EXTI на нажатие кнопки бесполезно, так как контроллеры GPIO будут отключены, пока микроконтроллер находится в режиме ожидания. Самый простой способ выйти из этого состояния – настроить часы реального времени на генерацию события пробуждения через фиксированный промежуток времени. Фактически во время фазы ожидания поддерживаются в рабочем состоянии только несколько периферийных устройств, и все они сгруппированы в специальном разделе конфигурации тактовой подсистемы, а именно в резервной области. Резервная область состоит из часов реального времени и небольшой части системы тактирования, содержащей внутренний и внешний низкоскоростные генераторы. Доступ для записи в регистры, относящиеся к резервной области, контролируется флагом отключения защиты резервной области POW_CR_DPB, расположенным в регистре POW_CR в бите 8.

Регистры конфигурации часов реального времени, отображаемые в области периферийных устройств, начиная с адреса 0x40002870, защищены от случайной записи из-за электромагнитных помех, а это означает, что перед доступом к другим регистрам в регистр защиты от записи необходимо записать специальную последовательность значений. Модуль RTC, интегрированный в нашу тестовую платформу, имеет множество функций, таких как

отслеживание даты и времени, а также установка настраиваемых сигналов пробуждения и регулярных событий с отметками времени. В этом примере нужно использовать только событие пробуждения, поэтому большинство регистров RTC здесь не рассматриваем.

Ограниченный перечень регистров, которые нам понадобятся для использования RTC, выглядит следующим образом:

- регистр управления `RTC_CR` предназначен для настройки различных функций, предоставляемых RTC. В этом примере используются значения, связанные с триггером пробуждения, включение прерывания с флагом разрешения прерывания таймера пробуждения `RTC_CR_WUTIE` и включение счетчика таймера пробуждения с помощью `RTC_CR_WUTE`;
- регистр инициализации и состояния `RTC_ISR` в этом примере используется для проверки состояния записи регистра настройки таймера пробуждения с помощью специального флага `RTC_ISR_WUTWF` во время установки таймера;
- регистр таймера пробуждения `RTC_WUTR` используется для установки интервала перед следующим событием пробуждения;
- регистр защиты от записи `RTC_WPR` используется для передачи последовательности разблокировки перед записью в другие регистры региона.

Макросы препроцессора, которые отображают эти регистры и значимые поля:

```
#define RTC_BASE (0x40002800)
#define RTC_CR (*(volatile uint32_t *)(RTC_BASE + 0x08))
#define RTC_ISR (*(volatile uint32_t *)(RTC_BASE + 0x0c))
#define RTC_WUTR (*(volatile uint32_t *)(RTC_BASE + 0x14))
#define RTC_WPR (*(volatile uint32_t *)(RTC_BASE + 0x24))

#define RTC_CR_WUP (0x03 << 21)
#define RTC_CR_WUTIE (1 << 14)
#define RTC_CR_WUTE (1 << 10)

#define RTC_ISR_WUTF (1 << 10)
#define RTC_ISR_WUTWF (1 << 2)
```

Процедура инициализации RTC для создания события пробуждения состоит из следующих шагов.

1. Включите тактовые импульсы для регистров конфигурации питания, если они еще не включены, чтобы активировать флаг `POW_CR_DPB` и начать настройку RTC:

   ```
   void rtc_init(void) {
     APB1_CLOCK_ER |= PWR_APB1_CLOCK_ER_VAL;
     POW_CR |= POW_CR_DPB;
   ```

2. Включите RTC, используя бит 15 в конфигурации резервной области в RCC:

   ```
     RCC_BACKUP |= RCC_BACKUP_RTCEN;
   ```

3. Включите резервный источник тактового сигнала, выбрав низкоскоростной внутренний генератор (LSI) или низкоскоростной внешний генератор (LSE), если он имеется.
4. В этом примере используются генератор LSI, поскольку генератор LSE отсутствует на тестовой платформе. Но внешние генераторы более точны и всегда предпочтительнее для надежного хронометража. После включения генератора процедура ждет, пока он не станет готов к работе, путем опроса бита в регистре состояния:

```
RCC_CSR |= RCC_CSR_LSION;
while (!(RCC_CSR & RCC_CSR_LSIRDY))
  ;
```

5. Выберите LSI в качестве источника для RTC:

```
RCC_BACKUP |= (RCC_BACKUP_RTCSEL_LSI <<
RCC_BACKUP_RTCSEL_SHIFT);
```

6. Включите генерацию прерываний и событий для линии 22 EXTI, связав событие с нарастающим фронтом:

```
EXTI_IMR |= (1 << 22);
EXTI_EMR |= (1 << 22);
EXTI_RTSR |= (1 << 22);
```

7. Разблокируйте запись в регистры RTC, записав последовательность разблокировки в RTC_WPR:

```
RTC_WPR = 0xCA;
RTC_WPR = 0x53;
```

8. Отключите RTC, чтобы разрешить запись в регистры конфигурации. Подождите, пока операция записи станет возможной, путем опроса RTC_ISR_WUTWF:

```
RTC_CR &= ~RTC_CR_WUTE;
DMB();
while (!(RTC_ISR & RTC_ISR_WUTWF))
  ;
```

9. Установите значение интервала до следующего события пробуждения. Частота LSI составляет 32 768 Гц, а делитель по умолчанию для регистра интервала пробуждения установлен на 16, поэтому каждая единица в RTC_WUTR представляет одну 2048-ю долю секунды. Чтобы установить интервал в 5 секунд, нужно использовать следующий код:

```
RTC_WUTR = (2048 * 5) - 1;
```

10. Включите событие пробуждения:

```
RTC_CR |= RTC_CR_WUP;
```

11. Сбросьте флаг пробуждения, который мог быть установлен при выходе из режима ожидания:

```
  RTC_ISR &= ~RTC_ISR_WUTF;
```

12. Для завершения последовательности записываем неверный байт в RTC_WPR. Таким образом, снова включается защита от записи в регистре RCC:

```
  RTC_WPR = 0xb0;
}
```

13. Чтобы включить RTC непосредственно перед переходом в режим ожидания, нужно выполнить следующую процедуру, которая гарантирует, что таймер активен и ведет отсчет, а генерация событий для события пробуждения активна:

```
void rtc_start(void)
{
  RTC_WPR = 0xCA;
  RTC_WPR = 0x53;
  RTC_CR |= RTC_CR_WUTIE |RTC_CR_WUTE;
  while (((RTC_ISR) & (RTC_ISR_WUTWF)))
    ;
  RTC_WPR = 0xb0;
}
```

Если только что показанная процедура вызывается перед переходом в ждущий режим, система снова включится, когда произойдет событие пробуждения, но она не возобновит выполнение с того места, где оно было приостановлено, как это происходит в других режимах с низким энергопотреблением. Вместо этого выполнение начинается с обработчика прерывания сброса в начале вектора прерывания. По этой причине в этом примере не требуется реализация для exit_lowpower_mode, а инструкция WFE, которая переводит систему в ждущий режим, никогда не вернется в тот же контекст выполнения. В итоге функция main для примера режима ожидания выглядит следующим образом:

```
void main(void) {
  int sleep = 0;
  clock_pll_on(0);
  led_setup();
  rtc_init();
  timer_init(cpu_freq, 1, 1000);
  while(1) {
    if (timer_elapsed) {
      WFE(); /* По событию таймера */
      led_toggle();
      timer_elapsed = 0;
    }
      if (tim2_ticks > 10) {
      sleep = 1;
```

```
      tim2_ticks = 0;
    }

    if (sleep) {
      enter_lowpower_mode();
      rtc_start();
      WFE(); /* Никогда не возвращается */
    }
    else
      WFI();
  }
}
```

8.3.4. Интервалы пробуждения

Важным аспектом, который следует учитывать при разработке стратегии с низким энергопотреблением, является длительность периода пробуждения, или, другими словами, сколько времени требуется системе для возврата в обычный режим работы после переключения в режим с низким энергопотреблением. Система реального времени может оставлять место для компромиссов между энергопотреблением и скоростью отклика, но важно понимать влияние операций пробуждения из различных режимов с низким энергопотреблением, чтобы предсказать задержку операций при наихудшем сценарии. Время пробуждения в значительной степени зависит от конструкции аппаратного обеспечения микроконтроллера и во многом определяется его архитектурой.

На нашей базовой платформе выход из спящего режима занимает небольшое количество циклов ЦП, но для режимов глубокого сна ситуация меняется. Выход из режима остановки занимает несколько микросекунд. Дальнейшие оптимизации, которые были активированы в режиме остановки, такие как изменение настройки регуляторов напряжения или отключение флеш-памяти, неизменно влияют на количество времени, затрачиваемого на возврат к нормальной работе. Сброс после режима ожидания приводит к еще более длительным интервалам пробуждения, порядка миллисекунд, поскольку система должна полностью перезагрузиться после события пробуждения, а ко времени выполнения кода запуска добавляются доли миллисекунд, необходимые ЦП для пробуждения.

При проектировании системы с низким энергопотреблением это время пробуждения необходимо учитывать и правильно измерять, особенно когда системе приходится иметь дело с ограничениями в реальном времени. Должен быть выбран оптимальный режим с низким энергопотреблением, который соответствует требованиям к временным характеристикам и профилю энергопотребления приложения, принимая во внимание накладные расходы, возникающие при выходе из режима с низким энергопотреблением, если система просыпается достаточно часто, чтобы этими интервалами нельзя было пренебречь.

Кроме того, если система спроектирована для работы в режимах с низким энергопотреблением, нам нужен надежный механизм для измерения мощ-

ности, потребляемой системой во время ее работы. В следующем разделе рассмотрен общий механизм отслеживания значений тока в тестируемой цепи для измерения влияния маломощных режимов работы микроконтроллера, а также для оценки всех введенных энергосберегающих оптимизаций.

8.4. Измерение мощности

Ток, потребляемый устройством, можно измерить в любой момент, подключив амперметр последовательно в цепь питания. Этот способ не показывает все колебания потребляемого тока в течение определенного интервала времени, поэтому часто бывает удобнее измерять значения падения напряжения на концах шунтирующего резистора с помощью осциллографа.

Шунтирующий резистор подключается последовательно с целевым устройством с любой стороны от источника питания (рис. 8.1). Его типичное сопротивление относительно невелико, в диапазоне нескольких ом, чтобы гарантировать, что падение напряжения остается низким, но все еще может быть измерено осциллографом.

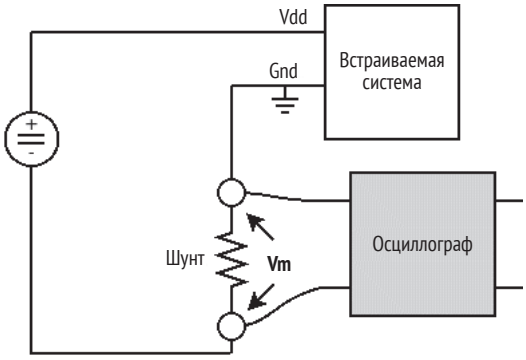

Рис. 8.1 ❖ Измерение тока с помощью осциллографа для измерения напряжения, падающего на шунтирующем резисторе

В последовательной цепи ток, протекающий через шунт, равен току, потребляемому устройством, поэтому напряжение на концах шунтирующего резистора изменяется пропорционально и синхронно с изменением тока.

8.4.1. Отладочные платы

Чтобы увидеть эффект от оптимизации энергопотребления, надо исключить из схемы электронные компоненты, не связанные с системой. Тестовая плата, например STM32F407DISCOVERY, имеет дополнительный микроконтроллер, который используется в качестве интерфейса отладки для хост-компьютера,

и питается от того же USB-разъема. Тем не менее комплекты для разработки часто предлагают способ измерения тока, исключающий постороннее оборудование, что позволяет нам должным образом оценить функции микроконтроллера с низким энергопотреблением.

На нашей базовой плате перемычку JP1 можно использовать для размыкания цепи между источником питания и схемой микроконтроллера. Заменив перемычку амперметром, подключенным к двум контактам, можно измерить ток, потребляемый непосредственно встраиваемой системой. Туда же можно подключить шунтирующий резистор для контроля тока, используя осциллограф для измерения напряжения на шунте.

Лаборатория, оснащенная надежными приборами для измерения потребляемой энергии, является хорошей отправной точкой для оценки маломощных проектов и помощи на этапе создания прототипа и проектирования оптимизации энергопотребления.

8.5. Проектирование встраиваемых приложений с низким энергопотреблением

В этом разделе предлагается несколько подходов к проектированию для достижения оптимального энергопотребления целевым устройством путем оценки энергопотребления всех компонентов и состояний разрабатываемой системы. Если есть возможность измерять ток, потребляемый целевым устройством, и в выбранной архитектуре и семействе микропроцессоров реализован режим пониженного энергопотребления, это позволяет сфокусировать усилия на нюансах разработки энергоэффективных встраиваемых приложений.

8.5.1. Замена циклов ожидания спящим режимом

Причина, по которой циклы ожидания очень популярны среди программистов-любителей, заключается в том, что их очень легко реализовать. Предположим, что системе необходимо дождаться перехода цифрового входа в состояние с низким логическим уровнем, и этот вход сопоставлен с определенным GPIO. Ожидание можно легко организовать с помощью следующей строки кода:

```
while((GPIOX_IDR & (1 << INPUT_PINX)) != 0)
  ;
```

Хотя этот способ отлично работает, как и ожидалось, он заставит ЦП войти в цикл выборки–декодирования–выполнения и непрерывно выполнять одни и те же инструкции, пока условие не станет ложным. Мощность, потребляемая микроконтроллером, в основном зависит от того, насколько быстро

работает ЦП. Более низкая частота соответствует меньшему количеству энергии, используемой на инструкцию. Выполнение инструкций в бесконечном цикле без переключения в режим пониженного энергопотребления приводит к максимальному потреблению энергии ЦП на измеримый период времени – в данном случае на все время, необходимое для изменения состояния логического входа.

Активный опрос линии – единственный выход, если прерывания не разрешены. Примеры, содержащиеся в этой книге, помогут вам выбрать правильный подход к обработке прерываний. Правильный способ обработки ожидания смены логического уровня предусматривает активацию линии прерывания, связанной со следующей операцией. В случае отслеживания линии GPIO можно использовать внешние триггеры прерывания, чтобы разбудить основной цикл при выполнении условия и переключиться в режим пониженного энергопотребления вместо цикла ожидания события.

Во многих других случаях искушения использовать циклы, подобные вышеупомянутому, можно было бы избежать, изучив другие способы работы с периферийным устройством, которое в настоящее время мешает системе перейти к следующему этапу выполнения. Современные контроллеры последовательных и сетевых каналов оснащены сигналами прерывания, и всегда есть способ определить событие периферийного устройства через внешнюю линию прерывания. Если же устройство действительно может функционировать только в режиме опроса, в крайнем случае частоту опроса можно уменьшить, связав действие с прерыванием по таймеру, что позволит проводить опрос несколько раз в секунду или даже реже, используя интервалы, которые больше соответствуют фактическому распорядку работы периферийного устройства. Выполнение операций по таймеру позволяет ЦП переключаться в режим пониженного энергопотребления в промежутках между ними.

Исключением из этого правила, неоднократно встречавшимся в этой главе, является ожидание флага готовности после активации системного компонента. Следующий код активирует внутренний низкоскоростной генератор, и он используется в примере режима ожидания перед входом в низкоскоростной режим. Регистр CSR опрашивается до тех пор, пока не заработает низкоскоростной генератор:

```
RCC_CSR |= RCC_CSR_LSION;
while (!(RCC_CSR & RCC_CSR_LSIRDY))
  ;
```

Операции, подобные этой, выполняемые на интегрированных периферийных устройствах в чипе микроконтроллера, имеют хорошо известную задержку в несколько тактов ЦП и, таким образом, не влияют на ограничения реального времени, поскольку максимальная задержка для подобных внутренних действий часто упоминается в документация на микроконтроллер. Ситуация меняется, если приходится опрашивать менее предсказуемый регистр, состояние которого и время реакции могут зависеть от внешних факторов, а в системе могут возникать длительные циклы занятости.

8.5.2. Глубокий сон во время длительных периодов бездействия

Как известно, режим ожидания позволяет «заморозить» систему с минимально возможным ультранизким энергопотреблением. Использование режима ожидания рекомендуется, когда техническими требованиями к конечному устройству жестко регламентировано сверхнизкое энергопотребление и выполняются следующие условия:

- существует жизнеспособная стратегия пробуждения, совместимая с текущим аппаратным обеспечением;
- система может восстановить выполнение, не полагаясь на свое предыдущее состояние, так как содержимое регистров ОЗУ и ЦП теряется, и система перезапускается из процедуры сброса при пробуждении.

Обычно более длительные периоды бездействия, когда, например, для пробуждения системы можно использовать часы реального времени, больше подходят для использования режима ожидания. Это относится к таким случаям, как считывание показаний датчиков и включение приводов через запрограммированные интервалы в течение дня, отслеживание времени и нескольких переменных состояния.

В большинстве других случаев режим остановки (stop mode) по-прежнему экономит достаточно много энергии, но обеспечивает более короткий интервал пробуждения. Еще одним важным преимуществом режима остановки является повышенная гибкость вариантов стратегии пробуждения. На самом деле любое событие на основе прерывания или настраиваемое событие может быть использовано для пробуждения системы из режима глубокого сна с низким энергопотреблением, поэтому оно больше подходит для состояний, в которых все еще остается некоторое взаимодействие с периферийными устройствами и интерфейсами, окружающими микроконтроллер.

8.5.3. Выбор тактовой частоты

Действительно ли платформа должна постоянно работать с максимальной вычислительной мощностью?

Производительность микроконтроллеров в настоящее время сравнима с характеристиками персональных компьютеров 20-летней давности, которые уже тогда были способны быстро выполнять вычисления и даже обрабатывать мультимедийный контент в реальном времени. Встроенным приложениям не всегда нужно, чтобы ЦП работал на полной частоте. Это особенно актуально при доступе к медленным периферийным устройствам, когда не имеет значения, насколько высокая тактовая частота у процессора и внутренней шины. Как в обычном рабочем режиме, так и в спящем режиме требуется гораздо меньше энергии, если тактовая частота уменьшается каждый раз, когда быстродействие процессора на самом деле не является узким местом системы.

Многие микроконтроллеры допускают уменьшение рабочей частоты ЦП и внутренних шин, что также обычно позволяет питать систему более низкими напряжениями. Как вы видели, изменение частоты тактовых импульсов можно выполнить «на ходу», тем самым придерживаясь компромисса между экономичностью и быстродействием. Но это означает, что все устройства, использующие тактовые импульсы в качестве эталона, должны быть перенастроены, поэтому изменение системной тактовой частоты связано с затратами времени на операции перенастройки и не должно применяться без необходимости. Удобный способ добавить изменения частоты в структуру системы – разделить параметры масштабирования частоты ЦП на фиксированные настраиваемые состояния и переключаться в требуемое состояние в соответствии с приоритетом быстродействия или экономии энергии.

8.5.4. Переключение профилей питания

Рассмотрим систему, подключенную к датчику, производящему и передающему данные через сетевой интерфейс. Датчик срабатывает, затем системе приходится ждать, пока он будет готов, что, как известно, занимает несколько секунд. Далее система считывает с датчика данные несколько раз подряд и отключается от него. Данные обрабатываются, шифруются и передаются с помощью сетевого устройства. Потом система бездействует в течение следующих нескольких часов, прежде чем повторить ту же операцию. Упрощенная модель конечного автомата показана на рис. 8.2.

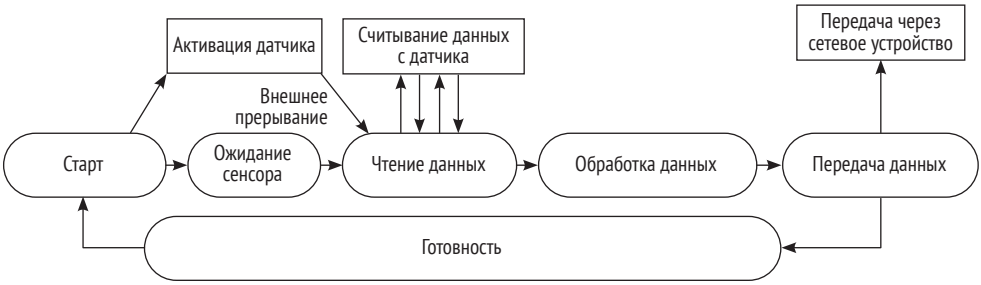

Рис. 8.2 ❖ Конечный автомат для гипотетической системы считывания данных с датчика

Поскольку между двумя последовательными циклами предусмотрен длительный интервал простоя, было бы хорошо перевести систему в режим ожидания на большую часть времени и запрограммировать пробуждение по сигналу RTC, чтобы система автоматически просыпалась вовремя для следующего сбора данных.

Другие, менее очевидные оптимизации возможны и для иных состояний. При получении данных от датчика полная вычислительная мощность ЦП, возможно, никогда не используется, так как система в основном занята обменом данными с датчиком или ожидает, возможно в спящем режиме, пока

не будет получено следующее значение. В этом случае можно обеспечить экономичный рабочий режим, при котором система работает на пониженной частоте, поэтому при чередовании между рабочим и спящим режимами оба они потребляют меньше энергии. Единственные этапы, на которых требуется более высокая производительность, – это когда данные обрабатываются, преобразуются и отправляются по сетевому устройству. В этом случае система будет оптимизирована для более быстрой работы и обработки данных в более короткие сроки. Сразу после активации датчика можно предусмотреть фазу остановки, если датчик способен отправить прерывание, чтобы разбудить систему, когда будут готовы данные.

После того как каждая фаза будет связана с соответствующим оптимальным режимом и выбранной рабочей частотой, можно внести изменения в проектную документацию, чтобы показать, как будет реализована оптимизация мощности в форме перехода состояний для достижения наилучшего сочетания производительности, экономии энергии и низкой задержки. На рис. 8.3 показаны переходы между фазами и связанные с ними режимы малой мощности.

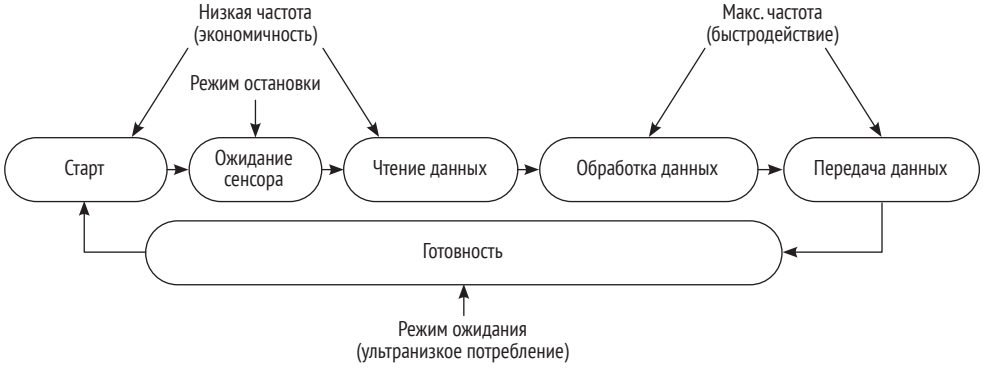

Рис. 8.3 ❖ Оптимизация энергопотребления для каждого рабочего состояния и режима ожидания

Настройка оптимальных энергетических профилей встраиваемой системы – это деликатный процесс, который сильно влияет на другие показатели производительности, приводя к задержкам и замедляя скорость выполнения. В большинстве случаев он заключается в поиске наилучшего компромисса между обеспечением приемлемой производительности при сохранении энергопотребления в желаемом диапазоне.

8.6. Заключение

Современные встраиваемые системы открывают множество возможностей для разработки маломощных и даже сверхмаломощных устройств. В этой

главе проанализированы различные профили энергопотребления, доступные для базового микроконтроллера, а также способы разработки, интеграции и оценки этих процедур для управления энергопотреблением во встраиваемых системах с приоритетом экономии энергии. Знание вариантов энергосбережения, доступных на целевом устройстве, и других методов энергосбережения является ключом к созданию долговечных и надежных устройств с питанием от батарей или источников энергии в окружающей среде.

Следующая глава посвящена знакомству с подключенными устройствами и описанию влияния работы с сетевыми протоколами и интерфейсами на архитектуру встраиваемой системы.

Глава 9

Распределенные системы и архитектура интернета вещей

Имея доступ к коммуникационным периферийным устройствам, таким как сетевые контроллеры и радиоинтерфейсы, микроконтроллеры могут устанавливать обмен данными с ближайшими устройствами и даже с удаленными серверами через интернет.

Набор встраиваемых систем, соединенных вместе и взаимодействующих друг с другом, можно рассматривать как одну автономную распределенную систему. Однородное межмашинное взаимодействие может быть реализовано с использованием нестандартных и даже проприетарных протоколов.

В зависимости от набора стандартных протоколов, которые она поддерживает, встраиваемая система может успешно взаимодействовать с разнородными удаленными системами. Использование популярных протоколов, которые стандартизированы или просто широко поддерживаются, дает возможность взаимодействовать как с ближайшими шлюзами, так и с удаленными облачными серверами через интернет.

Подключение небольших встраиваемых устройств к каналам межмашинной связи сделало возможным их взаимодействие с полномасштабными системами информационных технологий (information technology, IT). Своеобразное «столкновение двух миров» изменило интерпретацию распределенных систем: недорогие устройства с низким энергопотреблением теперь могут быть частью традиционных IT-структур, которые, в свою очередь, расширяют охват до локализованных и специализированных датчиков и исполнительных механизмов. Эта структура сегодня известна как *интернет вещей* (Internet of Things, IoT).

Этот технологический прорыв, который многие считают революционным, способен радикально изменить способ доступа к технологиям и процессы взаимодействия человека с машиной. К сожалению, аспектами безопасности связи IoT слишком часто пренебрегают, что приводит к неприятным инцидентам, которые могут поставить под угрозу конфиденциальность и целост-

ность передаваемых данных и позволить злоумышленникам получить контроль над удаленными устройствами.

В этой главе анализируются телекоммуникационные технологии и протоколы, которые можно интегрировать во встраиваемые системы. Они рассмотрены на всех уровнях встраиваемой системы, вплоть до интеграции в сети IoT.

Вы познакомитесь с сетевой моделью, начиная с физического уровня и возможных технологий для установления беспроводных или проводных соединений, вплоть до специализированных встроенных приложений, которые могут устанавливать безопасную связь с облачными службами, используя стандартные протоколы связи.

В частности, в этой главе рассмотрены следующие темы:

- сетевые интерфейсы;
- интернет-протоколы;
- TLS;
- протоколы приложений.

К концу этой главы вы будете иметь глубокое представление о возможностях современных микроконтроллеров в области IoT.

9.1. Технические требования

Предполагается, что вы знакомы с общими понятиями современных компьютерных сетей, хотя никакого предыдущего опыта работы с распределенными приложениями не требуется. Для получения более полной информации о сетевом программировании, которая имеет отношение к содержанию этой главы, рекомендуем в качестве дальнейшего чтения книгу *Hands On Network Programming with C*, L. Van Winkle; Packt Publishing (2019). В репозитории книги для этой главы нет конкретных примеров кода. Более полные примеры взаимодействия клиент-сервер через TCP и Transport Layer Security (TLS) можно найти в дистрибутиве исходного кода проектов с открытым исходным кодом, упомянутых далее в этой главе.

9.2. Сетевые интерфейсы

В состав встраиваемой системы часто входят один или несколько коммуникационных интерфейсов. Многие микроконтроллеры интегрируют часть *управления доступом к среде* (media access control, MAC) интерфейса Ethernet, поэтому подключение приемопередатчика физического уровня (PHY) позволит получить доступ к локальной сети. Некоторые устройства оснащены радиочастотными приемопередатчиками (трансиверами), работающими в фиксированных диапазонах частот и реализующими один или несколько протоколов для связи по беспроводным каналам связи. Часто используемые частоты для беспроводной связи – это полоса 2.4 ГГц, используемая Blue-

tooth и Wi-Fi 802.11, а также некоторые конкретные диапазоны частот ISM ниже 1 ГГц, которые зависят от местных законов и правил. К стандартным частотам ниже 1 ГГц относятся диапазон ISM 868 МГц в Европейском союзе и диапазон ISM 915 МГц в США. Трансиверы обычно предназначены для доступа к физическому уровню в соответствии с конкретными протоколами связи, регулирующими совместный доступ к физической среде между двумя или более устройствами. Хотя два интерфейса, обращающихся к одной и той же физической среде, могут иметь разные конфигурации, реализованная модель MAC должна соблюдать одинаковые спецификации на всех конечных точках, чтобы установить связь «точка–точка». Часть уровня MAC может быть реализована в самом устройстве, которое, в свою очередь, может использовать параллельный или последовательный интерфейс для передачи данных к микроконтроллеру и от него.

Производители оборудования могут распространять драйверы устройств для доступа к канальному уровню. Когда доступен полный исходный код, разработчику легче настроить доступ к среде, интегрировать функции связи устройства и адаптировать связь к любому стеку протоколов, поддерживаемому средой. Но многие драйверы устройств используют открытый исходный код лишь частично, что иногда ограничивает возможности интеграции с открытыми стандартами. Кроме того, интеграция стороннего проприетарного кода во встраиваемую систему негативно влияет на поддержку проекта и часто требует обходных путей для устранения известных проблем или включения функций, не предусмотренных производителем, и определенно влияет на модель безопасности системы.

Реализация драйверов устройств во встраиваемых системах для проводных или беспроводных сетевых интерфейсов включает в себя интеграцию соответствующего механизма управления доступом в коммуникационную логику и работу с конкретными функциями канала. Некоторые характеристики канала могут повлиять на структуру связи более высокого уровня, что повлияет на архитектуру всей распределенной системы. Наряду с надежным взаимодействием с механизмами MAC на этапе проектирования необходимо учитывать и оценивать такие аспекты, как скорость передачи данных, задержка и максимальный размер пакета, чтобы оценить необходимые для этого ресурсы системы.

В следующем разделе представлен обзор некоторых популярных сетевых интерфейсов в мире встраиваемых систем, которые обычно используются подключенными устройствами для связи с другими компонентами более широкой распределенной системы. Затем будут предложены некоторые критерии выбора наилучшей технологии для конкретной цели при проектировании коммуникационной инфраструктуры и протоколов.

9.2.1. MAC

Наиболее важные компоненты для реализации каналов связи через любые физические носители сгруппированы в логике MAC, ответственность за воплощение которой обычно делят между собой аппаратное и программное

обеспечения. Для организации межмашинного взаимодействия разработаны различные стандарты и технологии, но лишь немногие из них могут масштабироваться до уровня географически распределенной системы интернета вещей без промежуточных шлюзов, выполняющих преобразования протоколов.

Некоторые из стандартов напрямую заимствованы из мира «больших» IT и представляют собой адаптации существующих технологий TCP/IP, способных масштабироваться в соответствии с ограниченными ресурсами, доступными во встраиваемых системах. Другие стандарты специально разработаны для небольших встраиваемых систем, а взаимодействие с классической IT-инфраструктурой достигается за счет моделирования протоколов TCP/IP поверх беспроводных технологий с низким энергопотреблением. В обоих случаях исследования конвергенции продиктованы необходимостью более широкой интеграции небольших недорогих устройств с автономным питанием в службы IoT.

