Майа Пош

Программирование встроенных систем на C++ 17

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Майа Пош

Программирование встроенных систем на C++17

Hands-On Embedded Programming with C++17

Create versatile and robust embedded solutions for MCUs and RTOSes with modern C++

Maya Posch

Packt

BIRMINGHAM – MUMBAI

Программирование встроенных систем на C++17

Создание универсальных и надежных встроенных решений для микроконтроллеров и операционных систем реального времени на современной версии языка программирования C++

Майа Пош

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2020

УДК 004.4
ББК 32.973.202
П66

Пош М.
П66 Программирование встроенных систем на C++17 / пер. с анг. А. В. Снастина. – М.: ДМК Пресс, 2020. – 394 с.: ил.

ISBN 978-5-97060-785-5

Из этой книги вы узнаете, как создавать автономные и сетевые встроенные системы, обеспечивать их безопасность и рациональное использование памяти. Язык программирования C++ расширяет возможности сопровождения и обладает многочисленными преимуществами по сравнению с другими языками программирования, поэтому прекрасно подходит для такой разработки.

В книге описывается методика создания удобных графических интерфейсов пользователя (GUI) для встроенных систем, а также методы интеграции проверенных стратегий в конкретные проекты для достижения оптимальной производительности аппаратуры. Рассмотрены разнообразные аппаратные платформы – у вас есть возможность выбрать наилучший вариант для своего проекта.

Издание будет полезно архитекторам встроенных систем и опытным разработчикам на C++.

УДК 004.4
ББК 32.973.202

ISBN 978-1-78862-930-0 (анг.)
ISBN 978-5-97060-785-5 (рус.)

Содержание

Об авторе

Майа Пош (Maya Posch) – ведущий разработчик на языке C++, обладающий более чем 15-летним опытом практической работы. Открыв для себя сначала столь увлекательную область деятельности, как программирование, а немного позже не менее увлекательную электронику, Майа всегда проявляла живой интерес к технологиям и охотно разделяла свое страстное увлечение с другими.

Сама Майа называет себя разработчиком на языке C, который со временем увлекся языками C++ и Ada. Ей нравится работать в условиях, ограничивающих объем программного кода и возможности аппаратуры до минимума, и создавать в этих условиях новые, эффективные и удобные системы.

Майа также активно интересуется разработками с использованием программируемых пользователем вентильных матриц (ППВМ – FPGA), разработками в области искусственного интеллекта и исследованиями в сфере робототехники, а кроме того, обладает талантами литератора, музыканта и живописца.

О рецензентах

Франс Фаазе (Frans Faase) изучал информационные технологии в университете Твенте (Нидерланды) и получил степень магистра в области проектирования компиляторов и формальных методов. Принимал участие в нескольких научных исследовательских проектах, но в основном работал как инженер по программному обеспечению в промышленной сфере. Франс обладает почти 20-летним опытом практического использования C++ как профессиональный разработчик и исследователь, а кроме того, C++ – это его хобби. Франс принимал активное участие в нескольких проектах с открытым исходным кодом. В последнее время он также приобрел некоторый практический опыт разработки ПО для микроконтроллеров, в основном с использованием среды Arduino.

Патрик Минтрэм (Patrick Mintram) – инженер по программному обеспечению, который начинал профессиональную карьеру как техник по электронному оборудованию. В течение длительного времени работал со встроенными системами, в том числе занимался разработкой и тестированием в средах с повышенными требованиями к безопасности, а также линейным сопровождением и восстановлением систем. Женат, имеет двух кошек Дьюка и Дэзи, в свободное время – «домашний мастер на все руки» и бег трусцой.

Предисловие

Язык программирования C++ не добавляет никаких излишеств, расширяет возможности сопровождения и предлагает многочисленные преимущества над прочими языками программирования, следовательно, представляет собой удачный выбор для разработки встроенных систем. Если вам необходимо создавать автономные или сетевые встроенные системы, обеспечивать их безопасность и рациональное использование памяти, то из этой книги вы точно узнаете, как это делается. Также вы узнаете, как работает C++, будет проведено сравнение с другими языками, используемыми для разработки встроенных систем. Кроме того, описывается методика создания удобных графических интерфейсов пользователя (GUI) для встроенных систем, проектирование привлекательных и функциональных пользовательских интерфейсов, а также методы интеграции проверенных стратегий в конкретные проекты для достижения оптимальной производительности аппаратуры.

В предлагаемой книге подробно рассматриваются разнообразные аппаратные платформы для встроенных систем, поэтому вы получаете возможность выбрать наилучший вариант для своего конкретного проекта. Вы научитесь решать сложные архитектурные задачи после внимательного изучения всех тщательно проработанных программных шаблонов, представленных в этой книге.

Для кого предназначена эта книга

Если вы начинающий разработчик программ на C++ для встроенных систем, то эта книга предназначена для вас. Для полного понимания всех тем книги требуется хорошее знание конструкций языка C++. Какие-либо предварительные знания в области встроенных систем не требуются.

Краткое содержание книги

Глава 1 «Что такое встроенные системы» знакомит читателя со встроенными системами в целом. Обзор разнообразных категорий и примеров встроенных систем дает общее представление о том, что означает термин «встроенный», а также о разнообразии значений этого термина. Рассматриваются многочисленные ранние и современные доступные модели микроконтроллеров и решения систем на одном чипе (на одной плате), которые вы можете обнаружить в существующих системах, а также новые проектные решения.

Глава 2 «C++ как язык программирования встроенных систем» объясняет, почему C++ так же хорош, как C и другие подобные языки. Вообще говоря, C++ не только так же быстр, как C, но еще и не содержит каких-либо излишеств и предлагает многочисленные преимущества, связанные с парадигмами кодирования и удобством сопровождения.

В главе 3 «Разработка для встроенной ОС Linux и подобных систем» рассматриваются методы разработки для встроенных систем на основе ОС Linux и родственных систем на одноплатных компьютерах с учетом различий между разработкой для Linux-подобных систем и систем, основанных на PC.

В главе 4 «Встроенные системы с ограниченными ресурсами» обсуждается планирование эффективного использования ограниченных ресурсов. Основное внимание уделяется правильному выбору микроконтроллера для нового проекта, добавлению периферийных устройств и определению требований к Ethernet-интерфейсам и последовательному интерфейсу в проекте. Также рассматривается пример проекта с использованием микроконтроллера AVR, методы разработки для других архитектур микроконтроллеров и возможности использования операционной системы реального времени.

В главе 5 «Пример: монитор влажности почвы с использованием протокола Wi-Fi» описывается процесс создания системы наблюдения за влажностью почвы с применением протокола беспроводной связи Wi-Fi с дополнительными возможностями управления приводом водяного насоса или аналогичного механизма. При наличии встроенного веб-сервера можно воспользоваться его пользовательским интерфейсом на основе браузера для наблюдения и управления или же интегрировать веб-сервер в более крупную систему, применив REST API.

В главе 6 «Тестирование приложений, предназначенных для конкретных ОС» демонстрируются методы разработки и тестирования приложений, предназначенных для встроенных операционных систем. Вы узнаете, как установить и использовать рабочую среду и конвейер инструментов для кросс-компиляции, как выполнять отладку в удаленном режиме с помощью отладчика GDB, как описать процедуру сборки системы.

В главе 7 «Тестирование платформ с ограниченными ресурсами» рассматриваются методы эффективной разработки для конкретных целевых микроконтроллеров. Также демонстрируется реализация интегрированной среды, позволяющей отлаживать приложения для микроконтроллеров со всеми удобствами, присущими настольной ОС, и с предоставляемым ею комплектом инструментальных средств.

В главе 8 «Пример: информационно-развлекательная система на основе ОС Linux» демонстрируется возможность относительно простого процесса создания информационно-развлекательной системы на одноплатном компьютере с использованием механизма преобразования голоса в текст для формирования пользовательского интерфейса, управляемого голосом. Также описаны возможности расширения системы с добавлением функциональности.

В главе 9 «Пример: мониторинг и управление микроклиматом в здании» рассматривается процесс разработки системы мониторинга и управления климатическими условиями внутри здания, отдельные компоненты этой системы, а также уроки, извлеченные во время разработки.

Глава 10 «Разработка встроенных систем с использованием библиотеки Qt» содержит описание многочисленных способов применения библиотеки и рабочей среды Qt для разработки встроенных систем. Выполняется сравнение с другими рабочими средами, рассматривается оптимизация Qt для встроенных платформ, а также пример использования предварительно разработанного графического пользовательского интерфейса на основе QML, который можно добавить в ранее созданную информационно-развлекательную систему.

В главе 11 «Разработка для гибридных систем SoC/FPGA» рассматривается организация обмена информацией на стороне программируемой пользователем вентильной матрицы (FPGA) в гибридной системе FPGA/SoC. Это помогает читателю понять разнообразные методы реализации алгоритмов в FPGA и применение их на стороне системы на кристалле (SoC). Также описана реализация простого осциллографа на основе гибридной системы FPGA/SoC.

В приложении А «Эффективные практические методики» кратко описан ряд проблем и затруднений, наиболее часто возникающих в процессе проектирования ПО для встроенных систем.

Максимально эффективное использование книги

Требуется практический опыт работы с семейством одноплатных компьютеров Raspberry Pi. Необходим компилятор C++, комплект инструментальных средств для организации (кросс)конвейера GCC ARM Linux, полный набор инструментальных средств AVR, рабочие среды Sming, Valgrind и Qt, а также интегрированная среда разработки (IDE) Lattice Diamond.

Получение файлов исходного кода примеров

Файлы исходного кода примеров из этой книги можно загрузить из вашей учетной записи на сайте **www.packtpub.com**. Если вы приобрели книгу в другом месте, то можно посетить страницу поддержки **www.packtpub.com/support** и зарегистрироваться, чтобы получить эти файлы по электронной почте. Загрузка файлов исходного кода примеров выполняется следующим образом:

1. Войти/зарегистрироваться на сайт/сайте **www.packtpub.com**.
2. Перейти на закладку **SUPPORT**.
3. Щелкнуть по пункту **Code Downloads & Errata**.
4. Ввести название книги в панели ввода **Search**, далее выполнять выводимые на экран инструкции.

После загрузки файла архива необходимо распаковать его, чтобы извлечь содержимое, используя самые свежие версии программ-упаковщиков:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Комплект примеров исходного кода для этой книги размещен в репозитории GitHub **https://github.com/PacktPublishing/Hands-On-Embedded-Programming-with-CPP-17**. При любых изменениях в коде немедленно обновляются соответствующие файлы в существующем репозитории GitHub.

Кроме того, вы можете получить другие комплекты исходного кода из нашего обширного каталога книг и видеоматериалов по адресу **https://github.com/PacktPublishing/**. Рекомендуем регулярно проверять его содержимое.

Типографские соглашения, принятые в книге

В этой книге используется несколько стилей выделения некоторых элементов текста.

`Фрагмент кода в тексте` – ключевые слова, операторы, имена переменных и функций непосредственно в тексте. Пример: «Сам класс языка C++ реализован в языке C как структура `struct`, содержащая переменные этого класса».

Фрагмент кода отображается в следующем формате:

```
class B : public A {
   // Закрытые члены класса
public:
   // Дополнительные открытые члены класса
};
```

При необходимости привлечь внимание читателя к некоторой части фрагмента кода соответствующие строки или элементы строк выделяются полужирным шрифтом:

```
class B : public A {
   // Закрытые члены класса
public:
   // Дополнительные открытые члены класса
};
```

Ввод или вывод в командной строке отображается следующим образом:

```
sudo usermod -a -G gpio user
sudo usermod -a -G i2c user
```

Курсив – имена файлов, каталогов и прочих объектов.

Полужирный шрифт – важные (ключевые) слова, элементы пользовательского интерфейса или слова, которые выводятся на экран. Например, пункты меню или элементы диалоговых окон.

Этим значком обозначаются предупреждения или важные замечания.

Этим значком обозначаются советы, подсказки и полезные практические приемы.

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 прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы 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**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или вебсайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Часть I

ОСНОВЫ ПРОГРАММИРОВАНИЯ ВСТРОЕННЫХ СИСТЕМ И РОЛЬ C++

В этой части читатель познакомится с разнобразными существующими встроенными аппаратными платформами, а также с практическим примером проекта простой встроенной системы.

Часть I включает следующие главы:

- глава 1 «Что такое встроенные системы»;
- глава 2 «C++ как язык программирования встроенных систем»;
- глава 3 «Разработка для встроенной ОС Linux и подобных систем»;
- глава 4 «Встроенные системы с ограниченными ресурсами»;
- глава 5 «Пример: монитор влажности почвы с использованием протокола Wi-Fi».

Глава 1

Что такое встроенные системы

По существу слово «встроенная» в термине «встроенная система» обозначает состояние включения (встраивания, интеграции) такой системы в более крупную систему. Встраиваемая система – это компьютерная система определенного вида, которая обладает одной или несколькими специализированными функциями в объемлющей системе, в отличие от компонентов общего назначения. Более крупная объемлющая система по своей природе может быть цифровой, механической или аналоговой, в то время как дополнительная интегрированная цифровая схема взаимодействует непосредственно с данными, передаваемыми и получаемыми от интерфейсов, сенсоров и устройств памяти, для реализации истинной функциональности системы.

В этой главе рассматриваются следующие темы:

- различные категории встраиваемых платформ;
- примеры встраиваемых платформ каждой категории;
- особенности и трудности разработки для каждой категории.

Разнообразие встраиваемых систем

Любая компьютеризованная функция в современных устройствах реализована с использованием одного или нескольких микропроцессоров. Это означает, что процессор (CPU – central processing unit) обычно содержит одну интегральную микросхему (IC – integrated circuit). Микропроцессор состоит, как минимум, из арифметико-логического устройства (АЛУ) и управляющей схемы, но если рассуждать логически, то необходимы также регистры и блоки ввода/вывода (I/O), а кроме того, более развитые функциональные возможности, специально предназначенные для определенной категории продукции (носимые устройства, низкочастотные (слаботочные) сенсоры, микшеры сигналов и т. п.) или ориентированные на широкий потребительский рынок (бытовая электроника, медицина, автомобили и т. п.).

В настоящее время во встроенных системах можно обнаружить почти все типы микропроцессоров. Даже у людей, которые предпочитают пользоваться настольным компьютером, ноутбуком, смартфоном или планшетом, количество встроенных микропроцессоров в домашнем хозяйстве значительно превышает количество микропроцессоров общего назначения.

Внутри ноутбука или настольного компьютера имеется несколько встроенных микропроцессоров, дополняющих центральный процессор общего назначения. Эти микропроцессоры выполняют такие задачи, как обработка ввода с клавиатуры и мыши или ввода с сенсорных экранов, преобразование потоков данных в пакеты Ethernet или формирование выходных потоков видео либо аудиоданных.

Даже в более старых системах, например в компьютере Commodore 64, можно обнаружить точно такие же компоненты: микросхема ЦПУ, звуковая микросхема, микросхема видео и т. д. Центральный процессор выполняет любой код, написанный разработчиком приложения, тогда как другие микросхемы в этой системе предназначены для весьма специализированных конкретных целей, вплоть до микросхемы контроллера, управляющего приводами жесткого или гибкого диска.

Встроенные микропроцессоры можно найти не только в компьютерах общего назначения, но практически везде, зачастую в виде даже еще более интегрированных микроконтроллеров (MCU – microcontroller unit). Они управляют кухонными устройствами, стиральными машинами, двигателями автомобилей, дополняя функции более высокого уровня и обработку информации, получаемой от сенсоров.

Первые микроволновые печи были аналоговыми устройствами с механическими таймерами и переменными резисторами (реостатами) для установки мощности и продолжительности, но в современных микроволновых печах содержится по меньшей мере один микроконтроллер, отвечающий за обработку пользовательского ввода, управление дисплеем определенного типа и конфигурирование внутренних систем микроволновой печи. Дисплей может иметь собственный микроконтроллер в зависимости от сложности общей конфигурации.

Еще более интересен тот факт, что встроенные системы также обеспечивают контроль, автоматизированное управление и безопасность самолетов, гарантируют, что управляемые снаряды и космические ракеты будут работать, как предполагалось, и предоставляют постоянно расширяющиеся возможности в таких областях, как медицина и робототехника. Авиационная электроника любого самолета постоянно отслеживает множество параметров, получаемых от огромного количества сенсоров, выполняя код в конфигурации с тройной избыточностью, чтобы своевременно выявлять любые возможные сбои и проблемы.

Крошечные, но мощные микропроцессоры обеспечивают быстрый анализ химических соединений, нитей ДНК и РНК, а раньше для этого требовалось громоздкое лабораторное оборудование. При постоянно прогрессирующих технологиях встроенные системы настолько уменьшились в размерах, что появилась возможность их ввода в организм человека для наблюдения за его здоровьем.

Не только на Земле, но и на Луне, на Марсе и на астероидах космические зонды и вездеходы ежедневно выполняют бесчисленное множество задач с помощью многократно проверенных и испытанных встроенных систем. Исследования Луны стали возможными благодаря первому экземпляру встроенной системы в форме компьютерной системы Apollo Guidance Computer. В 1966 году была создана встроенная система, состоящая из соединенных проводами печатных плат и заполненная логическими вентилями НЕ с тройными входами, специально предназначенная для обработки навигационной информации, управления и контроля командным модулем и лунным модулем, носителем для которых стали ракеты Saturn V.

Широкое распространение и универсальная природа встроенных систем сделали их неотъемлемой частью современной жизни.

Встроенные системы обычно классифицируют по следующим категориям:

- микроконтроллеры (MCU);
- система на кристалле (System-on-Chip – SoC), которую часто называют одноплатным компьютером (Single-Board Computer – SBC).

Микроконтроллеры

Одним из решающих факторов при инновациях в области встроенных систем является стоимость, поскольку встроенные системы чаще всего должны представлять собой широко распространенную дешевую потребительскую продукцию. Такой подход помогает получить полноценный микропроцессор, память, устройство хранения данных и периферийные устройства ввода/вывода в одной микросхеме, упрощая реализацию, исключая необходимость печатной платы, но при этом добавляя преимущества более производительного и простого проектного решения и высокоэффективного производства. В 1970-е годы это привело к разработке микроконтроллеров (MCU): компьютерных систем на одной микросхеме, которые можно добавлять в новое проектное решение с минимальными затратами.

С внедрением энергонезависимого электрически стираемого перепрограммируемого ПЗУ (EEPROM) в технологию производства микроконтроллеров в начале 1990-х годов сначала появилась возможность многократной перезаписи программной памяти микроконтроллеров без операции стирания содержимого памяти с помощью ультрафиолетового луча, направляемого через специальное кварцевое окно в корпусе микроконтроллера. Это позволило существенно упростить прототипирование и в дальнейшем снизить стоимость, а также разработать и внедрить внутрисхемное программирование.

В результате многие системы, которые ранее управлялись сложными механическими и аналоговыми устройствами (например, лифты и регуляторы температуры), были оснащены одним или несколькими микроконтроллерами, которые обеспечивали ту же функциональность при более низкой цене и более высокой надежности. Используя возможности, управляемые программно, разработчики стали беспрепятственно добавлять расширенные функции, такие как сложные предварительно устанавливаемые программы (для стиральных машин, микроволновых печей и т. п.) и упрощенное управление комплексными дисплеями для обеспечения обратной связи с пользователем.

TMS 1000

Первым коммерчески распространяемым микроконтроллером был TMS 1000 компании Texas Instruments, универсальная 4-битовая система на кристалле. Этот микроконтроллер впервые поступил в широкую продажу в 1974 году. Первый вариант модели имел 1 Кб ПЗУ (ROM), 64×4 бита ОЗУ (RAM) и 23 контакта ввода/вывода. Частота варьировалась от 100 до 400 КГц, при этом каждая инструкция выполнялась за шесть циклов таймера.