Не существует окончательного универсального решения для определения доступа к сети для встраиваемых систем. Различия в требованиях в отрасли встраиваемых систем стимулировали разработку специализированных протоколов и технологий MAC, как стандартизированных, так и проприетарных, предназначенных для удовлетворения потребностей в конкретных функциях или использования в конкретных устройствах.

В следующих разделах описаны некоторые из наиболее успешных технологий MAC для межмашинного взаимодействия с учетом аспектов, связанных с внедрением технологии и способов интеграции.

Ethernet

Несмотря на то что это выглядит немного странным в ситуации, когда размер всей системы сравним с разъемом RJ-45, Ethernet по-прежнему остается самым надежным и быстрым каналом связи, доступным для интеграции во встраиваемые системы.

Многие микроконтроллеры Cortex-M оснащены одним MAC-контроллером Ethernet, который должен быть интегрирован с внешним уровнем PHY. Другие протоколы канального уровня реализуют тот же механизм адресации канального уровня, состоящий из 14-байтовой преамбулы, присоединяемой к каждому передаваемому пакету, с указанием исходного и конечного адресов канала и типа полезной нагрузки, содержащейся в транспортируемом пакете. MAC-адреса перезаписываются каждый раз, когда пакет направляется стеком TCP/IP к Ethernet-подобному интерфейсу, чтобы они соответствовали следующему каналу, который пакет должен пройти на своем пути к конечному пункту назначения.

Драйверы устройств могут активировать фильтры для отбрасывания всего трафика, который не связан с хостом, что избавляет от необходимости обрабатывать стеком TCP/IP весь поток коммуникационных данных.

Wi-Fi

Среди всех возможных вариантов стандарта беспроводной связи 802.11 именно протокол Wi-Fi заслужил наибольшую популярность за его высокоско-

ростной канал с малой задержкой и максимально широкую топологическую совместимость, в том числе с персональными компьютерами и мобильными устройствами. Тем не менее требования к мощности приемопередатчика Wi-Fi иногда могут быть трудновыполнимы для маломощных устройств. Сложность протоколов и механизмов для регулирования доступа к среде заставляет использовать довольно сложное программное обеспечение, которое часто распространяется в виде двоичных файлов, и поэтому его невозможно отлаживать и поддерживать без участия производителей.

Wi-Fi обеспечивает большую пропускную способность и достаточно низкую задержку и может реализовывать аутентификацию и шифрование на уровне канала передачи данных.

Хотя технически возможно реализовать локальную mesh-сеть, настроив приемопередатчики Wi-Fi для работы в режиме ad hoc, встраиваемые системы с поддержкой технологии 802.11 в основном подключаются к существующей внешней инфраструктуре для взаимодействия с другими портативными устройствами и доступа к интернету.

На рынке доступно несколько встраиваемых недорогих платформ, оснащенных стеком TCP/IP и встроенной RTOS, которые можно использовать как автономную платформу или интегрировать в полные системы для подключения к беспроводной локальной сети в качестве клиента или точки доступа.

Низкоскоростные беспроводные персональные сети (LR-WPAN)

Ячеистые сети (часто говорят *mesh-сети*) сенсоров широко используют беспроводные технологии для установления связи на малых расстояниях. Стандарт 802.15.4 регулирует доступ к частотам 2.4 ГГц и ниже 1 ГГц для построения локальных сетей ограниченного радиуса действия с типичной максимальной скоростью передачи данных 250 Кбит/с, доступ к которым можно получить с помощью недорогих приемопередатчиков с низким энергопотреблением. Доступ к физической среде не основан на инфраструктуре и поддерживает обнаружение и разрешение конфликтов на уровне MAC с использованием системы маяков.

Каждый узел может быть адресован с использованием 2 байт, а специальный адрес `0xFFFF` зарезервирован для широковещательного трафика, который получают все доступные узлы в сети. Максимальный размер полезной нагрузки для кадров 802.15.4 фиксирован и составляет 127 байт, поэтому в них невозможно инкапсулировать полноразмерные IP-пакеты, маршрутизируемые из Ethernet или канала беспроводной локальной сети. Реализации сетевых протоколов, способные обмениваться данными через интерфейсы 802.15.4, либо зависят от приложения, либо не поддерживают IP-сети, либо предлагают механизмы фрагментации и сжатия для передачи и приема каждого пакета по нескольким беспроводным кадрам.

Несмотря на то что этот стандарт не разработан специально для интернета вещей и не совместим напрямую с классической IP-инфраструктурой, существует множество вариантов построения сетей на основе 802.15.4. Кроме

того, хотя стандарт определяет протокол MAC для обмена кадрами только между узлами, которые находятся в зоне видимости, разработано несколько технологий канального уровня, стандартных и нестандартных, для построения сетей поверх 802.15.4.

Промышленные расширения канального уровня LR-WPAN

Благодаря гибкости приемопередатчиков и способности передавать и получать необработанные кадры 802.15.4 не составляет особого труда реализовать сетевые протоколы для сетей LR-WPAN.

В эпоху до IoT отрасль автоматизации процессов первой внедрила технологию 802.15.4 и долгое время искала стандартный стек протоколов, обеспечивающий совместимость устройств разных производителей. Можно сказать, что стек протоколов Zigbee де-факто стал отраслевым стандартом для сетей 802.15.4, несмотря на его проприетарность, закрытый исходный код и лицензионные отчисления при коммерческом использовании. Параллельно с этим Международное общество автоматизации (International Society of Automation, ISA) разработало предложение по открытому стандарту ISA100.11a, целью которого является разработка рекомендаций по построению сетей на основе каналов 802.15.4, которые будут использоваться в процессах промышленной автоматизации. Другой протокол промышленной автоматизации, изначально разработанный консорциумом предприятий, а затем одобренный Международной электротехнической комиссией (International Electrotechnical Commission, IEC) в качестве стандарта промышленной автоматизации, – это WirelessHART.

Такие технологии, как Zigbee, ISA100.1 и WirelessHART, определяют весь стек протоколов поверх 802.15.4, включая определение сети и транспортные механизмы, предоставление настраиваемых механизмов адресации и моделей связи, а также экспорт API, который можно использовать для интеграции приложений. С точки зрения конструкции распределенной системы, возможность подключения к интернету для устройств в пользовательской сети без реализации стека IP требует, чтобы одно или несколько устройств действовали в качестве шлюза, перенаправляя и преобразовывая каждый пакет для пользовательского стека протоколов LR-WPAN. Но процедура преобразования нарушает сквозную семантику связи TCP/IP, влияя на различные аспекты связи, включая сквозную безопасность.

6LoWPAN

Протокол 6LoWPAN, описанный в RFC 4944, представляет собой стандартизированный IETF протокол связи 802.15.4, который может передавать пакеты IPv6, и является фактическим стандартом для IP-совместимых сетей LR-WPAN. 6LoWPAN позволяет встраиваемым системам получать доступ к интернету с использованием интерфейсов 802.15.4, если узлы реализуют сеть TCP/IP, а канальный уровень предоставляет механизмы для передачи и получения полноразмерных IP-пакетов с использованием коротких кадров LR-WPAN. Содержимое пакета фрагментируется и передается в последова-

тельных транспортных блоках, а сетевые и транспортные заголовки дополнительно сжимаются для уменьшения служебных данных при передаче.

В настоящее время нет IPv4-аналога стандарта 6LoWPAN; тем не менее IETF рассматривает предложения, использующие аналогичный подход, чтобы добавить устаревшее подключение IPv4 для встроенных узлов.

6LoWPAN является частью нескольких реализаций сетевого стека, а также частью недавней попытки создать промышленный альянс – группу Thread, целью которой является продвижение технологии ячеистой сети с низким энергопотреблением, полностью совместимой с IPv6 и основанной на протоколах открытого стандарта, предназначенных для IoT. Многие бесплатные стеки TCP/IP с открытым исходным кодом и встраиваемые операционные системы поддерживают 6LoWPAN и могут использовать приемопередатчики 802.15.4 как узлы инфраструктуры связи для построения IP-сетей на основе встроенных функций и протоколов.

К канальному уровню может быть дополнительно добавлена mesh-сеть для формирования прозрачного мостового механизма, где все кадры повторяются канальным уровнем в удаленных узлах mesh-сети до тех пор, пока не будет достигнут их пункт назначения.

Поскольку 6LoWPAN предоставляет инфраструктуру для построения топологии сети, к mesh-сетям можно подходить по-разному, используя протоколы уровня приложений для обновления таблиц маршрутизации на уровне IP. Эти механизмы, известные как *маршрутизируемые mesh-сети* (route-over mesh network), основаны на стандартизированных механизмах динамической маршрутизации и могут также применяться для расширения ячеистой сети по различным физическим каналам.

Bluetooth

Еще одна технология межмашинной связи, находящаяся в постоянном развитии, – это Bluetooth. Его физический уровень основан на радиоканале 2.4 ГГц для связи между хостом и устройством или построения инфраструктуры *локальной персональной сети* (personal area network, PAN), поддерживающей несколько протоколов, включая связь TCP/IP. Благодаря своему давно состоявшемуся успеху и, как следствие, широкому распространению на рынке персональных компьютеров и портативных устройств Bluetooth-соединение начало набирать популярность и на рынке встраиваемых микроконтроллеров, в основном из-за недавней эволюции стандарта в сторону более низкого энергопотребления.

Первоначально разработанная как беспроводная замена последовательной связи для устройств на близком расстоянии, классическая технология Bluetooth эволюционировала к поддержке интегрированных выделенных каналов, включая сетевые интерфейсы с поддержкой TCP/IP и выделенные каналы потоковой передачи аудио и видео.

Вариант стека протоколов с низким энергопотреблением, представленный в четвертой версии стандарта, разработан специально для встраиваемых сенсорных узлов и представляет новый набор услуг. Сенсорное устройство может экспортировать *общий профиль атрибута* (generic attribute profile,

GATT), к которому может получить доступ клиент (обычно хост-компьютер) для установления связи с устройством. Когда приемопередатчик целевого устройства неактивен, он потребляет небольшое количество энергии, при этом остается возможность обнаружить его атрибут и инициировать передачу GATT от клиента. В настоящее время Bluetooth в основном используется для связи ближнего радиуса; для доступа к сенсорным узлам с персональных компьютеров и портативных устройств; для обмена мультимедийным контентом с удаленными аудиоустройствами, такими как динамики, гарнитуры и автомобильные голосовые интерфейсы громкой связи; а также в некоторых приложениях для здравоохранения благодаря профилям, специально разработанным для этой цели.

Сети мобильной связи

Подключение удаленных устройств, вокруг которых нет стационарной инфраструктуры, стало возможным благодаря той же технологии, которая используется портативными устройствами для доступа в интернет через мобильные сети, такие как GSM/GPRS, 3G и LTE. Возрастающая сложность устройств широкополосной мобильной связи на фоне снижения их стоимости и энергопотребления ускорили интеграцию этого вида сетевой связи во встроенные устройства на базе микроконтроллеров. Мобильные сети изначально поддерживают протоколы TCP/IP и обеспечивают прямое подключение к интернету или, в некоторых случаях, к ограниченным сетям, предоставляемым инфраструктурой доступа.

Несмотря на то что полнопрофильные устройства доступа к широкополосной сети по-прежнему популярны на отдельных рынках, таких как автомобильный и железнодорожный, их функциональность избыточна для передачи небольшого объема информации с удаленных сенсорных устройств, в то время как более простые модемы для доступа к старым технологиям с узкой полосой пропускания постепенно исчезают с рынка.

Поэтому сложилась ситуация, когда технологии мобильных сетей развивались, ориентируясь на требования рынка мобильных телефонов, и в то же время архитекторы встраиваемых устройств искали новые технологии, которые лучше соответствуют потребностям распределенных систем IoT. Перспективные технологии, такие как мобильные сети 6G, лучше соответствуют целям рынка встраиваемых систем и эволюционируют в направлении маломощной, экономичной и дальней связи.

Сети дальней связи с низким энергопотреблением (LPWAN)

LWPAN (low-power wide area networks) – это семейство новых технологий, которые заполняют рыночный пробел в области экономичной и маломощной узкополосной связи на больших расстояниях. Что касается LR-WPAN, производители оборудования сформировали различные промышленные альянсы в стремлении завоевать рынок, а в некоторых случаях и установить стандартный стек протоколов для универсальных сетей LPWAN. Этот процесс привел к развитию здоровой конкуренции по характеристикам, стоимости и функциям энергосбережения.

Технологии LPWAN обычно основаны на физических каналах с частотой ниже 1 ГГц, но, в отличие от локальных каналов, используют другие стандарты радиосвязи, что позволяет увеличить радиус действия. Устройства могут связываться друг с другом по радиоканалу и, в некоторых случаях, использовать коллективную инфраструктуру для увеличения покрытия даже на расстоянии в тысячи километров, когда они находятся в пределах видимости базовой станции.

К наиболее заметным новым технологиям в этой области относятся:

- **LoRa/LoRaWAN:** основанная на запатентованных механизмах беспроводного радиодоступа и полностью проприетарном стеке протоколов, эта технология обеспечивает связь на большие расстояния с более высокой скоростью передачи данных по сравнению с аналогичными технологиями. Несмотря на то что она предлагает несколько интересных функций, таких как локальная связь между узлами при отсутствии инфраструктуры, закрытость протокола делает эту технологию менее привлекательной для рынка встраиваемых систем. Поэтому в будущем она вряд ли сохранит свое место в конкурентной борьбе за рынок LPWAN, в конечном итоге уступив место более открытым стандартам;
- **Sigfox:** для работы этой сверхузкополосной радиотехнологии требуется специальная инфраструктура, и у нее низкая скорость передачи данных на очень больших расстояниях. Регулируемый доступ к инфраструктуре позволяет ежедневно передавать с узла или на узел ограниченное количество байтов, а полезная нагрузка сообщений зафиксирована на уровне 12 байт. Хотя реализация физического уровня является проприетарной, стек протоколов распространяется в виде исходного кода. Тем не менее во многих странах соответствие протокола регулирующим нормам остается открытым вопросом, и этот фактор может негативно повлиять на распространение технологии во всем мире, несмотря на ее заметный успех на европейском рынке;
- **Weightless:** еще одна технология, основанная на ультраузкополосном соединении. Weightless представляет собой полностью открытый стандарт для LPWAN, работающий в субгигагерцовом диапазоне. Напоминая Sigfox с точки зрения диапазона и скорости, он обеспечивает улучшенную модель безопасности в качестве альтернативы классическим механизмам развертывания предварительно согласованных общих ключей, позволяя использовать беспроводные механизмы согласования ключей шифрования;
- **DASH7:** самая младшая из описанных здесь технологий основана на полностью открытом стандарте. Альянс DASH7 предоставляет исходный код всего облегченного стека протоколов, что упрощает интеграцию технологии во встраиваемые системы. Этот стек протоколов ориентирован на гибкость при проектировании распределенных систем благодаря множеству вариантов определения топологии сети.

Протоколы LPWAN несовместимы напрямую с IP и требуют, чтобы один из узлов в сети генерировал трафик TCP/IP на основе данных дальней связи, полученных от узлов. Спорадический характер сетевого трафика с низкой

скоростью передачи ограничивает область применения этой технологии, а при необходимости иметь доступ к удаленным узлам в интернете приходится включать в состав инфраструктуры специальные узлы, способные конвертировать и перенаправлять трафик.

9.2.2. Выбор подходящих сетевых интерфейсов

В зависимости от варианта использования каждая встраиваемая система может извлечь свою выгоду от применения технологий связи. Из-за узкой специализации некоторых встраиваемых устройств их конструкция может даже выходить за рамки стандартной классификации и использовать технологии, разработанные специально для конкретной задачи. В некоторых случаях беспроводная связь невозможна из-за особенностей распространения излучения в некоторых средах, а также когда среда не способна надежно передавать радиоволны, например под водой или через тело человека.

Подводные лодки могут общаться через специальные акустические приемопередатчики, используя звуковые волны. Для проводной связи доступны различные широко распространенные технологии. Связь по линиям электропередач позволяет повторно использовать имеющиеся провода для переоборудования старых устройств и обеспечивает подключение к локальной сети, расширяя шины Ethernet или последовательных интерфейсов с использованием высокочастотной модуляции, которая не влияет на первоначальное назначение используемых электросетей.

На самом деле встраиваемым устройствам доступен широкий спектр возможностей подключения. Оптимальный выбор всегда зависит от назначения устройства и ресурсов, доступных в системе для реализации протоколов и стандартов связи. При выборе технологии связи следует учитывать несколько аспектов:

- дальность связи;
- необходимая скорость передачи данных;
- совокупная стоимость владения (цена приемопередатчика, усилия по интеграции и стоимость обслуживания);
- ограничения физической среды, такие как любая задержка, вносимая приемопередатчиком;
- влияние радиочастотных помех на требования к конструкции оборудования;
- максимальный блок передаваемых данных;
- энергопотребление и энергетический след;
- поддерживаемые протоколы или стандарты для совместимости со сторонними системами;
- соответствие интернет-протоколам для интеграции в системы IoT;
- гибкость топологии, динамическая маршрутизация и возможности mesh-сети;
- модель безопасности;
- ресурсы системы, необходимые для реализации драйверов и протоколов;

- наличие открытых стандартов во избежание проблем с обслуживанием долгосрочных проектов.

Каждая технология для подключенных устройств предлагает свои взгляды на вышеупомянутые аспекты. Имеет значение также, была ли технология заимствована из другого контекста, такого как Ethernet или GSM/LTE, или разработана специально для систем со сверхнизким энергопотреблением, как в протоколах LR-WPAN и LWPAN.

Выбор подходящих каналов связи при проектировании распределенных систем подразумевает строгое взаимное соответствие между аппаратурой и программным обеспечением. Создание подключенных устройств вносит в разработку встраиваемых систем дополнительный уровень сложности, особенно если требуется низкое энергопотребление.

В следующем разделе основное внимание уделяется адаптации интернет-протоколов к использованию во встроенных устройствах для создания конечных точек сети, работающих в соответствии со стандартами и имеющих множество функций. Реализация стека TCP/IP может быть расширена и доработана в соответствии с требованиями распределенной системы IoT. Случаи, когда протоколы, отличные от IP, транслируются пограничным шлюзом для интеграции нестандартной связи в системы IoT, здесь не рассматриваются, поскольку они часто объединяют более крупные выделенные системы с несколькими сетевыми интерфейсами.

Как вы могли заметить, отрасль встраиваемых систем весьма специализирована и потому вынуждена работать буквально на грани имеющихся стандартов, но сейчас просматривается устойчивая тенденция к возвращению TCP/IP в качестве общепринятого стандарта для сетевой связи из-за растущего влияния существующей IT-инфраструктуры в распределенных системах, включая небольшие, маломощные, экономичные встраиваемые системы. Эта тенденция привела к появлению во встраиваемых системах стандартных функций безопасности, включая различные безопасные протоколы сквозной связи, такие как TLS и DTLS.

9.3. Интернет-протоколы

Стандартизированный в начале 1980-х годов *стек интернет-протоколов* (IP), в настоящее время чаще всего называемый TCP/IP, представляет собой семейство сетевых, транспортных и прикладных протоколов, обеспечивающих стандартную связь с использованием широкого спектра технологий и интерфейсов. Далее будет показано, как использовать эти стандартные протоколы во встраиваемых системах, и будут описаны интерфейсы, которые приложения используют для связи с удаленными конечными точками. Далее также показано, как взаимодействовать с различными уровнями стека, от сетевых интерфейсов до абстракции сокета для установления соединений с удаленным узлом.

9.3.1. Частные реализации стандартных протоколов

Проектирование распределенной связи с использованием нестандартных стеков протоколов почти никогда не стоит усилий, необходимых для «изобретения велосипеда». Стандарты TCP/IP были предметом обширных исследований на протяжении многих десятилетий и являются основными функциональными компонентами интернета, каким мы его знаем сегодня, объединяя миллиарды разнородных устройств. Оснащение встраиваемой системы поддержкой протокола TCP/IP больше не является сложной задачей, поскольку существует несколько готовых реализаций с открытым исходным кодом, которые легко интегрируются в небольшие встраиваемые системы, если они могут получить доступ к физическим каналам связи, обеспечивающим передачу данных между двумя (или больше) конечными точками.

Сокеты (socket) – это стандартный способ доступа к коммуникациям транспортного уровня из сетевых приложений. Модель сокетов Беркли, позже стандартизированная POSIX, охватывает стандарт именования функций и компонентов, а также поведение в операционной системе UNIX. Если стек TCP/IP интегрирован с операционной системой, планировщик может предоставить механизм для приостановки вызывающего объекта в ожидании определенного ввода, а API вызов сокета может быть реализован в соответствии со спецификациями POSIX. Но во встроенном приложении, выполняемом без операционной системы, синхронизация с сокетами выполняется с помощью обратных вызовов, чтобы соответствовать модели главного цикла, основанной на событиях. По этой причине разработка встроенных приложений, взаимодействующих с сетевыми протоколами, немного отличается с точки зрения API и классических парадигм программирования. В неблокирующем сетевом приложении внутри одного потока никакая операция не должна загружать ЦП во время ожидания событий, за исключением самой функции `main()` главного цикла. Вызовы функций сокетов не являются исключением, требуя механизма для инициирования операции, регистрации функции обратного вызова для обработки ее завершения, а затем немедленного возврата в основной цикл.

9.3.2. Стек TCP/IP

Современный стек TCP/IP, возможно, является наиболее фундаментальной частью распределенной встраиваемой системы. Надежность связи зависит от того, насколько точно реализованы стандартные протоколы и в какой мере безопасность служб, работающих на устройстве, может быть скомпрометирована дефектами, скрытыми в реализации стека TCP/IP, его интерфейсных драйверах и связующем коде абстракций сокетов.

Наиболее популярной библиотекой TCP/IP с открытым исходным кодом для встраиваемых устройств является lightweight IP (облегченный стек IP), более известная под сокращенным названием lwIP. Интегрированная со многими операционными системами реального времени и даже распространяемая производителями оборудования, lwIP обеспечивает сеть IPv4 и IPv6, связь через сокеты UDP и TCP, клиент DNS и DHCP, а также богатый набор протоколов прикладного уровня, которые можно интегрировать во встраиваемую систему, использующую всего несколько десятков килобайт памяти. Несмотря на то что библиотека предназначена для небольших микроконтроллеров, ресурсы, необходимые для полнофункционального стека lwIP, выходят за пределы возможностей для некоторых небольших устройств, включая большинство беспроводных датчиков со сверхнизким энергопотреблением.

Библиотека Micro IP, чаще называемая uIP, представляет собой минималистическую реализацию TCP/IP, основанную на необычной, но блестящей идее обработки одного буфера за раз. Отсутствие необходимости выделения нескольких буферов в памяти максимально снижает объем оперативной памяти, необходимой для связи TCP/IP, и сложность реализации TCP и других протоколов и, как результат, сводит к минимуму совокупный код всего стека. uIP не предназначена для масштабирования до более высокой скорости передачи данных или для реализации расширенных функций, но иногда это лучший компромисс для подключения узлов с очень ограниченными ресурсами, в основном к сетям LR-WPAN.

picoTCP – это свободно распространяемый программный стек TCP/IP с более поздней историей. Он использует те же ресурсы и списки функций, что и lwIP, но имеет другую модульную конструкцию и больше внимания уделяет протоколам IoT, обеспечивая динамическую маршрутизацию, IP-фильтрацию и возможности NAT. Благодаря встроенной поддержке 6LoWPAN через устройства 802.15.4 picoTCP можно использовать для создания mesh-сетей, используя либо реализацию mesh-сети в 6LoWPAN, либо более классический подход с протоколами динамической маршрутизации, такими как OLSR и AODV, предусмотренными в модулях.

Существуют и другие реализации как открытых, так и проприетарных стеков TCP/IP, которые могут быть интегрированы как в приложения на «голом железе», так и во встроенные операционные системы, часто предоставляя аналогичные API для интеграции интерфейсных драйверов и взаимодействия с системой для обеспечения связи через сокеты с приложениями более высокого уровня. Встроенный стек TCP/IP подключается к сетевым устройствам через драйвер устройства, обеспечивая функцию отправки кадров в сеть и возможность доставлять полученные пакеты с помощью входной функции потока, которую стек TCP/IP использует для приема пакета. Пакеты, которые обрабатываются стеком TCP/IP, могут потребовать асинхронных операций, поэтому приложение или ОС должны обеспечить периодический вызов функции цикла стека, чтобы он мог обрабатывать пакеты в буферах. Наконец, на транспортном уровне предоставляется интерфейс сокета, чтобы приложение могло создавать и использовать сокет для связи с удаленными конечными точками.

9.3.4. Драйверы сетевых устройств

Для взаимодействия с драйвером сетевого интерфейса стек TCP/IP предоставляет интерфейс своим нижним уровням, отправляя и получая буферы, содержащие кадры или пакеты. Если устройство поддерживает Ethernet-адрес канального уровня, стеки TCP/IP должны подключить дополнительный компонент для работы с кадрами Ethernet и активировать протоколы обнаружения соседей, чтобы найти MAC-адрес принимающего устройства, прежде чем инициировать IP-связь.

Библиотека lwIP предоставляет структуру `netif`, описывающую сетевой интерфейс, который должен быть выделен кодом драйвера, но затем автоматически инициализируется стеком с помощью функции `netif_add`:

```
struct *netif netif_add(struct netif *mynetif,
    struct ip_addr *ipaddr,
    struct ip_addr *netmask,
    struct ip_addr *gw, void *state,
    err_t (* init)(struct netif *netif),
    err_t (* input)(struct pbuf *p, struct netif *netif));
```

Аргументы `ipaddr`, `netmask` и `gw` можно использовать для настройки начальной конфигурации IPv4, доступной по ссылке, созданной через этот интерфейс. lwIP поддерживает один адрес IPv4 и три адреса IPv6 на интерфейс, но все они могут быть перенастроены на более позднем этапе путем доступа к относительным полям в структуре `netif`. IP-адрес можно настроить либо с помощью присвоения статического IP-адреса, либо с помощью механизма автоматического назначения, такого как согласование DHCP, либо путем получения локальных адресов канала.

Переменная `state` – это определяемый пользователем указатель, который может создавать связь между сетевым устройством и скрытым полем, доступ к которому можно получить с помощью указателя `netif->state` в коде драйвера.

Указатель функции, предоставленный в качестве аргумента `init`, вызывается во время инициализации стека с тем же указателем `netif`, и драйвер должен использовать его для инициализации остальных полей для устройства `netif`.

Указатель функции, переданный во входном аргументе, описывает внутреннее действие, которое должен выполнить стек, когда он получает пакет из сети. Если устройство обменивается данными с использованием кадров Ethernet, должна быть определена функция `ethernet_input`, чтобы указать, что потребуется дополнительная обработка кадра Ethernet перед синтаксическим анализом содержимого кадра и что сеть поддерживает протоколы обнаружения соседей для связывания IP-адресов с MAC-адресами перед передачей данных. Если вместо этого драйвер обрабатывает «голые» IP-пакеты, функция приема, которую необходимо связать с указателем, – это `ip_input`.

Инициализация драйвера устройства завершается в функции `init`, которая также должна присвоить значение другим важным полям в структуре `netif`:

- `hw_addr`: содержит MAC-адрес устройства Ethernet, если он поддерживается;
- `mtu`: максимальный размер блока передачи, разрешенный этим интерфейсом;
- `name/num`: идентификатор устройства в системе;
- `output`: этот указатель функции вызывается стеком для добавления пользовательского заголовка соединения к IP-пакету, готовому к передаче. Для устройств Ethernet он должен указывать на `etharp_output` для запуска механизмов обнаружения соседей;
- `link_output`: этот указатель функции вызывается стеком, когда буфер готов к передаче.

После того как связь была помечена как работающая с помощью вызова `netif_up`, драйвер устройства может вызвать функцию ввода при получении новых пакетов, а сам стек вызовет функции `output/link_output` для взаимодействия с драйвером.

Библиотека picoTCP экспортирует аналогичный интерфейс для реализации драйверов устройств, но поддерживает несколько адресов для каждого интерфейса, поэтому конфигурация IP отделена от драйверов устройств. Каждое устройство имеет список ассоциированных с ним каналов IPv4 и IPv6, каждый со своей собственной конфигурацией IP, для реализации многосетевых сервисов (подключенных к нескольким физическим линиям). Структура драйвера устройства в picoTCP должна начинаться с физической записи структуры `pico_device` в качестве первого поля. Таким образом, обе структуры указывают на один и тот же адрес, и устройство может поддерживать свои собственные частные поля в конце структуры `pico_device`. Для инициализации устройства в драйвере выделяется структура и вызывается `pico_device_init`:

```
int pico_device_init(struct pico_device *dev, const char *name, const uint8_t *mac);
```

Требуются три аргумента: предварительно выделенная структура устройства, имя, используемое для идентификации в системе, и MAC-адрес Ethernet, если он присутствует. Если MAC равен нулю, стек обходит протокол Ethernet, и весь трафик, обрабатываемый драйвером, представляет собой чистые IP-пакеты без расширений канального уровня. Драйвер должен реализовать функцию `send`, которая используется стеком для доставки кадров или пакетов, подлежащих передаче интерфейсом, а ввод управляется через функцию `pico_stack_recv`:

```
int32_t pico_stack_recv(struct pico_device *dev, uint8_t *buffer, uint32_t len);
```

Устройство снова передается в качестве аргумента, чтобы стек автоматически распознавал, получает ли интерфейс кадр Ethernet или необработанный IP-пакет без заголовков, и реагировал соответствующим образом. IP-адреса можно настроить с помощью `pico_ipv4_link_add` и `pico_ipv6_link_add`, а доступ к таблице маршрутизации для добавления шлюзов и статических маршрутов к определенным сетям осуществляется через его API.

9.3.5. Выполнение стека TCP/IP

Чтобы интегрировать сетевой стек, система, как правило, должна обладать определенной функциональностью, такой как хронометраж и управление динамической памятью. Все системные функции, требуемые стеком, связываются во время компиляции с использованием системного заголовка конфигурации, который соответствующим образом связывает функции и глобальные значения.

В зависимости от характеристик физических каналов и требуемой пропускной способности стек TCP/IP может быть очень требовательным с точки зрения используемой динамической памяти, выделяя место для новых входящих буферов до тех пор, пока верхние уровни не смогут их обработать. Назначение отдельных пулов памяти для операций стека TCP/IP помогает в некоторых проектах контролировать использование памяти стека путем установки пороговых значений и жестких ограничений без влияния на функциональность других компонентов в системе.

Большинство библиотек реализуют свои собственные внутренние таймеры, используя монотонный счетчик, предоставляемый системой и увеличиваемый независимо другим компонентом в системе. Значение отслеживаемого времени можно увеличивать с помощью прерывания `SysTick`, обеспечивающего приемлемую точность, с которой стек выполняет синхронизированные операции для протоколов. Для lwIP достаточно экспортировать глобальную переменную с именем `lwip_sys_now`, которая содержит время, прошедшее с момента загрузки, выраженное в миллисекундах. Библиотеке picoTCP необходимо экспортировать макрос или встроенную функцию с именем `PICO_TIME_MS`, возвращающую то же значение. Оба стека ожидают, что основной цикл приложения обеспечивает повторяющиеся точки входа, вызывая в основном API функцию, необходимую для управления внутренними состояниями системных протоколов.

Чтобы проверить, истек ли срок действия любого из ожидающих таймеров, система вызывает `sys_check_timeouts` в lwIP или `pico_stack_tick` в picoTCP из основного цикла событий или выделенного потока при работе в ОС. Интервал между последовательными вызовами может повлиять на точность таймера и, как правило, не должен превышать несколько миллисекунд, чтобы гарантировать, что сетевой стек реагирует на синхронизированные события.

Сетевые интерфейсы также необходимо опрашивать на вход из сети либо непрерывно, либо с помощью соответствующей обработки прерываний, реализованной в системе. Когда новые данные доступны, драйверы устройств выделяют новые буферы и инициируют обработку, вызывая входные функции канала передачи данных или сетевого уровня.

Типичное приложение на «голом железе», использующее lwIP, начинается с выполнения всех шагов инициализации стека и драйвера устройства. Структура сетевого интерфейса размещается в основном стеке функций и инициализируется статической конфигурацией IPv4. В следующем коде предполагается, что драйвер устройства экспортирует функцию с именем

driver_netdev_create, которая заполняет специфичные для интерфейса поля и обратные вызовы:

```
void main(void)
{
  struct netif netif;
  struct ip_addr ipaddr, gateway, netmask;
  IP4_ADDR(&ipaddr, 192,168,0,2);
  IP4_ADDR(&gw, 192,168,0,1);
  IP4_ADDR(&netmask, 255,255,255,0);

  lwip_init();
  netif_add(&netif, &ipaddr, &netmask, &gw, NULL,
  driver_netdev_create, ethernet_input);
  netif_set_default(&netif);
```

Затем сетевой интерфейс активируется в стеке TCP/IP:

```
netif_set_up(&netif);
```

Перед входом в основной цикл приложение инициализирует связь, создавая и настраивая сокеты и связывая обратные вызовы:

```
application_init_sockets();
```

Основной цикл ориентирован на то, что драйвер экспортирует функцию с именем driver_netdev_poll, в данном случае это функция, в которой драйвер вызывает ethernet_input всякий раз, когда принимается новый кадр. Наконец, вызывается sys_check_timeouts, чтобы lwIP мог отслеживать ожидающие таймеры:

```
  while (1) {
    /* опрос netif, передача пакетов в lwIP */
    driver_netdev_poll(&netif);
    sys_check_timeouts();
    WFI();
  }
}
```

Аналогичная процедура ожидается и в приложениях на «голом железе», использующих picoTCP. Инициализация драйвера устройства не зависит от стека, и ожидается, что драйвер вызовет pico_device_init для структуры pico_device, содержащейся в пользовательском типе driver_device, в качестве обязательного первого члена. Единственная функция, экспортируемая драйвером, – это driver_netdev_create, которая также ассоциирует свой конкретный указатель функции опроса сети, который будет вызываться pico_stack_tick. Стек ожидает обратного вызова pico_stack_recv всякий раз, когда функция poll драйвера обрабатывает новые входящие пакеты:

```
void main(void)
{
```

```
  struct driver_device dev;
  struct ip4 addr, netmask, gw, zero, any;
  pico_string_to_ipv4("192.168.0.2", &ipaddr.addr);
  pico_string_to_ipv4("255.255.255.0", &netmask.addr);
  pico_string_to_ipv4("192.168.0.1", &gw.addr);
  any.addr = 0;

  pico_stack_init();
  driver_netdev_create(&dev);
```

Конфигурирование адреса IPv4 выполняется путем доступа к API модуля IPv4. Приложения могут связать одну или несколько конфигураций IP-адресов, вызвав `pico_ipv4_link_add` и указав адрес и сетевую маску. Для достижения всех соседей в подсети через интерфейс автоматически создается маршрут в IP:

```
pico_ipv4_link_add(&dev, ipaddr, netmask);
```

Чтобы добавить маршрут по умолчанию, шлюз связывается с адресом `0.0.0.0` (указывающим любой хост) с метрикой `1`. Шлюз по умолчанию можно позже переопределить, определив более конкретные маршруты для других подсетей:

```
pico_ipv4_route_add(any, any, gw, 1, NULL);
```

Как и в предыдущем примере, приложение теперь может инициализировать свои сокеты и привязывать обратные вызовы, которые будут вызываться стеком при необходимости:

```
application_init_sockets();
```

Следующий простой главный цикл неоднократно вызывает `pico_stack_tick`, который циклически опрашивает все связанные сетевые интерфейсы и выполняет все ожидающие действия во всех модулях протокола:

```
  while (1)
  pico_stack_tick();
  WFI();
}
```

Все действия TCP/IP связаны с обратными вызовами сокетов, которые вызываются всякий раз, когда ожидается, что приложение отреагирует на сетевые события и события тайм-аута, а тайм-ауты автоматически устанавливаются стеком, когда это необходимо для управления внутренними состояниями отдельных протоколов. Интерфейс, предоставляемый для доступа к сокету при отсутствии операционной системы, как упоминалось ранее, основан на пользовательских обратных вызовах, зависящих от реализации конкретного стека. В следующем разделе показано, как использовать API неблокирующих сокетов в двух разных реализациях стека TCP/IP.

9.3.6. Использование сокетов

Интерфейс, предоставляемый lwIP для взаимодействия с сокетами на «голом железе», также называемый *API необработанных сокетов* (raw socket API), состоит из настраиваемых вызовов, каждый из которых определяет обратный вызов всякий раз, когда ожидается событие из стека. Когда происходит определенное событие, lwIP вызывает обратный вызов из функции основного цикла.

Описание сокета TCP в lwIP содержится в структуре блока управления протоколом `tcp_pcb`. Чтобы выделить новый блок управления для прослушивающего TCP-сокета, используется следующая функция:

```
struct tcp_pcb *tcp_new(void);
```

Чтобы принять TCP-соединение, TCP-сервер lwIP, работающий без ОС, сначала вызовет следующие функции:

```
err_t tcp_bind(struct tcp_pcb *pcb, ip_addr_t *ipaddr, u16_t port);
err_t tcp_listen(struct tcp_pcb *pcb);
```

Эти неблокирующие функции привязывают сокет к локальному адресу и переводят его в состояние прослушивания.

В этот момент приложение POSIX, использующее блокирующие сокеты, должно вызвать функцию `accept`, которая бесконечно будет ждать следующего входящего соединения на сокете. Приложение lwIP, работающее без ОС, вместо этого вызывает следующую функцию:

```
void tcp_accept(struct tcp_pcb *pcb,
  err_t (* accept)(void *arg, struct tcp_pcb *newpcb,
  err_t err)
);
```

Она просто указывает на то, что сервер готов принимать новые соединения и хочет, чтобы его вызывали по адресу функции `accept`, которая была передана в качестве параметра при установлении нового входящего соединения.

Используя тот же механизм, для получения следующего сегмента данных приложение вызывает функцию:

```
void tcp_recv(struct tcp_pcb *pcb,
  err_t (* recv)(void *arg, struct tcp_pcb *tpcb,
  struct pbuf *p, err_t err)
);
```

Этот вызов указывает стеку TCP/IP, что приложение готово к приему следующего сегмента по соединению TCP, и операция может быть выполнена, когда доступен новый буфер, поскольку стек вызывает реальную функцию `recv`, указанную в качестве аргумента при вызове `tcp_recv`.

Точно так же picoTCP связывает один обратный вызов с каждым объектом сокета. Обратный вызов – это общая точка для реагирования на любые события, связанные с сокетом, такие как новое входящее TCP-соединение,

новые данные для чтения в буфере сокета или завершение предыдущей операции записи.

Обратный вызов указывается при создании сокета:

```
struct pico_socket *pico_socket_open(uint16_t net,
    uint16_t proto,
void (*wakeup)(uint16_t ev,
    struct pico_socket *s));
```

Предыдущая функция создает новый объект сокета для использования в заданном контексте сети и транспортного протокола (аргументы `net` и `proto` соответственно) и реагирует на все события сокета, вызывая функцию `wakeup`, предоставляемую приложением. Используя этот механизм, picoTCP успешно обнаруживает полузакрытые соединения сокетов и другие события, которые не связаны конкретно с текущей выполняемой операцией, но могут произойти из-за изменения состояния в модели связи сокета.

Сервер сокетов TCP можно настроить на вновь созданном сокете, используя следующие функции:

```
int pico_socket_bind(struct pico_socket *s,
    void *local_addr,
    uint16_t *port);
int pico_socket_listen(struct pico_socket *s, int backlog);
```

На этом этапе приложение должно ожидать входящие соединения, не вызывая `accept`. Всякий раз, когда устанавливается новое входящее соединение, генерируется событие, вызывающее функцию `wakeup`, и приложение, наконец, может вызвать `accept` для создания нового объекта сокета, соответствующего входящему соединению:

```
struct pico_socket *pico_socket_accept(
    struct pico_socket *s,
    void *orig,
    uint16_t *local_port);
```

Первый аргумент, передаваемый обратному вызову `wakeup` picoTCP, – это битовая маска, указывающая типы событий, произошедших в сокете. События могут быть следующими:

- `EV_RD`: указывает на наличие данных для чтения в буфере входящих данных;
- `EV_CONN`: указывает, что новое соединение установлено после вызова `connect` или во время ожидания в состоянии прослушивания перед вызовом `accept`;
- `EV_CLOSE`: срабатывает, когда другая сторона соединения отправляет TCP-сегмент `FIN`, указывая на то, что передача завершена. Сокет находится в состоянии `CLOSE_WAIT`, что означает, что приложение все еще может отправлять данные до разрыва соединения;
- `EV_FIN`: указывает, что сокет был закрыт и больше не может использоваться после возврата из обратного вызова;
- `EV_ERR`: произошла ошибка.

Интерфейс обратного вызова, предоставляемый стеками TCP/IP, вначале может показаться немного запутанным, но при правильной реализации в приложении он представляет собой очень эффективный способ добиться более высокой пропускной способности.

Оба проанализированных нами стека TCP/IP способны предоставлять более стандартизированные API в сочетании с операционной системой, запуская основной цикл библиотеки TCP/IP в отдельном потоке и предоставляя доступ к сокетам с помощью системных вызовов.

Связь через сокет – это только один из API, предоставляемых стеками TCP/IP. Другие протоколы, реализованные стеком, предоставляют собственные сигнатуры функций; они описаны в руководствах обеих библиотек.

9.3.7. Протоколы без установления соединения

TCP широко применяется во всех случаях, когда для приложения имеет смысл парадигма, ориентированная на соединение. Его аналог без установления соединения, протокол UDP, в основном применяется для решения других задач, но в некоторых случаях он может покрыть все потребности небольшой встроенной системы с ограниченными ресурсами. Реализации TCP на самом деле велики и на некоторых платформах занимают значительную часть доступной флеш-памяти. Это связано со сложными внутренними механизмами TCP, которые требуют включения большого количества кода для управления повторными передачами, тайм-аутами и подтверждениями, организацией буферов и отслеживанием нескольких конечных автоматов для каждого сокета.

UDP, в свою очередь, довольно прост и применяет несколько преобразований к данным, передаваемым из интерфейса сокета в сеть и наоборот. Как правило, реализации UDP намного меньше по размеру, и из-за отсутствия требований к надежности им не нужно отслеживать порядок и пробелы в уже переданных или полученных данных, что также влияет на использование оперативной памяти во время выполнения. Когда характеристики сети позволяют это, использование UDP для избыточной передачи данных с низким трафиком часто является вполне жизнеспособным вариантом.

9.3.8. Mesh-сети и динамическая маршрутизация

Как упоминалось ранее, некоторые протоколы канального уровня способны реализовать низкоуровневые механизмы mesh-соединения, скрывающие сложность топологии для верхних уровней. Другой подход применяется, когда протокол канального уровня не реализует эту функцию или когда mesh-сеть может быть расширена на различные сетевые интерфейсы и, таким образом, должна реализовывать стандартный протокол, не зависящий от интерфейса. Каждая связь соединяет два устройства в прямой видимости, которые, в свою очередь, координируют свои действия для определения оп-

тимального сетевого пути по направлению к заданному сетевому узлу на основе обнаруженной топологии. Промежуточные узлы на пути настраиваются для направления трафика к месту назначения на основе информации, доступной в текущей топологии (рис. 9.1).

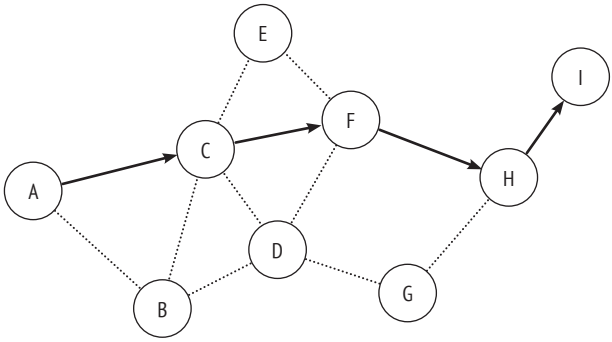

Рис. 9.1 ❖ Пример топологии mesh-сети
(узел A выбирает узел C для маршрутизации пакетов к I
после определения оптимального маршрута с четырьмя переходами)

В некоторых сценариях топология не является фиксированной, а развивается, когда узлы на пути становятся недоступными или меняют свое местоположение, что приводит к изменению видимости соседних узлов. Mesh-сети с нестатической топологией называются *мобильными одноранговыми сетями* (mobile ad-hoc network, MANET). Механизмы динамической маршрутизации, применяемые в сети MANET, должны иметь возможность реагировать на изменения топологии и соответствующим образом обновлять свои маршруты, поскольку сеть находится в постоянном развитии.

Механизмы маршрутизации mesh-сети реализованы в стеке TCP/IP, поскольку они должны иметь возможность реконфигурировать таблицу IP-маршрутизации во время выполнения и получать доступ к связи через сокеты. Mesh-сети, основанные на динамической IP-маршрутизации, используют разные протоколы, которые можно разделить на две категории:

- **проактивные протоколы динамической маршрутизации:** каждый сетевой узел отправляет широковещательное сообщение, чтобы объявить о своем присутствии в сети, а другие узлы могут обнаруживать присутствие соседа, читая сообщения и передавая список соседей соседям. Mesh-сеть готова к использованию в любое время и требует фиксированного времени на реконфигурацию при изменении топологии;
- **реактивные протоколы динамической маршрутизации:** узлы могут бездействовать, когда у них нет данных для обмена, и тогда путь настраивается за счет опроса каждого соседа с запросом маршрута к месту назначения. Затем сообщение повторяется, увеличивая значение счетчика отслеживания переходов, пока оно не достигнет пункта назначения, после чего, используя ответ, сеть может определить путь, запрошенный отправителем. Эти механизмы подразумевают, что ди-

намические маршруты формируются по запросу, поэтому первые сообщения могут иметь дополнительную задержку; с другой стороны, они требуют меньше энергии и могут быстрее реагировать на изменения топологии.

К первой группе относятся следующие наиболее популярные протоколы:

- **оптимизированная маршрутизация на основе состояния канала** (optimized link-state routing, OLSR), стандартизированная IETF в RFC3626 и RFC7181;
- **лучший подход к мобильной одноранговой сети** (better approach to mobile ad hoc networking, B.A.T.M.A.N.);
- **Babel** (IETF RFC6126);
- **вектор дальности последовательности до пункта назначения** (destination sequence distance vector, DSDV).

Реактивные протоколы маршрутизации по запросу, стандартизированные IETF, следующие:

- **вектор расстояния одноранговой связи по запросу** (ad-hoc, on-demand, distance vector, AODV), RFC3561;
- **динамическая маршрутизация источника** (dynamic source routing, RFC4728).

Выбор протокола маршрутизации опять же зависит от требований mesh-сети, которую необходимо построить. Реактивные протоколы по запросу лучше всего подходят для сетей со спорадическими данными и узлами с батарейным питанием, где допустимо более длительное время реакции протокола маршрутизации. И наоборот, непрерывно работающим встраиваемым системам больше подходят механизмы проактивной маршрутизации, которые гарантируют, что таблицы маршрутизации всегда обновлены до последнего известного состояния сети, и каждый узел всегда знает наилучший маршрут к каждому возможному пункту назначения, но в то же время требуют регулярной передачи по сети широковещательных пакетов для постоянного обновления статуса сетевых узлов и их соседей.

Протокол picoTCP, который был разработан как воплощение передовых технологий маршрутизации специально для устройств IoT, поддерживает один механизм mesh-сети нижнего уровня на канальном уровне 6LoWPAN и два протокола маршрутизации, а именно OLSR (реактивный) и AODV (проактивный), предоставляя более широкий выбор для интеграции связи TCP/IP в мобильные одноранговые сети. Например, чтобы включить OLSR, достаточно скомпилировать стек с поддержкой OLSR, и служба демона OLSR будет автоматически включена и запущена в основном цикле стека TCP/IP. Все устройства, которые должны участвовать в определении mesh-сети, должны быть добавлены вызовом `pico_olsr_add`:

```
pico_olsr_add(struct pico_device *dev);
```

Сеть AODV можно включить аналогичным образом, а интерфейсы добавляются с помощью функции `pico_aodv_add`:

```
pico_aodv_add(struct pico_devices *dev);
```

В обоих случаях службы будут работать прозрачно для пользователя и изменять таблицу маршрутизации каждый раз, когда в сети обнаруживается новый узел в случае OLSR, или каждый раз, когда запрашивается связь с удаленным узлом и до него выстраивается маршрут по запросу. Узлы, которые не находятся в прямой видимости, указывают шлюз первого перехода, который гарантирует, что узел назначения будет достигнут, используя показатель количества переходов, чтобы при обнаружении нового, более короткого пункта назначения маршрут заменялся и связь продолжалась, в идеале, без сбоев, вызванных заменой маршрута.

Протоколы маршрутизации, такие как OLSR, могут учитывать другие параметры, а не количество переходов, при расчете наилучшего пути к заданному месту назначения в mesh-сети. Например, при расчете наилучшего пути можно использовать информацию о качестве беспроводной связи, такую как отношение сигнал–шум или уровень принимаемого сигнала. Это позволяет нам выбирать маршруты на основе нескольких параметров и всегда использовать наилучший доступный вариант с точки зрения беспроводного сигнала.

Стратегии mesh-сетей с маршрутизацией не предусматривают механизмов пересылки широковещательных пакетов, которые должны повторяться протоколом канального уровня, чтобы достичь всех узлов в сети. Но известно, что реализация такого механизма может легко вызвать эффект пинг-понга, когда один пакет передается туда-обратно между двумя или более узлами, поэтому механизмы широковещательной пересылки, реализованные на канальном уровне, должны избегать повторной передачи одного и того же кадра дважды, отслеживая несколько последних кадров, переданных таким образом.

Системы IoT в реальном мире нуждаются в обеспечении безопасности передаваемых данных. Меры безопасности, направленные на конфиденциальность передаваемых данных, включают шифрование (но не ограничиваются им).

Внедрение стандартных протоколов безопасности гарантирует взаимодействие между разнородными компонентами в сети (например, между устройством и удаленным сервером) сквозным образом и с опорой на программные решения, полностью совместимые с протоколами, используемыми в обычном мире IT. В следующем разделе рассматривается безопасность транспортного уровня и ее перспективы.

9.4. TLS

Протоколы канального уровня часто предусматривают некоторые базовые механизмы безопасности, гарантирующие аутентификацию клиента, подключающегося к определенной сети, и шифрование данных с использованием симметричных ключей, таких как AES. В большинстве случаев аутентификации на канальном уровне достаточно, чтобы гарантировать начальный уровень безопасности. Тем не менее общие ключи, часто используемые в се-

тевых стеках LR-WPAN, потенциально уязвимы для нескольких видов атак, а использование общего ключа позволит злоумышленнику расшифровать любой трафик, который ранее был перехвачен в том же канале, если ключ был скомпрометирован. В других сценариях одного только шифрования недостаточно, чтобы гарантировать, что другая конечная точка является тем, за кого себя выдает, или что поток данных не был изменен во время передачи.

Устройство, которое является участником системы IoT, должно обеспечивать более высокий уровень безопасности, особенно во встроенных устройствах, которые никак не защищают память и где любая уязвимость означает, что злоумышленники могут перехватить контроль над устройством и получить всю конфиденциальную информацию, такую как закрытые ключи, используемые для аутентификации и шифрования при обмене данными с удаленными системами. TLS – это набор криптографических протоколов, предназначенных для обеспечения безопасной связи через стандартные сокеты TCP/IP. Обязанности этого компонента в основном сосредоточены на трех ключевых требованиях к безопасной связи в распределенных системах:

- **конфиденциальность** связи между участниками за счет использования симметричной криптографии. TLS определяет криптографические методы, направленные на создание одноразовых симметричных ключей, которые теряют свою актуальность в конце сеанса, для которого они были созданы;
- **аутентификация** участников обмена данными, с использованием криптографии с открытым ключом для подписи и проверки полезной нагрузки запроса. Благодаря свойствам асимметричного шифрования только тот участник, который владеет секретным закрытым ключом, может подписать полезную нагрузку, в то время как любой может проверить подлинность подписи с помощью открытого ключа, соответствующего закрытому ключу, которым подписано сообщение;
- **целостность** связи с использованием дайджестов сообщений, которые подтверждают, что сообщение не было изменено на своем пути.

На рынке встраиваемых систем доступно несколько реализаций набора протоколов с открытым исходным кодом, позволяющих использовать стандартные криптографические алгоритмы и стратегии для безопасного обмена данными через сокеты.

Примечание

По соображениям безопасности следует максимально избегать проприетарных реализаций компонентов безопасности с закрытым исходным кодом, потому что проблемы безопасности гораздо сложнее отследить в закрытой системе, и приходится слепо доверять поставщику решений.

Одна из наиболее полных и современных реализаций предоставляется бесплатной библиотекой программного обеспечения с открытым исходным кодом wolfSSL. Библиотека предлагает последнюю стандартную версию как TLS, так и DTLS и предназначена для повышения производительности и надежности небольших встроенных систем, включая поддержку аппаратных

ускорителей и генераторов случайных чисел для многих встроенных платформ, предназначенных для обеспечения безопасности системы.

Библиотека wolfSSL реализует криптографические примитивы в своей основной библиотеке (wolfCrypt) и группирует их в наборы шифров, используемые сокетами TLS, которые можно легко интегрировать как в сетевые приложения на «голом железе», так и в любую встроенную операционную систему, предоставляющую API связи транспортного сокета. Эти криптографические примитивы оптимизированы для встроенных устройств и используют ассемблерный код для наиболее важных операций для достижения наилучшей производительности.

Основное преимущество библиотеки TLS/SSL, разработанной для микроконтроллеров, заключается в том, что она реализует те же протоколы, что и любой ПК или сервер в интернете, но с меньшим размером кода и тщательно отслеживает использование ресурсов, таких как оперативная память, при выполнении наиболее ресурсоемких криптографических операций.

Использование библиотеки TLS с поддержкой передовых алгоритмов шифрования обеспечивает идеальную интеграцию с мерами безопасности, реализованными в классических компонентах IT-инфраструктуры сети IoT. На стороне облака сервисы, предназначенные для доступа к удаленным встраиваемым системам, должны допускать использование более эффективных наборов шифров на основе эллиптических кривых, поскольку классическое шифрование с открытым ключом на основе RSA требует более громоздких ключей и сложных вычислений для достижения того же уровня безопасности. Новые стандарты шифрования на основе открытых ключей, такие как Curve22519, добавленные в спецификацию TLS 1.3, обеспечивают более эффективную обработку ключей для систем с меньшим количеством ресурсов при сохранении того же уровня безопасности, что и у старых алгоритмов. При выборе подходящего набора криптографических алгоритмов для связи TLS между разнородными системами необходимо учитывать время вычислений операций, выполняемых на устройстве, таких как шифрование, генерация сеансового ключа, подписывание полезной нагрузки и проверка.

9.4.1. Защита связи через сокет

Библиотека wolfSSL поддерживает различные операционные системы и способна адаптироваться к конкретным конфигурациям памяти и интерфейсам сокетов, а также может быть интегрирована в систему без ОС с любым совместимым стеком TCP/IP или легко адаптирована благодаря универсальному интерфейсу ввода/вывода (I/O) на основе обратного вызова.

В любом случае, будь то на «голое железо» или ОС, приложение должно поддерживать доступ к *уровню защищенных сокетов* (secure socket layer, SSL) для связи с удаленной системой, в то время как библиотека отвечает за обеспечение абстракции для безопасного канала связи через транспортный уровень. Чтобы интегрировать сеансы TLS поверх существующей реализации TCP/IP на «голом железе», wolfSSL можно настроить для работы в неблокирующем режиме, опрашивая систему на наличие новых пакетов, полученных

через сокет, которые должны обрабатываться на уровне TLS. Приложение инициирует TCP-соединение, как обычно, либо подключаясь к удаленному сокету в режиме клиента, либо принимая новые соединения из локального прослушивающего сокета. После того как соединение установлено, wolfSSL назначает ему контекст, когда приложение вызывает `wolfSSL_accept` или `wolfSSL_connect` в режиме сервера или в режиме клиента, соответственно, чтобы инициировать «рукопожатие» TLS с удаленной системой. Последующая передача данных происходит с использованием функций `wolfSSL_read` и `wolfSSL_write` вместо обычных функций чтения/записи сокета, предоставляемых стеком TCP/IP, чтобы поток мог обрабатываться дополнительным SSL, созданным библиотекой TLS верхнего уровня.

Следующий пример иллюстрирует использование wolfSSL для создания сокета TLS поверх TCP-соединения. Подход к созданию сокета DTLS, эквивалента TLS для сокета без установления соединения, поверх UDP очень похож и по-прежнему использует ту же парадигму подключения/согласования, что и TLS, несмотря на то что UDP обычно применяется в одноранговой сети, которая не делает четкого различия между клиентом и сервером, как это делает TCP. Дополнительную информацию о создании безопасных сокетов DTLS без соединения можно найти в руководстве пользователя wolfSSL (https://www.wolfssl.com/documentation/manuals/wolfssl/index.html).

В нашем простом примере перед доступом к любому API библиотека сначала инициализируется с помощью `wolfSSL_Init`. Единственное обязательное требование заключается в инициализации и создании новых объектов, которые обычно называют *контекстами*. Один контекст реализует один конкретный метод (сервер TLS v. 1.2 в этом примере) и будет связан с одним или несколькими существующими сокетами через другую абстракцию, называемую SSL, которая в случае реализации wolfSSL представлена переменной типа `WOLFSSL`. Несколько объектов SSL, сгенерированных из одного и того же контекста, совместно используют один и тот же набор криптографических ключей и функций обратного вызова ввода-вывода, которые wolfSSL может использовать для запроса входящих данных в системе или передачи обработанных данных через сокет:

```
wolfSSL_Init();
wolfSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method());
wolfSSL_SetIORecv(ctx, wolfssl_recv_cb);
wolfSSL_SetIOSend(ctx, wolfssl_send_cb);
```

В системе реализованы два обратных вызова для доступа к сокетам в стеке TCP/IP с помощью системозависимого API сокетов TCP. Предположим, например, что пользовательская реализация TCP предоставляет функции чтения и записи `tcp_socket_write` и `tcp_socket_read` в контексте «голого железа», и эти функции возвращают `0`, когда не предпринимается никаких действий, поскольку стек TCP/IP занят или не готов к обработке буферов. Обратный вызов `wolfssl_send_cb` возвращает размер обработанных данных в случае успеха или специальное значение `WOLFSSL_CBIO_ERR_WANT_WRITE`, которое указывает, что операция ввода-вывода не может быть завершена без блокировки:

```
int wolfssl_send_cb(WOLFSSL* ssl, char *buf, int sz, void *sk_
ctx)
{
  tcp_ip_socket *sk = (tcp_ip_socket *)sk_ctx;
  int ret = tcp_socket_write(sk, buf, sz);
  if (ret > 0)
    return ret;
  else
    return WOLFSSL_CBIO_ERR_WANT_WRITE;
}
```

Обратный вызов чтения будет использовать специальное значение `WOLFSSL_CBIO_ERR_WANT_READ`, указывающее, что данные из стека недоступны для обработки:

```
int wolfssl_recv_cb(WOLFSSL *ssl, char *buf, int sz, void *sk_ctx)
{
  tcp_ip_socket *sk = (tcp_ip_socket *)sk_ctx;
  int ret = tcp_socket_read(sk, buf, sz);
  if (ret > 0)
    return ret;
  else
    return WOLFSSL_CBIO_ERR_WANT_READ;
}
```

Для наиболее часто используемых операционных систем и API стека TCP/IP библиотека wolfSSL уже предоставляет обратные вызовы ввода-вывода по умолчанию, поэтому, если вы правильно настроите конфигурацию, реализация пользовательских функций обратного вызова не потребуется.

Перед установлением любого соединения объекту `wolfSSL_CTX`, ассоциированному с объектами SSL для каждого соединения, необходимо предоставить набор сертификатов и ключей. В более сложной системе сертификаты и ключи хранятся в файловой системе, и к ним можно получить доступ, когда библиотека wolfSSL настроена на использование файловых операций. Во встраиваемых системах, где файловые хранилища часто не поддерживаются, сертификаты и ключи могут храниться в памяти и загружаться в контекст с помощью указателей на их расположение в памяти:

```
wolfSSL_CTX_use_certificate_buffer(ctx, certificate, len, SSL_FILETYPE_ASN1);
wolfSSL_CTX_use_PrivateKey_buffer(ctx, key, len,SSL_FILETYPE_ASN1);
```

Контекст сокета, передаваемый в обратные вызовы, назначается после установления несущего TCP-соединения. В случае сервера это может быть сделано контекстно для функции `accept`, в то время как клиент может связать сокет с конкретным контекстом SSL после успешного завершения функции `connect`. Для принятия соединения SSL на стороне сервера приложение должно вызвать `wolfSSL_accept`, чтобы согласование SSL могло быть завершено до фактической передачи данных. Процедура принятия SSL должна следовать за вызовом `accept` сокета после того, как указатель на объект сокета TCP/IP ассоциирован как контекст в объекте SSL, и будет использоваться в качестве аргумента `sk_ctx` для обратных вызовов, связанных с этим сокетом:

```
tcp_ip_socket new_sk = accept(listen_sk, origin);
WOLFSSL ssl = wolfSSL_new(ctx);

if (new_sk) {
  wolfSSL_SetIOReadCtx(ssl, new_sk);
  wolfSSL_SetIOWriteCtx(ssl, new_sk);
```

`wolfSSL_accept` вызывается после создания контекста сокета, потому что механизму `accept` может уже потребоваться вызов базового стека для прохождения через его состояния:

```
int ret = wolfSSL_accept(ssl);
```

Если согласование сеанса SSL прошло успешно, `wolfSSL_accept` возвращает специальное значение `WOLFSSL_SUCCESS`, и с этого момента безопасный сокет готов для связи через функции `wolfSSL_read` и `wolfSSL_write`. При работе без ОС `wolfSSL_read` и `wolfSSL_write` должны использоваться в неблокирующем режиме. Для этого необходимо установить соответствующий флаг во время выполнения в объекте сеанса SSL:

```
wolfSSL_set_using_nonblock(ssl, 1);
```

Использование неблокирующего ввода-вывода для функций wolfSSL гарантирует сохранение управляемой событиями модели основного цикла, ранее описанной для транспортных сокетов, поскольку вызов библиотечных функций никогда не останавливает систему. Функции API в wolfSSL предназначены для немедленного возврата определенных значений (таких как `WANT_WRITE` и `WANT_READ`), чтобы указать, что операция выполняется, а соответствующая функция (например, `wolfSSL_accept` в данном случае) должна быть вызвана снова позже, когда появятся новые доступные данные из базового сокета TCP.