В более поздних моделях была возможность увеличения размеров ПЗУ и ОЗУ, но общее проектное решение в основном оставалось неизменным вплоть до прекращения производства этого микроконтроллера в 1981 году.

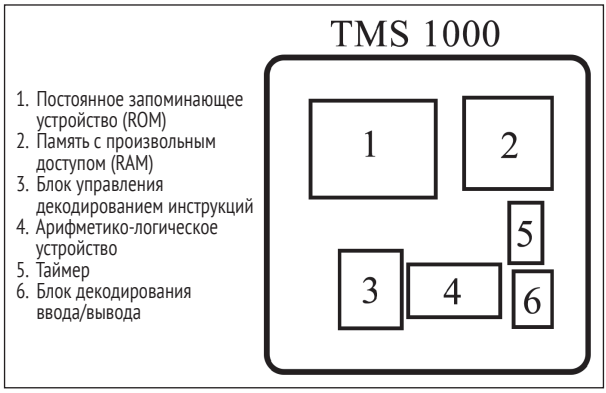

Рис. 1.1

Размер пластины этого микроконтроллера составлял приблизительно 5×5 мм, то есть был достаточно компактным для упаковки в корпус типа DIP (dual in-line package). В этом типе микроконтроллера использовалось маскируемое (программируемое при помощи маски) ПЗУ, то есть вы не могли получить «пустой» чип TMS 1000 и самостоятельно запрограммировать его. Вместо этого вы должны были передать отлаженную и адаптированную программу в компанию Texas Instruments для физического производства микросхемы с использованием фотолитографической маски, позволяющей создать металлический связующий мостик для каждого бита.

Это было достаточно примитивное проектное решение (по сравнению с более поздними микроконтроллерами), в котором отсутствовал стек и возможность обработки прерываний. Микроконтроллер работал с набором из 43 инструкций и имел два регистра общего назначения, что придавало ему некоторую схожесть с центральным процессорным устройством Intel 4004. Некоторые модели оснащались специализированными периферийными устройствами для управления вакуумно-люминесцентными (катодолюминесцентными) индикаторами (vacuum fluorescent displays – VFD) и для непрерывного считывания входных данных для обработки пользовательского ввода с клавиатуры без прерывания основной программы. Основная схема расположения контактов показана на рис. 1.2.

По рис. 1.2 становится понятно, что функции контактов являлись предшественниками функций контактов интерфейса ввода/вывода общего назначения (GPIO), хорошо известного нам сегодня, – контакты K могут использоваться только для ввода, в то время как контакты для вывода обозначены буквой O, а управляющие контакты помечены буквой R. Контакты OSC предназначены для соединения со схемой внешнего осциллографа. Как и в большинстве интегральных схем дискретной логики, контакт Init используется для инициализации микросхемы при подключении электроэнергии и должен сохранять высокую скорость выполнения, не более шести циклов, в то время как в более современные модели мик-

роконтроллеров интегрированы автоматические устройства реинициализации (power-on reset – POR), поэтому соответствующий контакт требует лишь наличия дискретного резистора или конденсатора.

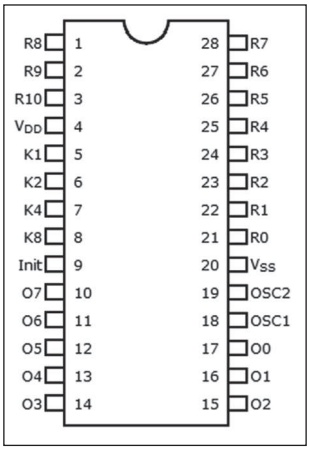

Рис. 1.2

В пресс-релизе компании Texas Instruments, выпущенном в 1974 году, указано, что этот микроконтроллер можно приобрести за 3 доллара, а при покупке крупной партии цена снижается. Предполагалось их использование не только в популярных электронных игрушках, таких как Speak and Spell, но почти везде, в том числе в бытовых приборах, автомобилях и научном оборудовании. До того, как производство TMS 1000 было прекращено в начале 1980-х годов, были проданы миллионы этих микроконтроллеров.

Также следует отметить, что стоимость однократно программируемых дешевых микроконтроллеров существенно снизилась, но этот тип продукции остается востребованным на рынке, например микроконтроллер Padauk PM150C сейчас можно приобрести за 0,03 долл., и хотя он предлагает 8-битовую архитектуру, 1 Кб слов ПЗУ и 64 байта ОЗУ кажутся подозрительно знакомыми.

Intel MCS-48

Ответом компании Intel на успешный микроконтроллер TMS 1000 компании Texas Instruments стала серия MCS-48, первые модели которой, 8048, 8035 и 8748, были выпущены в 1976 году. Модель 8048 имела 1 Кб ПЗУ и 64 байта ОЗУ. Это 8-битовое проектное решение с гарвардской архитектурой (раздельная память для кода/данных), в которой был введен внутренний 8-битовый размер слова и поддержка прерываний (на двух отдельных уровнях). Также обеспечивалась совместимость с микросхемами управления периферией 8080/8085, таким образом, MCS-48 представлял собой весьма универсальное семейство микроконтроллеров. Преимущество более развитых функций АЛУ и размера регистровых слов остается весьма заметным и по сей день, когда, например, 32-битовое расширение последовательно выполняется на 8-битовом микроконтроллере как группа 8-битовых расширений с соблюдением осторожности.

Для серии MCS-48 определено более 96 инструкций, большинство которых работают с данными длиной в один байт, а также допускается добавление расширенной памяти к внутреннему блоку ОЗУ. Благодаря усилиям сообщества вся доступная информация о семействе микроконтроллеров MCS-48 была собрана и опубликована на сайте **https://devsaurus.github.io/mcs-48/mcs-48.pdf**.

Здесь мы рассматриваем упрощенную функциональную блок-схему семейства микроконтроллеров MCS-48 и сравниваем ее с последующими семействами моделей.

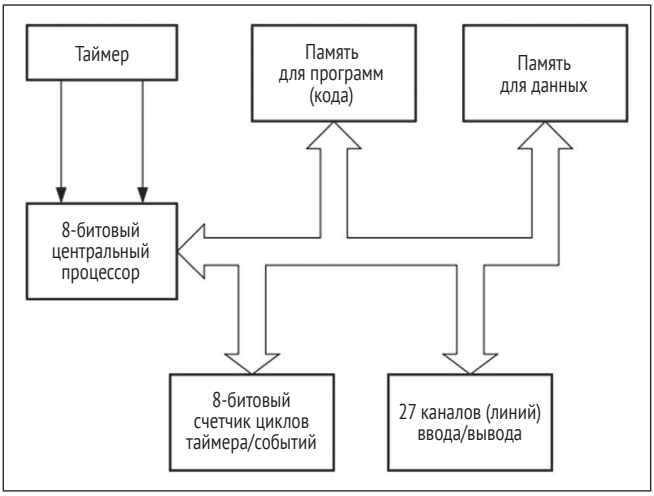

Рис. 1.3

Несмотря на то что это проектное решение было реализовано всего лишь несколькими годами позже TMS 1000, быстрое развитие архитектуры микроконтроллеров очевидно. Поскольку процесс проектирования микроконтроллеров происходил параллельно с проектированием широко распространенных в то время центральных процессоров, включая 16-битовую версию 6502, которая в конечном итоге стала основой семейства процессоров M68K, то в проектных решениях обнаруживается много похожих характеристик.

Благодаря гибкому и универсальному проектному решению это семейство микроконтроллеров оставалось широко распространенным и производилось до 1990-х годов, пока серия MCS-51 (8051) постепенно не заменила его. В следующем разделе микроконтроллер 8051 рассматривается более подробно.

Микроконтроллер MCS-48 использовался как контроллер клавиатуры в самой первой модели IBM PC. Он также применялся вместе с процессорами 80286 и 80386 как вентиль линии A20 и для выполнения функций рестарта для процессора 80286. В более поздних моделях PC эти функции были интегрированы в устройства Super I/O.

Еще один заслуживающий внимания вариант использования MCS-48 – игровая видеоконсоль Magnavox Odyssey и ряд моделей аналоговых синтезаторов Korg и Roland. Маскируемое ПЗУ (размером до 2 КБ) являлось дополнительной опцией для семейства MCS-48, но в микросхеме 87P50 уже использовался внешний

модуль ПЗУ для ее программирования, а микросхемы 8748 и 8749 имели до 2 Кб перезаписываемого ПЗУ (EPROM – Erasable Programmable Read Only Memory), позволяющего многократно перепрограммировать внутреннее содержимое микроконтроллера.

Как и для независимых модулей EPROM, для микроконтроллеров требовался специальный корпус со встроенным кварцевым окном, позволяющим ультрафиолетовому лучу света воздействовать на пластину микросхемы. Это кварцевое окно отчетливо видно на рис. 1.4 на фотографии микроконтроллера 8749 с модулем EPROM (автор фото Константин Ланзет (Konstantin Lanzet), лицензия CC BY-SA 3.0).

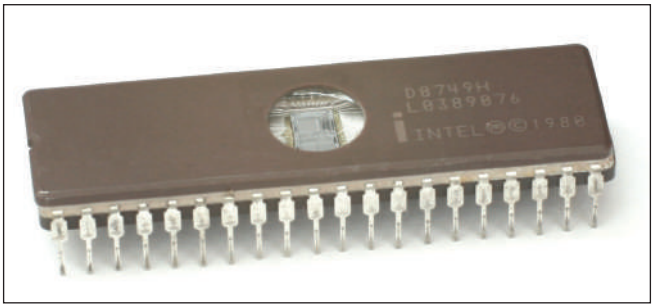

Рис. 1.4

Заряд ячеек EPROM, определяющий запись битов, уничтожается за 20–30 минут воздействия сильного ультрафиолетового излучения. Тот же эффект достигается при прямом воздействии солнечного света в течение нескольких недель. Цикл стирания обычно предполагает удаление корпуса и воздействие света непосредственно на стираемое устройство. После этого EPROM можно перепрограммировать заново. Спецификацией определен срок хранения данных в EPROM: приблизительно 10–20 лет при температуре около 85 °C, но поскольку срок хранения уменьшается экспоненциально при росте температуры, то при комнатной температуре можно рассчитывать на сохранность данных в течение 100 лет и более (для микросхемы 27C512A – 200 лет).

Из-за дополнительных затрат на создание кварцевого окна и вмонтирование его в корпус однократно программируемые EPROM использовались в течение некоторого времени, что позволяло легко программировать EPROM, но при помещении программируемой пластины микросхемы в непрозрачный корпус возможность перепрограммирования исчезла. В конце концов, в начале 1980-х годов стали доступными модули EEPROM (Electrically Erasable Programmable Read Only Memory), которые почти полностью заменили EPROM. Модули EEPROM можно перезаписывать около миллиона раз, прежде чем начнут возникать проблемы с сохранностью записанных в них данных. Надежность хранения данных почти ничем не отличается от модулей EPROM.

Intel MCS-51

Ряд недавно выпущенных микросхем – от Cypress CY7C68013A (контроллер периферийных USB-устройств) до Ti CC2541 (Bluetooth-система на кристалле) – содер-

жит ядра 8051, что свидетельствует о сохранении широкой распространенности архитектуры семейства Intel MCS-51 в наши дни. Существует огромное количество производных модификаций, в том числе и от других производителей, несмотря на то что компания Intel прекратила производство этой серии микроконтроллеров в марте 2007 года. Этот 8-битовый микроконтроллер, впервые появившийся в 1980-х годах, похож на серию 8048, но со значительным расширением набора функциональных возможностей.

Функциональная блок-схема, взятая из спецификации Intel 80xxAH, показана на рис. 1.5.

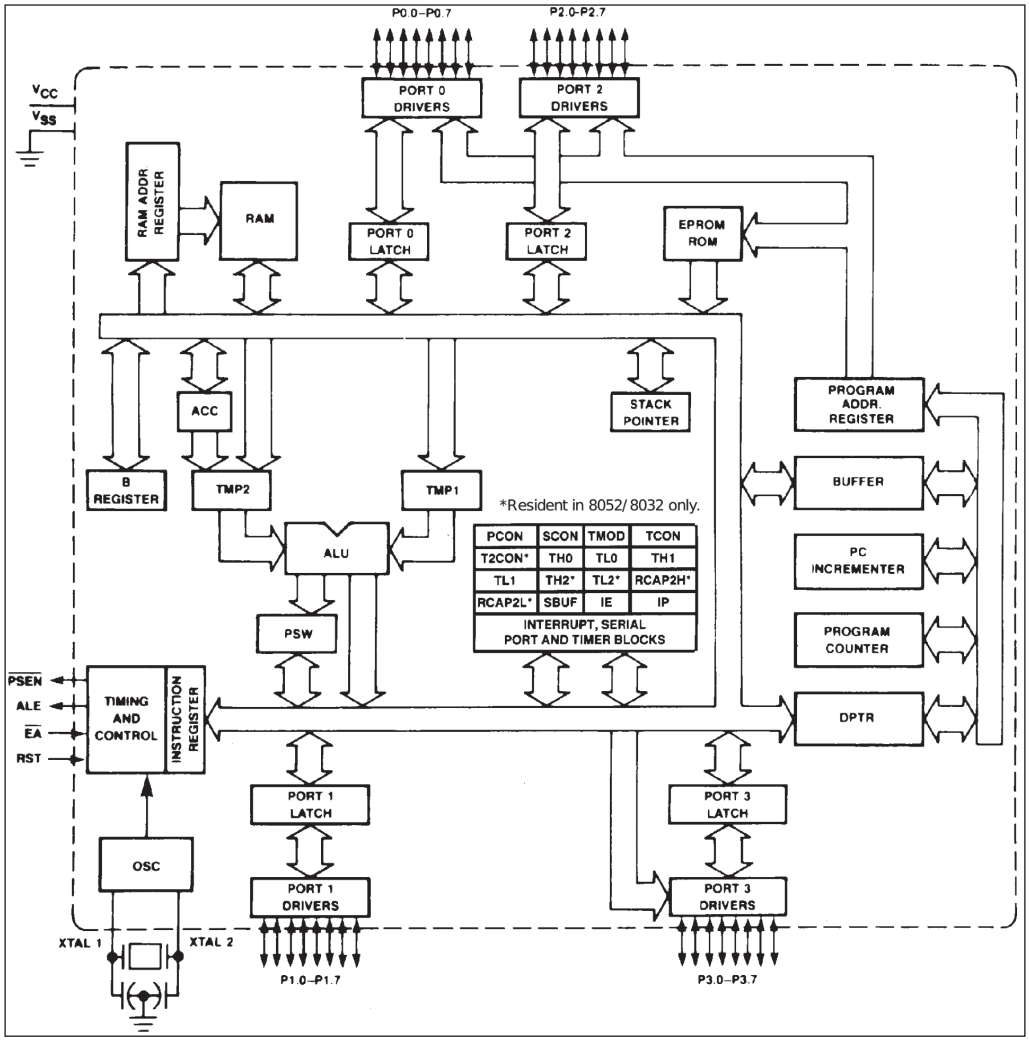

Рис. 1.5

Схема очень похожа на Atmel (сейчас это микрочип) AT89S51, который выпускается и в наши дни.

В спецификации определены общие рабочие характеристики в списке Features, часть которого, относящаяся к AT89S51, приведена ниже:

- 4 Кб программируемой в эксплуатируемой системе (in-system programmable – ISP) флеш-памяти – долговечность (выносливость): 10 000 циклов записи/стирания (ранее 1 000 000 для EEPROM);
- режим работы от 4,0 до 5,5 В;
- полностью статическая операция: от 0 Гц до 33 МГц (ранее 12 МГц);
- трехуровневая блокировка программной памяти;
- внутреннее ОЗУ (RAM) 128×8 бит;
- 32 программируемых канала (линии) ввода/вывода.

Затем в этом же списке перечисляются современные улучшенные характеристики: ядро, периферия, низкое энергопотребление и функциональные возможности, улучшающие удобство использования:

- два 16-битовых таймера/счетчика;
- шесть источников прерываний;
- полнодуплексный последовательный канал UART;
- режимы ожидания (Idle) и «сна» с пониженным энергопотреблением;
- прерывание для восстановления из режима «сна»;
- сторожевой таймер;
- двойной указатель на данные;
- флаг отключения электропитания;
- возможность быстрого программирования;
- гибкое программирование в режиме эксплуатации системы (ISP) в побайтовом или постраничном режиме.

За последние десятилетия в архитектуре 8051 значительным изменением стал только переход от первоначальной технологии транзисторов NMOS (n-type metal oxide semiconductor) к технологии CMOS (complementary MOS), обычно обозначаемый как 80C51, а также более позднее добавление интерфейсов USB, I2C и SPI и усовершенствованное управление энергопотреблением. Кроме того, были добавлены отладочные интерфейсы, которые стали вездесущими с начала XXI века. В примечаниях к спецификации Amtel 3487A не приводится даже краткое объяснение смысла буквы S, тем не менее ниже можно особо выделить новую функцию последовательного программирования микросхемы в эксплуатируемой системе (ISP).

Схема расположения контактов (ножек) микросхемы AT89S51 определяет назначение контактов SPI (Serial Peripherial Interface) (MOSI – Master Output Slave Input; MISO – Master Input Slave Output; SCK – Serial Clock), как показано на рис. 1.6.

Помимо автономных микроконтроллеров, ядра 8051 также включаются в состав более крупных систем, где простые микроконтроллеры с низким энергопотреблением предназначены для разнообразных задач, связанных с мониторингом операций ввода/вывода как с низкими, так и с высокими скоростями и даже в реальном времени. Широкий спектр микросхем от аналогов Ti CC2541 (Bluetooth-система на кристалле с низким энергопотреблением) до Cypress CY7C68013A (FX2LP™ контроллер периферийных USB-устройств) подчеркивает полезность и надежность архитектуры 8051 и в наши дни.

В процессе разработки программируемой пользователем вентильной матрицы (field-programmable gate array – FPGA) и интегральной схемы специального назна-

чения (application specific integration circuit – ASIC) процессоры типа 8051 широко использовались в качестве программируемых ядер. Этот тип микросхемы также адаптирован и добавлен в проекты VHDL и Verilog HDL для задач, которые проще решаются с помощью последовательного выполнения при отсутствии жестких требований к увеличенной пропускной способности (производительности) или ограничений по времени. Последнее, но не менее важное замечание: преимущество программируемых ядер заключается в их способности полноценного использования инструментальных средств разработки и отладки с обеспечением тесной интеграции при сохранении неизменности проектного решения аппаратной части. Всего лишь несколько сотен байтов программного кода, выполняемого программируемым ядром, могут успешно заменять крупные механизмы управления состояниями, большие блоки памяти, многочисленные счетчики и поддерживать логику уровня АЛУ. В результате возникает вопрос: какая реализация проще с точки зрения проверки и сопровождения?

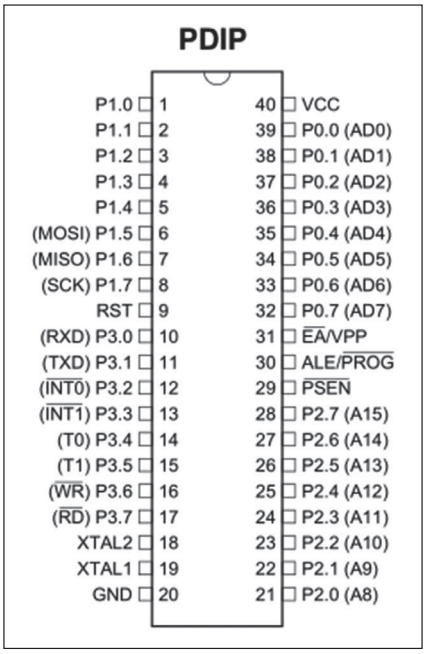

Рис. 1.6

PIC

Семейство микроконтроллеров PIC было выпущено в 1976 году компанией General Instrument, которая использовала для этих микроконтроллеров свой новый 16-битовый процессор CP1600. Этот центральный процессор был почти совместимым по набору инструкций с серией процессоров PDP-11 (компании Digital Equipment Corporation, DEC).