После того как связь между конечными точками транспортного канала защищена, можно обмениваться данными с использованием безопасных соединений через сокеты. Далее следует обзор некоторых наиболее распространенных прикладных протоколов, используемых системами IoT.

9.5. Протоколы приложений

Чтобы иметь возможность взаимодействовать с удаленными устройствами и облачными серверами в распределенной архитектуре, встраиваемые системы должны реализовывать стандартные протоколы, совместимые с существующей инфраструктурой. Два наиболее распространенных подхода, применяемых при проектировании удаленных сервисов:

- веб-сервисы;
- протоколы сообщений.

Первый подход – это в основном классический клиент-серверный обмен данными на основе *передачи состояния представления* (Representational State Transfer, REST), популярный в веб-службах, доступ к которым получают через персональные компьютеры или портативные устройства. Веб-службы не требуют адаптации, в частности на стороне облака, для поддержки встроенных систем, за исключением выбора шифров, подходящих для встраиваемых систем, как описано в разделе 9.4.1. Но модель связи «запрос–ответ» накладывает некоторые ограничения на разработку распределенных приложений. Протокол HTTP может быть обновлен в соответствии с общим соглашением о двух конечных точках HTTP и поддерживает протокол верхнего уровня WebSocket, обеспечивающий абстракцию симметричного двунаправленного канала поверх служб HTTP.

Протоколы сообщений – это другой подход, который лучше отражает функции системы со встроенными датчиками или исполнительными механизмами, где обмен информацией осуществляется с использованием коротких двоичных сообщений, которые могут передаваться промежуточными агентами и собираться или распространяться с серверных узлов. Протоколы сообщений являются предпочтительным выбором, когда сеть включает в себя небольшие узлы из-за более простого представления данных, в отличие от веб-служб, которые в основном основаны на человекочитаемых строках и предъявляют гораздо более высокие требования к размеру транспортных пакетов и памяти устройства, которое вынуждено обрабатывать строки ASCII.

В обоих случаях для сквозного шифрования и надежной идентификации участников требуется поддержка TLS на уровне инфраструктуры и устройств. Аутентификация открытым текстом и шифрование с общим ключом являются устаревшими методами и поэтому не должны быть частью стратегии безопасности современных распределенных систем.

9.5.1. Протоколы сообщений

Коммуникационные протоколы на основе сообщений не являются новинкой в программном обеспечении для компьютерных сетей, но особенно хорошо они вписались в распределенные системы IoT, особенно в сценариях, где модель на основе сообщений «один ко многим» позволяет обращаться ко многим устройствам одновременно и устанавливать двустороннюю связь, или несколько устройств из разных мест могут взаимодействовать друг с другом, используя внешний сервер, который выступает посредником. Отсутствие общепринятых стандартов в этой области привело к появлению нескольких различных моделей, каждая из которых имеет собственный API и определение сетевого протокола.

Но некоторые открытые стандарты, в частности, были разработаны для реализации безопасных распределенных архитектур обмена сообщениями, специально адаптированных для систем с ограниченными ресурсами и сетей

с ограниченной пропускной способностью, путем выработки спецификаций, которые разумно осуществимы в рамках небольшого объема кода. В первую очередь это относится к протоколу Message-Queuing Telemetry Transport (MQTT). Благодаря своей модели «издатель–подписчик» и возможности соединять встроенные устройства в разных физических расположениях по протоколу TCP/IP MQTT получил широкое распространение и поддерживается несколькими облачными архитектурами.

Протокол использует TCP для установления соединений с центральным посредником – брокером, который отправляет сообщения от издателей подписчикам. Издатели передают данные для определенной темы, описанной URI, а подписчики при подключении могут установить фильтр тем, за которыми они хотят следить, чтобы брокер выборочно пересылал им только сообщения, соответствующие фильтрам.

Несколько реализаций клиентской библиотеки существуют и для небольших встраиваемых устройств, хотя многие из них не поддерживают механизмы безопасности. Протокол поддерживает механизм аутентификации в виде текстового пароля, что неприемлемо по современным критериям безопасности, и его никогда не следует использовать поверх «чистого» канала TCP/IP, поскольку пароли могут быть легко перехвачены по пути.

Согласно стандарту, вместо TCP-соединения на основе сокетов через TCP-порт 1883, зарегистрированный в IANA, можно установить сеанс SSL, который использует TCP-порт 8883. Безопасная реализация MQTT, использующая сеансы SSL поверх TCP, представлена отдельной библиотеке GPL из пакета wolfSSL под названием wolfMQTT. Эта библиотека по умолчанию обеспечивает безопасные соединения сокетов MQTT. Она предоставляет как клиентскую, так и серверную аутентификацию с помощью сертификатов и открытых ключей и обеспечивает шифрование с симметричным ключом в текущем сеансе.

9.5.2. Архитектурный шаблон REST

Термин REST был предложен Роем Филдингом для описания шаблона, используемого веб-службами для связи с удаленными системами с использованием протокола без сохранения состояния. В REST-совместимой системе доступ к ресурсам осуществляется в виде HTTP-запросов, адресованных на определенный URI, с использованием того же стека протоколов, который применяется для отправки веб-страниц в ответ на запрос удаленного браузера. По сути, REST-запросы – это расширенные HTTP-запросы, представляющие все данные в виде закодированных строк, передаваемых через TCP в читаемом HTTP-потоке.

Использование этого шаблона обеспечивает ряд архитектурных преимуществ на стороне сервера и позволяет создавать распределенные системы с очень высокой масштабируемостью. Хотя встраиваемые системы не очень эффективны и обладают скромными ресурсами, они могут взаимодействовать с удаленными веб-службами, предоставляемыми системой RESTful, при помощи простого клиента REST.

9.5.3. Распределенные системы – единые точки отказа

При проектировании распределенных систем необходимо учитывать возможность возникновения дефектов каналов связи, недоступных шлюзов и других сбоев. Встраиваемые устройства не должны прекращать работу при потере связи с интернетом; на этот случай в них должны быть предусмотрены резервные механизмы на основе локальных шлюзов. Возьмем, к примеру, типичную систему IoT для управления всеми нагревательными и охлаждающими устройствами в доме, доступную с портативных устройств и координируемую удаленно с использованием любого механизма доступа к сети. Датчики температуры, нагреватели и охладители управляются с помощью mesh-сети встроенных устройств, а центральное управление осуществляется на удаленных облачных серверах. Система может дистанционно управлять приводами отопления/охлаждения на основе пользовательских настроек и показаний датчиков. Это дает нам возможность получить доступ к услуге даже из удаленного места, позволяя владельцу настраивать желаемую температуру в каждой комнате на основе команд, отправленных с пользовательских интерфейсов, которые обрабатываются и передаются облаком на встроенные устройства. Пока все компоненты подключены к интернету, система IoT работает должным образом.

Но в случае сбоя подключения пользователи не смогут управлять системой или активировать какую-либо функцию. Дублирование главной службы на устройстве в локальной сети обеспечивает устойчивую работу системы при сбоях соединения с интернетом и любых проблемах, которые могут помешать локальной сети получить доступ к удаленному облачному устройству. Если такой механизм существует, система, отключенная от интернета, попрежнему будет обеспечивать бесперебойную работу датчиков и исполнительных механизмов, предполагая, что все компоненты подключены к общей локальной сети. Более того, наличие локальной системы, обрабатывающей и ретранслирующей настройки и команды, снижает задержку отклика, поскольку запросы не должны проходить через интернет для обработки в облаке и пересылки обратно в ту же локальную сеть. Проектирование надежных сетей IoT должно включать в себя тщательную оценку единых точек отказа среди всех каналов и устройств, используемых для предоставления услуг, включая магистральный канал, используемый для доступа к службам, брокерам сообщений и удаленным устройствам, недоступность которых потенциально может вызвать отказ всей системы.

9.6. Заключение

В этой главе был представлен обзор принципов проектирования распределенных систем с межмашинной связью и сервисов IoT, в том числе подключенных конечных устройств, с акцентом на элементы безопасности, которые

слишком часто упускают из виду или недооценивают при разработке встраиваемых систем. Рассмотренная здесь технология обеспечивает полное, безопасное и быстрое подключение TCP/IP профессионального уровня к очень небольшим объектам и использует самые современные решения, такие как новейшие наборы шифров TLS. В качестве наиболее характерных примеров технологий, протоколов и алгоритмов безопасности, доступных для построения распределенных встроенных систем, было рассмотрено несколько аппаратных и программных технологий, доступных для маломощных устройств на основе микроконтроллеров.

В следующей главе рассмотрена многозадачность современных встраиваемых микроконтроллеров; в ней рассказывается, как с нуля разработать небольшой планировщик для микропроцессоров Cortex-M; кратко изложены ключевые роли операционной системы реального времени, работающей на встраиваемом устройстве.

Часть IV

МНОГОПОТОЧНОСТЬ

В заключительной части этой книги рассказано о разработке планировщика для параллельных многопоточных приложений и изменении контекста в ЦП ARM. В последней главе говорится о подходе TEE на примере системы, защищенной с помощью TrustZone-M.

Эта часть состоит из следующих глав:

- главы 10 «Параллельные задачи и планирование»;
- главы 11 «Надежная среда выполнения».

Глава 10

Параллельные задачи и планирование

Когда сложность системы увеличивается и программное обеспечение должно одновременно управлять несколькими периферийными устройствами и событиями, с определенного момента становится удобнее полагаться на операционную систему для координации и синхронизации всех операций. Разделение логики приложения на разные потоки дает несколько важных архитектурных преимуществ. Каждый компонент выполняет определенную операцию в своем работающем блоке и может освобождать ЦП, пока он находится в режиме ожидания ввода или события тайм-аута.

В этой главе будут рассмотрены механизмы, применяемые для реализации многопоточной встроенной операционной системы. Для наглядности будет создана с нуля минималистичная операционная система, адаптированная к тестовой платформе и содержащая работающий планировщик для параллельного выполнения нескольких задач.

Внутренние механизмы планировщика в основном реализованы в вызовах системных служб, и его строение влияет на производительность системы и другие функции, такие как различные уровни приоритета и временные ограничения для задач, зависящих от реального времени. Некоторые из возможных политик планирования для различных контекстов будут объяснены и реализованы в примерах кода.

Параллельное выполнение нескольких потоков подразумевает возможность совместного использования ресурсов и одновременного доступа к одной и той же памяти. Большинство микропроцессоров, предназначенных для запуска многопоточных систем, поддерживают примитивные функции для реализации механизмов блокировки, таких как семафоры, через специальные инструкции ассемблера. В нашем примере операционная система предоставляет примитивы мьютекса и семафора, которые могут использоваться потоками для управления доступом к общим ресурсам.

Внедряя механизмы защиты памяти, можно обеспечить разделение ресурсов на основе их адресов и позволить ядрам контролировать все операции, связанные с оборудованием, через интерфейс системных вызовов. Большинство встроенных операционных систем реального времени предпочитают плоскую модель без сегментации, чтобы код ядра был как можно меньше, и с минимальным API для оптимизации ресурсов, доступных для приложений. На примере ядра будет показано, как создать API системного вызова для централизации управления ресурсами, используя сегментацию физической памяти для защиты ресурсов ядра, системный блок управления, сопоставленные периферийные устройства и другие подходы, чтобы увеличить уровень безопасности системы.

В этой главе раскрыты следующие темы:

- управление задачами;
- реализация планировщика;
- синхронизация;
- разделение системных ресурсов.

К концу главы вы узнаете, как построить многопоточную встраиваемую среду.

10.1. Технические требования

Вы можете скачать файлы кода для этой главы на GitHub по адресу https://github.com/PacktPublishing/Embedded-Systems-Architecture-Second-Edition/tree/main/Chapter10.

10.2. Управление задачами

Операционная система обеспечивает абстракцию параллельно работающих процессов и потоков, чередуя приложения для параллельного выполнения. Фактически в системах с одним процессором одновременно может быть запущен только один поток. Пока запущенный поток выполняется, все остальные ждут в очереди до следующего переключения задачи.

В *совместной* (cooperative) модели переключение задачи всегда является добровольным действием, запрашиваемым со стороны потока. Противоположный подход, известный как *вытеснение* или *прерывание выполнения* (preemption), требует, чтобы ядро периодически прерывало задачи в произвольный момент их выполнения, сохраняло состояние текущей задачи и возобновляло выполнение следующей задачи в очереди.

Переключение выполняемой задачи на низком уровне представляет собой сохранение значений регистров ЦП в ОЗУ и загрузку из памяти значений следующей задачи, выбранной для выполнения. Эта операция более извест-

на как *переключение контекста* (context switching) и лежит в основе системы планирования.

10.2.1. Блок задач

Задачи представлены в системе в виде блочной структуры. Этот объект содержит всю информацию, необходимую планировщику для постоянного отслеживания состояния задачи, и зависит от конструкции планировщика. Задачи могут быть определены во время компиляции и запущены после загрузки ядра или созданы и завершены во время работы системы.

Каждый блок задачи может содержать указатель на функцию запуска, которая определяет начало кода, выполняемого при порождении задачи, и набор необязательных аргументов. Для каждой задачи назначается отдельная область памяти, которую можно использовать в качестве частной области стека. Таким образом, контекст выполнения каждого потока и процесса отделен от всех остальных, а значения регистров могут быть сохранены в области памяти, отведенной для задачи, когда задача прерывается. Указатель стека для конкретной задачи хранится в структуре блока задач и используется для хранения значений регистра ЦП при переключении контекста.

Запуск с отдельными стеками требует, чтобы часть памяти была зарезервирована заранее и связана с каждой задачей. В простейшем случае все задачи, использующие стек одинакового размера, создаются до запуска планировщика и не могут быть остановлены. В таком случае память, зарезервированная для частных стеков, может быть непрерывной и ассоциироваться с каждой новой задачей. Область памяти, используемая для областей стека, может быть определена в скрипте компоновщика.

Наша тестовая платформа имеет отдельную память, связанную с ядром, отображаемую по адресу `0x10000000`. Существует множество способов упорядочивания разделов памяти. Сопоставим начало пространства стека, используемого для связывания областей стека с потоками, с началом CCRAM. Оставшееся пространство CCRAM используется в качестве стека для ядра, а для динамического выделения полностью остается SRAM, за исключением разделов .data и .bss. Указатели экспортируются сценарием компоновщика со следующими инструкциями `PROVIDE`:

```
PROVIDE(_end_stack = ORIGIN(CCRAM) + LENGTH(CCRAM));
PROVIDE(stack_space = ORIGIN(CCRAM));
PROVIDE(_start_heap = _end);
```

В исходном коде ядра пространство стека `stack_space` объявлено как внешнее, поскольку оно экспортируется сценарием компоновщика. Объявим количество места, зарезервированного для стека выполнения каждой задачи (выраженное в четырехбайтных словах):

```
extern uint32_t stack_space;
#define STACK_SIZE (256)
```

Каждый раз, когда создается новая задача, следующий килобайт в пространстве стека назначается ее стеком выполнения, а начальный указатель стека устанавливается на наивысший адрес в области, поскольку стек выполнения растет в обратном направлении (рис. 10.1).

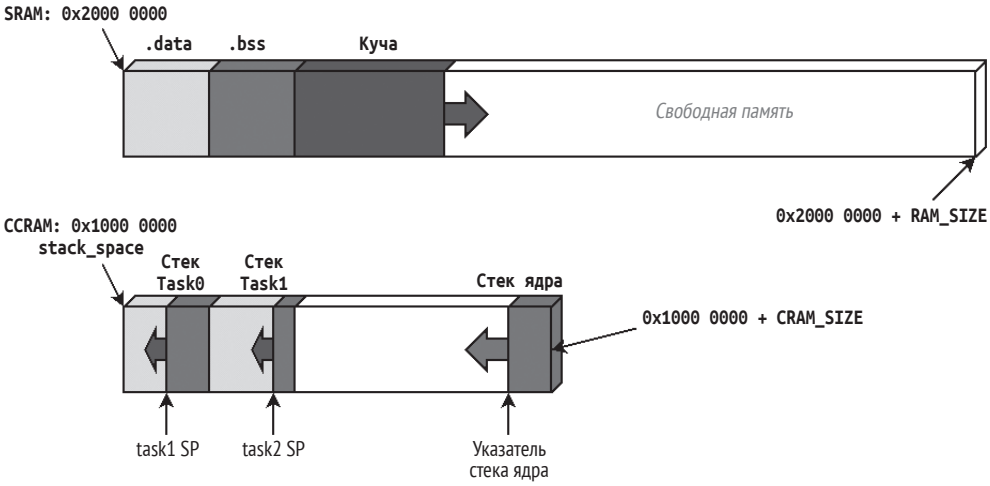

Рис. 10.1 ❖ Конфигурация памяти, используемая для выделения задачам отдельных стеков выполнения

Затем можно объявить простую структуру блока задач следующим образом:

```
#define TASK_WAITING 0
#define TASK_READY 1
#define TASK_RUNNING 2

#define TASK_NAME_MAXLEN 16;

struct task_block {
  char name[TASK_NAME_MAXLEN];
  int id;
  int state;
  void (*start)(void *arg);
  void *arg;
  uint32_t *sp;
};
```

Для хранения всех блоков задач системы определен глобальный массив. Нужно использовать глобальный индекс для отслеживания уже созданных задач, чтобы иметь возможность использовать позицию в памяти относительно идентификатора текущей задачи:

```
#define MAX_TASKS 8
static struct task_block TASKS[MAX_TASKS];
static int n_tasks = 1;
```

```
static int running_task_id = 0;
#define kernel TASKS[0]
```

В этой модели блок задач предварительно размещается в разделе данных, а поля инициализируются по ходу выполнения, отслеживая индекс. Первый элемент массива зарезервирован для блока задач ядра, которое является запущенным в данный момент процессом.

В нашем примере задачи создаются путем вызова функции `task_create` и указания имени, точки входа и ее аргумента. Для статической конфигурации с предопределенным количеством задач это делается при инициализации ядра, но более продвинутые планировщики могут позволить нам выделять новые блоки управления для порождения новых процессов во время выполнения, пока планировщик работает:

```
struct task_block *task_create(char *name, void (*start)(void
*arg), void *arg)
{
  struct task_block *t;
  int i;
  if (n_tasks >= MAX_TASKS)
    return NULL;
  t = &TASKS[n_tasks];
  t->id = n_tasks++;
  for (i = 0; i < TASK_NAME_MAXLEN; i++) {
    t->name[i] = name[i];
    if (name[i] == 0)
      break;
  }
  t->state = TASK_READY;
  t->start = start;
  t->arg = arg;
  t->sp = ((&stack_space) + n_tasks * STACK_SIZE);
  task_stack_init(t);
  return t;
}
```

Чтобы реализовать функцию `task_stack_init`, которая инициализирует значения в стеке для запуска процесса, нам нужно понять, как работает переключение контекста и как запускаются новые задачи при запуске планировщика.

10.2.2. Переключение контекста

Процедура переключения контекста состоит из получения значений регистра ЦП во время выполнения и сохранения их в нижней части стека текущей задачи. Затем нужно восстановить значения для следующей задачи, чтобы возобновить ее выполнение. Эта операция должна выполняться в контексте прерывания, и ее внутренние механизмы зависят от процессора. На нашей тестовой платформе любой обработчик прерывания может заменить теку-

щую задачу и восстановить другой контекст, но эта операция чаще выполняется в подпрограммах обслуживания прерываний, связанных с системными событиями. Cortex-M предоставляет два исключения ЦП, которые предназначены для обеспечения базовой поддержки переключения контекста, поскольку они могут запускаться произвольно в любом контексте:

- **PendSV:** это способ по умолчанию для вытесняющего ядра принудительно запустить прерывание вскоре после установки одного бита в определенном регистре в блоке управления системой, и обычно он связан с переключением контекста следующей задачи;
- **SVCall:** это основная точка входа пользовательского приложения для подачи формального запроса на доступ к ресурсу, управляемому ядром. Эта функция предоставляет API безопасного доступа к ядру для запроса операций от компонента или драйвера. Поскольку результат операции может быть недоступен сразу, SVCall также может разрешить вытеснение вызывающей задачи, чтобы реализовать абстракцию блокирующих системных вызовов.

Подпрограммы, используемые для сохранения значений регистров ЦП в память и восстановления из памяти во время переключения контекста, в процессоре Cortex-M частично реализованы аппаратно. Это означает, что при входе в прерывание копия части регистра автоматически помещается в стек. Копия регистров в стеке называется *кадром стека* (stack frame) и содержит регистры с R0 по R3, R12, LR, PC и xPSR в порядке, показанном на рис. 10.2.

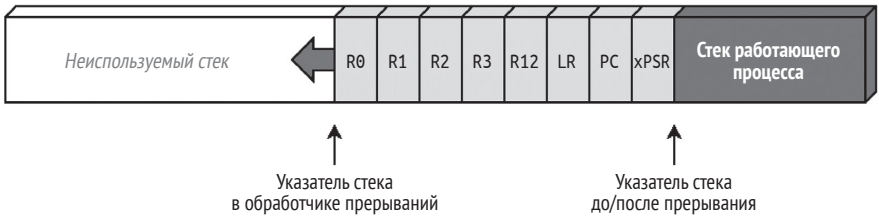

Рис. 10.2 ❖ Регистры автоматически копируются в стек при входе в обработчик прерывания

Но указатель стека не захватывает другую половину регистров ЦП – с R4 по R11. По этой причине для успешного завершения переключения контекста системный обработчик, намеревающийся заменить работающий процесс, должен сохранить *дополнительный кадр стека*, содержащий значение для этих регистров, и восстановить дополнительный кадр стека следующей задачи непосредственно перед возвратом из обработчика. Набор инструкций ARM Thumb-2 содержит команды для размещения значений смежных регистров ЦП в стеке и возврата их на место. Следующие две функции используются для добавления дополнительного кадра в стек и извлечения из стека:

```
static void __attribute__((naked)) store_context(void)
{
  asm volatile("mrs r0, msp");
```

```
  asm volatile("stmdb r0!, {r4-r11}");
  asm volatile("msr msp, r0");
  asm volatile("bx lr");
}

static void __attribute__((naked)) restore_context(void)
{
  asm volatile("mrs r0, msp");
  asm volatile("ldmfd r0!, {r4-r11}");
  asm volatile("msr msp, r0");
  asm volatile("bx lr");
}
```

Атрибут `((naked))` используется для того, чтобы GCC не помещал последовательности пролога и эпилога, состоящие из нескольких ассемблерных инструкций, в скомпилированный код. В прологе будут изменены значения некоторых регистров в области дополнительного кадра стека, которые будут восстановлены в эпилоге, и это противоречит назначению функций, получающих доступ к значениям регистров с помощью ассемблерных инструкций. Из-за отсутствия эпилога функции `naked` возвращаются, переходя к вызывающей инструкции, которая хранится в регистре LR.

В результате ассемблерной операции `push` стек вытесняемого процесса выглядит, как показано на рис. 10.3.

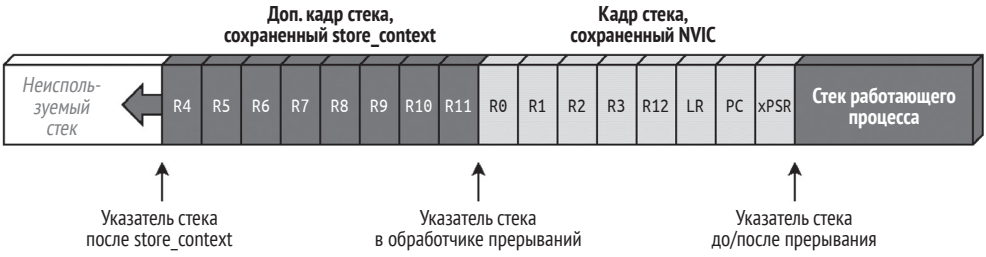

Рис. 10.3 ❖ Копирование значений остальных регистров в стек для завершения переключения контекста

10.2.3. Создание задач

Когда система работает, все задачи, кроме одной, находятся в состоянии ожидания. Это означает, что полный кадр стека сохраняется в нижней части стека, а указатель стека хранится в блоке управления, чтобы им мог воспользоваться планировщик для возобновления каждого процесса.

Вновь созданная задача впервые проснется в середине переключения контекста. Ожидается, что в этот момент задача сохранит предыдущее состояние регистров ЦП, но очевидно, что у новой задачи его просто нет. При создании стека в конец помещается фиктивный кадр стека, чтобы при возобновлении задачи сохраненные значения копировались в системные регистры и задача могла возобновиться с точки входа.

Функция task_create опирается на функцию инициализации стека, task_stack_init, которая помещает начальные значения в системные регистры, чтобы разрешить восстановление задачи, и перемещает сохраненный указатель стека в начало дополнительного кадра, который можно оставить неинициализированным. Для доступа к сохраненному регистру в кадре стека нужно объявить структуру stack_frame, которая использует по одному полю для каждого регистра, и структуру extra_frame, просто для полноты картины:

```
struct stack_frame {
  uint32_t r0, r1, r2, r3, r12, lr, pc, xpsr;
};

struct extra_frame {
  uint32_t r4, r5, r6, r7, r8, r9, r10, r11;
};

static void task_stack_init(struct task_block *t)
{
  struct stack_frame *tf;
  t->sp -= sizeof(struct stack_frame);
  tf = (struct stack_frame *)(t->sp);
  tf->r0 = (uint32_t) t->arg;
  tf->pc = (uint32_t) t->start;
  tf->lr = (uint32_t) task_terminated;
  tf->xpsr = (1 << 24);
  t->sp -= sizeof(struct extra_frame);
}
```

После восстановления контекста обработчик исключений процедуры возврата автоматически восстанавливает контекст из кадра стека, который, как вы помните, был просто сфабрикован. Регистры для стартующей задачи инициализируются следующим образом:

- **счетчик программ** (program counter, PC) содержит адрес функции запуска, куда система перейдет, чтобы переключиться на эту задачу в первый раз;
- **R0–R3** могут содержать необязательные аргументы для передачи функции запуска в соответствии с ABI ЦП. В данном случае происходит перенос значения единственного аргумента, заданного для функции запуска вызывающей стороной task_create;
- **регистр состояния исполняемой программы** (execution program status register, xPSR) должен быть запрограммирован так, чтобы в бите 24 был установлен только обязательный флаг набора инструкций Thumb;
- **регистр ссылки** (link register, LR) содержит указатель на процедуру, которая вызывается при возврате функции запуска. В нашем случае задачам не разрешается возвращаться из функции запуска, поэтому функция task_terminated – это просто бесконечный цикл, и это считается системной ошибкой. В других случаях, если задачам разрешено завершаться, функция может играть роль общей точки выхода для задач, чтобы выполнять операции очистки, необходимые при возврате из функции запуска.

После создания начального кадра стека задача начинает участвовать в многозадачности и может быть выбрана планировщиком в любое время для возобновления выполнения из того же состояния, как и остальные неработающие задачи (рис. 10.4).

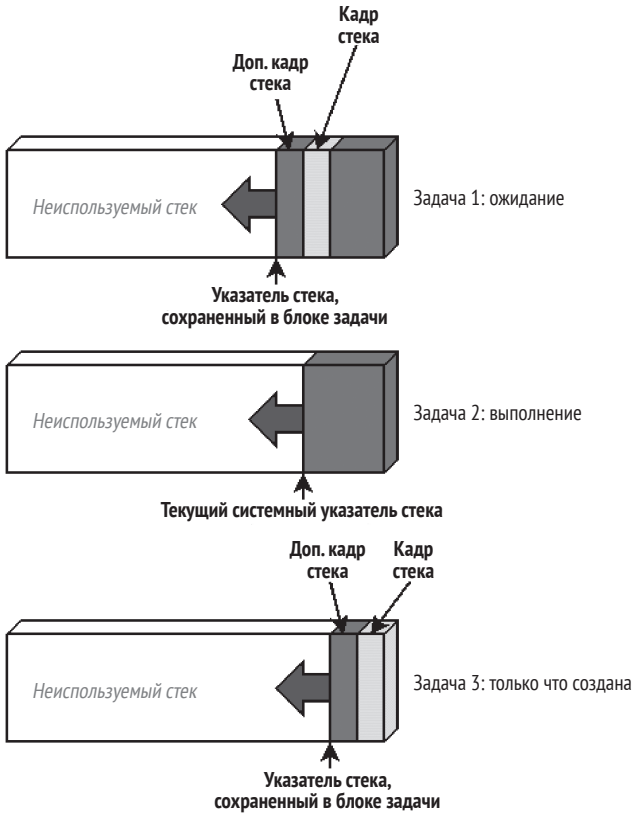

Рис. 10.4 ❖ Указатели стека трех задач в разных состояниях выполнения

Простая функция `main` ядра к этому моменту может создавать процессы и подготавливать стек, но еще не может их запускать, пока не будут реализованы внутренние компоненты планировщика. В этом случае полезен хронометраж, поэтому при запуске нужно включить `SysTick`, чтобы отслеживать время в системе. Следующий код инициализирует блок задач ядра и создает две новые задачи:

```
void main(void) {
  clock_pll_on(0);
  systick_enable();
  led_setup();
  kernel.name[0] = 0;
  kernel.id = 0;
  kernel.state = TASK_RUNNING;
```

```
  task_create("test0",task_test0, NULL);
  task_create("test1",task_test1, NULL);
  while(1) {
    schedule();
  }
}
```

Две задачи созданы так, что они указывают на разные функции запуска, и обе получают NULL в качестве аргумента. Обе функции никогда не должны возвращаться и могут быть прерваны и возобновлены в соответствии с реализованной политикой планировщика.

Теперь удалить необходимо реализовать внутренний механизм планировщика, чтобы запускать и чередовать выполнение только что определенных нами параллельных задач.

10.3. Реализация планировщика

Архитектура системы зависит от того, как реализован планировщик. Задачи могут выполняться в кооперативной модели до тех пор, пока они добровольно не уступят ЦП следующей задаче, или за кулисами системы ОС решает инициировать прерывание, чтобы сменить текущую задачу на следующую, применяя определенную стратегию формирования интервала между переключением задач и назначая приоритеты для выбора следующей задачи. В обоих случаях переключение контекста происходит в рамках одного из доступных вызовов супервизора, предназначенного для принятия решения о том, какие задачи планировать следующими, и для выполнения переключения контекста. В этом разделе к нашему примеру будет добавлена полная процедура переключения контекста через PendSV, а затем будут проанализированы и реализованы несколько возможных стратегий планирования.

10.3.1. Вызовы супервайзера

Основной компонент планировщика представляет собой обработчик исключений, связанных с событиями прерывания системы, таких как PendSV и SVCall. В Cortex-M исключение PendSV может быть вызвано программным обеспечением в любое время путем установки флага PENDSET, соответствующего биту 28 регистра управления прерыванием и состояния, расположенного в SCB по адресу 0xE000ED04. Определим простой макрос для инициирования переключения контекста путем установки флага:

```
#define SCB_ICSR (*((volatile uint32_t *)0xE000ED04))
#define schedule() SCB_ICSR |= (1 << 28)
```

Вызов schedule из ядра и все последующие вызовы приведут к переключению контекста, которое теперь можно реализовать в обработчике PendSV.

Чтобы завершить переключение контекста, обработчик должен выполнить следующие шаги:

1) сохранить текущий указатель стека из регистра указателя в блоке задач;
2) поместить дополнительный кадр в стек, вызвав `store_context`;
3) изменить состояние текущей задачи на `TASK_READY`;
4) выбрать новую задачу для возобновления;
5) изменить состояние новой задачи на `TASK_RUNNING`;
6) получить новый указатель стека из связанного блока задачи;
7) извлечь дополнительный кадр из стека, вызвав `restore_context`;
8) установить специальное возвращаемое значение для обработчика прерывания, чтобы активировать режим потока в конце служебной подпрограммы PendSV.

Функция `isr_pendsv` должна иметь атрибут `naked`, поскольку она обращается к регистру ЦП напрямую через функции `store_context` и `restore_context`:

```
void __attribute__((naked)) isr_pendsv(void)
{
  store_context();
  asm volatile("mrs %0, msp" : "=r"(
          TASKS[running_task_id].sp));
  TASKS[running_task_id].state = TASK_READY;
  running_task_id++;
  if (running_task_id >= n_tasks)
    running_task_id = 0;
  TASKS[running_task_id].state = TASK_RUNNING;
  asm volatile("msr msp, %0"::"r"(
                 TASKS[running_task_id].sp));
  restore_context();
  asm volatile("mov lr, %0" ::"r"(0xFFFFFFF9));
  asm volatile("bx lr");
}
```

Значение, которое загружается в LR перед возвратом, указывает, что происходит возврат в потоковый режим в конце этого прерывания. В зависимости от значения последних 3 бит служебная подпрограмма сообщает ЦП, какой указатель стека использовать при возврате из прерывания. Используемое в данном случае значение `0xFFFFFFF9` обозначает потоковый режим, применяющий указатель основного стека. Другие значения потребуются позже, когда в пример будет добавлена поддержка отдельных указателей стека для ядра и процесса.

Полный контекст был реализован внутри служебной подпрограммы PendSV, которая на данный момент просто выбирает следующую задачу и выполняет ядро с ID `0` после последней задачи в массиве. Служебная подпрограмма запускается в режиме обработчика каждый раз, когда вызывается макрос расписания.

10.3.2. Планировщик совместного выполнения

Для чередования выполнения задач в системе можно применять разные стратегии. В простейшем случае функция `main()` каждой задачи добровольно приостанавливает ее выполнение, вызывая макрос расписания.

В следующем примере определены два потока. Оба включают светодиод и удерживают ЦП в цикле занятости в течение 1 с, прежде чем выключить светодиод и явным образом вызвать функцию `schedule()` для запуска переключения контекста:

```
void task_test0(void *arg)
{
  uint32_t now = jiffies;
  blue_led_on();
  while(1) {
    if ((jiffies - now) > 1000) {
      blue_led_off();
      schedule();
      now = jiffies;
      blue_led_on();
    }
  }
}

void task_test1(void *arg)
{
  uint32_t now = jiffies;
  red_led_on();
  while(1) {
    if ((jiffies - now) > 1000) {
      red_led_off();
      schedule();
      now = jiffies;
      red_led_on();
    }
  }
}
```

Наша небольшая операционная система наконец-то заработала, и ядро последовательно планирует две задачи. Задача с идентификатором `0` также возобновляется в начале каждого цикла, но в этом простом случае задача ядра только вызывает расписание в цикле, немедленно возобновляя задачу с идентификатором 1. В данном случае реактивность системы полностью зависит от реализации задач, так как каждая задача может бесконечно удерживать ЦП и препятствовать запуску других задач. Совместная модель используется только в очень специфических сценариях, где каждая задача напрямую отвечает за использование своих циклов ЦП и взаимодействие с другими потоками и может влиять на скорость отклика и поведение всей системы.

Для простоты в этой реализации не учитывается обработка переменной `jiffies`. При приращении каждую миллисекунду переполнение значения `jiffies` произойдет примерно через 42 дня. Реальные операционные системы, в отличие от нашего упрощенного примера, должны реализовывать соответствующий механизм для сравнения переменных времени, не показанный здесь, который может обнаруживать переход при вычислении разницы во времени.

10.3.3. Параллелизм и кванты времени

Другой подход заключается в назначении каждой задаче коротких интервалов процессорного времени и непрерывном переключении процессов через очень короткие промежутки выполнения. Вытесняющий планировщик автономно прерывает текущую задачу, чтобы возобновить следующую без явного запроса от самой задачи. Он также может навязывать свою политику в отношении выбора следующей задачи для запуска и продолжительности интервала, в течение которого ЦП выделяется каждой задаче, а именно ее *кванта времени* (timeslice).

С точки зрения задачи выполнение теперь выглядит непрерывным и полностью независимым от планировщика, который действует за кулисами, непрерывно прерывая и возобновляя каждую задачу, создавая тем самым иллюзию, что все задачи выполняются одновременно. Потоки можно переопределить, чтобы светодиоды мигали с двумя разными интервалами:

```
void task_test0(void *arg)
{
  uint32_t now = jiffies;
  blue_led_on();
  while(1) {
    if ((jiffies - now) > 500) {
      blue_led_toggle();
      now = jiffies;
    }
  }
}

void task_test1(void *arg)
{
  uint32_t now = jiffies;
  red_led_on();
  while(1) {
    if ((jiffies - now) > 125) {
      red_led_toggle();
      now = jiffies;
    }
  }
}
```

Чтобы чередовать задачи в циклическом режиме, можно инициировать выполнение PendSV из обработчика SysTick, что приводит к переключению задач через регулярные промежутки времени. Новый обработчик SysTick запускает переключение контекста каждые TIMESLICE миллисекунд:

```
#define TIMESLICE (20)
void isr_systick(void)
{
  if ((++jiffies % TIMESLICE) == 0)
  schedule();
}
```

В этой новой конфигурации у нас теперь есть более полная модель, позволяющая нескольким задачам выполняться независимо, а планирование полностью контролируется ядром.

10.3.4. Блокировка задач

Простой планировщик, который был до сих пор, обеспечивает только два состояния задач: TASK_READY и TASK_RUNNING. Третье состояние может быть реализовано для определения задачи, которую не нужно возобновлять, поскольку она была заблокирована и ожидает события или тайм-аута. Задача может ожидать системного события определенного типа, например:

- события прерывания от устройства ввода/вывода (I/O), используемого задачей;
- связь с другой задачей, такой как стек TCP/IP;
- механизмы синхронизации, такие как мьютекс или семафор, для доступа к общему ресурсу в системе, который в данный момент недоступен;
- события тайм-аута.

Для управления различными состояниями планировщик может вести два или более списков, чтобы отделить задачи, выполняемые в данный момент или готовые к выполнению, от задач, ожидающих события. Затем планировщик выбирает следующую задачу среди тех, что находятся в состоянии TASK_READY, и игнорирует те, которые находятся в списке заблокированных задач (рис. 10.5).

Эта вторая версия планировщика отслеживает текущую задачу, используя глобальный указатель, а не индекс массива, и организует задачи в два списка:

- tasklist_active: содержит блок для текущей задачи и всех задач в состоянии TASK_READY, ожидающих планирования;
- tasklist_waiting: содержит блок задач, заблокированных в данный момент.

Рис. 10.5 ❖ Конечный автомат, описывающий состояния выполнения задачи

Простейшим примером этого нового механизма может служить функция `sleep_ms`, которую задачи используют для временного переключения в состояние ожидания и установки точки возобновления в будущем для повторного планирования. Реализация такого подхода позволяет нашим задачам спать между действиями переключения светодиодов вместо запуска цикла занятости, который неоднократно проверяет, не истек ли таймер. Эти новые задачи не только более эффективны, поскольку они не тратят впустую ресурсы ЦП в цикле занятости, но и более читаемы:

```
void task_test0(void *arg){
  blue_led_on();
  while(1) {
    sleep_ms(500);
    blue_led_toggle();
  }
}

void task_test1(void *arg)
{
  red_led_on();
  while(1) {
    sleep_ms(125);
    red_led_toggle();
  }
}
```

Для упорядочивания блоков задач в списки в структуру необходимо добавить указатель на следующий элемент, чтобы два списка заполнялись во время выполнения. Для управления функцией `sleep_ms` необходимо добавить новое поле системного времени. Нужно отслеживать это время, чтобы знать, когда перемещать каждую задачу в список активных задач для возобновления:

```
struct task_block {
  char name[TASK_NAME_MAXLEN];
  int id;
```

```
  int state;
  void (*start)(void *arg);
  void *arg;
  uint8_t *sp;
  uint32_t wakeup_time;
  struct task_block *next;
};
```

Этими списками можно управлять с помощью двух простых функций для вставки/удаления элементов:

```
struct task_block *tasklist_active = NULL;
struct task_block *tasklist_waiting = NULL;

static void tasklist_add(struct task_block **list,struct task_
block *el)
{
  el->next = *list;
  *list = el;
}

static int tasklist_del(struct task_block **list, struct task_
block *delme)
{
  struct task_block *t = *list;
  struct task_block *p = NULL;
  while (t) {
    if (t == delme) {
      if (p == NULL)
        *list = t->next;
      else
        p->next = t->next;
      return 0;
    }
    p = t;
    t = t->next;
  }
  return -1;
}
```

Необходимо также добавить две дополнительные функции для перемещения задач из активного списка в список ожидания и наоборот, которые дополнительно изменяют состояние самой задачи:

```
static void task_waiting(struct task_block *t)
{
  if (tasklist_del(&tasklist_active, t) == 0) {
    tasklist_add(&tasklist_waiting, t);
    t->state = TASK_WAITING;
  }
}

static void task_ready(struct task_block *t)
```

```
{
  if (tasklist_del(&tasklist_waiting, t) == 0) {
    tasklist_add(&tasklist_active, t);
    t->state = TASK_READY;
  }
}
```

Функция `sleep_ms` устанавливает время возобновления и переводит задачу в состояние ожидания, а затем активирует планировщик, чтобы задача была вытеснена:

```
void sleep_ms(int ms)
{
  if (ms < TASK_TIMESLICE)
    return;
  t_cur->wakeup_time = jiffies + ms;
  task_waiting(t_cur);
  schedule();
}
```

Новый обработчик PendSV выбирает следующую задачу для запуска из активного списка, который, как предполагается, всегда содержит хотя бы одну задачу, поскольку основная задача ядра никогда не переходит в состояние ожидания. Новый поток выбирается с помощью функции `tasklist_next_ready`, которая также гарантирует, что если текущая задача была перемещена из активного списка или является последней в очереди, заголовок активного списка будет выбран для следующего кванта времени:

```
static inline struct task_block *tasklist_next_ready(struct
task_block *t)
{
  if ((t->next == NULL) || (t->next->state != TASK_READY))
    return tasklist_active;
  return t->next;
}
```

Эта небольшая функция является ядром нового планировщика, основанного на двойном списке, и вызывается в середине каждого переключения контекста для выбора следующей активной задачи в PendSV:

```
void __attribute__((naked)) isr_pendsv(void)
{
  store_context();
  asm volatile("mrs %0, msp" : "=r"(t_cur->sp));
  if (t_cur->state == TASK_RUNNING) {
    t_cur->state = TASK_READY;
  }
  t_cur = tasklist_next_ready(t_cur);
  t_cur->state = TASK_RUNNING;
  asm volatile("msr msp, %0" ::"r"(t_cur->sp));
  restore_context();
```

```
  asm volatile("mov lr, %0" ::"r"(0xFFFFFFF9));
  asm volatile("bx lr");
}
```

Наконец, чтобы проверить время пробуждения каждой спящей задачи, ядро обращается к списку ожидающих задач и перемещает блоки задач обратно в активный список по истечении времени пробуждения. Теперь добавим к инициализации ядра несколько дополнительных шагов, чтобы гарантировать, что сама задача ядра будет помещена в список запущенных задач при загрузке:

```
void main(void) {
  clock_pll_on(0);
  led_setup();
  button_setup();
  systick_enable();
  kernel.name[0] = 0;
  kernel.id = 0;
  kernel.state = TASK_RUNNING;
  kernel.wakeup_time = 0;
  tasklist_add(&tasklist_active, &kernel);
  task_create("test0",task_test0, NULL);
  task_create("test1",task_test1, NULL);
  task_create("test2",task_test2, NULL);
  while(1) {
    struct task_block *t = tasklist_waiting;
    while (t) {
      if (t->wakeup_time && (t->wakeup_time < jiffies)) {
        t->wakeup_time = 0;
        task_ready(t);
      }
      t = t->next;
    }
    WFI();
  }
}
```

10.3.5. Ожидание ресурсов

Блокировка на заданный интервал времени – это только одна из возможностей временно исключить задачу из списка активных. Ядро может реализовать другие обработчики событий и прерываний, чтобы вернуть задачи в цикл планировщика. Например, задача может заблокироваться на какое-то время, ожидая событий ввода-вывода от определенного набора ресурсов в состоянии `TASK_WAITING`.

В нашем примере кода можно реализовать функцию чтения состояния кнопки из задачи, которая будет выходить из блокировки только после нажатия кнопки. До этого момента вызывающая задача остается в списке ожидания и никогда не планируется. Задача, которая переключает зеленый

светодиод каждый раз, когда нажимается кнопка, использует button_read() в качестве точки блокировки:

```
#define BUTTON_DEBOUNCE_TIME 120
void task_test2(void *arg)
{
  uint32_t toggle_time = 0;
  green_led_off();
  while(1) {
    if (button_read()) {
      if((jiffies - toggle_time) > BUTTON_DEBOUNCE_TIME)
      {
        green_led_toggle();
        toggle_time = jiffies;
      }
    }
  }
}
```

Функция button_read указывает на вызывающую задачу, поэтому указатель button_task используется для ее пробуждения при нажатии кнопки. Задача перемещается в список ожидания, и в драйвере инициируется операция чтения, после чего задача вытесняется:

```
struct task_block *button_task = NULL;
int button_read(void)
{
  if (button_task)
    return 0;
  button_task = t_cur;
  task_waiting(t_cur);
  button_start_read();
  schedule();
  return 1;
}
```

Чтобы уведомить планировщик о нажатии кнопки, драйвер использует обратный вызов, указанный ядром во время инициализации, и передает его в качестве аргумента в button_setup:

```
static void (*button_callback)(void) = NULL;

void button_setup(void (*callback)(void))
{
  AHB1_CLOCK_ER |= GPIOA_AHB1_CLOCK_ER;
  GPIOA_MODE &= ~ (0x03 << (BUTTON_PIN * 2));
  EXTI_CR0 &= ~EXTI_CR_EXTI0_MASK;
  button_callback = callback;
}
```

Ядро связывает функцию button_wakeup с обратным вызовом драйвера, поэтому при возникновении события, если задача ожидает уведомления о на-

жатии кнопки, она перемещается обратно в список активных задач и возобновляется, как только планировщик выбирает ее для запуска:

```
void button_wakeup(void)
{
  if (button_task) {
    task_ready(button_task);
    button_task = NULL;
    schedule();
  }
}
```

В драйвере кнопки для инициации операции блокировки разрешено прерывание, связанное с нарастающим фронтом сигнала, который соответствует событию нажатия кнопки:

```
void button_start_read(void)
{
  EXTI_IMR |= (1 << BUTTON_PIN);
  EXTI_EMR |= (1 << BUTTON_PIN);
  EXTI_RTSR |= (1 << BUTTON_PIN);
  nvic_irq_enable(NVIC_EXTI0_IRQN);
}
```

При обнаружении события выполняется обратный вызов в контексте прерывания. Прерывание будет отключено до следующего вызова `button_start_read`:

```
void isr_exti0(void)
{
  nvic_irq_disable(NVIC_EXTI0_IRQN);
  EXTI_PR |= (1 << BUTTON_PIN);
  if (button_callback)
    button_callback();
}
```

Любой драйвер устройства или системный модуль, использующий обработку прерываний для разблокировки соответствующей задачи, для взаимодействия с планировщиком может использовать механизм обратного вызова. Используя аналогичную стратегию блокировки, можно реализовать операции чтения и записи, чтобы держать вызывающую задачу в списке ожидания до тех пор, пока желаемое событие не будет обнаружено и обработано через обратный вызов в коде планировщика.

Системным компонентам и библиотекам, предназначенным для встраиваемых приложений без ОС, может потребоваться дополнительный уровень для интеграции в операционную систему с блокировкой вызовов. Встроенные реализации стека TCP/IP, такие как lwIP и picoTCP, обеспечивают уровень интеграции с портативной RTOS, включая блокировку вызовов сокетов, реализованную путем запуска функций цикла в выделенной задаче, которая управляет связью с API сокета, занятым в других задачах. Ожидается, что механизмы блокировки, такие как мьютексы и семафоры, реализуют бло-

кирующие вызовы, которые приостанавливают выполнение задачи, когда запрошенный ресурс недоступен.

Стратегия планирования, которая использовалась, является очень реактивной и обеспечивает хороший уровень взаимодействия между задачами, но не предусматривает уровни приоритета, что необходимо при разработке систем реального времени.

10.3.6. Планирование в реальном времени

Одним из ключевых требований к операционным системам реального времени является способность реагировать на заданное количество событий, выполняя соответствующий код в течение короткого и предсказуемого промежутка времени. Чтобы реализовать функции со строгими требованиями ко времени, операционная система должна сосредоточиться на быстрой обработке и диспетчеризации прерываний, а не на других показателях, таких как пропускная способность или равнодоступность ресурсов. Каждая задача может иметь определенные требования, например крайние сроки, указывающие точное время, когда выполнение должно начаться или закончиться, или быть связанной с общими ресурсами, что ведет к зависимостям от других задач в системе. Система, которая может выполнять задачи с детерминированным требованием времени, должна быть в состоянии уложиться в сроки в течение измеримого фиксированного периода времени.

Организация планирования в реальном времени – сложная задача. По этой теме опубликовано много авторитетных статей и книг, поэтому здесь не будет подробного рассмотрения деталей. Исследования показали, что несколько подходов, основанных на приоритетах задач в сочетании с правильно подобранной стратегией переключения задач во время выполнения, способны обеспечить хорошее приближение к идеальной системе реального времени.

Чтобы поддерживать задачи жесткого реального времени с детерминированными сроками выполнения, операционная система должна обладать следующими характеристиками:

- процедура быстрого переключения контекста, реализованная в планировщике;
- измеримые интервалы, когда система работает с отключенными прерываниями;
- короткие обработчики прерываний;
- поддержка приоритетов прерываний;
- поддержка приоритетов задач для минимизации задержек при выполнении сложных задач в реальном времени.

С точки зрения планирования задач задержка в реальном времени в основном связана со способностью системы возобновлять выполнение задачи при возникновении внешнего события.

Чтобы гарантировать детерминированную задержку для выбранной группы задач, RTOS часто реализуют уровни фиксированного приоритета, кото-

рые назначаются задачам при создании и определяют порядок, в котором следующая задача выбирается при каждом вызове планировщика супервизором.

Критические по времени операции должны быть реализованы в задачах с более высоким приоритетом. Было исследовано множество стратегий планировщика, чтобы оптимизировать время реакции задач в реальном времени, сохраняя при этом отзывчивость системы и разрешая проблемы, связанные с возможным ресурсным голоданием задач с более низким приоритетом. Найти оптимальную стратегию планирования для конкретного прикладного сценария может быть очень сложно; подробности, касающиеся детерминированного расчета задержки и джиттера системы реального времени, выходят за рамки этой книги.

Среди разработчиков операционных систем реального времени очень популярен один из вышеупомянутых подходов. Он обеспечивает немедленное переключение контекста для задач реального времени, выбирая задачу с наивысшим приоритетом среди готовых к выполнению при каждом вызове диспетчера планировщика. Эта стратегия планирования, известная как *статическое вытесняющее планирование на основе приоритетов* (static priority-driven preemptive scheduling), не является оптимальной во всех случаях, поскольку задержка выполнения задач зависит от количества задач с одним и тем же уровнем приоритета и не предусматривает механизма предотвращения потенциального голодания задач с более низким приоритетом в случае высокой загрузки системы. Но этот механизм достаточно прост, чтобы его можно было легко реализовать и продемонстрировать влияние механизма приоритетов на задержку выполнения задач в реальном времени.

Другой возможный подход состоит в динамическом переназначении приоритетов во время выполнения на основе характеристик задач. Планировщикам реального времени может пойти на пользу механизм, который гарантирует, что задача с ближайшим критическим сроком выполнения выбирается первой. Этот подход, известный как *планирование с учетом ближайшего дедлайна*, или просто EDF (earliest-deadline-first), лучше подходит для соблюдения сроков в режиме реального времени в системе с более высокой нагрузкой. Наиболее известной реализацией этого механизма является планировщик `SCHED_DEADLINE`, включенный в Linux начиная с версии 3.14. Он менее популярен во встраиваемых операционных системах, несмотря на относительно простую реализацию.

В следующем примере показана упрощенная реализация статического планировщика на основе приоритетов. Используется четыре отдельных списка для хранения активных задач, по одному для каждого уровня приоритета, поддерживаемого в системе. Уровень приоритета присваивается каждой задаче при создании, а приоритет ядра сохраняется на уровне 0, при этом его основная задача выполняется только тогда, когда все остальные задачи находятся в спящем режиме, и единственной целью ядра является проверка таймеров спящих задач. Задачи могут быть помещены в список активных задач с соответствующим уровнем приоритета, когда они станут готовы, и перемещены в список ожидания, когда они заблокированы. Для

отслеживания статического приоритета задачи в блок задач добавлено поле приоритета:

```
struct task_block {
  char name[TASK_NAME_MAXLEN];
  int id;
  int state;
  void (*start)(void *arg);
  void *arg;
  uint8_t *sp;
  uint32_t wakeup_time;
  uint8_t priority;
  struct task_block *next;
};
```

Для быстрого добавления и удаления блока задач из списка задач с одинаковым приоритетом необходимо определить две функции быстрого доступа:

```
static void tasklist_add_active(struct task_block *el)
{
  tasklist_add(&tasklist_active[el->priority], el);
}

static int tasklist_del_active(struct task_block *el)
{
  return tasklist_del(&tasklist_active[el->priority], el);
}
```

Затем их можно будет использовать в новых версиях функций `task_waiting` и `task_ready`, когда задача будет удалена или вставлена в соответствующий список активных задач с заданным приоритетом:

```
static void task_waiting(struct task_block *t)
{
  if (tasklist_del_active(t) == 0) {
    tasklist_add(&tasklist_waiting, t);
    t->state = TASK_WAITING;
  }
}
static void task_ready(struct task_block *t)
{
  if (tasklist_del(&tasklist_waiting, t) == 0) {
    tasklist_add_active(t);
    t->state = TASK_READY;
  }
}
```

В системе создаются три задачи, но та, которая будет заблокирована до нажатия кнопки, создается с более высоким уровнем приоритета:

```
void main(void) {
  clock_pll_on(0);
  led_setup();
```

```
  button_setup(button_wakeup);
  systick_enable();
  kernel.name[0] = 0;
  kernel.id = 0;
  kernel.state = TASK_RUNNING;
  kernel.wakeup_time = 0;
  kernel.priority = 0;
  tasklist_add_active(&kernel);
  task_create("test0",task_test0, NULL, 1);
  task_create("test1",task_test1, NULL, 1);
  task_create("test2",task_test2, NULL, 3);

  while(1) {
    struct task_block *t = tasklist_waiting;
    while (t) {
      if (t->wakeup_time && (t->wakeup_time < jiffies)) {
        t->wakeup_time = 0;
        task_ready(t);
      }
      t = t->next;
    }
    WFI();
  }
}
```

Функция, которая выбирает следующую задачу, переработана таким образом, чтобы она находила задачу с наивысшим приоритетом среди готовых к запуску. Для этого просматриваются списки приоритетов, от высшего к низшему. Если одна из текущих задач входит в список с наивысшим приоритетом, по возможности выбирается следующая задача на том же уровне, чтобы гарантировать механизм циклической обработки в случае конкурирующих задач с одинаковым уровнем приоритета. В любом другом случае выбирается первая задача в списке с наивысшим приоритетом:

```
static int idx;
static inline struct task_block *
tasklist_next_ready(struct task_block *t)
{
  for (idx = MAX_PRIO - 1; idx >= 0; idx--) {
     if ((idx == t->priority) && (t->next != NULL) &&
        (t->next->state == TASK_READY))
       return t->next;
     if (tasklist_active[idx])
       return tasklist_active[idx];
  }
  return t;
}
```

Основное отличие этого планировщика от планировщика с одним уровнем приоритета в плане реакции на событие нажатия кнопки в задаче с ID 2 заключается во временном интервале между событием нажатия кнопки и реакцией самой задачи. Оба планировщика реализуют вытеснение, не-

медленно переводя задачу обратно в состояние готовности в обработчике прерывания события кнопки.

Но в первом случае задача возвращается в карусель запланированных задач, чтобы конкурировать с другими задачами того же уровня приоритета, что может вызвать задержку реакции задачи. Эту задержку можно оценить как *N* * `TIMESLICE` в худшем случае, где *N* – количество процессов, готовых к запуску в момент возникновения прерывания.

При планировании на основе приоритетов в определенной степени можно быть уверенными в том, что задача реального времени является первой задачей, которая должна быть запланирована после возникновения прерывания, так что время, необходимое от прерывания до возобновления задачи, является измеримым (порядка нескольких микросекунд), поскольку ЦП выполняет предсказуемое количество инструкций для выполнения всех промежуточных действий.

Встроенные ОС реального времени имеют основополагающее значение для реализации жизненно важных систем, в основном в транспортной и медицинской отраслях. С другой стороны, они основаны на упрощенных моделях, чтобы сделать базовые системные операции как можно более легкими и с минимальными издержками для интерфейсов системных вызовов и системных API. Противоположный подход может заключаться в увеличении сложности ядра для оптимизации пропускной способности, взаимодействия задач, улучшения безопасности памяти и других показателей производительности, что лучше подходит для встраиваемых систем с более свободными или отсутствующими требованиями в работе в режиме реального времени. Более строгие политики планирования на основе приоритетов уменьшают задержку и гарантируют отклик в режиме реального времени в строгих сценариях использования, но им не всегда хватает гибкости для использования во встраиваемых системах общего назначения, где другие требования могут быть важнее, чем задержка задачи. В таких случаях планирование вытеснения на основе времени может дать лучшие результаты.

10.4. Синхронизация

В многопоточной среде, где доступ к памяти, периферийным устройствам и системе является общим, система должна предоставлять механизмы синхронизации, позволяющие задачам взаимодействовать при арбитраже доступа к общесистемным ресурсам.

Мьютексы и семафоры являются двумя наиболее часто используемыми механизмами синхронизации между параллельными потоками, поскольку они обеспечивают минимальный набор для решения большинства проблем параллельного выполнения. Функции, которые могут блокировать вызывающие задачи, должны иметь возможность взаимодействовать с планировщиком, чтобы переводить задачу в состояние ожидания всякий раз, когда ресурс недоступен, и удерживать его до снятия блокировки или увеличения семафора.

10.4.1. Семафоры

Семафор (semaphore) является наиболее распространенным базовым механизмом синхронизации, который предоставляет счетчик с монопольным доступом и используется двумя или более потоками для арбитража доступа к определенному общему ресурсу. API, предоставляемый задачам, должен гарантировать возможность использования объекта для реализации счетчика с монопольным доступом, что, как правило, требует от ЦП наличия некоторых вспомогательных возможностей. По этой причине внутренняя реализация стратегий синхронизации зависит от микрокода в целевом процессоре.

В Cortex-M3/M4 реализация механизмов блокировки зависит от инструкций, предоставляемых ЦП для выполнения эксклюзивных операций. Набор команд тестовой платформы содержит следующие две инструкции:

- **эксклюзивная загрузка регистра** (load register exclusive, LDREX): загружает значение из адреса в памяти в регистр ЦП;
- **эксклюзивное сохранение регистра** (Store Register Exclusive, STREX): пытается сохранить новое значение, содержащееся в регистре, по адресу в памяти, соответствующему последней инструкции LDREX. Если STREX завершается успешно, ЦП гарантирует, что запись значения в память была эксклюзивной и что значение не было изменено с момента последнего вызова LDREX. При наличии двух конкурирующих секций LDREX/STREX только одна приведет к успешной записи в регистр; вторая инструкция STREX завершится ошибкой и вернет ноль.

Свойства этих инструкций гарантируют эксклюзивный доступ к счетчику, который затем используется для реализации примитивных функций на базе семафоров и мьютексов.

Функция `sem_trywait` пытается уменьшить значение семафора. Операция разрешена всегда, если только значение семафора не равно `0`, что приводит к немедленному сбою. Функция возвращает `0` в случае успеха и `-1`, если значение семафора равно нулю, и в этот момент невозможно уменьшить значение семафора.

Последовательность событий в `sem_trywait` такова:

1. Значение переменной семафора (целое число, доступ к которому осуществляется с помощью эксклюзивных инструкций загрузки и сохранения) считывается из ячейки памяти, на которую указывает аргумент функции, в регистр R1.
2. Если значение R1 равно `0`, семафор не может быть запрошен, и функция возвращает `-1`.
3. Значение R1 уменьшается на единицу.
4. Значение R1 сохраняется в ячейке памяти, на которую указывает аргумент функции, а результат операции `STREX` помещается в R2.
5. Если операция выполнена успешно, R2 содержит `0`, семафор получен и успешно декрементирован, и функция может вернуть состояние успеха.

6. Если операция сохранения не удалась (предпринята попытка параллельного доступа), процедура немедленно повторяется для следующей попытки.