В 1987 году компания General Instrument расширила и укрепила свой отдел микроэлектроники, создав крупное подразделение Microchip Technology, которое

стало независимой компанией в 1989 году. Компания Microchip Technology производит новые модификации PIC и сейчас. Вместе с развитием ядер и периферии PIC разработка памяти, интегрированной непосредственно в микросхему, позволила начать внедрение слабосвязанной инкапсулированной памяти EPROM для однократно программируемого чипа, а в дальнейшем – EEPROM для обеспечения возможностей многократного перепрограммирования в период эксплуатации микроконтроллера. Как и большинство микроконтроллеров, PIC имеет гарвардскую архитектуру. В наше время выпускается линия микроконтроллеров PIC от 8-битовых до 32-битовых с широким спектром функциональных возможностей. Во время написания данной книги микроконтроллер PIC был представлен несколькими семействами, описанными в табл. 1.1.

Таблица 1.1

Семейство	Контакты	Память	Дополнительные характеристики
PIC10	6–8	384–896 байт ПЗУ, 64–512 байт ОЗУ	8-битовый, 8–16 МГц, модифицированная гарвардская архитектура
PIC12	8	2–16 Кб ПЗУ, 256 байт ОЗУ	8-битовый, 16 МГц, модифицированная гарвардская архитектура
PIC16	8–64	3,5–56 Кб ПЗУ, 1–4 Кб ОЗУ	8-битовый, модифицированная гарвардская архитектура
PIC17	40–68	4–16 Кб ПЗУ, 232–454 байта ОЗУ	8-битовый, 33 МГц, вытеснен моделью PIC18, хотя продолжают выпускаться клоны сторонних производителей
PIC18	28–100	16–128 Кб ПЗУ, 3728–4096 байт ОЗУ	8-битовый, модифицированная гарвардская архитектура
PIC24 (dsPIC)	14–144	64–1024 Кб ПЗУ, 8–16 Кб ОЗУ	16-битовый, в микроконтроллеры DsPIC (dsPIC33) встроены периферийные устройства для цифровой обработки сигналов
PIC32MX	64–100	32–512 Кб ПЗУ, 8–32 Кб ОЗУ	32-битовый, 200 МГц MIPS M4K с режимом MIPS16e, выпущен в 2007 году
PIC32MZ EC PIC32MZ EF PIC32MZ DA	64–288	512–2048 Кб ПЗУ, 256–640 Кб статического ОЗУ (32 Мб DDR2 DRAM)	32-битовые, MIPS ISA (2013), версия PIC32MZ DA (2017) содержит графическое ядро. Тактовые частоты ядер 200 МГц (ec, DA) и 252 МГц (EF)
PIC32MM	20–64	16–256 Кб ПЗУ, 4–32 Кб ОЗУ	32-битовый microMIPS, 25 МГц, вариант, оптимизированный по низкой цене и низкому энергопотреблению
PIC32MK	64–100	512–1024 Кб ПЗУ, 128–256 Кб ОЗУ	32-битовый, 120 МГц, MIPS ISA, вариант, созданный в 2017 году. Ориентирован на управление в производстве и другие формы глубоко интегрированных приложений

Семейство PIC32 интересно тем, что основано на ядре процессора MIPS и использует соответствующую архитектуру набора инструкций (Instruction Set Architecture – ISA) вместо PIC ISA, который применялся во всех прочих микроконтроллерах PIC. Используемая во всех микроконтроллерах этого семейства модель ядра процессора – M4K, 32-битовое ядро MIPS32 компании MIPS Technology. Различия между семействами PIC легко обнаружить, изучая блок-схемы из соответствующих спецификаций.

Вероятнее всего, историю развития линии микроконтроллеров PIC в течение нескольких десятилетий лучше всего изучать по функциональным блок-схемам, поэтому начнем с модели PIC10 (рис. 1.7).

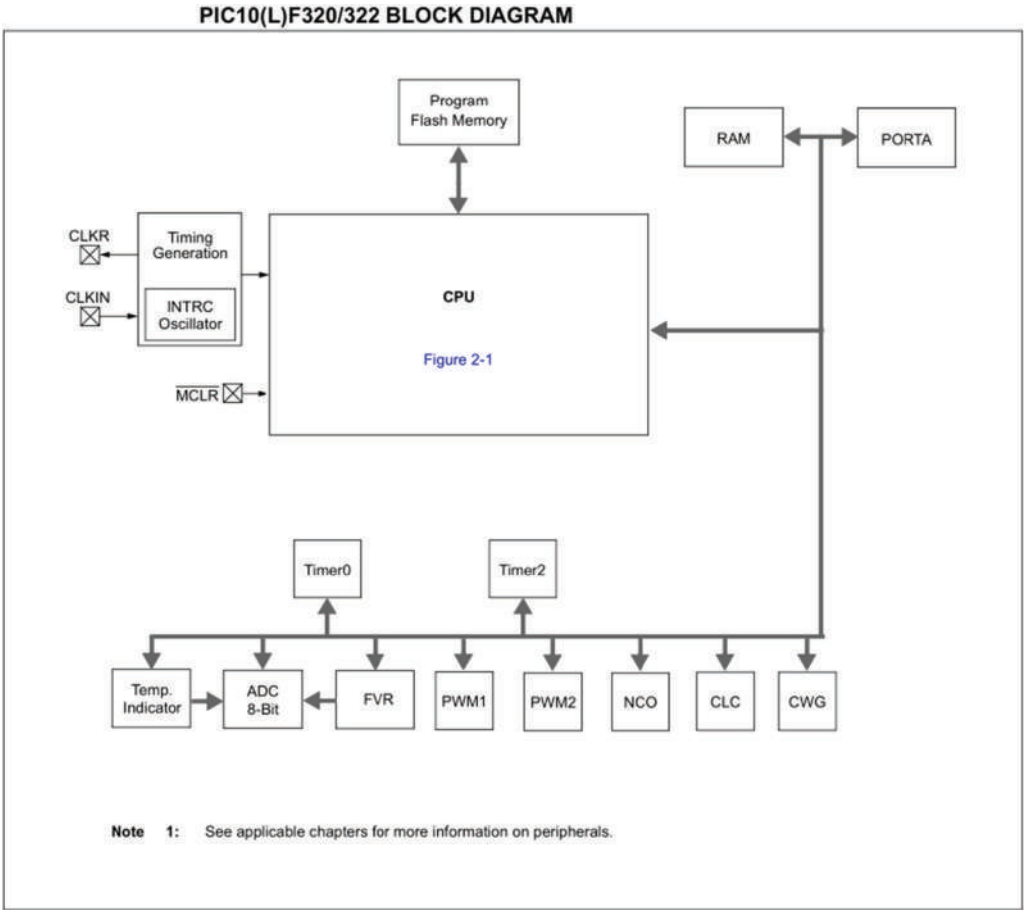

Рис. 1.7

Это очень маленькие микроконтроллеры с практически любыми периферийными устройствами, расположенными вокруг процессорного ядра, не описываются здесь более подробно, а в справочной таблице указаны только характеристики памяти. Количество портов ввода/вывода минимально, а интерфейсы I2C и UART, широко известные сегодня, не реализованы как периферийная логика. Для выбора следующего микроконтроллера этого семейства обратим внимание на то, что спецификация PIC16F84 весьма подробно описывает архитектуру процессора и сообщает, что добавлены схемы управления включением электропитания и перезагрузкой. Кроме того, расширены функции GPIO и добавлена память EEPROM для упрощения интеграции энергонезависимого устройства хранения данных. Последовательные периферийные устройства, размещенные непосредственно на пластине микроконтроллера, пока еще отсутствуют.

На рис. 1.8 показана функциональная блок-схема микроконтроллера PIC18.

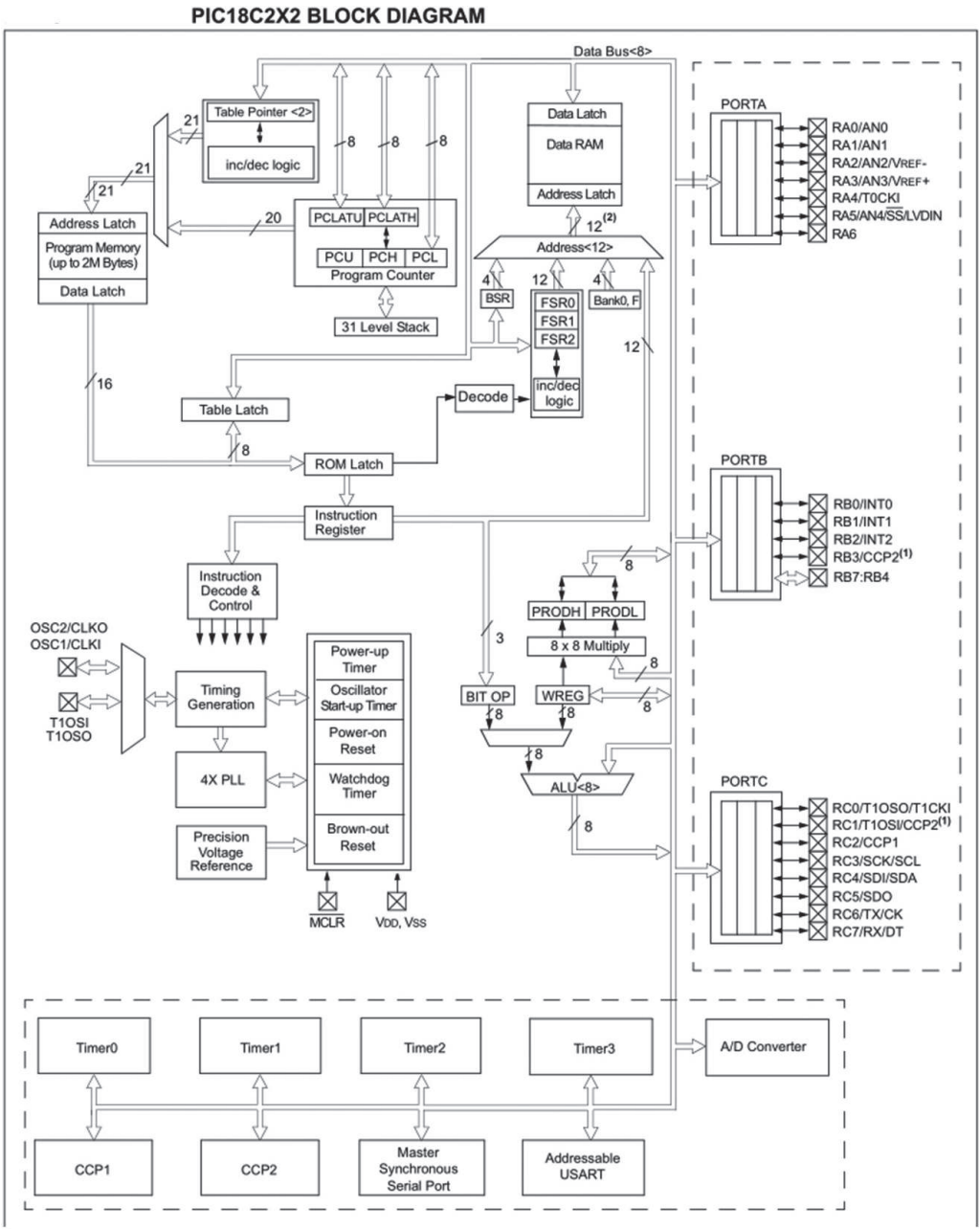

Рис. 1.8

Семейство PIC18 представляет самую последнюю версию 8-битовой архитектуры PIC, которая охватывает широкий диапазон приложений. Эта модель обеспе-

чивает значительно больше функций ввода/вывода по сравнению с семействами PIC10, PIC12 и PIC16, а кроме того, предлагает больше вариантов расширений ПЗУ и ОЗУ и содержит универсальный синхронный и асинхронный приемопередатчик (USART) в сочетании с синхронизированным последовательным портом для 4-проводного внешнего последовательного интерфейса (SPI). Также следует отметить, что теперь порты имеют изменяемые функции контактов, но маршруты от периферийных модулей к контактам и соответствующие конфигурационные регистры здесь не показаны для упрощения блок-схемы.

Теперь перейдем от ядра микроконтроллера к рассмотрению функциональных возможностей портов и периферийных модулей на функциональной блок-схеме PIC24 (рис. 1.9).

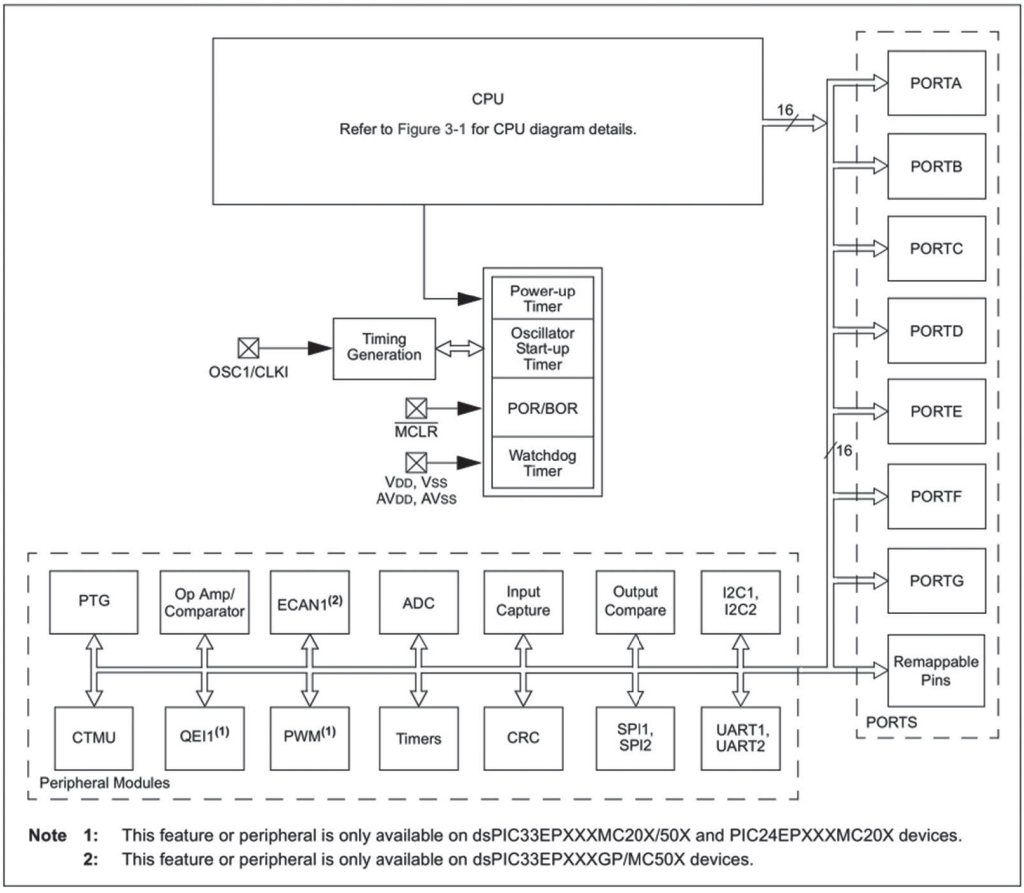

Рис. 1.9

Эта блок-схема похожа на схему для PIC10, здесь центральное процессорное устройство показано условно как единый блок по отношению к остальным компонентам микроконтроллера. Каждый из блоков портов имеет собственный на-

бор контактов ввода/вывода, но на этой блок-схеме невозможно показать все возможные функции контактов.

Каждый контакт ввода/вывода может иметь жестко закрепленную функцию (связанную с периферийным модулем) или переназначаемую функцию (изменение маршрута на аппаратном уровне или перепрограммирование). В общем случае чем более сложным является микроконтроллер, тем более вероятно, что его контакты ввода/вывода имеют обобщенные (изменяемые), а не жестко определенные функции.

Последней рассматриваемой здесь моделью является PIC32 (рис. 1.10).

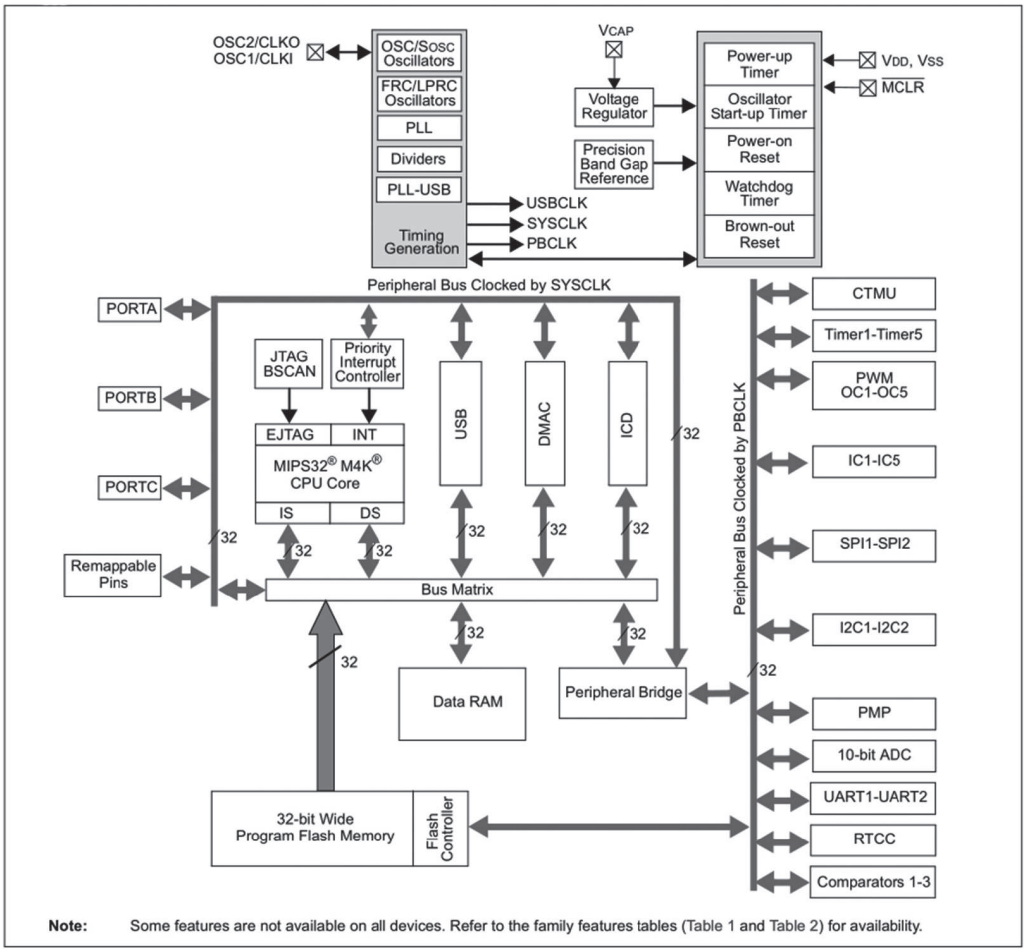

Рис. 1.10

На рис. 1.10 показана блок-схема для устройств PIC32MX1XX/2XX из семейства PIC32MX. Обычно эти устройства работают на тактовой частоте 50 МГц.

Интересным свойством архитектуры PIC32 является эффективное преобразование гарвардской архитектуры ЦПУ M4K MIPS в архитектуру, более похожую на

фон-неймановскую (John von Neumann), с передачей программных инструкций и данных по системной матрице шин (System Bus Matrix). Отметим, что блок, обозначающий единственный регистр процессора на блок-схеме PIC10, теперь абстрактно отображает сложный периферийный модуль, цифровой или со смешанным сигналом, или мощный стандартный аппаратный интерфейс тестирования и отладки (JTAG), размещенный непосредственно на панели микроконтроллера.

AVR

Архитектура AVR была разработана двумя студентами Норвежского технологического института, а самый первый микроконтроллер AVR был разработан в компании Nordic VLSI (теперь Nordic Semiconductor). Сначала микроконтроллер назывался µRISC и для обеспечения лицензионной защиты технологии был продан компании Atmel. Первый микроконтроллер Atmel AVR был выпущен в 1997 году.

В настоящее время можно вспомнить некоторые модели из многочисленных семейств 8-битовых микроконтроллеров AVR (табл. 1.2).

Таблица 1.2

Семейство	Контакты	Память	Дополнительные характеристики
ATtiny	6–32	0,5–16 Кб ПЗУ, 0–2 Кб ОЗУ	1,6–20 МГц, компактный микроконтроллер с низким энергопотреблением и ограниченными периферийными модулями
ATmega	32–100	4–256 Кб ПЗУ, 0,5–32 Кб ОЗУ	
ATxmega	44–100	16–384 Кб ПЗУ, 1–32 Кб ОЗУ	32 МГц, самый крупный микроконтроллер AVR с расширяемыми периферийными модулями и функциональными возможностями, улучшающими производительность, такими как DMA

Также использовалась 32-битовая архитектура AVR32, но не была принята компанией Atmel, так как компания перешла на другую 32-битовую архитектуру ARM (SAM). Немного больше об архитектуре SAM можно узнать из подраздела «Микроконтроллеры на основе ARM». Но самая подробная информация содержится в соответствующем руководстве «Product Selection Guide».

Кроме того, компания Atmel применяла так называемые контроллеры с программируемой пользователем на системном уровне интегральной микросхемой (Field Programmable System Level Integrated Circuit – FPSLIC): гибрид систем AVR/FPGA. По существу, такой вариант позволяет пользователю добавлять собственные периферийные модули и функциональность непосредственно в аппаратуру микроконтроллера AVR.

Рассмотрим подробнее семейство микроконтроллеров ATtiny. На рис. 1.11 показана блок-схема серии ATtiny212/412.

Эта серия микроконтроллеров ATtiny может работать на тактовых частотах до 20 Мгц, иметь до 4 Кб флеш-ПЗУ и 256 байт статической оперативной памяти (SRAM), а также до 128 байт EEPROM. Все это размещено в корпусе с 8 контактами. Несмотря на малый размер, микроконтроллер содержит множество периферий-

ных модулей, которые могут быть соединены с любым поддерживаемым контактом, как показано на рис. 1.12.

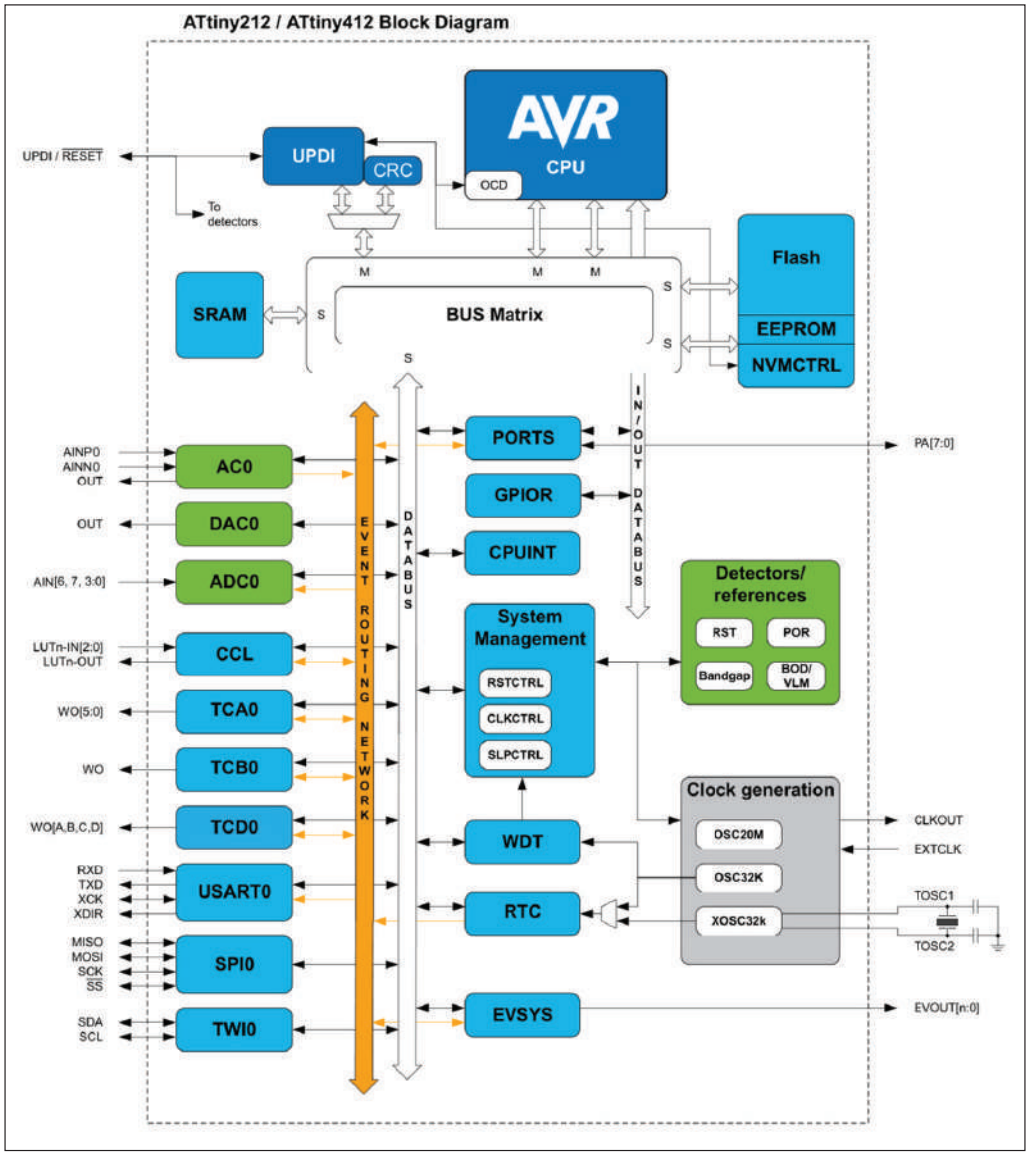

Рис. 1.11

Рис. 1.12

Для сравнения в табл. 1.3 приведены характеристики широко распространенного микроконтроллера ATmega2560 и некоторых других представителей этого семейства.

Таблица 1.3

Устройство	Флеш-память (Кб)	EEPROM (Кб)	ОЗУ (Кб)	Контакты ввода/вывода общего назначения	16-битовые каналы широтно-импульсной модуляции	UART	Каналы аналого-цифрового преобразования
ATmega640	64	4	8	86	12	4	16
ATmega1280	128	4	8	86	12	4	16
ATmega1281	128	4	8	54	6	2	8
ATmega2560	256	4	8	86	12	4	16
ATmega2561	256	4	8	54	6	2	8

В этом семействе микроконтроллеров насчитывается несколько десятков контактов интерфейса ввода/вывода общего назначения (GPIO), поэтому блок-схема существенно усложняется и содержит гораздо больше блоков портов для контактов ввода/вывода, как показано на рис. 1.13.

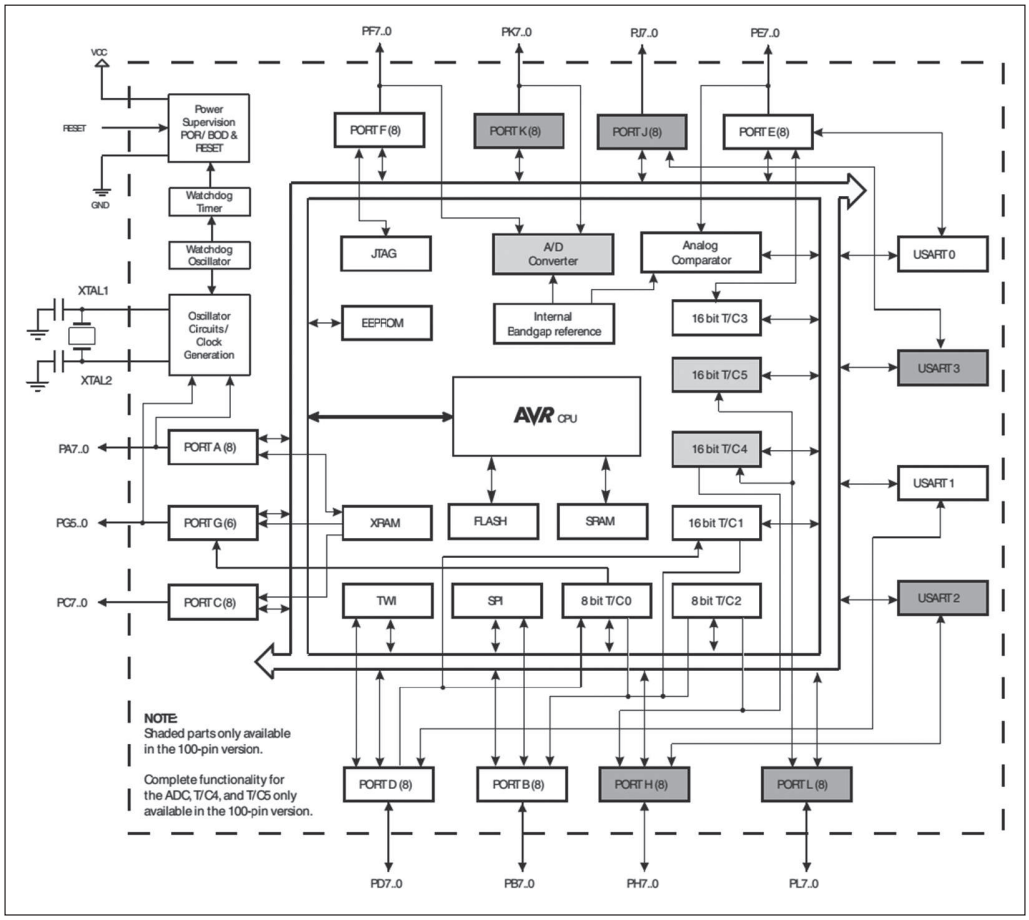

Рис. 1.13

На этой схеме все входящие и исходящие стрелки обозначают один контакт или блок контактов, большинство из которых имеет общее назначение. Из-за большого количества контактов больше нет смысла использовать на практике формат корпуса с расположением контактов в ряд (DIP, SOIC и проч.) для физической микросхемы.

Для моделей ATmega640, 1280 и 2560 используется корпус TQFP со 100 контактами, а функциональность каждого контакта описана в соответствующей спецификации (рис. 1.14).

Семейство ATxmega очень похоже на семейство ATmega с аналогичным расположением контактов, но различия в основном относятся к изменениям архитектуры и к дополнительной оптимизации. Кроме того, модели ATxmega оснащены ПЗУ и ОЗУ большего размера, а также более развитыми вариантами периферийных модулей.

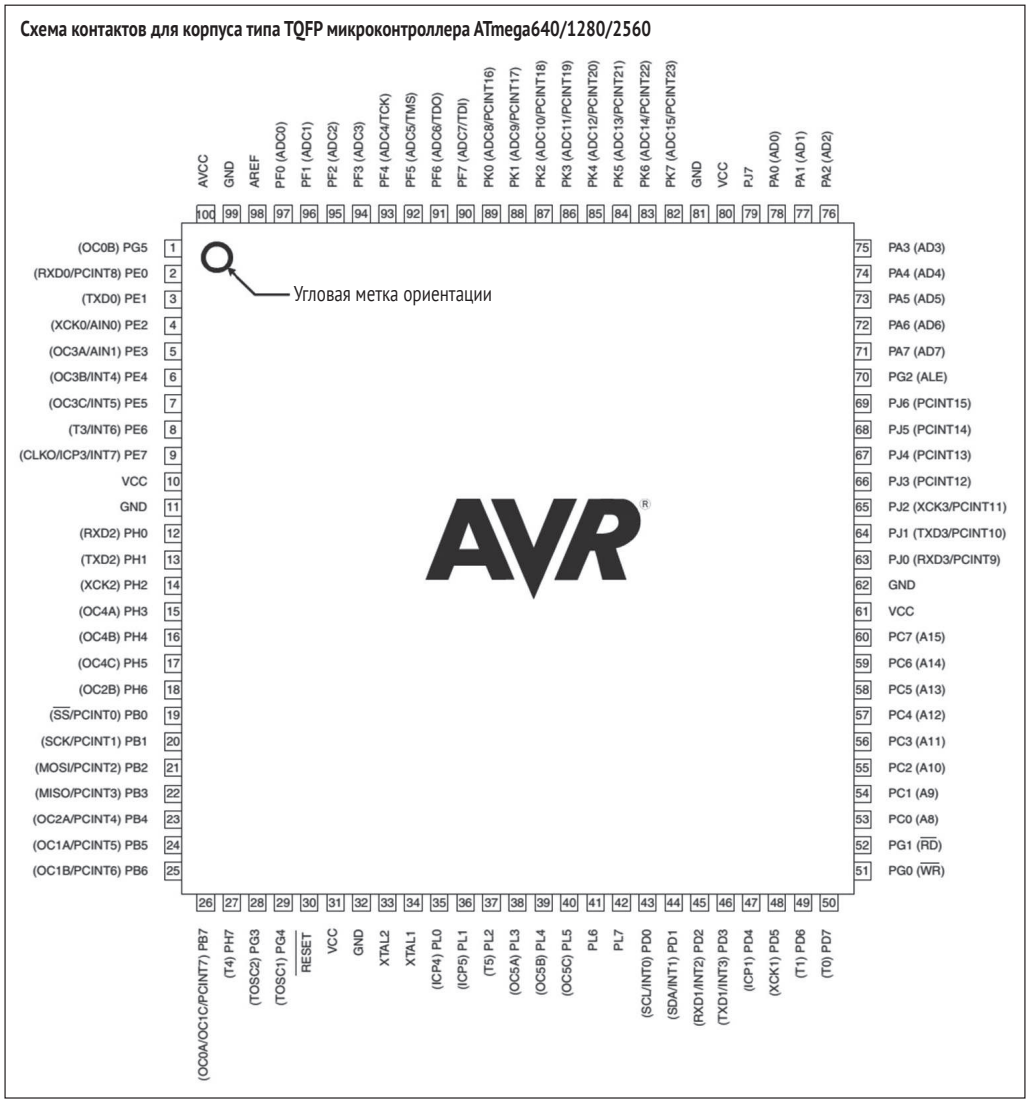

Рис. 1.14

Выбор микроконтроллера ATtiny, ATmega или ATxmega прежде всего зависит от требований конкретного проекта, в особенности от требований к вводу и выводу, от необходимых типов периферийных устройств (последовательные устройства, SPI, I2C, CAN и т. д.), а также от размера программного кода и от размера ОЗУ, требуемого для выполнения этого кода.

M68k и микроконтроллеры на основе Z80

8-битовый процессор Zilog Z80 совместим с процессором Intel 8080, который конкурировал с другими микропроцессорами на протяжении 1980-х годов. Борьба шла за преобладание в домашних компьютерах и в игровых системах, в том числе в Nintendo Game Boy, Sega Master System, Sinclair ZX80/ZX81/Spectrum, MSX и Tandy TRS-80.

Компания Zilog представила микроконтроллер Z380 на основе микропроцессора Z80 в 1994 году и в течение нескольких лет обновляла модели, в том числе Z8, eZ80 и др. Кроме того, широко распространены клоны Z80.

Еще одним известным в 1980-е годы микропроцессором являлась модель Motorola 68k (или 68000). АЛУ и внешняя шина данных этого микропроцессора 16-битовые, но внутренняя шина данных и регистры 32-битовые. После появления M68k в 1979 году его архитектура не изменялась и используется до настоящего времени – компания Freescale Semiconductor (сейчас NXP) продолжает производство большого количества микропроцессоров 68k.

Компания Motorola создавала множество вариантов микроконтроллеров на основе архитектуры 68k, в том числе контроллер коммуникационной связи (обмена данными) MC68320 в 1989 году. В настоящее время к проектным решениям микроконтроллеров на основе 68k относится модель ColdFire, полностью 32-битовая микросхема.

ARM Cortex-M

Весьма распространенным типом 32-битовых микроконтроллеров является семейство ARM Cortex-M. К нему относятся модели M0, M0+, M1, M3, M4, M7, M23 и M33, большинство из которых предоставляет возможность использования сопроцессора операций с плавающей точкой (floating point unit – FPU) для улучшения производительности математических операций.

Это семейство используется не только в качестве автономных микроконтроллеров, но часто модели интегрируются в устройства типа «система на кристалле» (System-on-Chip – SoC) для обеспечения особой функциональности, например для управления сенсорным экраном, датчиками-сенсорами или энергопотреблением. Поскольку компания Arm Holdings сама не производит различные варианты и модификации микроконтроллеров, многие сторонние производители приобрели лицензии на эти проектные решения и иногда вносят собственные изменения и усовершенствования.

В табл. 1.4 приведены некоторые характеристики микроконтроллеров семейства ARM Cortex-M.

Таблица 1.4

Ядро	Год анонсирования	Архитектура	Набор инструкций
M0	2009	Armv6-M	Thumb-1, частично Thumb-2
M0+	2012	Armv6-M	Thumb-1, частично Thumb-2
M1	2007	Armv6-M	Thumb-1, частично Thumb-2
M3	2004	Armv7-M	Thumb-1, Thumb-2
M4	2010	Armv7-M	Thumb-1, Thumb-2, дополнительно FPU
M7	2014	Armv7E-M	Thumb-1, Thumb-2, дополнительно FPU
M23	2016	Armv8-M	Thumb-1, частично Thumb-2
M33	2016	Armv8-M	Thumb-1, Thumb-2, дополнительно FPU

Thumb – это компактный набор 16-битовых инструкций, практически идеальный для встроенных систем с ограниченными ресурсами. Другие семейства микропроцессоров ARM также поддерживают набор инструкций Thumb в дополнение к основному набору 32-битовых инструкций.

H8 (SuperH)

Семейство микроконтроллеров H8 широко использовалось в 8-, 16- и 32-битовых вариантах. Это семейство в начале 1990-х годов начала выпускать компания Hitachi, а новые проектные решения продолжала создавать компания Renesas Technology до недавнего времени, хотя в последние годы рекомендуется использовать новейшие решения – семейства RX (32-битовое) и RL78 (16-битовое). Заслуживает внимания использование микроконтроллера H8 в контроллере Lego Mindstorms RCX, где применяется модель H8/300.

ESP8266/ESP32

ESP – это семейство 32-битовых микроконтроллеров, выпускаемых компанией Espressif Systems, в которые включена функциональная поддержка протоколов Wi-Fi (в обе модели) и Bluetooth (ESP32).

Микроконтроллер ESP8266 появился в 2014 году и сразу был продан стороннему производителю – компании Ai-Thinker в виде модуля (ESP-01), который мог использоваться другим микроконтроллером или системой на основе микропроцессоров для обеспечения функциональной поддержки протокола Wi-Fi. Модуль ESP-01 содержал встроенное программное обеспечение для этой цели, которое позволяло работать с модулем, применяя команды в стиле модемов семейства Hayes.

Ниже приведены некоторые системные характеристики модуля ESP-01:

- 32-битовый микропроцессор Tensilica Xtensa Diamond Standard L106;
- тактовая частота ЦПУ 80–160 МГц;
- до 50 Кб ОЗУ, доступного пользовательским приложениям (с загруженным стеком Wi-Fi);
- ПЗУ внешнего последовательного интерфейса SPI ROM (от 512 Кб до 16 Мб);
- поддержка Wi-Fi для 802.11 b/g/n.