Вот код ассемблерной процедуры, которая реализует все вышеперечисленные шаги и возвращает `0` в случае успеха и `-1` в случае сбоя декрементирования:

```
sem_trywait:
  LDREX r1, [r0]
  CMP r1, #0
  BEQ sem_trywait_fail
  SUBS r1, #1
  STREX r2, r1, [r0]
  CMP r2, #0
  BNE sem_trywait
  DMB
  MOVS r0, #0
  BX lr
sem_trywait_fail:
  DMB
  MOV r0, #-1
  BX lr
```

Следующий код представляет собой функцию для увеличения семафора, которая похожа на процедуру ожидания, за исключением того, что вместо этого увеличивается счетчик семафора, и операция в конечном итоге будет успешной, даже если несколько задач пытаются получить доступ к семафору в одно и то же время. Функция возвращает `0` при успешном выполнении, за исключением случаев, когда значение перед увеличением было равно нулю; в этом случае она возвращает `1`, чтобы напомнить вызывающей стороне о необходимости уведомить любого слушателя в состоянии ожидания о том, что значение увеличилось и связанный ресурс теперь доступен:

```
.global sem_dopost
sem_dopost:
  LDREX r1, [r0]
  ADDS r1, #1
  STREX r2, r1, [r0]
  CMP r2, #0
  BNE sem_dopost
  CMP r0, #1
  DMB
  BGE sem_signal_up
  MOVS r0, #0
  BX lr
sem_signal_up:
  MOVS r0, #1
  BX lr
```

Чтобы интегрировать статус блокировки функции `sem_wait` в планировщик, интерфейс семафора, предоставляемый ОС задачам, обертывает неблоки-

рующий вызов sem_trywait в его блокирующую версию, которая блокирует задачу, когда значение семафора равно нулю.

Чтобы реализовать блокирующую версию интерфейса семафора, объект semaphore может отслеживать задачи, обращающиеся к ресурсам и ожидающие события post. В этом случае идентификаторы задач хранятся в массиве с именем listeners:

```
#define MAX_LISTENERS 4
struct semaphore {
  uint32_t value;
  uint8_t listeners[MAX_LISTENERS];
};

typedef struct semaphore semaphore;
```

При сбое операции ожидания задача блокируется и повторяется только после успешной операции публикации из другой задачи. Для этого ресурса в массив слушателей добавляется идентификатор задачи:

```
int sem_wait(semaphore *s)
{
  int i;
  if (s == NULL)
    return -1;
  if (sem_trywait(s) == 0)
    return 0;
  for (i = 0; i < MAX_LISTENERS; i++) {
    if (!s->listeners[i])
      s->listeners[i] = t_cur->id;
    if (s->listeners[i] == t_cur->id)
      break;
  }
  task_waiting(t_cur);
  schedule();
  return sem_wait(s);}
```

Ассемблерная подпрограмма sem_dopost возвращает положительное значение, если операция post инициировала приращение от нуля до единицы; это означает, что прослушиватели, если они есть, должны быть возобновлены, чтобы попытаться получить ресурс, который только что стал доступным.

10.4.2. Мьютексы

Термин «мьютекс» (mutex) – это сокращение от *взаимного исключения* (mutual exlusion), и этот механизм настолько тесно связан с семафором, что его можно реализовать с помощью тех же ассемблерных процедур. Мьютекс – это не что иное, как двоичный семафор, который инициализируется значением 1, чтобы разрешить первую операцию блокировки.

Из-за свойства семафора, которое не дает возможности уменьшить счетчик после того, как его значение достигает 0, в нашей быстрой реализации

интерфейса мьютекса примитивы семафора `sem_wait` и `sem_post` переименованы в `mutex_lock` и `mutex_unlock` соответственно.

Две задачи могут одновременно попытаться уменьшить незаблокированный мьютекс, но только одна из них добьется успеха; другая потерпит неудачу. В блокирующей версии мьютекса для примера планировщика обертки для API мьютекса, построенные поверх функций семафора, будут такими:

```
typedef semaphore mutex;
#define mutex_init(m) sem_init(m, 1)
#define mutex_trylock(m) sem_trywait(m)
#define mutex_lock(x) sem_wait(x)
#define mutex_unlock(x) sem_post(x)
```

Пример операционной системы, разработанный нами ранее, предлагает полный API механизмов синхронизации, интегрированных с планировщиком, как для семафоров, так и для мьютексов.

10.4.3. Инверсия приоритета

Явление, которое часто встречается при разработке операционных систем с вытесняющими планировщиками на основе приоритетов, использующими встроенные механизмы синхронизации, – это *инверсия приоритетов* (priority inversion). Это явление влияет на скорость реакции задач реального времени, которые совместно используют ресурсы с другими задачами с более низким приоритетом, и в некоторых случаях может привести к тому, что задачи с более высоким приоритетом не будут выполняться в течение непредсказуемого периода времени. Так получается, когда задача с высоким приоритетом ожидает освобождения ресурса задачей с более низким приоритетом, которая тем временем может быть вытеснена другими несвязанными задачами в системе.

В частности, пример последовательности событий, которые могут спровоцировать это явление, выглядит следующим образом:

1. T1, T2 и T3 – три запущенные задачи с приоритетом 1, 2 и 3 соответственно.
2. T1 получает блокировку с помощью мьютекса на ресурсе X.
3. T1 вытесняется T3, которая имеет более высокий приоритет.
4. T3 пытается получить доступ к общему ресурсу X и блокируется мьютексом.
5. T1 возобновляет выполнение в критической секции.
6. T1 вытесняется T2, который имеет более высокий приоритет.
7. Произвольное количество задач с приоритетом больше 1 может прервать выполнение задачи T1, прежде чем она сможет снять блокировку с ресурса и разбудить T3.

Один из возможных механизмов, который можно реализовать, чтобы избежать этой ситуации, называется *наследованием приоритетов* (priority inheritance). Этот механизм состоит во временном повышении приоритета

задачи, совместно использующей ресурс, до наивысшего приоритета среди всех задач, обращающихся к ресурсу. Таким образом, задача с более низким приоритетом не вызывает задержек планирования для задач с более высоким приоритетом, а требования реального времени по-прежнему выполняются.

10.5. Разделение системных ресурсов

Пример операционной системы, которая была реализована в этой главе, уже имеет много интересных функций, но по-прежнему характеризуется плоской моделью без сегментации памяти или разделения привилегий. Минималистские системы не предоставляют каких-либо механизмов для разделения системных ресурсов и регулирования доступа к памяти. Вместо этого задачам в системе разрешено выполнять любые привилегированные операции, включая чтение и изменение памяти других задач, выполнение операций в адресном пространстве ядра и прямой доступ к периферийным устройствам и регистрам ЦП во время выполнения.

На нашей тестовой платформе доступны различные подходы, направленные на повышение уровня безопасности в системе путем внесения небольшого количества модификаций ядра, что позволит:

- реализовать разделение привилегий ядра и процесса;
- интегрировать защиту памяти в планировщик;
- обеспечить интерфейс системного вызова через вызов супервизора для доступа к ресурсам.

Давайте подробно обсудим каждый из этих пунктов.

10.5.1. Уровни привилегий

Процессор Cortex-M предназначен для запуска кода с двумя разными уровнями привилегий. Разделение привилегий требуется всякий раз, когда в системе выполняется ненадежный код приложения, позволяя ядру постоянно контролировать выполнение и предотвращать системные сбои из-за неправильного поведения пользовательского потока. Уровень выполнения по умолчанию при загрузке – привилегированный, чтобы ядро могло загрузиться. Приложения можно настроить для выполнения на уровне пользователя и использования другого регистра указателя стека во время операций переключения контекста.

Изменение уровней привилегий возможно только во время выполнения обработчика исключений, и это делается с помощью специального значения, возвращаемого исключением, которое хранится в LR до того, как оно будет возвращено обработчиком исключений, выполнившим переключение контекста. Флаг, управляющий уровнем привилегий, является младшим битом регистра `CONTROL`, который можно изменить во время переключения контекс-

та перед возвратом из обработчика исключений, чтобы перевести потоки приложений на уровень привилегий пользователя.

Более того, большинство ЦП Cortex-M предоставляют два отдельных регистра указателя стека:

- главный указатель стека (main stack pointer, MSP);
- указатель стека процессов (process stack pointer, PSP).

В соответствии с рекомендациями ARM операционные системы должны использовать PSP для выполнения пользовательских потоков, тогда как MSP предназначен для обработчиков прерываний и ядра. Выбор стека зависит от специального возвращаемого значения в конце обработчика исключений. В планировщике, который был реализован до сих пор, жестко запрограммировано значение `0xFFFFFFF9`, которое используется для возврата в режим потока после прерывания и продолжения выполнения кода на привилегированном уровне. Возвращаемое обработчиком прерывания значение 0xFFFFFFFD указывает ЦП выбрать PSP в качестве регистра указателя стека при возврате в режим потока.

Чтобы корректно реализовать разделение привилегий, в обработчик PendSV, используемый для переключения задач, необходимо внести изменения для сохранения и восстановления контекста с использованием правильного указателя стека для вытесняемой задачи и выбранного стека. Функции `store_context` и `restore_context`, которые использовались до сих пор, переименуем в `store_kernel_context` и `restore_kernel_context` соответственно, поскольку ядро по-прежнему использует главный указатель стека. В обновленную процедуру переключения контекста добавим две новые функции, которые используют регистр PSP для хранения и восстановления контекстов потоков:

```
static void __attribute__((naked)) store_user_context(void)
{
  asm volatile("mrs r0, psp");
  asm volatile("stmdb r0!, {r4-r11}");
  asm volatile("msr psp, r0");
  asm volatile("bx lr");
}
static void __attribute__((naked)) restore_user_context(void)
{
  asm volatile("mrs r0, psp");
  asm volatile("ldmfd r0!, {r4-r11}");
  asm volatile("msr psp, r0");
  asm volatile("bx lr");
}
```

В безопасной версии планировщика служебная подпрограмма PendSV выбирает правильный указатель стека для сохранения и восстановления контекста и вызывает связанные подпрограммы. В зависимости от нового контекста возвращаемое значение, хранящееся в LR, применяется для выбора регистра, используемого в качестве нового указателя стека, а пользовательский или привилегированный уровень устанавливается в регистре `CONTROL` (значения `1` или `0` соответственно):

```
void __attribute__((naked)) isr_pendsv(void)
{
  if (t_cur->id == 0) {
    store_kernel_context();
    asm volatile("mrs %0, msp" : "=r"(t_cur->sp));
  } else {
    store_user_context();
    asm volatile("mrs %0, psp" : "=r"(t_cur->sp));
  }
    if (t_cur->state == TASK_RUNNING) {
      t_cur->state = TASK_READY;
  }
  t_cur = tasklist_next_ready(t_cur);
  t_cur->state = TASK_RUNNING;
  if (t_cur->id == 0) {
    asm volatile("msr msp, %0" ::"r"(t_cur->sp));
    restore_kernel_context();
    asm volatile("mov lr, %0" ::"r"(0xFFFFFFF9));
    asm volatile("msr CONTROL, %0" ::"r"(0x00));
  } else {
    asm volatile("msr psp, %0" ::"r"(t_cur->sp));
    restore_user_context();
    asm volatile("mov lr, %0" ::"r"(0xFFFFFFFD));
    asm volatile("msr CONTROL, %0" ::"r"(0x01));
  }
  asm volatile("bx lr");
}
```

Задача, запущенная с установленным битом привилегированного режима в регистре CONTROL, имеет ограниченный доступ к ресурсам системы. В частности, потоки не могут получить доступ к регистрам в области SCB, а это означает, что некоторые базовые операции, такие как включение и отключение прерываний через NVIC, зарезервированы для исключительного использования ядром. При использовании в сочетании с MPU разделение привилегий еще больше повышает безопасность системы за счет разделения памяти на уровне доступа, что позволяет обнаруживать и прерывать некорректно работающий код приложения.

10.5.2. Сегментация памяти

В планировщик можно интегрировать стратегии динамической сегментации памяти, чтобы гарантировать, что отдельные задачи не будут обращаться к областям памяти, связанным с критически важными для системы компонентами, и что ресурсы, требующие контроля ядра, могут быть доступны из пользовательского пространства.

В главе 5 было показано, как можно использовать MPU для разграничения смежных сегментов памяти и запрета доступа к определенным областям для любого кода, работающего в системе. Контроллер MPU предоставляет маску разрешений для более детального изменения атрибутов областей па-

мяти. В частности, можно разрешить доступ к некоторым областям только в том случае, если ЦП работает на привилегированном уровне, что эффективно предотвращает доступ пользовательских приложений к определенным областям системы без контроля ядра. Безопасная операционная система может принять решение полностью исключить доступ приложений к периферийной области, и система регистрируется, используя для этих областей флаги разрешений «только ядро». Значения, связанные с конкретными разрешениями в регистре атрибутов региона MPU, могут быть определены следующим образом:

```
#define RASR_KERNEL_RW (1 << 24)
#define RASR_KERNEL_RO (5 << 24)
#define RASR_RDONLY (6 << 24)
#define RASR_NOACCESS (0 << 24)
#define RASR_USER_RW (3 << 24)
#define RASR_USER_RO (2 << 24)
```

Ядро может применить конфигурацию MPU при загрузке. В следующем примере установим глобально доступную для чтения область флеш-памяти как область `0`, используя `RASR_RDONLY`, а область SRAM, доступную глобально, как область `1`, сопоставленную с адресом `0x20000000`:

```
int mpu_enable(void)
{
  volatile uint32_t type;
  volatile uint32_t start;
  volatile uint32_t attr;
  type = MPU_TYPE;
  if (type == 0)
    return -1;
  MPU_CTRL = 0;
  start = 0;
  attr = RASR_ENABLED | MPUSIZE_256K | RASR_SCB | RASR_RDONLY;
  mpu_set_region(0, start, attr);
  start = 0x20000000;
  attr = RASR_ENABLED | MPUSIZE_128K | RASR_SCB | RASR_USER_RW | RASR_NOEXEC;
  mpu_set_region(1, start, attr);
```

Более строгая политика безопасности может даже ограничить использование SRAM пользовательскими задачами в непривилегированном режиме, но для этого потребуется реорганизация регионов `.data` и `.bss`, которые сопоставляются при запуске задачи. В следующем примере показано, как интегрировать политику защиты памяти для каждой задачи в планировщик, чтобы предотвратить доступ к системным ресурсам и защитить области стека других задач. Цель: защитить область CCRAM, поскольку она содержит стек выполнения ядра, а также стек других задач в системе. Для этого область CCRAM должна быть помечена как область 2 с исключительным доступом ядра. Позже для выбранной задачи во время переключения контекста должно быть создано исключение, чтобы разрешить доступ к собственному пространству стека:

```
start = 0x10000000;
attr = RASR_ENABLED | MPUSIZE_64K | RASR_SCB | RASR_KERNEL_RW | RASR_NOEXEC;
mpu_set_region(2, start, attr);
```

Периферийные регионы и системные регистры являются областями ограниченного доступа в нашей системе, поэтому они также отмечены для монопольного доступа ядра во время выполнения. В нашей безопасной архитектуре ОС задачи, которые хотят получить доступ к периферийным устройствам, должны использовать системные вызовы для выполнения контролируемых привилегированных операций:

```
start = 0x40000000;
attr = RASR_ENABLED | MPUSIZE_1G | RASR_SB | RASR_KERNEL_RW | RASR_NOEXEC;
mpu_set_region(4, start, attr);
start = 0xE0000000;
attr = RASR_ENABLED | MPUSIZE_256M | RASR_SB | RASR_KERNEL_RW | RASR_NOEXEC;
mpu_set_region(5, start, attr);
SHCSR |= MEMFAULT_ENABLE;
MPU_CTRL = 1;
return 0;
}
```

Во время переключения контекста, непосредственно перед возвратом из служебной подпрограммы `isr_pendsv`, планировщик может вызвать функцию, экспортируемую нашим пользовательским модулем MPU, чтобы временно разрешить доступ к области стека задаче, выбранной для выполнения следующей в непривилегированном режиме:

```
void mpu_task_stack_permit(void *start)
{
  uint32_t attr = RASR_ENABLED | MPUSIZE_1K | RASR_SCB | RASR_USER_RW;
  MPU_CTRL = 0;
  DMB();
  mpu_set_region(3, (uint32_t)start, attr);
  MPU_CTRL = 1;
}
```

Эти дополнительные мероприятия ограничили возможности прямого доступа к любым ресурсам для реализованных в настоящее время задач. Чтобы сохранить те же функции, что и раньше, эта демонстрационная система теперь должна экспортировать новый безопасный API для задач, запрашивающих системные операции.

10.5.3. Системные вызовы

Последняя версия операционной системы, которая была разработана в этой главе, больше не позволяет задачам управлять системными ресурсами, такими как периферийные устройства ввода и вывода, и даже не позволяет задачам блокироваться добровольно, поскольку функции `sleep_ms` не раз-

решено устанавливать флаг ожидания, чтобы инициировать переключение контекста.

Операционная система экспортирует API, доступный для задач, через исключение системных вызовов SVCall, которое обрабатывается служебной подпрограммой `isr_svc` и запускается в любое время из задач с помощью инструкции `svc`.

В следующем простом примере показано использование ассемблерной инструкции `svc 0` для переключения в режим обработчика, определив макрос быстрого доступа `SVC()`:

```
#define SVC() asm volatile ("svc 0")
```

Эта инструкция обернута в функцию C, чтобы иметь возможность передавать ей аргументы. API для тестовой платформы предоставляет первые четыре аргумента вызова через переключатель режима внутри регистров R0–R3. API не позволяет передавать какие-либо аргументы системным вызовам, но использует первый аргумент в R0 для идентификации запроса, переданного из приложения в ядро:

```
static int syscall(int arg0)
{
  SVC();
}
```

Реализован весь интерфейс системных вызовов для операционной системы, который состоит из перечисленных ниже системных вызовов без аргументов. С каждым системным вызовом связан идентификационный номер, передаваемый как `arg0`. Список системных вызовов – это контракт на интерфейс между задачами и ядром и единственный способ для задач использовать защищенные ресурсы в системе:

```
#define SYS_SCHEDULE 0
#define SYS_BUTTON_READ 1
#define SYS_BLUELED_ON 2
#define SYS_BLUELED_OFF 3
#define SYS_BLUELED_TOGGLE 4
#define SYS_REDLED_ON 5
#define SYS_REDLED_OFF 6
#define SYS_REDLED_TOGGLE 7
#define SYS_GREENLED_ON 8
#define SYS_GREENLED_OFF 9
#define SYS_GREENLED_TOGGLE 10
```

Каждый из этих системных вызовов должен обрабатываться в `isr_svc`. Управление периферийными устройствами и регистрами системного блока может быть выполнено путем вызова функций драйвера в контексте обработчика, хотя здесь это делается только для краткости. В правильно разработанной системе операции, для выполнения которых требуется более не-

скольких инструкций, должны быть отложены для выполнения задачей ядра в следующий раз, когда она будет запланирована. Код следующего примера предназначен только для демонстрации возможной реализации isr_svc, которая реагирует на пользовательские запросы, разрешенные системным API, для управления светодиодом и кнопкой на плате, а также предоставляет механизм, который можно расширить для реализации блокировки системных вызовов.

Служебная процедура svc выполняет запрошенную команду, переданную в качестве аргумента самому обработчику. Если системный вызов является блокирующим, как, например, системный вызов SYS_SCHEDULE, то новая задача выбирается для завершения переключения в обработчике.

Подпрограмма svc теперь может обрабатывать внутренние команды, как показано в следующем примере функции-обработчика:

```
void __attribute__((naked)) isr_svc(int arg)
{
  store_user_context();
  asm volatile("mrs %0, psp" : "=r"(t_cur->sp));
  if (t_cur->state == TASK_RUNNING) {
    t_cur->state = TASK_READY;
  }
  switch(arg) {
    case SYS_BUTTON_READ: /* команда чтения кнопки */
      button_start_read();
      break;
    case SYS_SCHEDULE: /* команда планирования след. задачи */
      t_cur = tasklist_next_ready(t_cur);
      t_cur->state = TASK_RUNNING;
      break;
    case SYS_BLUELED_ON: /* команда включения синего светодиода */
      blue_led_on();
      break;
    /* case ... (другие светодиоды и другие команды) */
}
```

Контекст возобновляется в конце подпрограммы так же, как и в PendSV. Хотя это необязательно, если вызов должен быть блокирующим, может произойти переключение задач:

```
  if (t_cur->id == 0) {
      asm volatile("msr msp, %0" ::"r"(t_cur->sp));
      restore_kernel_context();
      asm volatile("mov lr, %0" ::"r"(0xFFFFFFF9));
      asm volatile("msr CONTROL, %0" ::"r"(0x00));
  } else {
      asm volatile("msr psp, %0" ::"r"(t_cur->sp));
      restore_user_context();
      mpu_task_stack_permit(((uint8_t *)((&stack_space))
```

```
          +(t_cur->id << 10))); asm volatile("mov lr, %0" ::"r"(0xFFFFFFFD));
      asm volatile("msr CONTROL, %0" ::"r"(0x01));
  }
  asm volatile("bx lr");}
```

Несмотря на ограниченные функциональные возможности, новая система экспортирует все API, необходимые для повторного запуска потоков нашего приложения. Для этого необходимо удалить все запрещенные вызовы из кода задач и вставить вместо них вновь созданные системные вызовы:

```
void task_test0(void *arg)
{
  while(1) {
    syscall(SYS_BLUELED_ON);
    mutex_lock(&m);
    sleep_ms(500);
    syscall(SYS_BLUELED_OFF);
    mutex_unlock(&m);
    sleep_ms(1000);
  }
}

void task_test1(void *arg)
{
  syscall(SYS_REDLED_ON);
  while(1) {
    sleep_ms(50);
    mutex_lock(&m);
    syscall(SYS_REDLED_TOGGLE);
    mutex_unlock(&m);
  }
}

void task_test2(void *arg)
{
  uint32_t toggle_time = 0;
  syscall(SYS_GREENLED_OFF);
  while(1) {
    button_read();
    if ((jiffies - toggle_time) > 120) {
      syscall(SYS_GREENLED_TOGGLE);
      toggle_time = jiffies;
    }
  }
}
```

Размер кода безопасной операционной системы может значительно увеличиться, если она реализует все операции в пространстве ядра и должна обеспечивать реализацию всех разрешенных системных вызовов. С другой стороны, это обеспечивает разделение физической памяти между задачами и защищает системные ресурсы и другие области памяти от случайных ошибок в коде приложения.

10.6. Встраиваемые операционные системы

Как было показано в предыдущих разделах этой главы, разработка собственного планировщика, адаптированного к задачам пользователя, – вполне решаемая задача, и если все сделано правильно, такой планировщик обеспечит наилучшее приближение к желаемой архитектуре и будет учитывать конкретные характеристики целевого устройства. Но в реальной жизни рекомендуется начать с изучения различных вариантов готовых встраиваемых операционных систем, выбрав те из них, которые поддерживают нужную аппаратную платформу и предоставляют функции, упомянутые в этой главе.

Многие из доступных реализаций ядра ОС для микроконтроллеров имеют открытый исходный код и находятся в здоровом состоянии активной разработки, поэтому они вполне заслуженно обладают популярностью и хорошей репутацией на рынке встраиваемых систем. Некоторые из них достаточно широко распространены и протестированы, поэтому их можно брать за основу для создания надежных встроенных многозадачных приложений.

10.6.1. Выбор операционной системы

Выбор операционной системы, которая наилучшим образом соответствует поставленной цели и платформе разработки, является деликатной задачей, которая влияет на общую архитектуру, может отразиться на процессе проектирования и даже привести к недоступности API в кодовой базе приложения. Критерии выбора зависят от характеристик оборудования, интеграции с другими компонентами, такими как сторонние библиотеки, возможностей, предлагаемых для взаимодействия с периферийными устройствами и интерфейсами, и, что наиболее важно, диапазона вариантов использования, для которых разработана система.

За некоторыми исключениями, операционные системы включают в себя, помимо планировщика и управления памятью, набор интегрированных библиотек, модулей и инструментов. В зависимости от назначения функциональность встраиваемой ОС может охватывать несколько областей, включая следующие:

- уровни аппаратной абстракции для конкретных платформ;
- драйверы для обычных периферийных устройств;
- интеграция стека TCP/IP для подключенных устройств;
- файловые системы и файловые абстракции;
- интегрированные системы управления питанием.

В зависимости от реализации потоковой модели в планировщике некоторые системы работают с фиксированным количеством предопределенных задач, настроенных во время компиляции, в то время как другие выбирают более сложные процессы и иерархии потоков, которые позволяют разработчикам создавать и завершать новые потоки в любой момент во время выполнения. Но динамическое создание и завершение задач редко применяются во

встраиваемых системах, и в большинстве случаев существуют более простые альтернативные решения.

Более сложные системы вносят дополнительные накладные расходы из-за развитой логики в коде системных исключений и менее подходят для критических операций в реальном времени, поэтому большинство успешных RTOS в настоящее время сохраняют свою простую архитектуру, предоставляя необходимый минимум для запуска нескольких потоков в режиме плоской памяти, который прост в управлении и не требует дополнительных переключений контекста для управления привилегиями операций, а также обеспечивает измеримую низкую задержку для соответствия требованиям к системам реального времени.

Из-за множества доступных опций и непрерывного развития кода вслед за прогрессом оборудования предоставление исчерпывающего списка операционных систем для встраиваемых устройств выходит за рамки этой книги. В отличие от отрасли персональных компьютеров, где на рынке доминирует всего несколько операционных систем, встраиваемые ОС сильно отличаются друг от друга с точки зрения их архитектуры, API, драйверов, поддерживаемого оборудования и инструментов сборки.

В завершающем разделе этой главы рассмотрены две самые популярные операционные системы с открытым исходным кодом для встраиваемых устройств – FreeRTOS и Riot. Проведено сравнение их подходов к параллельному выполнению задач и управлению памятью, а также имеющиеся дополнительные функции.

10.6.2. FreeRTOS

Возможно, самая популярная среди операционных систем с открытым исходным кодом для встраиваемых устройств, на момент создания которой прошло около 20 лет активной разработки, FreeRTOS часто переносится на многие встроенные платформы, имеет десятки различных аппаратных версий и обеспечивает поддержку большинства, если не всех, архитектур процессоров.

Разработанная с прицелом на небольшой объем кода и простые интерфейсы, эта система не предлагает полные наборы драйверов или расширенные функции специально для конкретных ЦП, а фокусируется на двух вещах: планировании потоков в реальном времени и управлении динамической памятью. Простота архитектуры этой ОС позволяет переносить ее на большое количество платформ, а разработчики могут сосредоточиться на ограниченном наборе хорошо протестированных и надежных операций.

В свою очередь, производители оборудования часто интегрируют FreeRTOS в свой комплект программного обеспечения, добавляя сторонние библиотеки и примеры кода, в большинстве случаев как единственную альтернативу подходу на «голом железе», особенно для тестовых приложений и примеров. Поскольку сторонний код не включен напрямую в FreeRTOS, это способствует конкуренции между различными решениями. Например, FreeRTOS можно объединить со многими реализациями стека TCP/IP для поддержки сетевых

подключений, даже если ни одно из них не является частью ядра системы. Драйверы устройств не входят в ядро, но есть несколько примеров полных систем, основанных на интеграции FreeRTOS с пакетами поддержки плат, распространяемыми как производителями, так и частью более широкой экосистемы, где FreeRTOS включена в архитектуру как ядро.

Планировщик работает с вытеснением, с фиксированными уровнями приоритета и наследованием приоритетов через общие мьютексы. Уровни приоритета и размеры стека для всех потоков определяются при создании потока. Типичное приложение FreeRTOS начинается со своей функции `main`, которая отвечает за инициализацию потоков и запуск планировщика. Новую задачу можно создать с помощью функции `xTaskCreate`:

```
xTaskCreate(task_entry_fn, "TaskName", task_stack_size,
      ( void * ) custom_params, priority, task_handle);
```

Первый параметр – это указатель на функцию `main`, которая будет точкой входа для задачи. Когда вызывается точка входа задачи, настраиваемые параметры, указанные здесь как четвертый параметр, будут переданы в качестве единственного аргумента функции, что позволит нам совместно использовать пользовательский параметр с потоком при создании задачи. Второй аргумент `xTaskCreate` – это просто имя задачи в виде строки текста, которая используется в целях отладки. Третий и пятый параметры определяют размер стека и приоритет для этой задачи соответственно. Наконец, последний аргумент – это необязательный указатель на внутреннюю структуру задачи, которая будет заполнена при возврате `xTaskCreate`, если был предоставлен допустимый указатель. Этот объект типа `TaskHandle_t` требуется для доступа к некоторым функциям задачи, таким как уведомления о задаче или общие утилиты задачи.

После того как приложение создало свои задачи, функция `main` вызывает основной планировщик при помощи следующей команды:

```
vTaskStartScheduler();
```

Если все идет хорошо, эта функция никогда не возвращается, и главной функцией приложения становится фактическая задача ядра, которая отвечает за планирование задач, определенных ранее, и новых задач, которые могут быть добавлены позже.

Одной из самых интересных функций, предлагаемых FreeRTOS, является управление динамической памятью, которое доступно в пяти вариантах, оптимизированных для различных сценариев:

- **Heap 1**: разрешается только однократное статическое выделение памяти в куче без возможности освобождения памяти. Это полезно, если приложения могут с самого начала занять все необходимое пространство, поскольку дальнейшее выделение памяти в системе будет невозможно;
- **Heap 2**: позволяет освободить память, но не выполняет повторную сборку освобожденных блоков. Этот механизм подходит для реализаций с ограниченным количеством фрагментов кучи, особенно если

они сохраняют тот же размер, что и ранее освобожденные объекты. При неправильном использовании эта модель может привести к сильно фрагментированному стеку с риском исчерпания кучи в долгосрочной перспективе из-за отсутствия реорганизации памяти, даже если общий размер выделенного объекта не увеличивается;

- **Heap 3**: этот метод представляет собой обертку для реализации функционала `malloc/free`, предоставляемого сторонней библиотекой, которая гарантирует, что операции с обернутой памятью станут потокобезопасными при использовании в контексте многопоточности FreeRTOS. Эта модель позволяет нам определить пользовательский метод управления памятью, определив функцию `malloc/free` в отдельной модели или используя библиотечную реализацию и присоединив системный вызов `sbrk()`, как показано в главе 5;
- **Heap 4**: это более продвинутый менеджер памяти с поддержкой объединения памяти. Смежные блоки `free` объединяются, и выполняется определенное наведение порядка, чтобы оптимизировать использование кучи при гетерогенных выделениях из разных потоков. Этот метод уменьшает фрагментацию кучи и улучшает использование памяти в долгосрочной перспективе;
- **Heap 5**: этот метод использует тот же механизм, что и Heap 4, но позволяет определить несколько несмежных областей памяти как часть одного и того же пространства кучи. Этот метод представляет собой готовое к использованию решение для физической фрагментации, при условии что регионы определены во время инициализации и предоставлены системе через доступный API.

Выбор конкретной модели кучи сводится к включению в сборку одного из доступных исходных файлов, определяющих одни и те же функции с разными реализациями. Эти файлы являются частью дистрибутива FreeRTOS под узнаваемыми именами (`heap_1.c`, `heap_2.c` и т. д.). Нужно выбрать только один файл, и он должен быть связан с конечным приложением для управления памятью.