Поскольку 32-битовый микроконтроллер на модуле ESP-01 обладал гораздо большей функциональностью, чем простейший модем, вскоре он стал использоваться для более общих задач, тем более что появились обновленные модули ESP8266 (со встроенной микросхемой EEPROM), а также переходные платы (адаптеры). Не так давно тип адаптера NodeMCU стал весьма распространенным, несмотря на то что другие производители создали свои версии переходных плат в разнообразных форм-факторах и с различной функциональностью.

На рис. 1.15 показана упрощенная блок-схема микроконтроллера ESP8266EX.

После невероятно успешной модели ESP8266 компания Espressif Systems разработала модель ESP32, в которой использовался обновленный двухъядерный центральный процессор, а также были внесены другие изменения. На рис. 1.16 показана блок-схема ESP32.

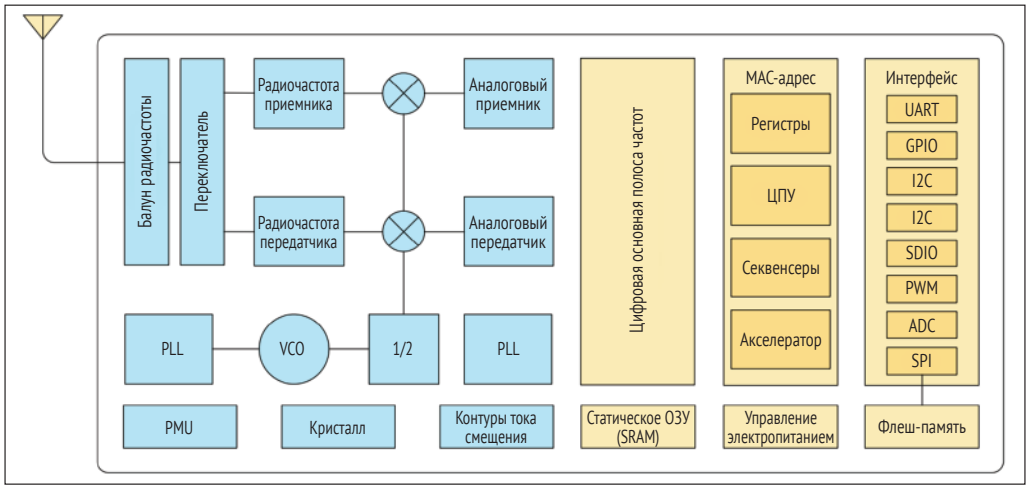

Рис. 1.15

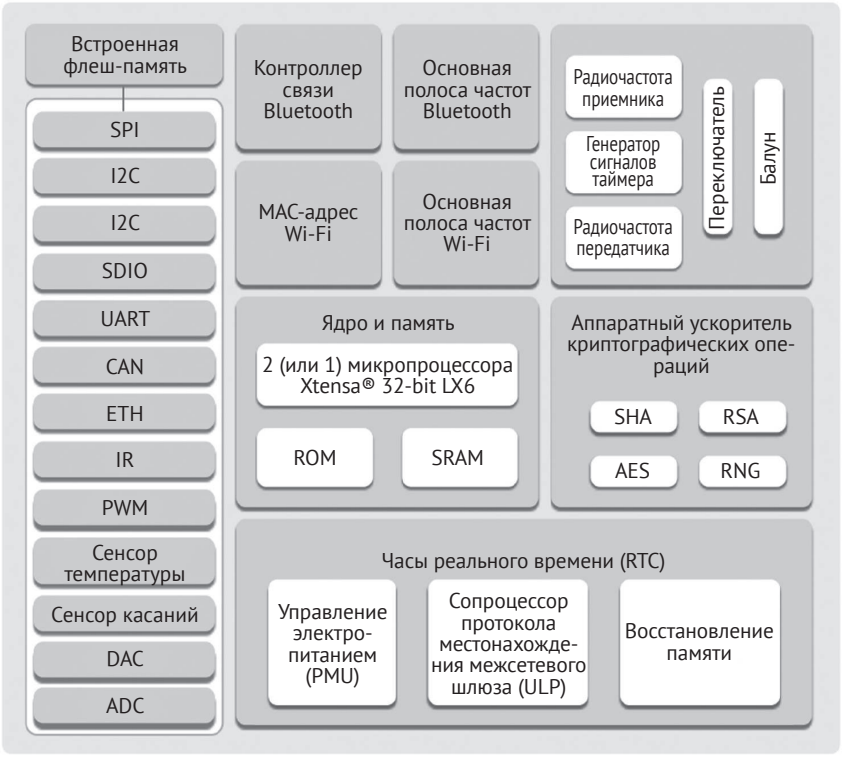

Рис. 1.16

Ниже перечислены некоторые технические характеристики ESP32:

- ❍ микропроцессор Xtensa 32-bit LX6 (двухъядерный);
- ❍ тактовая частота ЦПУ 160–240 МГц;

- 520 Кб статического ОЗУ (SRAM);
- поддержка Wi-Fi для 802.11 b/g/n;
- поддержка Bluetooth v4.2 и BLE (с низким энергопотреблением).

Обе модели ESP8266 и ESP32 массово продаются как полнофункциональные модули с микроконтроллером, модулем внешней памяти ROM и антенной Wi-Fi, либо смонтированной непосредственно на плате, либо во внешнем исполнении (рис. 1.17).

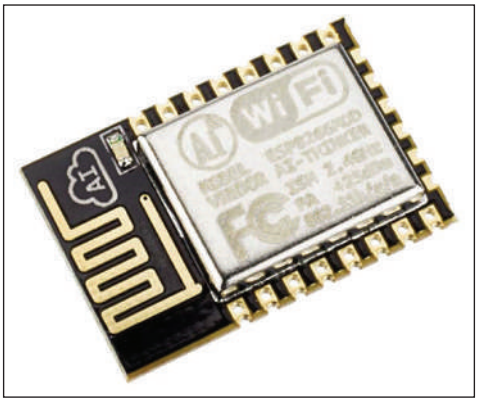

Рис. 1.17

Возможен вариант с металлическим корпусом, защищающим плату от электромагнитного воздействия и улучшающим характеристики трансивера Wi-Fi (и Bluetooth в модели ESP32), но в целом такое проектное решение со смонтированной на плате антенной и жестко определенной геометрией требует сертификации FCC и последующего использования как апробированный и санкционированный модуль. Несанкционированное подключение внешней антенны может нарушать местные законы. Поэтому необходимо получение идентификатора FCC ID, для того чтобы устройство, содержащее такой модуль, можно было применять в коммерческих целях.

Другие микроконтроллеры

Кроме описанных выше микроконтроллеров, существует множество микроконтроллеров с другими архитектурами от различных производителей. Некоторые из них, например Propeller компании Parallax, имеют многоядерную архитектуру и в определенной степени уникальны, тогда как в большинстве микроконтроллеров реализована обычная архитектура с одноядерным ЦПУ, набором периферийных модулей, ОЗУ и внутренним или внешним ПЗУ.

Кроме физических микросхем компании Altera (сейчас Intel), Lattice Semiconductor и Xilinx предлагают так называемые программные ядра, представляющие собой микроконтроллеры, работающие на программируемой пользователем вентильной матрице (FPGA), либо как независимые компоненты, либо как часть более крупного проектного решения на этой матрице FPGA. Для таких решений также существуют соответствующие компиляторы C/C++.

Особенности

Основные особенности и затруднения разработки для микроконтроллеров заключаются в относительно ограниченных доступных ресурсах. Особенно это касается небольших микроконтроллеров с малым количеством внешних контактов (ножек), для которых необходимо принять правильное проектное решение о том, сколько ресурсов (циклов ЦПУ, ОЗУ и ПЗУ) потребует конкретный блок кода и насколько реальным является добавление некоторой особенной функциональной возможности.

Кроме того, это означает, что при выборе правильного микроконтроллера для конкретного проекта нужны не только технические знания, но и практический опыт. Теоретические знания требуются для выбора микроконтроллера, полностью соответствующего задаче. Практический опыт очень полезен для подбора оптимальной модели (варианта) микроконтроллера, к тому же помогает сэкономить время на этапе выбора.

Одноплатный компьютер, или система на кристалле

Система на кристалле (System-on-Chip – SoC) похожа на микроконтроллер, но отличается от этих типов встроенных систем определенным уровнем интеграции при сохранении требований к наличию некоторого количества внешних компонентов для обеспечения функциональности. Системы на кристалле широко используются как часть реализации одноплатного компьютера (Single Board Computer – SBC), в том числе по стандарту PC/104, а также для более современных форм-факторов, таких как Paspberry Pi и производных от этой модели плат (см. рис. 1.18).

Схема на рис. 1.18 взята с сайта **https://xdevs.com/article/rpi3_oc/**. На ней четко показана компоновка одноплатного компьютера, в данном случае Raspberry Pi 3. Микросхема BCM2837 – это система на кристалле на основе ARM, предоставляющая ядро ЦПУ и основные периферийные модули (в основном описанные в соответствующих отдельных разделах заголовков). Вся оперативная память (ОЗУ) представляет собой внешний модуль, как и периферийные модули Ethernet и Wi-Fi. ПЗУ представлено в виде твердотельной (флеш-) карты, которая также предоставляет возможность хранения данных.

Большинство систем на кристалле создаются на основе микросхем ARM (семейство Cortex-A), хотя не менее часто применяются чипы MIPS. Системы на кристалле широко используются в промышленности.

Другими вариантами являются платы массового потребления, например для смартфонов, для которых не установлен предопределенный форм-фактор, тем не менее производители следуют некоторому шаблону системы на кристалле с учетом необходимости внешнего ПЗУ, ОЗУ и устройства хранения данных, а также разнообразных периферийных устройств. Такой подход противоположен проектным решениям микроконтроллеров, описанных в предыдущем разделе, которые всегда способны самостоятельно выполнять свои функции, за исключением разве что выполнения некоторых требований к внешним ПЗУ.

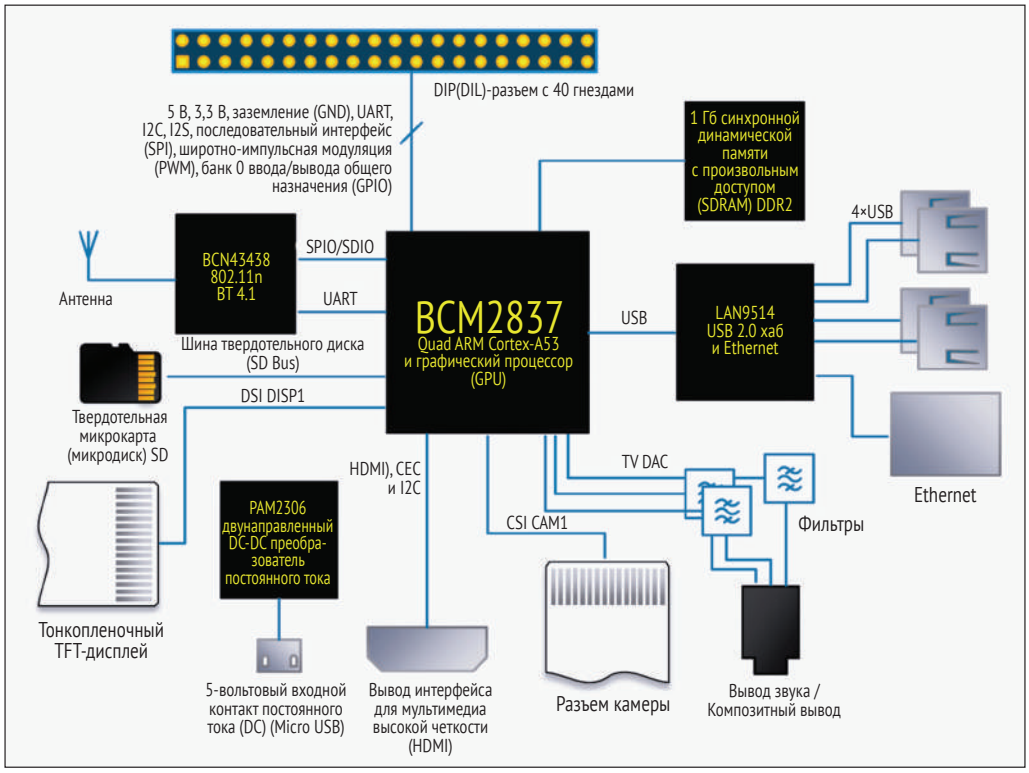

Рис. 1.18

Особенности

По сравнению с микроконтроллерами особенности и трудности разработки для систем на кристалле не столь многочисленны. Некоторые системы на кристалле предоставляют открытый интерфейс, который можно даже использовать для разработки непосредственно на устройстве, то есть с выполнением циклов компиляции на самом устройстве без кросс-компиляции на настольном компьютере и копирования (переноса) бинарных файлов. Кроме того, это позволяет установить и использовать полноценную операционную систему вместо разработки непосредственно на «голой» аппаратуре.

Очевидным недостатком является увеличение сложности при наращивании функциональных возможностей, в результате приводящее к дополнительным затруднениям, таким как обеспечение работы с учетными записями пользователей, установка прав доступа, управление драйверами устройств и т. п.

Резюме

В этой главе были подробно рассмотрены компоненты встроенных систем. Были наглядно продемонстрированы различия между разнообразными типами встроенных систем, а также определены основные принципы выбора правильного микроконтроллера или системы на кристалле для конкретного проекта.

После ознакомления с этой главой читатель более уверенно будет разбираться в спецификациях на микроконтроллеры и системы на кристалле, определять различия между ними и выбирать необходимые модели и варианты для своих проектов.

В следующей главе объясняется, почему язык C++ является наиболее подходящим вариантом выбора для программирования встроенных систем.

Глава 2

C++ как язык программирования встроенных систем

При возникновении необходимости в разработке встроенных компонентов систем с ограниченными ресурсами чаще всего рассматривают только язык программирования C или ассемблер в качестве наиболее оптимальных вариантов выбора. При этом считается, что язык C++ более громоздок, чем C, или что он вносит слишком большой объем дополнительной сложности. В этой главе мы подробно рассмотрим все эти кажущиеся очевидными проблемы и определим достоинства C++ как языка программирования встроенных систем. Темы главы:

- связь C++ с C;
- преимущества языка C++ как мультипарадигмового языка;
- совместимость с существующими решениями на C и ассемблере;
- изменения в стандартах C++11, C++14 и C++17.

Связь C++ с C

Линии родословных языков C и C++ восходят к языку программирования Algol, первая версия которого появилась в 1958 году (ALGOL 58) с обновлениями в 1960 и 1968 годах. Язык программирования Algol представил концепцию императивного программирования – стиля программирования, согласно которому инструкции (команды) языка в явной форме указывали компьютеру, как вносить изменения в данные для получения выходного результата и управления потоком выполнения.

Парадигмой, естественным образом следующей из концепции императивного программирования, является использование процедур. Мы начнем с простого примера, для того чтобы ознакомиться с терминологией. Процедура – это синоним подпрограмм и функций. Все эти понятия определяют группы инструкций (команд), которые являются самодостаточными. Эффект воздействия этих групп инструкций ограничен областью видимости (scope) в том блоке, в котором эти инструкции размещены, таким образом создается иерархия и последовательное представление этих процедур как новых, более абстрактных инструкций. Обя-

зательное использование такого стиля процедурного программирования нашло свое выражение в так называемом структурном программировании в совокупности с применением управляющих структур – циклов и разветвлений (условных выражений).

Со временем были введены стили структурного и модульного программирования как методики усовершенствования процесса разработки, качества и удобства сопровождения программного кода приложений. Язык C является императивным структурированным языком программирования, поскольку в нем используются инструкции (команды), управляющие структуры и функции.

Рассмотрим стандартный пример приветствия миру на языке C:

```
#include <stdio.h>
int main(void)
{
    printf("hello, world");
    return 0;
}
```

Точкой входа в любое приложение, написанное на языке C (и C++), является функция (процедура) `main()`. В первой строке инструкций в этой функции вызывается другая процедура `printf()`, которая содержит собственные инструкции и, возможно, вызывает другие блоки инструкций в виде дополнительных функций.

Таким образом, мы уже воспользовались стилем процедурного программирования, реализовав логический блок `main()` (функцию `main()`), который вызывается, когда это необходимо. Несмотря на то что функция `main()` будет вызываться только однократно, процедурный стиль обнаруживается и в команде `printf()`, которая вызывает соответствующие инструкции в любой точке приложения без необходимости их явного копирования. Применение процедурного программирования значительно упрощает сопровождение итогового программного кода и позволяет создавать библиотеки кода, которые можно использовать в многочисленных приложениях, при этом выполняется сопровождение только одной кодовой базы.

В 1979 году Бьярн Страуструп (Bjarne Stroustrup) начал работу по созданию языка C with Classes (C с классами), для которого он воспользовался существующими парадигмами программирования на C и добавил элементы из других языков, в частности из языка Simula (объектно-ориентированное программирование: императивное и структурированное) и ML (обобщенное (generic) программирование в форме шаблонов). Кроме того, была обеспечена скорость языка BCPL (Basic Combined Programming Language), но без присущих ему ограничений, заставлявших разработчика слишком большое внимание уделять подробностям низкого уровня.

Разработанный в итоге мультипарадигмовый язык в 1983 году был переименован в C++, язык с дополнительными функциональными возможностями, которых не было в C: перегрузка операторов и функций, виртуальные функции, модификация работы со ссылками и адресами, а кроме того, началась разработка отдельного компилятора для языка C++.

Важнейшей целью языка C++ оставалось обеспечение практических решений для реальных задач в окружающем мире. Кроме того, всегда существовало стремление сделать C++ лучше, чем C, в соответствии с его именем. Страуструп опреде-

лил набор правил (перечисленных в работе «Evolving C++ 1991–2006»), управляющих разработкой C++ в наши дни, в том числе следующие:

- развитие C++ обязательно должно управляться реальными задачами;
- каждая функциональная возможность должна иметь разумную очевидную реализацию;
- C++ – это язык программирования, а не полностью завершенная система;
- не следует пытаться заставить людей использовать какой-либо конкретный стиль программирования;
- недопустимы неявные (скрытые) нарушения статической системы типов;
- необходимо обеспечить полноценную поддержку типов, определяемых пользователем, такую же, как для встроенных типов;
- не следует оставлять возможность применения языка более низкого уровня, чем C++ (за исключением ассемблера);
- вы не платите за то, что не используете (принцип нулевых накладных расходов (издержек));
- при возникновении сомнений необходимо предоставить средства ручного управления и контроля.

Различия по сравнению с языком C относятся не только к объектно-ориентированному программированию. Несмотря на существующее с давних времен мнение о том, что C++ представляет собой лишь набор расширений языка C, C++ уже давно стал самостоятельным и независимым языком программирования со строгой системой типов (по сравнению со слишком гибкой и слабо контролируемой системой типов языка C), с более мощными парадигмами программирования и с функциональными возможностями, отсутствующими в C. Таким образом, совместимость C++ с C может выглядеть в большей степени как стечение обстоятельств, с учетом того, что C был успешным языком программирования, появившимся в нужное время, и использовался как основа для разработки нового языка.

Основной проблемой языка Simula в тот период была чрезвычайно низкая скорость выполнения при использовании в общих целях, а BCPL являлся языком слишком низкого уровня. Для того времени язык C++ являлся относительно новым языком программирования, представлявшим правильный баланс между богатством функциональных возможностей и производительностью.

C++ как язык программирования встроенных систем

Приблизительно в 1983 году, когда C++ обрел свою первоначальную форму и получил новое имя, распространенные персональные компьютеры для общего использования, а также для бизнеса обладали характеристиками, приведенными в табл. 2.1.

Теперь сравним эти компьютерные системы с ранее рассмотренным 8-битовым микроконтроллером, например с AVR ATmega 2560 со следующими характеристиками:

- тактовая частота 16 МГц;
- ОЗУ 8 Кб;
- ПЗУ 256 Кб (программы);
- ПЗУ 4 Кб (данные).

Таблица 2.1

Система	ЦПУ	Тактовая частота, МГц	ОЗУ, Кб	ПЗУ, Кб	Устройство внешней памяти, Кб
BBC Micro	6502 (B+ 6512A)	2	16–128	32–128	Максимум 1280 (флоппи-диск с файловой системой ADFS) Максимум 20 Мб (жесткий диск)
MSX	Zilog Z80	3.58	8–128	32	720 (флоппи-диск)
Commodore 64	6510	~1	64	20	1000 (магнитная лента) 170 (флоппи-диск)
Sinclair ZX81	Zilog Z80	3.58	1	8	15 (картридж)
IBM PC	Intel 8080	4.77	16–256	8	360 (флоппи-диск)

Микроконтроллер ATmega 2560 был выпущен в 2005 году, но остается доступным и сегодня наряду с более мощными 8-битовыми микроконтроллерами. Его функциональные возможности в основном остались на уровне компьютерных систем 1980-х годов, но, в отличие от них, этот микроконтроллер не зависит от каких-либо компонентов внешней памяти.

Таковая частота (скорость) микроконтроллера значительно выше благодаря современному усовершенствованному процессу производства кремниевых интегральных микросхем, который также позволяет уменьшить размеры микрочипов, повысить пропускную способность (производительность), следовательно, снизить стоимость. Более важно, что архитектуры 1980-х годов в общем требовали от 2 до 5 циклов для извлечения, декодирования и выполнения инструкции и сохранения результата, в отличие от одного цикла выполнения микропроцессора AVR.

Ограничения на размер (статического) ОЗУ микроконтроллера в настоящее время главным образом связаны с ограничениями по стоимости и энергопотреблению, но их можно с легкостью обойти, воспользовавшись микросхемами внешнего ПЗУ в совокупности с добавочными дешевыми устройствами внешней памяти на основе флеш-памяти и других средств хранения данных.

Системы, подобные Commodore 64 (C64), обычно программировались на языке C в дополнение к встроенному интерпретатору Basic (во встроенном ПЗУ). Широко распространенной системой разработки на языке C для Commodore 64 была среда Power C, предлагаемая компанией Spinnaker (см. рис. 2.1).

Power C был коммерческим продуктом, представляющим полный комплект эффективного программного обеспечения, ориентированный на разработчиков, использующих C. Комплект поставлялся на одном двустороннем флоппи-диске и позволял писать C-код в редакторе, затем компилировать его в приложенном компиляторе и обрабатывать редактором связей (линкером). В комплект также входили заголовочные файлы и библиотеки, обеспечивающие создание выполняемых файлов для целевой системы.

После этого появлялось множество таких укомплектованных компиляторов, ориентированных на разнообразные системы, формирующих развитую, богатую возможностями экосистему для разработки программного обеспечения. Разумеется, в этой среде C++ стал новым явлением. Первое издание книги Страуструпа «The C++ Programming Language» («Язык программирования C++») было опубликовано в 1985 году, но без прилагаемой полноценной реализации описываемого языка.

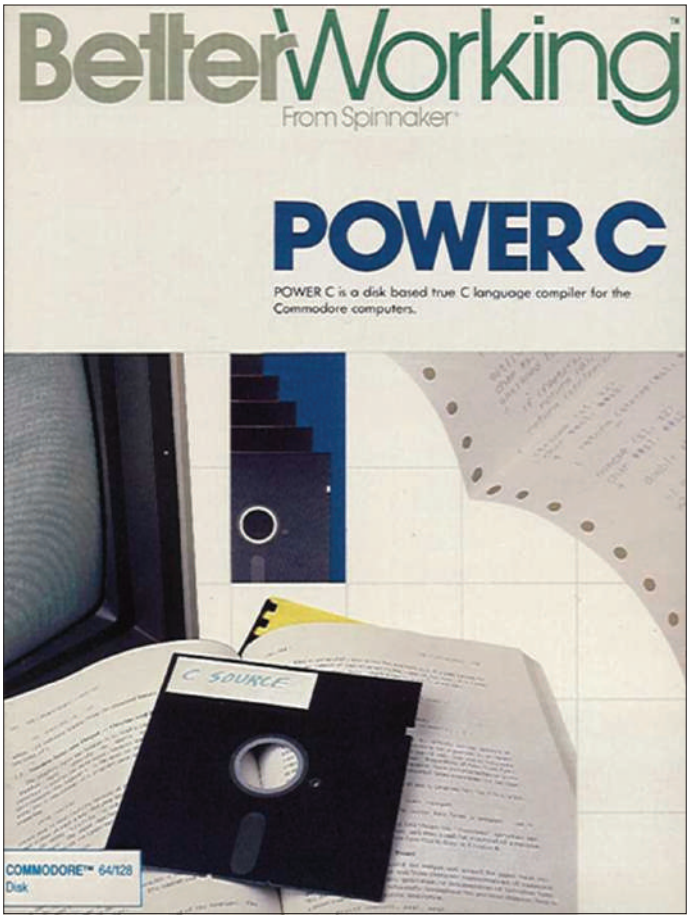

Рис. 2.1

Тем не менее вскоре сформировалась коммерческая поддержка C++, и появились мощные интегрированные среды разработки, такие как Borland C++ 1.0, выпущенная в 1987 году и обновленная до версии 2.0 в 1991 году. Подобные интегрированные среды разработки стали использоваться в основном на компьютерах IBM PC и на многочисленных клонах, где, кроме Basic, фактически отсутствовал язык программирования как основной инструмент разработки.

Язык C++ начал свое существование как неофициальный стандарт в 1985 году и до 1989 года фактически оставался «неофициальным» языком. Выпуск второго издания «The C++ Programming Language» позволил с полным основанием утверждать, что C++ практически достиг уровня функциональных свойств, на котором становится возможной первая стандартизация, выполненная международной рабочей группой ISO, то есть стандарт ISO/IEC 14882:1998, общеизвестный как C++98. Можно считать, что C++ уже выглядел вполне проработанным и адаптированным языком программирования еще до появления микропроцессоров Motorola 68040 в 1990 году и Intel 486DX в 1992 году, которые подняли мощность процессоров выше отметки 20 MIPS.

После того как мы рассмотрели хронологию спецификаций аппаратного оборудования и историю развития языка C++ в сравнении с языком C и прочими языками программирования, а также использование C++ на существующих в то время системах с относительно ограниченными ресурсами, можно с достаточной уверенностью утверждать, что C++ вполне способен работать на такой аппаратуре, в том числе и на современных микроконтроллерах. Но при этом необходимо точно выяснить степень сложности, внесенную в язык C++, поскольку она напрямую воздействует на потребление памяти и на производительность всей компьютерной системы.

Функциональные возможности языка C++

Ранее мы отметили сущность явных изменений данных и состояния системы, которая определяет императивное программирование, в отличие от декларативного программирования, согласно которому вместо обработки данных в цикле аналогичная функциональность может быть объявлена как отображение (сопоставление) оператора на некоторые данные, следовательно, определяется лишь функциональность (сущность выполняемой операции), а не конкретный порядок операций. Но почему в языках программирования возникает необходимость выбора между императивной и декларативной парадигмами?

В действительности одной из основных отличительных особенностей языка C++ является его мультипарадигмовая природа, создающая возможность использования как императивной, так и декларативной парадигмы. После внедрения объектно-ориентированного, обобщенного и функционального программирования в C++ в дополнение к процедурному программированию, присущему языку C, вполне естественно предположить, что все эти нововведения имеют некоторую цену – либо в форме более интенсивного использования ЦПУ, либо в форме увеличения потребления памяти ОЗУ и/или ПЗУ.

Но, как было отмечено выше в этой главе, функциональные возможности языка C++ создавались на основе языка C, поэтому они не должны стать причиной дополнительных издержек (или издержки должны быть весьма небольшими) по сравнению с реализацией аналогичных конструкций в чистом C. Для решения этой задачи и для подтверждения правильности гипотезы о минимальных издержках мы более подробно рассмотрим ряд функциональных возможностей и свойств языка C++, а также методы их реализации с соответствующей оценкой стоимости в бинарном представлении и в плане потребления памяти.

Некоторые примеры, специально предназначенные для демонстрации возможностей C++ как низкоуровневого языка программирования встроенных систем, с разрешения автора взяты из комплекта Rud Merriam's Code Craft, опубликованного на сайте Hackaday: **https://hackaday.io/project/8238-embedding-c**.

Пространства имен

Пространства имен – это способ ввода дополнительных уровней области видимости (scope) в приложение. Это концепция уровня компилятора.

Основная цель использования пространств имен – модуляризация кода, разделение его на логические сегменты в тех случаях, когда классы не являются под-

ходящим решением или если необходимо явно распределить классы по конкретным категориям, соответствующим пространствам имен. При этом исключаются конфликты имен и типов между классами, типами и перечислениями с одинаковыми именами.

Строгая типизация

Информация о типах необходима для проверки легальности доступа и правильности интерпретации данных. Значительным нововведением в C++ по сравнению с C стало введение системы строгой типизации. Это означает, что многие проверки соответствия типов, выполняемые компилятором, стали значительно строже, чем допускалось в языке C, который является языком с нестрогой типизацией.

Это можно увидеть более наглядно, если рассмотреть правильный код C, который при обработке компилятором C++ сгенерирует ошибку:

```
void* pointer;
int* number = pointer;
```

В другой форме это можно записать следующим образом:

```
int* number = malloc(sizeof(int) * 5);
```

Язык C++ запрещает неявные приведения типов, требуя, чтобы приведенные выше примеры были записаны в следующей форме:

```
void* pointer;
int* number = (int*) pointer;
```

Правильная запись второго примера для C++:

```
int* number = (int*) malloc(sizeof(int) * 5);
```

Поскольку тип, к которому выполняется приведение, задан явно, мы можем быть вполне уверены в том, что во время компиляции приведение типов будет выполнено именно так, как предполагалось.

Кроме того, компилятор выдает ошибку при попытке присваивания переменной с квалификатором `const` значения по ссылке, не имеющей такого квалификатора:

```
const int constNumber = 42;
int number = &constNumber; // Ошибка: запрещенная инициализация по ссылке
```

Чтобы избавиться от этой ошибки, необходимо в явной форме выполнить следующее приведение типа:

```
const int constNumber = 42;
int number = const_cast<int&>(constNumber);
```

Выполнение явного приведения типа, подобного показанному в этом примере, вполне допустимо и корректно. Возможно, в дальнейшем возникнут проблемы при использовании такой ссылки для изменения содержимого переменной, значение которой предполагается постоянным. Со временем вы тоже будете писать похожий код, но при условии, что полностью понимаете последствия его применения.

Жесткие требования к явному указанию типов дают существенное преимущество при выполнении статического анализа, которое намного более полезно и эффективно по сравнению с языками с нестрогой типизацией. В свою очередь, это создает преимущество, обеспечивающее безопасность во время выполнения, так как любые преобразования и присваивания, вероятнее всего, будут корректными без неожиданных побочных эффектов.

Таким образом, система типов в большей степени является функциональностью компилятора, нежели кода времени выполнения, за исключением (дополнительной) функции получения информации о типе во время выполнения. Накладные расходы, связанные со строгой системой типов в C++, возникают только во время компиляции, так как более строгие проверки должны выполняться при каждом присваивании значения переменной, при каждой операции и преобразовании.

Преобразование типов

Необходимость преобразования типа возникает в том случае, когда значение присваивается совместимой переменной, имеющей тип, не совпадающий точно с типом значения. Если существует правило такого преобразования, то оно может быть выполнено в неявной форме, иначе необходимо предоставить компилятору явное указание (приведение типа) для обращения к конкретному правилу при возникновении неоднозначности.

В языке C применяется только явное и неявное приведение типов, но C++ расширяет набор этих операций с помощью группы функций на основе шаблонов, позволяя выполнять приведение обычных (встроенных) типов и объектов (классов) несколькими различными способами:

- `dynamic_cast <new_type> (`*`expression`*`);`
- `reinterpret_cast <new_type> (`*`expression`*`);`
- `static_cast <new_type> (`*`expression`*`);`
- `const_cast <new_type> (`*`expression`*`).`

Приведение типа `dynamic_cast` гарантирует, что полученный в результате объект является корректным в соответствии с информацией о типе объекта во время выполнения (RTTI – runtime type information) (см. следующий раздел). Приведение типа `static_cast` выполняется похожим образом, но корректность полученного в результате объекта не проверяется.

Приведение типа `reinterpret_cast` может выполнить преобразование любого типа в любой тип даже для совершенно не связанных друг с другом классов. Необходимость и смысл применения этого преобразования определяет сам разработчик, как, впрочем, и для обычного явного преобразования (приведения типов).

Приведение типа `const_cast` интересно тем, что устанавливает или удаляет статус `const` для значения. Это может быть полезным, если необходима неконстантная версия значения для всего лишь одной функции. Но такое преобразование создает угрозу для безопасности системы типов, поэтому следует применять его с чрезвычайной осторожностью.

Классы

Объектно-ориентированное программирование (ООП) существует со времен Simula, который был известен как весьма медленный язык программирования.

Поэтому Бьярн Страуструп решил для реализации ООП взять за основу быстрый и эффективный язык C.

Язык C++ для реализации объектов использует языковые конструкции в стиле C. Это очень хорошо видно, если рассмотреть фрагмент кода C++ и соответствующий фрагмент кода C.

Исследуя класс C++, можно видеть его обычную структуру:

```
namespace had {
using uint8_t = unsigned char;
const uint8_t bufferSize = 16;
    class RingBuffer {
        uint8_t data[bufferSize];
        uint8_t newest_index;
        uint8_t oldest_index;
        public:
        enum BufferStatus {
            OK, EMPTY, FULL
        };
        RingBuffer();
        BufferStatus bufferWrite(const uint8_t byte);
        enum BufferStatus bufferRead(uint8_t& byte);
    };
}
```

Этот класс также размещен в пространстве имен с переопределением типа `unsigned char`, с глобальным для этого пространства имен определением переменной. Далее следует само определение класса, включающее закрытую (private) и открытую (public) части.

В приведенном выше коде C++ определяется несколько различных областей видимости, начиная с пространства имен и заканчивая классом. Сам класс добавляет области видимости, определяя открытый (public), защищенный (protected) и закрытый (private) уровни доступа.

Тот же код можно написать на обычном языке C следующим образом:

```
typedef unsigned char uint8_t;
enum BufferStatus {BUFFER_OK, BUFFER_EMPTY, BUFFER_FULL};
#define BUFFER_SIZE 16
struct RingBuffer {
   uint8_t data[BUFFER_SIZE];
   uint8_t newest_index;
   uint8_t oldest_index;
};
void initBuffer(struct RingBuffer* buffer);
enum BufferStatus bufferWrite(struct RingBuffer* buffer, uint8_t byte);
enum BufferStatus bufferRead(struct RingBuffer* buffer, uint8_t *byte);
```

Ключевое слово `using` в первом примере является аналогом ключевого слова `typedef` во втором примере, позволяющего выполнить прямое отображение (преобразование). Вместо квалификатора `const` используется директива препроцессора `#define`. Перечисление `enum`, в сущности, одинаково в C и в C++, за исключением того, что компилятор C++ не требует явного назначения имени перечислению `enum`, когда оно используется как тип. То же самое относится и к структурам, когда они применяются для упрощения кода C++.

Класс C++ реализован на языке C как структура `struct`, содержащая переменные класса. Создание экземпляра класса по существу означает, что инициализируется экземпляр этой структуры `struct`. Затем указатель на этот экземпляр `struct` передается при каждом вызове функции, которая является частью класса C++.

Эти простые примеры показывают отсутствие каких-либо издержек во время выполнения при использовании любой функциональной возможности C++ по сравнению с кодом на языке C. Пространство имен, уровни доступа класса (`public`, `private` и `protected`) и прочие особенности используются только компилятором для проверки корректности компилируемого кода.

Важным преимуществом кода C++ при одинаковой производительности является то, что требуется меньший объем исходного кода, а также возможность определения строго регламентированного интерфейса к уровням доступа и наличие метода-деструктора класса, который вызывается при удалении экземпляра класса для автоматического освобождения ресурсов, выделенных при создании этого экземпляра.

Пример использования определенного выше класса:

```
had::RingBuffer r_buffer;
int main() {
    uint8_t tempCharStorage;
    // Заполнение буфера
    for (int i = 0; r_buffer.bufferWrite('A' + i) ==
    had::RingBuffer::OK; i++)    {
        //
    }
    // Чтение буфера
    while (r_buffer.bufferRead(tempCharStorage) == had::RingBuffer::OK)
    {
         //
    }
}
```

Сравним с C-версией того же кода:

```
struct RingBuffer buffer;
int main() {
    initBuffer(&buffer);
    uint8_t tempCharStorage;
    // Заполнение буфера
    uint8_t i = 0;
    for (; bufferWrite(&buffer, 'A' + i) == BUFFER_OK; i++) {
        //
    }
    // Чтение буфера
    while (bufferRead(&buffer, &tempCharStorage) == BUFFER_OK) { //
    }
}
```

Использование класса C++ не слишком отличается от использования аналогичной структуры в стиле C. Отсутствие необходимости передачи вручную созданного экземпляра `struct`, а вместо этого вызов метода класса, возможно, является

самым существенным различием. Созданный экземпляр всегда остается доступным в форме указателя `this`, который указывает именно на этот экземпляр класса.

Хотя в примере на языке C++ используется пространство имен и перечисление, включенное в класс `RingBuffer`, это всего лишь дополнительные (не обязательные) функциональные возможности. Можно использовать глобальные перечисления, размещать их в области видимости пространства имен или создавать несколько уровней пространств имен. Эти решения определяются требованиями к конкретному приложению.

Для оценки стоимости (накладных расходов) использования классов версии примеров из текущего раздела были скомпилированы для упомянутого выше комплекта Code Craft для плат разработки Arduino UNO (ATMega328 MCU) и Arduino Due (AT91SAM3X8E MCU), а результаты сравнения размеров файлов, содержащих скомпилированный бинарный код, приведены в табл. 2.2.

Таблица 2.2

	Плата Uno		**Плата Due**	
	C	**C++**	**C**	**C++**
Данные в глобальной области видимости	614	652	11 184	11 196
Данные в области видимости функции `main()`	664	664	11 200	11 200
Четыре экземпляра	638	676	11 224	11 228

Настройка оптимизации для получения файлов с таким размером кода была выполнена с помощью ключа компилятора `-O2`.

Здесь можно видеть, что после компиляции код C++ практически идентичен коду C, за исключением того случая, когда выполняется инициализация глобального экземпляра класса с учетом автоматического добавления кода выполнения этой инициализации, увеличивающего размер на 38 байт для платы Uno.

Поскольку должен существовать только один экземпляр этого кода, необходимо платить постоянную цену: в первой и в последней строках табл. 2.2 указаны один и четыре экземпляра класса соответственно или их эквивалент, хотя требуются всего лишь дополнительные 38 байт в ПО Uno. Для ПО Duo можно наблюдать нечто подобное, хотя в явной форме это не определено. Вероятнее всего, на это различие влияют некоторые параметры настройки или оптимизации.

Из этого можно сделать вывод, что иногда не следует отдавать компилятору процедуру инициализации класса. Мы должны сделать это самостоятельно, если действительно необходимы эти несколько последних байтов ПЗУ или ОЗУ. Хотя в подавляющем большинстве случаев это не является проблемой.