Важные функции, предоставляемые диспетчером динамической памяти в FreeRTOS, – это `pvPortMalloc` и `pvPortFree`; обе они имеют такие же сигнатуры и значение, как функции `malloc` и `free`, о которых рассказывалось в главе 5.

Доступна поддержка режимов MPU и потока, а потоки могут выполняться в ограниченном режиме, когда доступна только память, назначенная конкретному потоку. При выполнении потоков в ограниченном режиме системный API по-прежнему доступен, поскольку системные функции отображаются в определенной области памяти. Основная стратегия безопасности состоит в преднамеренном переводе задач в ограниченный режим и определении границ доступа к памяти, позволяя задаче обращаться только к своему собственному стеку и не более чем к трем настраиваемым областям в отображаемой памяти.

Управление низким энергопотреблением ограничено спящим режимом, и по умолчанию механизм глубокого сна не реализован. Но система позволяет нам переопределить функции обратного вызова планировщика для входа в настраиваемые режимы с низким энергопотреблением, которые можно

использовать в качестве отправных точек для реализации индивидуальных стратегий энергосбережения.

Последние версии FreeRTOS содержат дистрибутивы со сторонним кодом, которые можно взять за основу при создании защищенной подключенной платформы для систем IoT. Те же авторы создали стек TCP/IP, предназначенный для FreeRTOS, и он распространяется в пакете FreeRTOS Plus вместе с ядром и библиотекой wolfSSL для поддержки защищенной связи через сокеты.

10.6.3. Riot

В основном построенные на основе ограниченных микроконтроллеров, таких как Cortex-M0, встраиваемые системы с низким энергопотреблением часто представляют собой небольшие устройства с питанием от батареи или с извлечением энергии из окружающей среды, время от времени подключающиеся к удаленным службам с использованием беспроводных технологий. Эти небольшие недорогие системы обычно используются в проектах интернета вещей и в других подобных сценариях «установил и забыл», где они могут годами работать от одного встроенного источника питания практически без затрат на обслуживание.

В таких вариантах использования по-прежнему популярны архитектуры без ОС. Тем не менее было разработано несколько очень легких операционных систем для организации и синхронизации задач с использованием как можно меньшего количества ресурсов, с акцентом на энергосбережение и подключенность. Главная задача при разработке такой операционной системы состоит в том, чтобы найти способ разместить сложные сетевые протоколы в нескольких килобайтах памяти. Перспективные системы, разработанные для служб IoT, предлагают как собственные сети IPv6, часто через 6LoWPAN, так и полностью функциональные, но минималистские стеки TCP/IP, разработанные для того, чтобы пожертвовать пропускной способностью в пользу экономии объема памяти.

Из-за небольшого размера кода в этих системах могут отсутствовать некоторые дополнительные функции. Например, они могут не использовать стратегии безопасности памяти или иметь ограниченный стек подключения для экономии ресурсов. Нередко такие системы оснащены только сетевым стеком UDP.

Операционная система Riot имеет быстрорастущее сообщество энтузиастов и профессиональных разработчиков. Цель проекта – создать встраиваемую систему, рассчитанную на низкое энергопотребление, с учетом требований по интеграции устройства в более крупные распределенные системы. Базовая система очень масштабируема, так как отдельные компоненты могут быть исключены во время компиляции.

Подход, используемый в ОС Riot, отличается от минималистской концепции, которая реализована в FreeRTOS, где ядро операционной системы занимает минимальный объем, а все остальное интегрировано в виде внешних компонентов. Riot предлагает широкий выбор библиотек и кода поддержки

устройств, включая сетевые стеки и драйверы беспроводной связи, что делает эту систему особенно удобной для IoT. Компоненты, которые не являются частью основных функций ОС, выделены в дополнительные модули с настраиваемой системой сборки на основе make-файлов, предназначенной для облегчения включения модулей в приложение.

С точки зрения API выбор сообщества Riot – это попытка максимально имитировать интерфейс POSIX. Такой подход облегчает разработку встроенных приложений программистами с разным опытом и позволяет использовать API стандартного языка C для доступа к ресурсам в системе. Но система по-прежнему основана на плоской модели. Разделение привилегий не реализовано на системном уровне, и приложения пользовательского пространства по-прежнему должны получать доступ к системным ресурсам, напрямую обращаясь к системной памяти.

В качестве дополнительной меры безопасности можно использовать MPU для обнаружения переполнения стека в отдельных потоках путем размещения небольшой области только для чтения в нижней части стека, которая вызывает исключение, если потоки пытаются записывать за пределами отведенного им пространства стека.

Riot реализует несколько коммуникационных стеков в виде модулей, в том числе минимальный IP-стек под названием GNRC. Это реализация только для IPv6, адаптированная к функциям базовой сети 802.15.4 и обеспечивающая механизм сокетов для разработки облегченных приложений IoT. Поддержка сети включает совместимость с lwIP. lwIP включен в качестве модуля для предоставления более полных реализаций TCP/IP, когда это необходимо. WolfSSL также доступен в виде модуля для защиты связи через сокеты с использованием последней версии TLS, а еще использования криптографических функций, например для защиты данных в дежурном режиме.

Одной из возможностей Riot является доступ к настройке режимов пониженного энергопотребления, который интегрирован в систему через модуль управления питанием. Этот модуль обеспечивает абстракцию для управления специфичными для платформы функциями, такими как режимы остановки и ожидания на платформах Cortex-M. Режимы с низким энергопотреблением можно активировать во время выполнения из кода приложения, что упрощает интеграцию стратегий с низким энергопотреблением в архитектуру. Для возврата в нормальный режим работы применяются часы реального времени, сторожевой таймер или другие внешние сигналы.

Планировщик в Riot не использует системные такты SysTick и основан преимущественно на совместном режиме. Задачи могут приостанавливаться явным образом, вызывая функцию `task_yield` или любую из блокирующих функций для доступа к функциям ядра (таким как IPC и таймеры) и аппаратным периферийным устройствам. ОС Riot не применяет какой-либо параллелизм на основе временных интервалов; задача принудительно прерывается в случае получения аппаратного прерывания. Программирование приложений с помощью этого планировщика требует особого внимания, потому что случайное создание цикла занятости в одной задаче может привести к голоданию всей системы.

Задачи в Riot OS можно создавать через функцию `thread_create`:

```
thread_create(task_stack_mem, task_stack_size, priority,
    flags, task_entry_fn, (void*)custom_args, "TaskName");
```

Хотя синтаксис `thread_create` похож на синтаксис эквивалентной функции в FreeRTOS, можно отметить несколько различий в подходе к двум планировщикам. В Riot, например, память, зарезервированная для пространства стека создаваемой задачи, должна быть выделена вызывающей стороной. Пространство стека не может быть автоматически выделено при создании задачи, что означает больше кода в вызывающей программе, но также и большую гибкость для настройки расположения каждого пространства стека в памяти. Как было показано ранее, планировщик не использует системные такты, поэтому нет необходимости запускать его вручную. Задачи можно создавать и останавливать в любой момент их выполнения.

В операционной системе Riot не рекомендуется использовать динамически выделяемую память, поскольку эта ОС разработана для встраиваемых систем с небольшим объемом доступной оперативной памяти. Но система предлагает три разных подхода к управлению динамически выделяемой памятью (кучей):

- **однократное статическое выделение**: подобно модели Heap 1 в FreeRTOS, этот распределитель не предлагает никаких средств для освобождения или повторного использования областей памяти. Однажды выделенная память резервируется и никогда не освобождается. По умолчанию функция `malloc` использует эту реализацию, а функция `free` не действует;
- **выделение массива памяти**: в качестве пула памяти для псевдодинамических запросов на выделение фиксированного, предопределенного размера может использоваться статически выделенный буфер. Этот распределитель может быть полезен в тех случаях, когда приложение обрабатывает несколько буферов одинакового размера. Он имеет собственный API и не изменяет поведение функции `malloc` по умолчанию;
- **распределитель с двухуровневой раздельной подгонкой** (two-level segregate fit, TLSF): этот распределитель поддерживает несколько пулов памяти на основе алгоритма, оптимизированного для RTOS, и обеспечивает поддержку динамической памяти при работе с дедлайнами в реальном времени. Поддержка TLSF `malloc` доступна в качестве дополнительного модуля. При компиляции модуль заменяет функции `malloc` и `free`, предоставляемые однократным распределителем, который затем отключается.

Операционная система Riot – интересный выбор в качестве отправной точки для устройств IoT. Она предлагает широкий спектр драйверов устройств и модулей, созданных и интегрированных поверх легкой и энергоэффективной базовой системы, включая микроядро с упреждающим планировщиком.

10.7. Заключение

В этой главе были показаны типичные компоненты встраиваемой ОС, разработав ее с нуля с единственной целью изучения внутреннего устройства системы – как различные механизмы могут быть интегрированы в планировщик, как блокировать вызовы, как построен API драйверов и как работают механизмы синхронизации задач.

Затем были проанализированы компоненты двух наиболее популярных операционных систем реального времени с открытым исходным кодом для встраиваемых микроконтроллеров, а именно FreeRTOS и Riot, чтобы выделить различия в архитектуре, реализации и API для работы с потоками и управления памятью.

Теперь вы можете выбрать наиболее подходящую ОС для своей задачи и даже разработать ее самостоятельно, когда это необходимо, путем реализации планировщика, механизмов приоритетов, разделения привилегий между задачами и ядром и сегментации памяти.

В следующей главе будут более подробно рассмотрены *среды доверенного выполнения* (trusted execution environment, TEE), уделив особое внимание функциям TrustZone-M, недавно представленным ARM в их последнем семействе микроконтроллеров, которые добавляют новый независимый уровень разделения привилегий.

Глава 11

Доверенная среда выполнения

Важный шаг в технологической эволюции аппаратной архитектуры микроконтроллеров недавно был достигнут благодаря внедрению механизма разделения рабочих областей, который уже присутствует в других архитектурах, где его обычно называют *доверенной средой выполнения* (trusted execution environment, TEE).

TEE – это абстракция, которая предоставляет две или более отдельные среды выполнения (или, как иногда говорят, «мира») с разными возможностями и разрешениями для доступа к устройствам, ресурсам и периферийным модулям.

Изоляция среды выполнения одного или нескольких программных компонентов и модулей, также известная как *песочница* (sandbox), заключается в ограничении их доступа к ресурсам системы без влияния на производительность и нормальную работу. Это распространенный подход во многих областях информатики, а не только в области безопасности встраиваемых систем.

Подобные аппаратные механизмы изоляции являются традиционными функциональными компонентами инфраструктуры облачных серверов, какой она является сегодня, в форме расширений виртуализации и безопасных механизмов изоляции, которые позволяют одновременно запускать несколько «гостевых» виртуальных машин или контейнеров на одном физическом сервере.

В этой главе рассмотрены следующие концепции и технологии:

- песочница;
- TrustZone-M;
- разделение системных ресурсов;
- сборка и запуск примера.

К концу главы вы познакомитесь с механизмом TEE и научитесь настраивать и использовать TrustZone-M на микроконтроллерах Cortex-M для получения двух отдельных областей выполнения.

11.1. Технические требования

Для запуска примера, доступного для скачивания в репозитории этой книги, требуется микроконтроллер STM32L552. Технология TrustZone-M поддерживается только новейшим семейством микроконтроллеров ARM Cortex-M. STM32L552 – это Cortex-M33, полностью поддерживающий TrustZone-M, что делает его удобным и доступным выбором для первых шагов в изучении этой технологии.

Файлы кода для этой главы доступны по адресу https://github.com/PacktPublishing/Embedded-Systems-Architecture-Second-Edition/tree/main/Chapter11.

11.2. Песочница

Песочница – это общее понятие в сфере компьютерной безопасности, которое обозначает комплекс аппаратных и программных мер, которые ограничивают доступ к системе для одного или нескольких ее компонентов, чтобы ограничить область системы, затронутую случайными сбоями или преднамеренными атаками вредоносных программ, и предотвратить распространение последствий по всей системе. Песочница может иметь разные формы и реализации, которые используют или не используют определенные аппаратные функции для повышения безопасности и эффективности.

Под TEE подразумевают те механизмы песочницы, в которых ЦП постоянно отслеживает безопасный статус работающего кода без существенного влияния на его производительность. Из-за того, что механизмы TEE неразрывно связаны с конструкцией ЦП, модели поведения, управления и связи TEE в песочницах различаются на разных платформах и сильно зависят от архитектуры. Более того, TEE можно использовать для разных целей, часто в сочетании с криптографией для сохранения целостности и подлинности программного обеспечения через аппаратный *корень доверия* (root of trust).

В 2005 г. Intel ввела первые инструкции по виртуализации (Intel VT) для процессоров x86, чтобы напрямую запускать изолированный код виртуальной машины (в отличие от эмуляции ЦП в выделенном процессе на хост-машине), предоставляя аппаратную виртуализацию ключевых компонентов (процессор, оперативная память и периферийные устройства). Процессоры Intel ограничивают доступ гостевых виртуальных машин к реальному оборудованию, используя расширение существующих иерархических областей защиты, часто называемых просто *кольцами защиты* (protection ring), которые уже применяются для разделения ядра и пользовательского пространства.

Виртуальные машины – не единственный вариант использования TEE на процессорах x86. *Intel Software Guard Extensions* (SGX, расширения для защиты программ) – это набор связанных с безопасностью инструкций, присутствующих во многих процессорах x86 и защищающих определенные области памяти или анклавы от несанкционированного доступа. Хотя эти инструкции недавно перестали применять в потребительских процессорах Intel, они все

еще встречаются в процессорах для облачного и корпоративного оборудования. SGX можно использовать для нескольких целей, таких как обеспечение защищенного хранилища для сокрытия секретных ключей, которые будут безопасно использоваться приложениями. Но изначально они были разработаны для использования в системе *управления цифровыми правами* (digital rights management, DRM) на ПК, которая обеспечивает защиту авторских прав на носитель и контент проприетарного программного обеспечения, разрешая доступ к защищенному контенту только предварительно авторизованному, подписанному программному обеспечению. В этом сценарии противниками, от которых TEE защищает систему, являются конечные пользователи системы.

Позже AMD добавила в свои процессоры расширения архитектуры для конкретных поставщиков, сгруппированные в технологию под названием *Secure Encrypted Virtualization* (SEV, безопасная зашифрованная виртуализация). В дополнение к песочнице для запуска виртуальных машин, управляемых гипервизором, SEV использует аппаратное шифрование для обеспечения конфиденциальности содержимого отдельных страниц памяти и даже регистров ЦП во время выполнения.

Но архитектуры Intel были не первыми, где нашли применение встроенные расширения безопасности с поддержкой ЦП. ARM начала исследования в области доверенных вычислений в начале 2000-х гг. и объявила о поддержке технологии TrustZone в 2003 г. Современные микропроцессоры ARM, например в семействе Cortex-A, поддерживают технологию TrustZone-A, которая реализует две отдельные среды выполнения – *защищенную* (secure, S) и *незащищенную* (non-secure, NS), причем последняя имеет ограниченное предварительно настроенное представление фактической системы, в то время как первая имеет прямой доступ ко всем аппаратным ресурсам.

Что касается первых микроконтроллеров, реализующих TEE, следует упомянуть недавно разработанную архитектуру RISC-V. Как микропроцессоры, так и микроконтроллеры в семействах RISC-V предлагают раздельные полные песочницы, доступные как в 32-битной, так и в 64-битной архитектуре с расширениями «S» или «U». Каждый контейнер с аппаратной поддержкой предоставляет подмножество ресурсов, доступных в системе, и запускает собственную прошивку.

Наконец, новейшее семейство микроконтроллеров ARMv8-M содержит расширения и микрокод, необходимые для изоляции между защищенными и незащищенными областями выполнения, на основе существующей и хорошо отлаженной технологии TrustZone. Эта технология называется TrustZone-M, и она будет более подробно рассмотрена немного позже в данной главе. ARMv8-M – это прямой эволюционный потомок семейства микроконтроллеров ARMv7-M, которые использовались в качестве тестовой платформы во всех предыдущих главах этой книги.

В оставшейся части главы речь пойдет исключительно о TrustZone-M и о том, как настраивать и разрабатывать компоненты встроенной системы, реализующей TEE на микроконтроллерах семейства ARMv8-M. Термин TrustZone отныне будет относиться конкретно к технологии TrustZone-M.

11.3. TrustZone-M

Ядра ARMv7-M, такие как микроконтроллеры Cortex-M0+ и Cortex-M4, десятилетиями доминировали на рынке встраиваемых систем и до сих пор остаются самым популярным выбором для многих разработчиков. Несмотря на ряд изменений и дополнений, новые ядра Cortex-M23 и Cortex-M33, а также более новые ядра M35P и M55 унаследовали и расширили многие успешные функции микроконтроллеров Cortex-M0, Cortex-M4 и Cortex-M7.

В типичном случае применения TrustZone несколько разработчиков могут участвовать в различных этапах создания программного обеспечения. Владелец устройства может предоставить базовую систему, уже оснащенную всем программным обеспечением, разрешенным для работы в защищенной среде. Это оставляет системному интегратору возможность настраивать незащищенную часть, но с ограниченным представлением системы, которое зависит от конфигурации ресурсов, разрешенных защищенной средой. В этом случае системный интегратор получает частично заблокированную систему с включенным механизмом TrustZone и защитой флеш-памяти от внешнего доступа. Предоставленное безопасное программное обеспечение контролирует выполнение любого пользовательского программного обеспечения в незащищенной среде, сохраняя при этом ресурсы, сопоставленные в защищенной среде, и ограничивая доступ из работающего приложения. Системные интеграторы, не имеющие авторизованного доступа к защищенной среде выполнения, могут по-прежнему запускать привилегированное или непривилегированное программное обеспечение в незащищенной среде, включая операционные системы и драйверы устройств, которые имеют доступ к авторизованным интерфейсам либо напрямую, либо с некоторой помощью от защищенного супервизора.

Пример кода для этой главы можно скомпилировать и запустить на тестовой платформе. Этот пример основан на процедуре загрузчика, представленного в главе 4. Выбор этого примера связан со схожестью структуры решения на основе TrustZone, которое будет описано далее, поскольку программное обеспечение для двух областей выполнения поставляется в виде отдельных двоичных файлов. В случае с TrustZone разделение кода загрузчика, выполняемого в защищенной среде, и приложения, работающего в незащищенной области, поможет вам разобраться с элементами и инструментами, используемыми для создания, настройки и запуска компонентов в реальной системе.

Следующий раздел содержит описание новой тестовой платформы, а затем кратко представлена модель исполнения в защищенной и незащищенной средах, что приведет нас к более глубокому анализу блоков и контроллеров TrustZone-M, регулирующих разделение ресурсов в системе.

11.3.1. Тестовая платформа

В качестве тестовой платформы в примерах этой главы используется микроконтроллер STM32L552 на базе процессора Cortex-M33, который можно найти

на макетных платах в удобном формате Nucleo-144. Серию микроконтроллеров STM32L5 можно считать наиболее близким потомком более старой серии STM32F4, нацеленным на тот же сегмент рынка за счет сочетания режимов с низким энергопотреблением и высокой производительностью. Поэтому микроконтроллер этой серии был выбран в качестве тестовой платформы для последующих примеров. Но большинство концепций и компонентов технологии TrustZone-M, которые будут описаны, применимы ко всем микроконтроллерам семейства ARMv8-M, выпускаемым ST Microelectronics и некоторыми другими производителями микросхем.

На STM32L552ZE тактовую частоту процессора можно настроить на 110 МГц. Микроконтроллер оснащен 256 Кбайт SRAM, разделенной на два банка SRAM1 и SRAM2, отображаемых на отдельные области. 512 Кбайт флеш-памяти можно использовать как одно непрерывное пространство или настроить как два отдельных банка. Производитель предоставляет специфичные для платформы библиотеки и инструменты, которые не являются частью предоставленных примеров, потому что их использование нужно, как обычно, начинать с изучения фирменной документации, а это выходит за рамки данной книги. Единственным исключением из этого подхода в примере, который описан далее, является использование интерфейса командной строки программирования `STM32_Programmer_CLI`, который можно использовать для отображения текущего значения байтов программируемых опций, просто подключившись через встроенный отладчик ST-Link к ПК с помощью USB-кабеля и выполнив команду

```
STM32_Programmer_CLI -c port=swd -ob displ
```

Этот инструмент будет полезен для установки *байтов параметров*, необходимых для включения и выключения TrustZone, а также для настройки других параметров разделения областей флеш-памяти. Значения байтов опций хранятся в энергонезависимой памяти и сохраняются после отключения питания платы.

Важное примечание

Модификация некоторых байтов параметров через программатор может быть необратимой, что приведет к полной неработоспособности устройства без возможности восстановления. Пожалуйста, очень внимательно изучите техническую документацию вашего микроконтроллера перед изменением параметров.

Один из байтов параметров содержит флаг `TZEN`, который должен быть отключен в соответствии с заводскими настройками по умолчанию. Только когда механизм TrustZone-M будет полностью настроен, его можно включить на целевом устройстве для загрузки и запуска примера. Часть загрузчика в защищенной области будет отвечать за настройку среды для приложения, установленного как отдельный двоичный файл, и выполнение его в незащищенной среде. Затем будут продемонстрированы переходы между двумя областями, представив новые инструкции ассемблера ARMv8, введенные для этой цели.

В следующем разделе вы познакомитесь с расширениями, включенными в архитектуру ARMv8-M, для выполнения кода и управления областями

выполнения. Эти расширения являются общими, включены во все микроконтроллеры ARMv8-M, поддерживающие TrustZone, и являются основным компонентом для организации выполнения в раздельных областях.

11.3.2. Защищенные и незащищенные области выполнения

В главе 10 было сказано, что разделение ресурсов между разными потоками, а также между потоками и операционной системой возможно с помощью сегментации памяти. В семействе микроконтроллеров ARMv8-M механизм TrustZone-M часто называют расширением безопасности, поскольку он фактически добавляет один дополнительный уровень разделения привилегий между программными компонентами, работающими на целевом устройстве. Это расширение безопасности не заменяет существующее разделение потоков, которое было реализовано ранее в безопасной версии планировщика. Напротив, оно вводит дополнительный режим безопасности поверх существующего разделения.

Подобно тому как ОС, работающая без этого расширения, обеспечивает разделение между режимом потока и привилегированным режимом и может устанавливать границы для доступа к областям, отображаемым в память, с помощью MPU, TrustZone-M добавляет *защищенные* (secure, S) и *незащищенные* (non-secure, NS) среды выполнения с доступом к отдельным ресурсам под контролем процессора.

В каждой из этих сред по-прежнему возможно реализовать разделение привилегированных потоков, используя существующий механизм, основанный на бите `CONTROL`. Каждая область безопасности может иметь свои собственные привилегированные и непривилегированные режимы выполнения. Операционная система, работающая в среде NS, по-прежнему может использовать классическое разделение привилегий, унаследованное от предыдущей архитектуры ARMv7-M. Это создает в общей сложности четыре доступных контекста выполнения, за которыми может одновременно следить ЦП, обобщенные в табл. 11.1 как комбинация уровней среды и привилегий.

Таблица 11.1. Доступные режимы выполнения в защищенных/незащищенных областях

Защищенная среда выполнения	**Незащищенная среда выполнения**
Защищенное привилегированное выполнение	Незащищенное привилегированное выполнение
Защищенное потоковое выполнение	Незащищенное потоковое выполнение

Как было показано в главе 10, Cortex-M4 предоставляет два отдельных указателя стека (MSP и PSP) для отслеживания различных контекстов при выполнении потоков или кода ядра. Всего в Cortex-M33 имеется четыре различных указателя стека: MSP_S, PSP_S, MSP_NS и PSP_NS. Каждый указатель

стека присваивается реальному регистру SP во время выполнения в зависимости от текущей области и контекста.

В архитектуру ARMv8-M добавлена очень удобная функция, когда на ЦП присутствуют расширения MAIN, как в нашей тестовой платформе. Каждый из четырех указателей стека имеет соответствующий регистр *ограничения указателя стека* (SPLIM) (называемый соответственно MSPLIM_S, PSPLIM_S, MSPLIM_NS и PSPLIM_NS). Эти регистры указывают нижний предел значения указателя стека в четырех случаях. На самом деле это эффективная мера противодействия проблемам, проанализированным в разделе 5.3.2. ЦП будет постоянно следить за тем, чтобы стек никогда не превышал своего нижнего предела в памяти, генерируя исключение, когда это происходит. Этот механизм обеспечивает лучший аппаратный способ защиты памяти от случайных переполнений стека и конфликтов с другими областями памяти, чем тот, который был предложен в примерах из главы 5, где была введена защитная область между двумя областями памяти, отведенными под стек и динамическое выделение.

Было уже показано, как переключаться между режимами выполнения и как установка или очистка бита `CONTROL` при возврате из системных вызовов влияет на переходы между режимами привилегированного и потокового выполнения. Механизмы переключения между защищенным и незащищенным выполнением реализуются с помощью специальных инструкций ассемблера, которые будут объяснены позже после знакомства с разделением ресурсов между защищенными и незащищенными средами.

Чтобы лучше понять, к каким системным ресурсам может или не может получить доступ программное обеспечение, работающее в незащищенной среде, в следующем разделе будут подробно описаны различные возможности, предоставляемые модулями контроллера TrustZone-M для изоляции и разделения аппаратных ресурсов.

11.4. Разделение системных ресурсов

Когда модуль TrustZone-M включен, все области, отображаемые в памяти, включая ОЗУ, периферийные устройства и даже флеш-память, получают новый атрибут безопасности. Помимо принадлежности к защищенной и незащищенной областям, атрибут безопасности может принимать третье значение – *доступность для незащищенного вызова* (non-secure callable, NSC). Этот последний атрибут определяет специальные области памяти, используемые для реализации переходов из незащищенной среды в защищенную с помощью специального механизма, который будет объяснен в разделе 11.5. Область NSC используется для предложения безопасных API, которые действуют как системные вызовы с новыми возможностями. Защищенная среда предоставляет служебные подпрограммы, которые могут выполнять определенные контролируемые действия при доступе к защищенным ресурсам из незащищенной среды.

11.4.1. Атрибуты безопасности и области памяти

Микроконтроллеры Cortex-M33 предлагают различные уровни защиты. Комбинация влияния этих уровней определяет, какие из отображаемых в память адресов, связанных с ресурсом в системе, доступны для обоих вариантов выполнения, а какие из них доступны только из безопасной среды.

Включение TrustZone-M также дублирует представление некоторых системных ресурсов. Флеш-память, обычно отображаемая с начального адреса `0x08000000`, имеет псевдоним в регионе `0x0C000000`, который используется для доступа к тому же хранилищу из безопасной среды. Многие системные регистры имеют защищенные и незащищенные версии в разных местах памяти. Например, контроллер GPIOA сопоставляется с адресом `0x42020000`, когда TrustZone отключен. Когда TrustZone включен, тот же адрес используется программным обеспечением, работающим в незащищенной среде, если контроллер GPIOA для нее доступен. Но программное обеспечение, работающее в безопасной среде, будет использовать тот же контроллер, сопоставленный с адресом `0x52020000`. Та же система банков памяти применяется ко многим другим регистрам в области адресов периферийных устройств, которые имеют защищенные и незащищенные версии одних и тех же регистров, сопоставленные с двумя отдельными областями.

Прежде чем поступить на обработку операции другими компонентами, поддерживающими TrustZone, каждый доступ к памяти отслеживается и фильтруется двумя модулями, ответственными за настройку атрибутов. Это *модуль атрибуции безопасности* (security attribution unit, SAU) и *модуль атрибуции, определяемый реализацией* (implementation-defined attribution unit, IDAU). Упомянутые модули влияют на доступность всего сопоставления памяти независимо от типа ресурса, связанного с каждой областью. В то время как SAU настраивается с помощью набора регистров, IDAU содержит жестко запрограммированные сопоставления, принудительно заданные производителем чипа. Сочетание атрибутов IDAU и SAU влияет на доступность каждой отображаемой в памяти области, в частности:

- на области, обозначенные IDAU как защищенные, не влияют атрибуты SAU, и они всегда будут отображаться как защищенные;
- области, отображаемые IDAU как NSC, могут быть защищенными или NSC в зависимости от атрибута SAU;
- области, обозначенные IDAU как незащищенные, будут следовать сопоставлению SAU.

Комбинация атрибутов и результирующее сопоставление для каждой области сведены в табл. 11.2.

По умолчанию наш IDAU на тестовой платформе STM32L552 обеспечивает сопоставление атрибута защищенный/NSC с несколькими ключевыми областями:

- отображение флеш-памяти в защищенном пространстве, начиная с адреса `0x0C000000`;
- второй банк SRAM (SRAM2) сопоставлен с началом адреса `0x30000000`;

- память между `0x50000000` и `0x5FFFFFFF`, зарезервированная для безопасного управления и настройки периферийных устройств.

Таблица 11.2. Комбинация атрибутов IDAU и SAU

Атрибут IDAU	Атрибут SAU	Результирующий атрибут
Защищенный	Защищенный, NSC или незащищенный	Защищенный
NSC	Защищенный	Защищенный
Незащищенный	Защищенный	Защищенный
NSC	NSC или незащищенный	NSC
Незащищенный	NSC	NSC
Незащищенный	Незащищенный	Незащищенный

После сброса SAU определяет все области как защищенные и по умолчанию отключен. Чтобы выполнить незащищенный код, нужно определить как минимум две незащищенные области в пределах интервалов, разрешенных конфигурацией IDAU.

В рассматриваемом примере происходит инициализация нескольких областей памяти, чтобы разрешить доступ из приложений, прежде чем включить SAU. SAU управляется через четыре основных 32-битных регистра:

- **SAU_CTRL** (управление SAU): используется для активации SAU. Он содержит флаг для «инвертирования» логики фильтра SAU, определяя все области памяти как незащищенные;
- **SAU_RNR** (регистр номера области SAU): содержит номер области для выбора в начале процедуры настройки для области памяти. Дальнейшие записи в SAU_RBAR и SAU_RLAR будут относиться к этой пронумерованной области;
- **SAU_RBAR** (регистр базового адреса области SAU): указывает базовый адрес региона, который нужно настроить;
- **SAU_RLAR** (регистр ограничения адресов области SAU): содержит конечный адрес области для настройки. Младшие 5 бит зарезервированы для флагов. Бит 1, когда он установлен, указывает, что область является защищенной или вызываемой незащищенным приложением. Бит 0 включает область и указывает, что ее конфигурация завершена.