Наследование

В дополнение к возможности организации кода в форме объектов классы также могут быть представлены как шаблоны для других классов, используя механизм полиморфизма. В C++ можно объединять свойства любого количества классов в новом классе, создавая при этом его собственные свойства и методы.

Это весьма эффективный способ создания типов, определенных пользователем (UDT – user-defined type), особенно в сочетании с оператором перегрузки, позво-

ляющим использовать операторы общего назначения для переопределения операторов сложения, вычитания и т. п. для типа, определенного пользователем.

Наследование в C++ выполняется по следующему образцу:

```
class B : public A {
// Закрытые члены класса
public: // Дополнительные открытые члены класса
};
```

Здесь объявляется класс `B`, который является производным от класса `A`. Это позволяет использовать все открытые методы, определенные в классе A, в любом экземпляре класса B, как если бы они были изначально определены в этом классе.

Все это выглядит достаточно легким для понимания, даже если ситуация кажется слегка запутанной в самом начале описания наследования от более одного базового класса. Но при правильном планировании и проектировании полиморфизм может стать чрезвычайно мощным инструментом.

К сожалению, это не дает ответа на вопрос о величине накладных расходов при использовании полиморфизма в исходном коде. Ранее мы видели, что добавление классов C++ само по себе не увеличивает накладные расходы во время выполнения, хотя при наследовании от одного или нескольких базовых классов можно предположить, что итоговый код становится значительно более сложным.

К счастью, этого не происходит. Если в основном пользоваться несложными классами, то получаемые в итоге производные классы становятся простым объединением базовых структур, определяемых при реализации класса на более низком уровне. Сам по себе процесс наследования в совокупности с проверкой всех соответствующих выполняющихся операций происходит главным образом во время компиляции, что предоставляет разнообразные преимущества разработчику.

Виртуальные базовые классы

Иногда не имеет особого смысла реализация какого-либо метода базового класса, но в то же время необходимо определять реализацию этого метода в каждом производном классе. В таких случаях используются виртуальные методы.

Рассмотрим следующее определение класса:

```
class A {
public:
   virtual bool methodA() = 0;
   virtual bool methodB() = 0;
};
```

При попытке наследования от этого класса обязательно необходимо определить (в производном классе) реализацию двух виртуальных методов. Поскольку оба метода в базовом классе являются виртуальными, то и весь базовый класс обозначается как виртуальный базовый класс. Это особенно полезно, когда необходимо определить интерфейс, который может быть реализован несколькими различными классами, но при этом нужно сохранить преимущество обращения только к одному типу, определенному пользователем.

Во внутреннем представлении виртуальные методы похожи на методы, реализованные обычным способом, но для них используются виртуальные таблицы

vtables. Это структуры данных, содержащие для каждого виртуального метода адрес памяти (указатель), соответствующий реализации этого метода:

```
VirtualClass* → vtable_ptr → vtable[0] → methodA()
```

Можно сравнить характеристики производительности на этом уровне косвенного обращения к методу с кодом на языке C и с классами, использующими прямые вызовы методов. В статье по комплекту Code Craft, посвященной измерениям времени обращения к виртуальным функциям (**https://hackaday.com/2015/11/13/code-craft-embedding-c-timing-virtual-functions/**), описана подобная методика и приведены интересные результаты, показанные в табл. 2.3.

Таблица 2.3

	Uno		**Due**	
Уровень оптимизации →	**Os**	**O2**	**Os**	**O2**
Вызов функции C	10,4	10,2	3,7	3,6
Прямой вызов метода C++	10,4	10,3	3,8	3,8
Вызов виртуального метода C++	11,1	10,9	3,9	3,8
Несколько вызовов C	110,4	106,3	39,4	35,5
Вызовы по указателю на функцию C	105,7	102,9	38,6	34,9
Несколько виртуальных вызовов C++	103,2	100,4	39,5	35,2

Время в табл. 2.3 указано в микросекундах.

Для этого теста использовались те же две платы Arduino, что и для приведенного в предыдущем разделе сравнения размеров скомпилированных файлов для C и классов C++. Здесь при сравнении применялись два различных уровня оптимизации, устанавливаемых флагами компилятора: `-Os` выполняет оптимизацию по размеру итогового бинарного файла (в байтах), `-O2` выполняет оптимизацию по скорости выполнения более агрессивным способом, чем уровень оптимизации `-O1`.

По результатам измерений, приведенным в табл. 2.3, можно с уверенностью сказать, что уровень косвенности, вводимый виртуальными методами, ухудшает производительность заметно, но не критически, добавляя 0.7 микросекунды на плате Arduino Uno ATmega328 и около 0.1 микросекунды на более быстрой плате Arduino Due на основе ARM.

Даже в абсолютном измерении использование виртуальных методов класса почти не ухудшает производительность, то есть нет оснований для отказа от них, если только производительность не является первостепенным фактором, например для более медленных микроконтроллеров. Чем быстрее ЦПУ микроконтроллера, тем меньшим будет отрицательное воздействие виртуальных методов.

Встроенные функции

Ключевое слово `inline` в C++ указывает компилятору, что он должен пытаться каждый раз генерировать в месте этого вызова код, соответствующий функции, имя которой следует за ключевым словом, а не создавать отдельно код этой функции и затем вызывать ее посредством обычного механизма вызова. Такой подход исключает накладные расходы на вызов функции.

Это способ оптимизации во время компиляции, при котором в выходной файл компилятора добавляются только размеры реализаций каждой встроенной функции.

Информация о типах во время выполнения

Главной целью механизма RTTI является обеспечение применения безопасного преобразования типов, как при использовании оператора `dynamic_cast<>`. Механизм информации о типе объекта во время выполнения подразумевает хранение дополнительной информации о каждом полиморфном классе, что связано с некоторыми дополнительными накладными расходами.

Само название механизма говорит о том, что это функциональная особенность времени выполнения, следовательно, ее можно отключить, если эта функция не нужна. Отключение механизма RTTI является часто применяемой практикой на некоторых встроенных платформах, особенно если этот механизм редко применяется на платформах с ограниченными ресурсами, такими как 8-битовые микроконтроллеры.

Обработка исключений

Исключения широко используются на настольных платформах, предоставляя способ генерации исключений в условиях возникновения ошибки, которые можно перехватить и обработать в блоках `try/catch`.

Хотя сама по себе обработка исключений не связана с какими-либо издержками, процесс генерации исключения требует некоторых накладных расходов, в частности существенного процессорного времени и объема ОЗУ для подготовки и обработки исключения. Кроме того, необходима полная уверенность в том, что перехватывается каждое исключение, в противном случае велик риск аварийного завершения приложения без видимых причин.

Если противопоставить обработку исключений и код проверки возвращаемых значений из вызванного метода, то решение о предпочтении того или иного метода следует принимать отдельно в каждом конкретном случае. Кроме того, это может быть вопросом личных предпочтений разработчика. При этом требуются различные стили программирования, которые могут оказаться не подходящими для всех разработчиков.

Шаблоны

Часто встречается мнение, что шаблоны в C++ представляют собой весьма тяжелые и громоздкие конструкции, а их использование ухудшает характеристики программы. При этом совершенно упускается из вида тот факт, что шаблоны очевидно предназначены для применения в качестве весьма лаконичного метода для автоматизации генерации почти идентичного кода из единственного шаблона, что, собственно, и определяет их название.

В действительности это означает, что для любого определенного разработчиком шаблона функции или класса компилятор сгенерирует встроенную реализацию при каждом обращении к такому шаблону.

Такие шаблоны можно видеть в стандартной библиотеке шаблонов STL (standard template library) языка C++, которая обеспечивает поддержку интенсивного использования шаблонов. Например, рассмотрим структуру данных в форме простого отображения (map):

```
std::map<std::string, int> myMap;
```

Здесь компилятор обрабатывает один шаблон `std::map` вместе с параметрами этого шаблона, указанными в угловых скобках, заполняет шаблон реальными характеристиками и записывает встроенную реализацию в этой точке программы.

В действительности мы получаем ту же реализацию, как если бы вручную написали реализацию всей этой структуры данных для двух указанных типов. Поскольку пришлось бы писать вручную отдельные реализации для каждого возможного встроенного типа, а кроме того, и для всех дополнительных типов, определяемых пользователем, применение обобщенного шаблона позволяет сэкономить огромное количество времени, не жертвуя производительностью.

Стандартная библиотека шаблонов

Стандартная библиотека шаблонов C++ STL (standard template library) содержит исчерпывающий и постоянно пополняемый набор функций, классов и т. п., позволяющий решать практически любые задачи без необходимости обращения к внешним библиотекам. Одним из самых распространенных является класс STL `string`, поддерживающий безопасную и удобную обработку строк без обязательного отслеживания завершающих нуль-символов и прочих неудобств.

Большинство встроенных платформ поддерживают библиотеку STL в полном объеме, или, по крайней мере, ее значительную часть, учитывая ограничения по доступному объему ОЗУ, то есть исключаются, например, такие реализации, как полные хеш-таблицы и прочие сложные структуры данных. Многие реализации STL для встроенных систем содержат оптимизации для целевой платформы, позволяющие минимизировать использование ОЗУ и ЦПУ.

Удобство сопровождения

В предыдущих разделах мы кратко рассмотрели некоторые функциональные возможности, предлагаемые языком C++, и степень применимости их на платформах с ограниченными ресурсами. Большим преимуществом C++ является сокращение размера кода при правильном применении шаблонов в сочетании с правильной организацией и модульностью кодовой базы с помощью классов, пространств имен и т. п.

Попытки улучшить модульность кода с корректно организованными интерфейсами между модулями увеличивают возможность повторного использования кода в нескольких проектах. Кроме того, упрощается сопровождение кода, поскольку предназначение любого конкретного фрагмента кода становится более понятным, а также четко определяются цели модульного и интегрированного (комплексного) тестирования.

Резюме

В этой главе мы получили ответ на вопрос, почему C++ следует использовать для разработки ПО для встроенных систем. Гибкость разработки на языке C++ позволяет оптимизировать его для платформ с ограниченными ресурсами, кроме того, C++ предлагает множество функциональных возможностей, весьма важных для управления процессом проектирования и организации разработки.

После изучения этой главы предполагается, что читатель может описать основные функциональные характеристики C++ и привести конкретные примеры их применения. При написании кода C++ читатель будет ясно понимать стоимость (издержки) каждой конкретной функциональной возможности и сможет обосновать причину выбора одной реализации фрагмента кода из нескольких других вариантов с учетом доступного пространства и ограничений ОЗУ.

В следующей главе будет рассматриваться процесс разработки для встроенной системы ОС Linux и аналогичных систем на основе одноплатных компьютеров и подобных платформ.

Глава 3

Разработка для встроенной ОС Linux и подобных систем

Небольшие системы на одноплатных компьютерах (SoC) в наше время можно встретить везде – от смартфонов, консолей видеоигр и комплектов smart-телевидения до информационно-развлекательных систем в автомобилях и самолетах. Особенно часто такие системы используются в устройствах массового потребления.

Кроме применения в потребительских устройствах, такие системы также можно обнаружить в виде систем управления в промышленности, строительстве и разработках различного уровня, где они наблюдают за оборудованием, обеспечивают ответную реакцию на входные данные и выполняют запланированные задачи для комплексных сетей сенсорных датчиков и силовых приводов рабочих органов (манипуляторов). В отличие от микроконтроллеров, одноплатные компьютеры не ограничены в ресурсах, поэтому обычно работают под управлением полноценной операционной системы (ОС), например ОС на основе Linux, VxWorks или QNX.

В этой главе рассматриваются следующие темы:

- разработка драйверов для встроенных систем под управлением ОС;
- способы интеграции периферийных устройств;
- обеспечение и реализация требований к производительности в реальном времени;
- определение ограничений ресурсов и методы их обхода.

Встроенные операционные системы

Операционная система (ОС) обычно используется для встроенной системы, когда вы пишете приложение непосредственно для системной аппаратуры, которую планируется эксплуатировать необычным способом. Для такого приложения ОС предоставляет набор прикладных программных интерфейсов – API (application programming interface), которые позволяют абстрагироваться от конкретной аппаратуры и функциональности, реализованной с помощью этой аппаратуры, как, например, функции обмена данными в сети или выходной видеосигнал.

Здесь должен быть найден компромисс между удобством разработки и размером и сложностью кода.

Когда «голое железо» идеально реализует только одну необходимую функцию, операционная система предлагает планировщик задач с функциональностью, которая, возможно, вообще не нужна выполняемому приложению. Поэтому важно знать, когда следует использовать ОС вместо разработки непосредственно для аппаратного устройства, и хорошо понимать сложности и особенности в обоих случаях.

Разумным обоснованием использования ОС является реальная необходимость одновременного выполнения различных задач (многозадачность или многопоточность). Реализация собственного планировщика задач с нуля в подавляющем большинстве случаев не оправдывает трудозатраты. Необходимость запуска неопределенного количества приложений и возможности добавлять и удалять их в любое время также может быть удовлетворена значительно проще при использовании ОС.

Наконец, такие функциональные возможности, как усовершенствованный вывод графики, ускорение графики (например, OpenGL), сенсорные экраны и развитые сетевые функции (например, SSH и шифрование), могут существенно упростить реализацию, если вы получаете доступ к ОС, к предоставляемым ею готовым к работе драйверам и разнообразным API.

Наиболее широко распространенные встроенные операционные системы перечислены в табл. 3.1.

Таблица 3.1

Наименование	Производитель	Лицензия	Платформы	Дополнительные характеристики
Raspbian	Разработка сообщества	В основном GPL и аналогичные	ARM (Raspberry Pi)	ОС на основе Debian Linux
Armbian	Разработка сообщества	GPLv2	ARM (различные платы)	ОС на основе Debian Linux
Android	Google	GPLv2, Apache	ARM, x86, x86_64	ОС на основе Linux
VxWorks	Wind River (Intel)	Проприетарная	ARM, x86, MIPS, PowerPC, SH-4	RTOS, монолитное ядро
QNX	BlackBerry	Проприетарная	ARMv7, ARMv8, x86	RTOS, микроядро
Windows IoT	Microsoft	Проприетарная	ARM, x86	Ранее известна как Windows Embedded
NetBSD	NetBSD Foundation	Упрощенная BSD из двух пунктов	ARM, 68k, MIPS, PowerPC, SPARC, RISC-V, x86 и т. д.	ОС на основе BSD с максимальной степенью переносимости

Все эти ОС объединяет то, что они предоставляют основную необходимую функциональность, как, например, управление памятью и задачами, в то же время обеспечивая доступ к аппаратуре и внутренним функциям ОС с помощью прикладных программных интерфейсов API.

В этой главе основное внимание уделяется системам на кристалле (SoC) и системам на основе одноплатных компьютеров (SBC), работающих под управлением перечисленных выше операционных систем. Каждая из этих ОС адаптирована

для использования на системе, обладающей по меньшей мере несколькими мегабайтами ОЗУ и оснащенной хранилищем данных с размером порядка нескольких мегабайтов или даже гигабайтов.

Если целевая система на кристалле или одноплатный компьютер пока еще не адаптирован для существующих дистрибутивов Linux или необходима чрезвычайно высокая специализация системы, то можно воспользоваться инструментальными средствами из проекта Yocto Project (**http://www.yoctoproject.org/**).

Встроенные системы на основе ОС Linux являются преобладающими, и самый известный пример этого – ОС Android. Android в основном используется на смартфонах, планшетах и аналогичных устройствах, в которых главным компонентом является взаимодействие с пользователем в графическом режиме, основанное на инфраструктуре приложений Android и на соответствующих API. Из-за такого уровня специализации Android не подходит для других вариантов использования.

Система Raspbian основана на широко распространенном дистрибутиве Debian Linux, адаптированном в основном только для одноплатных компьютеров серии Raspberry Pi. Armbian – похожая система, но применимая для гораздо более широкого диапазона одноплатных компьютеров. Raspbian и Armbian представляют собой результаты разработки неформального сообщества. Это похоже на проект Debian, который также может использоваться непосредственно для встроенных систем. Основным преимуществом Raspbian, Armbian и других аналогичных проектов является то, что они предоставляют готовые к работе образы для целевых одноплатных компьютеров.

Как и Linux-подобные ОС, NetBSD обладает всеми преимуществами ПО с открытым исходным кодом, то есть предоставляет полный доступ ко всем исходным кодам, следовательно, имеется возможность высокой степени специализации и адаптации любого аспекта ОС, включая поддержку узкоспециализированной аппаратуры. Еще одним большим преимуществом NetBSD и ОС на основе BSD является то, что все версии и варианты создаются на единой кодовой базе, управляемой одной группой разработчиков. Зачастую это упрощает разработку и сопровождение проекта встроенной системы.

Лицензия BSD (с тремя и двумя пунктами) предоставляет значительную выгоду для коммерческих проектов, поскольку требует только предоставления ссылки на авторство вместо требования от производителя предоставления полного исходного кода ОС по запросу. Это может стать весьма важным условием, если кто-то вносит некоторые изменения в исходный код, добавляя модули кода, которые желательно сохранить недоступными для посторонних (закрыть доступ к части исходного кода).

Например, последние модели игровых консолей PlayStation используют модифицированную версию FreeBSD, позволяющую компании Sony обеспечить высокую степень оптимизации ОС для своей аппаратуры и использовать ее как игровую консольную систему без обязательства открытия собственного специализированного кода вместе с остальным открытым исходным кодом ОС.

Также существуют и проприетарные (полностью закрытые) варианты, например предлагаемые компаниями BlackBerry (QNX) и Microsoft (Windows IoT, ранее известная как Windows Embedded, а до этого Windows CE). Эти варианты требуют лицензионной платы за каждое устройство, а для выполнения любой специализации необходимо обращаться в компанию-производитель.

Операционные системы реального времени

Главное требование к операционной системе реального времени (real-time OS – RTOS) – гарантия того, что задачи будут выполнены и завершены в течение определенного заданного интервала времени. Это позволяет использовать их для приложений реального времени, где непостоянство (так называемый джиттер) между временами выполнения комплекта экземпляров одной и той же задачи абсолютно неприемлемо.

Отсюда можно вывести основное различие между жестким и мягким режимами реального времени для ОС: при малом джиттере (непостоянстве) ОС поддерживает жесткий режим реального времени, то есть способна гарантировать постоянное выполнение конкретной задачи практически с одинаковой задержкой. При более высоком джиттере ОС способна в большинстве случаев, но не всегда выполнять задачу с одинаковой задержкой.

Внутри этих двух категорий можно выполнить еще одно разделение между планировщиками, управляемыми событиями, и планировщиками с разделением времени. Планировщик, управляемый событиями, переключает задачи на основе их приоритетов (планирования приоритетов), тогда как планировщик с разделением времени использует таймер для регулярного переключения задач. Какое проектное решение лучше, напрямую зависит от цели использования проектируемой системы.

Здесь главное то, что планировщики с разделением времени имеют преимущество перед планировщиками, управляемыми событиями, состоящее в том, что они выделяют гораздо больше процессорного времени задачам с низкими приоритетами, что может стать причиной того, что многозадачные системы будут выглядеть работающими более уравновешенно и плавно.

Вообще говоря, рекомендуется использовать ОС реального времени только в том случае, если проектные требования определяют неизбежную необходимость обработки входных данных за строго определенный интервал времени (временно́е окно). Для программ из группы робототехнических и промышленных приложений, возможно, наиболее важно выполнение заданной операции в точно заданный одинаковый интервал времени каждый раз. Любой сбой приведет к нарушению технологического процесса производства и к потере качества продукции.

В примере проекта, рассматриваемого ниже в этой главе, используется не ОС реального времени, а обычная ОС на основе Linux, поскольку не определяются строгие требования ко времени выполнения задач. Использование ОС реального времени в подобных случаях приводит к ненужной перегруженности системы и, вероятно, к увеличению сложности и накладных расходов.