В следующем примере кода показана вспомогательная функция `sau_init_region`. Получив идентификатор области, базовый адрес, конечный адрес и значение бита безопасности, она соответствующим образом установит все значения регистра:

```
static void sau_init_region(uint32_t region,
    uint32_t start_addr,
    uint32_t end_addr,
    int secure)
{
  uint32_t secure_flag = 0;
  if (secure)
      secure_flag = SAU_REG_SECURE;
  SAU_RNR = region & SAU_REGION_MASK;
```

```
  SAU_RBAR = start_addr & SAU_ADDR_MASK;
  SAU_RLAR = (end_addr & SAU_ADDR_MASK) | secure_flag | SAU_REG_ENABLE;
}
```

Следующая функция вызывается функцией инициализации `secure_world_init` для сопоставления четырех областей SAU, которые нужно настроить для этого примера:

- **Region 0**: вызываемый раздел, где будет храниться наш безопасный API, который можно вызывать из незащищенного приложения. В этом примере используется функция с именем `nsc_blue_led_toggle`, которая будет единственным способом, с помощью которого приложение может получить доступ к защищенному GPIO, подключенному к синему светодиоду на плате Nucleo;
- **Region 1**: незащищенная область во флеш-памяти начиная с адреса `0x08040000`. Здесь будет находиться код нашего незащищенного приложения;
- **Region 2**: незащищенная часть банка SRAM1, которую может использовать незащищенное приложение для хранения стека и переменных. Это необходимый шаг, чтобы приложение могло получить доступ к адресам ОЗУ;
- **Region 3**: управление незащищенными периферийными модулями, сопоставленное с адресом `0x40000000`, включая незащищенные контроллеры GPIO. К этой области будет обращаться незащищенное приложение для настройки системных часов и управления зеленым светодиодом в нашем примере.

В рассматриваемом примере инициализация SAU происходит следующим образом:

```
static void secure_world_init(void)
{
  /* Незащищенный вызов: область функций NSC*/
  sau_init_region(0, 0x0C001000, 0x0C001FFF, 1);

  /* Не защищено: область флеш-памяти приложения */
  sau_init_region(1, 0x08040000, 0x0804FFFF, 0);

  /* Незащищенная область RAM в SRAM1 */
  sau_init_region(2, 0x20018000, 0x2002FFFF, 0);

  /* Не защищено: внутренние прерывания */
  sau_init_region(3, 0x40000000, 0x4FFFFFFF, 0);
```

Код в окончании этой функции активирует SAU и включает специальный обработчик, обнаруживающий ошибки безопасности:

```
  /* Включение SAU */
  SAU_CTRL = SAU_INIT_CTRL_ENABLE;

  /* Включение обработчика ошибок безопасности */
  SCB_SHCSR |= SCB_SHCSR_SECUREFAULT_EN;
}
```

По умолчанию включение SAU помечает все области как защищенные, поэтому конфигурация каждой области обрезает незащищенное или вызываемое незащищенным приложением «окно» в адресуемом пространстве памяти. Region 0 – единственная область, помеченная флагом NSC в нашей демонстрационной конфигурации. Это означает, что безопасным приложением здесь будет установлен код NSC (поясняется позже). Области 1, 2 и 3 – это единственные области памяти, к которым можно получить доступ при работе в незащищенной среде с включенным TrustZone-M.

Как упоминалось ранее, IDAU/SAU – это лишь первый уровень фильтров для механизмов защиты TrustZone-M. Флеш-память и ОЗУ защищены дополнительными шлюзами безопасности, которые могут быть основаны на блоках или на водяных знаках. Микроконтроллер STM32L552 оснащен *глобальным контроллером TrustZone* (GTZC), который включает в себя один контроллер шлюза на основе водяных знаков для флеш-памяти и один на основе блоков для определения защищенных/незащищенных блоков ОЗУ.

11.4.2. Флеш-память и водяные знаки

На целевой платформе флеш-память можно настроить так, чтобы она отображалась как единое непрерывное пространство или разделялась пополам путем активации конфигурации с двумя банками. В рассматриваемом примере использования TrustZone-M флеш-память хранится в одном банке.

В этой конфигурации, когда TrustZone включен, можно назначить незащищенную область в старшей половине непрерывного пространства флеш-памяти начиная с адреса `0x08040000`. Когда флеш-память разделена на два банка, каждый банк может настроить свой собственный независимый защищенный *водяной знак* (watermark). Область флеш-памяти между начальным и конечным адресами помечается как защищенная, а все, что остается за пределами меток, является незащищенным. Защищенная область в каждом банке ограничена значением байтов параметров, SECWMx_PSTRT и SECWMx_PEND. Если разделители перекрываются, то есть когда значение SECWMx_PEND больше, чем значение SECWMx_PSTRT, вся область помечается как незащищенная.

Их значение можно изменить с помощью утилиты программатора:

```
STM32_Programmer_CLI -c port=swd -ob SECWM1_PSTRT=0 SECWM1_PEND=0x39
```

В однобанковом режиме каждый сектор флеш-памяти имеет размер 4096 байт. Установив соответствующие байты параметров, помечаются первые 64 сектора (от `0x00` до `0x39`) как защищенные, что оставляет другую половину флеш-памяти, начиная с адреса `0x08040000`, для использования незащищенным приложением из нашего примера. Программатор, запущенный с опцией `-ob displ`, выведет следующее сообщение:

```
Secure Area 1:
  SECWM1_PSTRT : 0x0 (0x8000000)
  SECWM1_PEND : 0x39 (0x8039000)
```

11.4.3. Конфигурация GTZC и защита SRAM на основе блоков

На нашей текущей тестовой платформе присутствует дополнительный шлюз для управления доступом в контроллере TrustZone. Компонент шлюза на основе блоков GTZC позволяет нам настроить бит защиты для частей SRAM. Весь объем SRAM на STM32L552 разделен на два основных банка:

- SRAM1: 192 Кбайта ОЗУ, отображаемые по адресу 0x08000000;
- SRAM2: 64 Кбайта ОЗУ, отображаемые по адресу 0x30000000 и установленные как NSC в IDAU.

В этом примере помечена старшая половина SRAM1, начиная с адреса 0x2018000, как незащищенная. Для этого GTZC предоставляет два набора регистров, по одному для каждого банка, для настройки блочного шлюза для каждой страницы в ОЗУ. Каждый блок представляет 256 байт, и каждый 32-битный регистр, содержащий один бит защиты на блок, может отображать 32 страницы, также определяемые как суперблок размером 8 Кбайт. 24 регистра требуются для отображения 24 суперблоков, т. е. 192 Кбайта в SRAM1, и только восемь требуется для отображения области 64 Кбайт в SRAM2.

Как и при инициализации SAU, снова используется подход, основанный на удобном макросе, который, зная банк памяти, номер суперблока и значение его регистра, вычисляет адрес для правильного регистра, который ссылается на суперблок, и генерирует правильный оператор присваивания:

```
#define SET_GTZC_MPCBBx_S_VCTR(bank,n,val) \
(*((volatile uint32_t *)(GTZC_MPCBB##bank##_S_VCTR_BASE )\ + n ))= val
```

Таким образом, можно легко настроить блочные шлюзы смежных областей внутри цикла. В примере приложения для безопасной среды используется следующая функция для настройки блочных шлюзов для двух банков:

```
static void gtzc_init(void)
{
  int i;
  /* Конфигурируем нижнюю половину SRAM1 как защищенную */
  for (i = 0; i < 12; i++) {
    SET_GTZC_MPCBBx_S_VCTR(1, i, 0xFFFFFFFF);
  }

  /* Конфигурируем верхнюю половину SRAM1 как незащищенную */
  for (i = 12; i < 24; i++) {
    SET_GTZC_MPCBBx_S_VCTR(1, i, 0x0);
  }

  /* Конфигурируем память SRAM2 как защищенную */
  for (i = 0; i < 8; i++) {
    SET_GTZC_MPCBBx_S_VCTR(2, i, 0xFFFFFFFF);
  }
}
```

Теперь есть все необходимое для запуска простейшего незащищенного приложения в системе; определены незащищенные области в SAU, установлен водяной знак для разделения флеш-памяти и настроены блочные шлюзы, чтобы разрешить незащищенный доступ к старшей половине SRAM1.

Но есть еще один аспект, заслуживающий внимания, а именно возможность настройки защищенного доступа к периферийным устройствам.

11.4.4. Настройка безопасного доступа к периферийным устройствам

На нашей тестовой платформе периферийные устройства делятся на две категории:

- **защищаемые периферийные устройства:** периферийные устройства подключаются не напрямую к локальной шине, а через систему шлюзов, управляемую *контроллером безопасности TrustZone* (TrustZone secure controller, TZSC);
- **периферийные устройства, поддерживающие TrustZone:** это периферийные устройства, которые интегрируются с механизмами TrustZone – например, предлагая отдельные интерфейсы для доступа к своим ресурсам в зависимости от среды выполнения.

Для первой категории периферийных устройств конфигурация безопасного доступа и привилегированного доступа в защищенных и незащищенных доменах может быть настроена через регистры TZSC в GTZC. При запуске системы все устройства по умолчанию настраиваются как защищенные, поэтому для включения доступа к UART, I2C, таймерам и другим периферийным устройствам необходимо будет снять бит защиты, связанный с конкретным контроллером.

Периферийные устройства, поддерживающие TrustZone, имеют банки регистров для обеих защищенных зон. В следующем примере настраиваются три контроллера GPIO (GPIOA, GPIOB и GPIOC), которые подключены к светодиодам на плате Nucleo-144 через контакты C7 (зеленый светодиод), B7 (синий светодиод) и A9 (красный светодиод). Регистры контроллера GPIO при включении TrustZone объединяются в две области. Вы заметите в примере кода разницу между двумя интерфейсами драйвера светодиодов в защищенном и незащищенном приложениях. В защищенной версии `led.h` определены следующие базовые адреса для регистров контроллера GPIO:

```
#define GPIOA_BASE 0x52020000
#define GPIOB_BASE 0x52020400
#define GPIOC_BASE 0x52020800
```

Те же самые контроллеры в незащищенном приложении отображаются в незащищенном адресном пространстве периферийных устройств:

```
#define GPIOA_BASE 0x42020000
#define GPIOB_BASE 0x42020400
#define GPIOC_BASE 0x42020800
```

Это гарантирует, что при работе в незащищенной среде конфигурация GPIO доступна только через интерфейс, назначенный незащищенному пространству.

Кроме того, каждый контроллер GPIO предоставляет интерфейс для защиты каждого отдельного управляемого вывода. Это достигается с помощью регистра только для записи, определяющего защищенный и незащищенный доступы с помощью флага, соответствующего каждому выводу. Регистр называется GPIOx_SECCFG и располагается по смещению `0x24` в пространстве каждого контроллера GPIO. Он доступен для записи только при работе в защищенной среде.

В следующем примере кода определены функции для установки/сброса бита защиты для каждого контакта GPIO, подключенного к трем светодиодам. Например, можно перевести в защищенное состояние красный светодиод перед созданием незащищенного приложения, чтобы запретить изменение состояния светодиода в приложении, вызвав функцию `red_led_secure(1)`, которая реализована следующим образом:

```
void red_led_secure(int onoff)
{
  if (onoff)
      GPIOA_SECCFG |= (1 << RED_LED);
  else
      GPIOA_SECCFG &= ~(1 << RED_LED);
}
```

Наш пример приложения для безопасной среды ограничивает доступ к синему и красному светодиодам перед отправкой данных, но разрешает доступ к зеленому светодиоду:

```
red_led_secure(1);
green_led_secure(0);
blue_led_secure(1);
```

После смены среды выполнения незащищенное приложение попытается включить все три светодиода, но фактически будет включен только зеленый, а остальные останутся выключенными, поскольку доступ через незащищенный интерфейс блокируется битом `SECCFG`, установленным в защищенной среде, и не влияет на GPIO.

Но мигание синим светодиодом по-прежнему будет выполняться с использованием специального незащищенного вызываемого интерфейса, описанного далее в разделе 11.5.5.

После настройки всех защищаемых и поддерживающих TrustZone периферийных устройств все готово, чтобы создать и установить образы прошивки для двух сред выполнения и наблюдать за их влиянием на систему.

11.5. Сборка и запуск примера

Наконец-то пришло время применить на практике все, что вы узнали о TrustZone-M, активируя флаги параметров, необходимые для включения TrustZone-M, и запуская два программных компонента, связанных со средами выполнения.

11.5.1. Включение TrustZone-M

По умолчанию TrustZone-M выключен на нашем микроконтроллере, когда он находится в заводском состоянии. Включение TrustZone – это односторонняя операция, но обычно она не является необратимой, если только она не сочетается с другими аппаратными механизмами защиты, которые делают невозможным ее отключение при развертывании встроенной системы. Но отключение TrustZone после включения требует более сложной процедуры, чем просто очистка одного бита в регистре.

Важное примечание

Пожалуйста, внимательно изучите техническую документацию к вашему микроконтроллеру и рекомендации по применению и убедитесь, что вы понимаете процедуру и последствия включения или попытки отключения TrustZone-M на вашем устройстве.

На тестовой платформе, чтобы включить TrustZone, установим соответствующий флаг в байтах параметров с помощью следующей команды:

```
STM32_Programmer_CLI -c port=swd mode=hotplug -ob TZEN=1
```

После включения TrustZone можно собрать и установить защищенную прошивку. Далее рассмотрены некоторые важные аспекты, которые следует учитывать при создании защищенной части системы.

11.5.2. Безопасная точка входа в приложение

Области, определенные в сценарии компоновщика защищенной среды, отражают системные ресурсы, видимые защищенной прошивкой. Выделим область ОЗУ, охватывающую нижнюю половину банка SRAM1:

```
RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00018000
```

Наши LMS `.text` и `.data` оказываются в области FLASH, сопоставленной с ее адресом в защищенной среде:

```
FLASH (rx) : ORIGIN = 0x0C000000, LENGTH = 0x1000
```

В нашем простом примере для хранения образа загрузчика достаточно 4 Кбайт. Кроме того, определим вызываемую область, которая будет содер-

жать реализацию наших защищенных заглушек. Это область, предназначенная для доступа к защищенным API из незащищенной среды посредством предопределенных вызовов специальных функций:

```
FLASH_NSC(rx): ORIGIN = 0x0C001000, LENGTH = 0x1000
```

Точка входа защищенного приложения на эталонной платформе жестко запрограммирована в байтах параметров. Перед записью образа прошивки нужно убедиться, что байты опции для `SECBOOTADD0` настроены так, чтобы они указывали на адрес `0x0C000000`, который является началом флеш-памяти в представлении безопасной среды. Если по какой-либо причине значение было ранее изменено, его можно восстановить с помощью следующей команды:

```
STM32_Programmer_CLI -c port=swd mode=hotplug -ob
SECBOOTADD0=0x180000
```

Это связано с тем, что размер `SECBOOTADD0` составляет 128 байт, поэтому запись значения `0x180000` даст нам указатель на адрес `0x0C000000`.

Это последнее значение завершает настройку байтов параметров, поэтому все готово, чтобы собрать и установить безопасное приложение.

Список байтов параметров и их значений, назначенных для настройки и запуска кода примера, предоставлен в репозитории этой книги в файле `Chapter11/option-bytes.txt`.

11.5.3. Компиляция и компоновка приложений защищенной среды

Если вы посмотрите в make-файле на секцию приложения для защищенной среды, то заметите, что в процессе сборки были введены два новых флага. GCC использует флаг `-mcmse`, чтобы указать, что компилируется безопасный код для системы TrustZone. Добавляя этот флаг, происходит указание компилятору генерировать заглушки *Secure Gateway* (SG) для функций, помеченных как вызываемые незащищенным приложением. При компиляции источников, помеченных как защищенные, GCC позволяет использовать определенные атрибуты для обозначения наших безопасных вызовов API. В рассматриваемом примере определен безопасный вызов API, который позволит незащищенному коду переключать значение контакта GPIO, подключенного к синему светодиоду. Вызываемая функция определяется в коде защищенного приложения; в нашем случае она содержится в файле `nsc_led.c`:

```
void __attribute__((cmse_nonsecure_entry))
    nsc_blue_led_toggle(void)
{
  if ((GPIOB_ODR & (1 << BLUE_LED)) == (1 << BLUE_LED))
    blue_led_off();
  else
```

```
    blue_led_on();
}
```

Атрибут компилятора `__attribute__((cmse_nonsecure_entry))` указывает GCC сгенерировать заглушку SG для этой функции. Раздел `FLASH_NSC`, который был определен в скрипте компоновщика, используется для хранения заглушек SG для настраиваемого нами защищенного API. Заглушки SG автоматически помещаются в раздел с именем `.gnu.sgstubs`, который был помещен в регион `FLASH_NSC` в примере скрипта компоновщика:

```
.gnu.sgstubs :
{
  . = ALIGN(4);
  *(.gnu.sgstubs*) /* Заглушки SG */
  . = ALIGN(4);
} >FLASH_NSC
```

Дополнительные флаги компоновщика, `--cmse-implib` и `--out-implib=led_cmse.o`, имеют другое назначение, которое напрямую не влияет на защищенную среду. При компоновке безопасного приложения, добавляя эти флаги, происходит указание компоновщику создать новый объектный файл, который не будет скомпонован в окончательном защищенном приложении. Вместо этого для приложения незащищенной среды будет скомпонован новый объектный файл, содержащий оболочки для безопасного API. Этот файл оболочек, или так называемый *винир* (veneer), подготавливает переход из незащищенной среды в среду с вызовами от незащищенных приложений. Другими словами, этот новый файл `led_cmse.o` представляет собой аналог реализации безопасных вызовов через незащищенную вызываемую заглушку SG. Генерируемый файл содержит код, необходимый для перехода к заглушке вызова незащищенного приложения. Для создания защищенного приложения нужно ввести два специальных набора флагов:

- флаг компиляции `-mcmse`, который сообщает gcc, что происходит генерация безопасного кода для TrustZone, и включает заглушки SG для незащищенных точек входа;
- флаги компоновщика `-cmse-implib` и `-out-implib=…`, которые указывают компоновщику генерировать виниры в формате объектных файлов, которые, в свою очередь, будут связаны с незащищенной средой для доступа к связанным безопасным вызовам API.

После сборки с помощью `make` защищенный образ прошивки можно загрузить во флеш-память устройства с помощью следующей команды:

```
STM32_Programmer_CLI -c port=swd -d bootloader.bin 0x0C000000
```

Флеш-память микроконтроллера теперь заполнена защищенной прошивкой – нашим усовершенствованным загрузчиком, который настроит все параметры в контроллере TrustZone и подготовит незащищенное приложение. Очевидным следующим шагом является компиляция и установка приложения для незащищенной среды.

11.5.4. Компиляция и компоновка приложений незащищенной среды

Сценарий компоновщика для нашего незащищенного приложения определяет границы видимой для него незащищенной среды. Безопасные регионы и регионы NSC отсюда недоступны. Видимость флеш-памяти ограничена ее верхней половиной, а доступное ОЗУ ограничено верхней половиной банка SRAM1. Сценарий компоновщика `target.ld` в незащищенном приложении определяет эти регионы следующим образом:

```
FLASH (rx) : ORIGIN = 0x08040000, LENGTH = 256K
RAM (rwx) :  ORIGIN = 0x20018000, LENGTH = 96K
```

С этого момента процесс сборки аналогичен созданию обычных приложений без поддержки TrustZone. В отличие от своего защищенного аналога, незащищенные приложения не требуют каких-либо специальных флагов компилятора или компоновщика.

Заметным исключением является упомянутый ранее дополнительный объектный файл, созданный в процессе сборки защищенного приложения, который позволяет незащищенному приложению на короткое время взаимодействовать с защищенной средой. Контракт между защищенной и незащищенной средами состоит из безопасного API, определенного защищенной средой. В рассматриваемом примере определена только одна безопасная функция `nsc_blue_led_toggle`. Объектный файл, содержащий заглушки (называемый в нашем примере `cmse_led.o`), автоматически сгенерированный при компиляции кода для защищенной среды, компонуется внутри незащищенного приложения, и фактически это код, который удовлетворяет символьные зависимости в защищенном приложении для этих специальных символов. Детали этой процедуры рассмотрены в следующем разделе.

Собрав незащищенное приложение с помощью `make`, загрузим незащищенный образ во внутреннюю флеш-память целевого устройства начиная с адреса `0x08040000`:

```
STM32_Programmer_CLI -c port=swd -d image.bin 0x08040000
```

Теперь стоит более подробно рассмотреть переходы из защищенной среды в незащищенную и наоборот, чтобы понять, как новые инструкции ARMv8-M участвуют в операциях перехода и как их следует использовать в таких случаях.

11.5.5. Переходы между средами выполнения

Когда загрузчик защищенной среды готов к использованию приложения, есть некоторые отличия в ассемблерном коде, который подготавливает регистры ЦП и выполняет переход в незащищенную область. Во-первых, когда TrustZone включен, системный регистр `VTOR` хранится в определенном банке.

Это означает, что есть два отдельных регистра, которые содержат смещение для таблицы векторов, по одному для каждой среды выполнения, – VTOR_S и VTOR_NS, для защищенной и незащищенной среды соответственно. Прежде чем перейти к точке входа для незащищенного кода, в регистр VTOR_NS необходимо поместить смещение вектора прерывания для приложения незащищенной среды. Вектор прерывания находится в начале двоичного образа, поэтому следующее присвоение в процедуре main загрузчика гарантирует, что в конечном итоге наш незащищенный код сможет выполнять процедуры обслуживания прерываний:

```
/* Обновление вектора прерываний */
VTOR_NS = ((uint32_t)app_IV);
```

После того как этот системный регистр настроен, имеется два важных указателя, необходимых для запуска приложения, аналогично тому, как это было бы с загрузчиком без поддержки TrustZone-M (описано в главе 4). Эти указатели, хранящиеся в первых двух 32-битных словах двоичного образа незащищенного приложения, являются начальным указателем стека и фактической точкой входа, содержащей адрес обработчика isr_reset соответственно. Прочитаем эти два адреса в локальные переменные стека перед запуском:

```
app_end_stack = (*((uint32_t *)(NS_WORLD_ENTRY_ADDRESS)));
app_entry = (void *)(*((uint32_t *)(NS_WORLD_ENTRY_ADDRESS + 4)));
```

В рассматриваемом примере заранее определен размер области стека для незащищенного приложения, вычисляя наименьший адрес, разрешенный для стека, следующим образом:

```
app_stack_limit = app_end_stack - MAX_NS_STACK_SIZE;
```

Затем это значение присваивается регистру MSPLIM_NS. Это специальный регистр, поэтому нужно использовать инструкцию msr:

```
asm volatile("msr msplim_ns, %0" ::"r"(app_stack_limit));
```

Установим значение для нового указателя стека, который заменит SP после завершения перехода между средами:

```
asm volatile("msr msp_ns, %0" ::"r"(app_end_stack));
```

Фактический переход к незащищенному коду сильно отличается от предыдущего загрузчика, представленного в главе 4. Сначала нужно убедиться, что адрес для перехода настроен в соответствии с соглашением, используемым в переходах ARMv8. Значение, которое будет прочитано из двоичного образа в локальную переменную app_entry, на самом деле является нечетным, что является классическим требованием при присвоении нового значения регистру PC при переходе в пределах одной среды – например, при использовании инструкции mov pc,... в ARMv7-M. Эта инструкция уже использовалась в примере загрузчика из главы 4. В ARMv8-M инструкция, которая одновременно выполняет переход по заданному адресу и в незащищенную среду, называ-

ется blxns. Но при вызове blxns или любой другой инструкции, подразумевающей переход к незащищенному адресу, и нужно убедиться, что в адресе назначения для перехода обнулен младший значащий бит. По этой причине надо уменьшить значение app_entry на единицу перед выполнением blxns:

```
/* Переход к незащищенной точке app_entry */
asm volatile("mov r12, %0" ::"r"
((uint32_t)app_entry - 1));
asm volatile("blxns r12" );
```

Это последняя инструкция, выполняемая в защищенной среде перед окончательным запуском нашего незащищенного приложения. Если использовать отладчик для проверки значений регистров при выполнении этих последних инструкций, можно убедиться, что значения регистров ЦП обновляются, и затем, наконец, регистр SP будет указывать на новый контекст в незащищенной среде.

С этого момента любая попытка вернуться в защищенную среду, конечно же, запрещена и будет генерировать исключение. Но назначение функций, размещенных в области NSC, – обеспечить временное и контролируемое выполнение безопасных функций, запрашиваемых из незащищенной среды.

В рассматриваемом примере перед переходом к незащищенной среде выполнения накладывается некоторое ограничение на доступ к линиям GPIO, связанным с тремя светодиодами, путем установки соответствующих битов в регистре GPIOx_SECCFG, как описано в разделе 11.4.4.

Когда оба образа, составляющих пример, загружены на тестовую платформу, можно выключить и снова включить питание и наблюдать за эффектами, глядя на три светодиода. После перезагрузки красный светодиод должен быть включен при запуске и гореть, пока в загрузчике работает безопасный код. После выполнения безопасного кода в течение произвольного количества циклов, чтобы дать достаточно времени для проверки состояния светодиода, красный светодиод выключается, и его состояние фиксируется. Синий светодиод также защищен с помощью вызова blue_led_secure(1), выполняемого перед запуском незащищенного приложения. Зеленый светодиод никак не защищен и обычно доступен в незащищенной среде.

Когда запускается незащищенное приложение, зеленый светодиод постоянно горит, а синий светодиод быстро мигает. Последнее возможно только благодаря тому, что незащищенные приложения могут получить доступ к функции в безопасных API.

Чтобы получить ассемблерный код, сгенерированный для этой функции, нужно выполнить команду arm-none-eabi-objdump -D для elf-файла защищенного приложения. Сразу видно, что сгенерированная вызываемая функция-заглушка на самом деле является короткой процедурой, помещенной в начало невызываемой секции:

```
0c001000 <nsc_blue_led_toggle>:
c001000:   e97f e97f sg
c001004:   f7ff bdd2 b.w c000bac
              <__acle_se_nsc_blue_led_toggle>
```

Наиболее интересной частью кода, работающего в области NSC, является использование специальной ассемблерной инструкции `sg`, которая представляет собой новую инструкцию, представленную в ARMv8-M с конкретной целью реализации безопасных вызовов из незащищенных сред выполнения. Эта инструкция подготавливает ответвление к безопасному вызову в защищенном флеш-пространстве, и она разрешена только тогда, когда выполняется при вызове из незащищенной среды.

Также обратите внимание, что реальная реализация на самом деле содержится в функции `__acle_se_nsc_blue_led_toggle`, сгенерированной компилятором и помещенной в область S флеш-памяти.

Ассемблерный код, сгенерированный виниром для `nsc_blue_led_toggle`, как видно из дизассемблирования незащищенного приложения после включения сгенерированного объекта в финальный образ, должен выглядеть следующим образом:

```
080408e8 <__nsc_blue_led_toggle_veneer>:
80408e8:   f85f f000   ldr.w   pc, [pc]  ; 80408ec \
                       <__nsc_blue_led_toggle_veneer+0x4>
80408ec:   0c001001
```

Завершение процедуры вызова безопасной функции из незащищенной среды находится в конце фактической реализации безопасной функции, переключающей синий светодиод `__acle_se_nsc_blue_led_toggle`:

```
c000bee:   4774        bxns    lr
```

Этот код должен показаться вам знакомым, так как на самом деле это версия инструкции `bxlns`, которую вы видели ранее, выполняющая переход на адрес возврата, хранящийся в регистре ссылки, а также переход обратно в незащищенную среду. В следующем списке кратко перечислены шаги, выполняемые при вызове безопасной функции из незащищенной среды выполнения в примере этой главы.

1. Незащищенный код вызывает винир для функции `nsc_blue_led_toggle`, реализованной в объекте `cmse_leds.o`, который генерируется при компиляции защищенного кода и связан с незащищенным приложением.
2. Виниру известно расположение заглушки SG в области NSC. Эта область доступна для выполнения из защищенной среды, при этом она находится в определенном регионе внутри защищенной прошивки. Затем винир переходит к заглушке SG.
3. Заглушка SG вызывает инструкцию `sg`, инициируя переход в защищенную среду, а затем переходит к фактической реализации `acle_se_nsc_blue_led_toggle`. Теперь код выполняется в защищенной среде, выполняя запрошенное действие (в нашем примере это смена логического уровня на линии GPIO, подключенной к синему светодиоду).
4. Когда процедура завершается, безопасная функция выполняет переход обратно в незащищенную среду с помощью инструкции `bxns`, одновременно возвращаясь к адресу исходного вызывающего объекта в незащищенной среде.

Несмотря на простоту, в нашем примере показано, как настроить и использовать все функции, необходимые для разделения двух сред выполнения, а также механизмы, которые будут использоваться для реализации взаимодействия между этими средами. Схема этих взаимодействий в защищенной среде будет определять возможности, предлагаемые незащищенным приложениям. Границы и интерфейс для переходов действуют как контракт между двумя частями системы, который реализован на уровне оборудования благодаря механизму TrustZone-M.

11.6. Заключение

ARMv8-M – новейшая архитектура, разработанная ARM для современных микроконтроллеров. Она расширяет и дополняет возможности своего предшественника, ARMv7-M, за счет нескольких новых функций. Наиболее важным улучшением этой новой архитектуры является возможность реализации TEE путем разделения сред выполнения и создания изолированной среды для выполнения незащищенных приложений.

В реальных сценариях это обеспечивает гибкость развертывания приложений от разных поставщиков с различными уровнями доверия в отношении доступа к функциям и ресурсам в системе.

В этой последней главе проведен анализ механизмов, доступных в технологии TrustZone-M. Их можно активировать в системах ARMv8-M для использования мощного аппаратного решения, предназначенного для защиты системных компонентов от любого доступа, который не был явно разрешен компонентом системного супервизора, работающим в защищенной среде.

Предметный указатель

И

К

Л

М

Н

О

П

Р

С

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
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: **http://www.galaktika-dmk.com/**.

Даниэле Лакамера

Архитектура встраиваемых систем

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Научный редактор *Романов А. Ю.*
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 26,98. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Данная книга посвящена разработке встраиваемых систем с использованием STM32 с процессорными ядрами ARM. Она будет интересна инженерам и программистам для изучения системного программирования и устройства встраиваемых систем, а также в качестве учебного пособия студентам соответствующих специальностей как продвинутое изучение C/C++ для его применения на низком уровне операционной системы и драйверов устройств.

В первой главе рассказывается о встраиваемых системах и как они устроены. **Во второй** главе даются практические рекомендации, как организовать рабочий процесс и настроить среду и другие инструменты разработчика встраиваемых систем. **Третья** глава посвящена описанию жизненного цикла разработки встраиваемой системы. **В четвертой** главе описана процедура загрузки и выполнения программного кода. **В пятой** главе рассказано про управление памятью: отображение памяти, стек выполнения, динамическое выделение памяти, защиту памяти. **В шестой** главе рассказывается, как взаимодействовать с периферийными устройствами. **Седьмая** глава посвящена стандартным локальным шинам (UART, SPI, I2C). **В восьмой** главе рассказывается про управление питанием и энергосбережением встраиваемой системы. **Девятая** глава затрагивает подключение встраиваемых систем к сетям передачи данных и Интернет вещей. **В десятой** главе описаны основы параллельного выполнения задач и работы планировщика. **В одиннадцатой** главе описаны принципы управления системными ресурсами и организации доверенной среды выполнения.

Книга снабжена множеством подробных примеров программного кода, доступного в репозитории, который можно апробировать на отладочных платах с STM32.

ДМК ПРЕСС
издательство
www.дмк.рф

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК “Галактика”
books@alians-kniga.ru

ISBN 978-5-93700-206-8
9 785937 002068