Применение ОС реального времени разумно рассматривать в случае, когда необходимо максимальное приближение к сущности программирования реального времени непосредственно для аппаратного устройства («голого железа») без потери всех удобств и преимуществ использования обычной полноценной операционной системы.

Специализированные периферийные устройства и драйверы

Периферийное устройство определяется как дополнительное (вспомогательное) устройство, которое добавляет функции ввода/вывода или другую функциональность в компьютерную систему. Устройство может быть любым – от контроллера I2C, SPI или SD-карты до графического или аудиоустройства. Большинство периферийных устройств являются частью физической системы на кристалле (или одноплатного компьютера), другие добавляются с подключением через интерфейсы, которые система на кристалле предоставляет для связи с внешней средой. Примерами внешних периферийных устройств могут быть модули ОЗУ (через контроллер ОЗУ) и часы реального времени (real-time clock – RTC).

Проблема, которая часто возникает при использовании дешевых одноплатных компьютеров, таких как Raspberry Pi, Orange Pi, и их многочисленных клонов, заключается в отсутствии (в большинстве случаев) на этих платах часов реального времени (RTC), поэтому при выключении такой компьютер не отслеживает реальное время. Причина в том, что обычно предполагается, что такие платы (почти) постоянно будут подключены к интернету, поэтому ОС может воспользоваться онлайновой службой времени (Network Time Protocol – NTP) для синхронизации системного времени, что позволяет сэкономить место на плате.

Тем не менее возможны ситуации использования одноплатного компьютера при отсутствии доступа к интернету, или в условиях неприемлемой задержки синхронизации времени в режиме онлайн, или по многим другим причинам. В этом случае, возможно, потребуется добавление периферийного устройства RTC на плату и соответствующее конфигурирование ОС, чтобы появилась возможность использования добавленного устройства.

Добавление часов реального времени

Можно недорого приобрести часы реального времени (RTC) как готовый к использованию модуль, чаще всего выполненный на основе микросхемы DS1307. Для этого модуля требуется напряжение 5 В, а к одноплатному компьютеру (или к микроконтроллеру) подключается он через шину I2C.

На рис. 3.1 показан небольшой модуль часов реального времени на основе микросхемы DS1307. Здесь можно видеть саму микросхему часов реального времени, кристалл и микроконтроллер, который используется для обмена данными с хост-системой вне зависимости от того, является ли хост-система системой на кристалле или платой на основе микроконтроллера. Требуется лишь обеспечить необходимое напряжение (и силу тока) для работы модуля часов реального времени, а также физический доступ к шине I2C.

После подключения модуля часов реального времени к одноплатному компьютеру следующая задача – обеспечить возможность его использования операционной системой. Для этого необходимо убедиться в том, что модуль ядра I2C загружен и можно использовать устройства I2C.

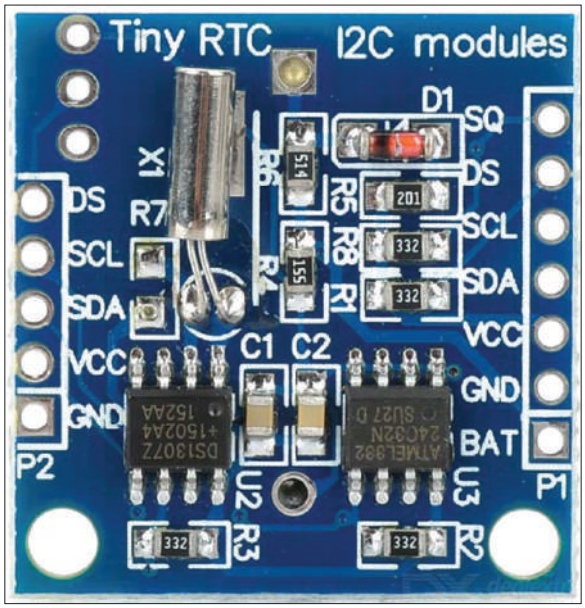

Рис. 3.1

В дистрибутивы Linux для одноплатных компьютеров, такие как Raspbian и Armbian, обычно включены драйверы для ряда модулей часов реального времени. Это позволяет относительно быстро настроить модуль RTC и интегрировать его в ОС. Для модуля, рассматриваемого в этом разделе, требуются модули ядра I2C и микросхемы DS1307. В Raspbian OS для первого поколения Raspberry Pi эти модули называются `i2c-dev`, `2cbcm2708` и `rtc-ds1307`.

Сначала необходимо разрешить использование этих модулей, чтобы они загружались при запуске системы. В ОС Raspbian Linux можно отредактировать файл */etc/modules* или воспользоваться другими инструментами конфигурации, доступными на этой платформе. После перезагрузки устройство часов реального времени должно быть обнаружено на шине I2C с помощью инструментального средства I2C scanner.

При работающих часах реального времени можно удалить пакет fake-hwclock, предоставляемый дистрибутивом Raspbian. Это простой модуль, имитирующий работу часов реального времени, но явно сохраняющий текущее время в файле перед завершением работы системы, так что при следующей загрузке метки времени и даты файловой системы будут согласованными и непротиворечивыми благодаря восстановлению сохраненной даты и времени. Таким образом, исключается ситуация, когда все новые создаваемые файлы оказываются более старыми, чем ранее существующие файлы.

Вместо модуля имитации будет использоваться утилита hwclock, которая работает с любыми часами реального времени для синхронизации файловой системы. При этом требуется изменить способ запуска ОС, указав локацию модуля RTC, передаваемую как загрузочный параметр в следующей форме:

```
rtc.i2c=ds1307,1,0x68
```

Это позволяет инициализировать модуль RTC (`/dev/rtc0`) на шине I2C с адресом 0x68.

Специализированные драйверы

Точный формат и методы включения драйверов (модулей ядра) в ядро ОС различны для каждой операционной системы, поэтому вряд ли возможно привести здесь полное их описание. Но мы все же рассмотрим реализацию для ОС Linux драйвера модуля часов реального времени, который был описан в предыдущем подразделе.

Кроме того, несколько позже в этой главе мы рассмотрим, как используются периферийные устройства I2C из пространства пользователя на примере организации мониторинга клубного зала. Применение драйвера (библиотеки), размещенного в пространстве пользователя, часто является неплохой альтернативой реализации драйвера как модуля ядра.

Функциональность модуля часов реального времени включена в ядро Linux, а соответствующий код можно найти в репозитории GitHub **https://github.com/torvalds/linux/tree/master/drivers/rtc**.

Файл *rtc-ds1307.c* содержит две функции, которые необходимы для чтения данных и установки часов реального времени, соответственно `ds1307_get_time()` и `ds1307_set_time()`. Основная функциональность этих функций очень похожа на ту, которую мы будем использовать в примере мониторинга клубного зала несколько позже в этой главе, где поддержка устройства I2C будет просто включена непосредственно в приложение.

Главным преимуществом обмена информацией с I2C, SPI и другими подобными периферийными устройствами из пространства пользователя является то, что мы не ограничены скомпилированной средой, поддерживаемой ядром ОС. Возьмем в качестве примера ядро Linux, которое написано на языке C с небольшими фрагментами на ассемблере. API ядра организованы в C-стиле, следовательно, нам неизбежно придется использовать стиль кодирования на C при написании собственных модулей ядра.

При этом мы лишимся большинства преимуществ, которые имелись бы при попытке написать эти модули на C++. Перемещая код наших модулей в пространство пользователя и используя его как часть приложения или как совместно используемую библиотеку (shared library), мы устраняем эти ограничения и можем свободно применять все концепции и функциональные возможности C++.

Для полноты описания ниже приведен основной шаблон модуля ядра Linux:

```
#include <linux/module.h>       // Необходимо для всех модулей
#include <linux/kernel.h>       // Необходимо для модуля KERN_INFO

int init_module() {
        printk(KERN_INFO "Hello world.n");
        return 0;
}

void cleanup_module() {
        printk(KERN_INFO "Goodbye world.n");
}
```

Это тривиальный пример Hello World, написанный в стиле C++.

Заключительное замечание при рассмотрении версий драйвера в пространстве пользователя и как модуля ядра касается переключения контекста. С точки зрения эффективности (производительности) модули ядра работают быстрее и с меньшими задержками, поскольку ЦПУ не должен постоянно переключаться из контекста пространства пользователя в контекст пространства ядра и обратно, чтобы обеспечить обмен данными с устройством и передавать сообщения от устройства в код, взаимодействующий с ним.

Для устройств с высокой пропускной способностью (таких как устройства хранения данных и устройства захвата данных), возможно, потребуется определить различие между бесперебойно (равномерно) работающей системой и системой, в которой регулярно возникают задержки и затруднения при выполнении задач.

Тем не менее при рассмотрении примера мониторинга клубного зала в этой главе и использовании в нем произвольно выбранного устройства I2C должно быть понятно, что применение модуля ядра привело бы к существенным издержкам без каких-либо заметных преимуществ.

Ограничение ресурсов

Несмотря на то что одноплатные компьютеры и системы на кристаллах становятся все более мощными, их нельзя сравнивать напрямую с современными настольными системами или серверами. Им присущи существенные ограничения по объему ОЗУ, размерам внешней памяти, а также по отсутствию (или недостатку) возможностей расширения.

При широком диапазоне вариантов объемов (постоянно устанавливаемого) ОЗУ приходится предварительно определять размер памяти, необходимый для приложений, которые планируется выполнять на этой системе, особенно при ЦПУ с относительно низкой производительностью.

Обычно на одноплатных компьютерах отсутствуют устройства хранения данных (то есть устройства внешней памяти) с высокой износоустойчивостью или объем такой внешней памяти весьма незначителен (имеется в виду возможность многократной перезаписи без ограничения количества циклов). Поэтому на одноплатных компьютерах нет пространства для свопинга, и все необходимое хранится в доступном ОЗУ. Без экстренного обращения к операции свопинга любые утечки памяти, а также интенсивное использование оперативной памяти быстро приводит к нарушению функциональности системы или к постоянным ее перезагрузкам.

Даже притом, что производительность ЦПУ на одноплатных компьютерах значительно возросла за последние годы и для массово доступных моделей, в общем случае остается в силе рекомендация по использованию кросс-компилятора для генерации бинарного кода для одноплатного компьютера на высокопроизводительной настольной системе или на мощном сервере.

Более подробно проблемы и решения разработки для одноплатных компьютеров будут рассматриваться в главе 6 «Тестирование приложений, предназначенных для конкретных ОС» и в приложении А «Эффективные практические методики».

Пример: мониторинг клубного зала

В этом разделе рассматривается практическая реализация решения на основе одноплатного компьютера, который обеспечивает следующую функциональность для клубного зала:

- мониторинг (отслеживание) состояния замка входной двери клуба;
- мониторинг (отслеживание) состояния коммутатора (переключателя) в помещении клуба;
- передача сообщений об изменении состояния через упрощенный сетевой протокол MQTT;
- поддержка REST API для текущего состояния помещения клуба;
- управление состоянием освещения;
- управление электропитанием клубного зала.

Здесь основной вариант использования – работа в клубном зале, для которого необходимо обеспечить мониторинг состояния замка входной двери, а также коммутатора (переключателя) в самом клубном зале для регулирования подключения электропитания к гнездам (розеткам) в помещении клуба. Переключение коммутатора состояния в положение on (Вкл) должно обеспечивать подачу электропитания ко всем розеткам. Также необходимо организовать передачу сообщений по протоколу MQTT, чтобы все прочие устройства в клубном зале или в других помещениях клуба могли обновить свое состояние.

MQTT (message queuing telemetry transport) – это упрощенный бинарный сетевой протокол, работающий поверх TCP/IP, предназначенный для обмена сообщениями между устройствами по принципу издатель/подписчик (publisher/subscriber). Этот протокол хорошо подходит для приложений с ограниченными ресурсами, такими как сети сенсорных датчиков. Каждый MQTT-клиент обменивается данными с центральным сервером – MQTT-брокером.

Аппаратные устройства

Блок-схема системы clubstatus показана на рис. 3.2.

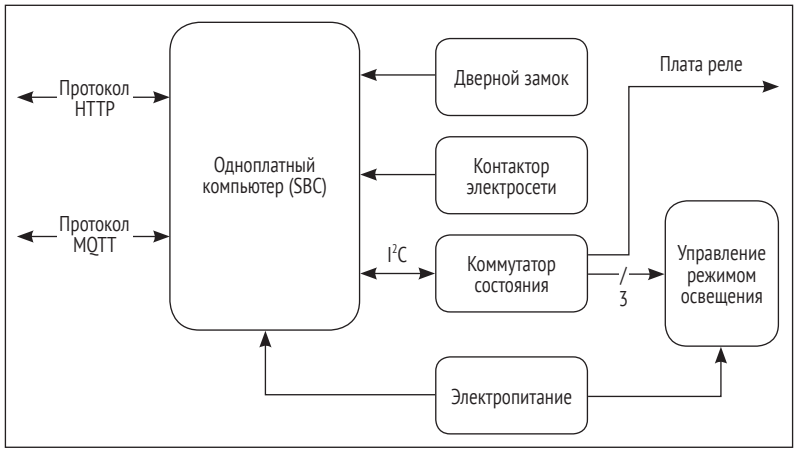

Рис. 3.2

Для платформы одноплатного компьютера используется модель Raspberry Pi, Raspberry Pi B+ или более новые модели серии B, например Raspberry Pi 3 Model B (рис. 3.3).

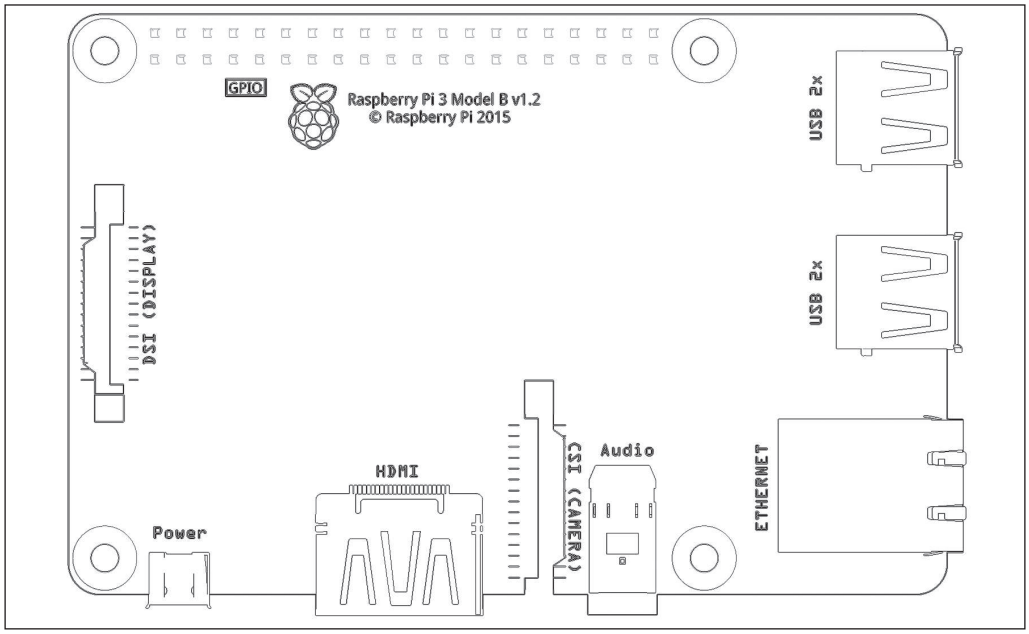

Рис. 3.3

Основными функциональными возможностями, на которые мы обращаем особое внимание при проектировании этой SBC-системы, являются разъем Ethernet-соединения и, разумеется, совместимый с Raspberry Pi многоштырьковый разъем GPIO (general-purpose input/output), интерфейса ввода/вывода общего назначения.

Для этой платы используется стандартная установка операционной системы Raspbian OS на твердотельном микродиске µSD card. Какая-либо специализированная конфигурация не требуется. Главная причина выбора модели B+ или аналогичной модели заключается в наличии на ней стандартных монтажных гнезд по общепринятому шаблону.

Реле

Для управления состоянием освещения и непостоянной подачи электропитания к розеткам в зале используется несколько реле, описанных в табл. 3.2.

Таблица 3.2

Реле	Функция
0	Состояние электропитания для непостоянно подключаемых розеток
1	Состояние зеленого света
2	Состояние желтого света
3	Состояние красного света

Здесь проектное решение состоит в том, что реле состояния электропитания соединяется с коммутатором, управляющим электросетью, подающей электропитание на розетки, которые отключаются, когда общее состояние помещения клуба «отключено» (off). Состояние освещения показывает текущее состояние помещения клуба. В следующем разделе описаны подробности реализации этой концепции.

Для упрощения проектного решения воспользуемся готовой к использованию платой реле, содержащей четыре реле, управляемых микросхемой NXP PCAL9535A через порт ввода/вывода (расширение интерфейса GPIO), соединенный с шиной I2C на плате Raspberry Pi (рис. 3.4).

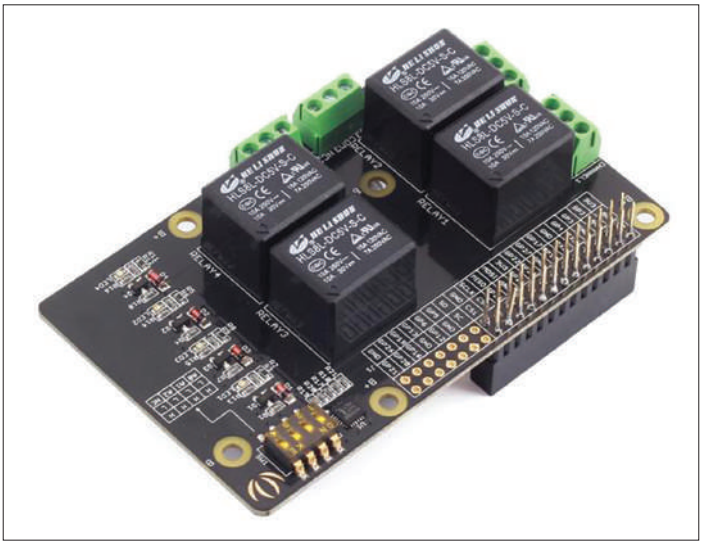

Рис. 3.4

В данном случае используется плата реле Seeed Studio Raspberry Pi Relay Board v1.0 (**http://wiki.seeedstudio.com/Raspberry_Pi_Relay_Board_v1.0/**). На ней размещены четыре необходимых для рассматриваемого примера реле, позволяющих переключать режимы освещения и регулировать режим электропитания до 30 В постоянного тока или 250 В переменного тока. Это позволяет подключать практически любой тип осветительных приборов и других реле или аналогичных устройств.

Соединение с одноплатным компьютером обеспечивается размещением платы реле на плате компьютера с подключением через многоштырьковый разъем GPIO, что позволяет добавлять другие платы поверх платы реле. Кроме того, это дает возможность добавления функциональности устранения дребезга контактов в систему, как показано на принципиальной электрической схеме.

Устранение дребезга контактов

Основным требованием к такому устройству является наличие функции устранения дребезга контактов при переключении сигналов, а также обеспечение электропитанием платы Raspberry Pi. Теоретическое обоснование функции устра-

нения дребезга механических контактов заключается в том, что сигнал, генерируемый такими переключателями, не является чистым, то есть они не способны мгновенно переходить из разомкнутого состояния в замкнутое. Контакт переключателя на короткое время замыкается (обеспечивая соединение), прежде чем упругость металлических пластин контакта заставляет его снова разомкнуться, затем очень быстро колебаться между этими двумя состояниями до тех пор, пока контакт в итоге не будет зафиксирован в своей конечной позиции. Этот процесс можно наблюдать на схеме (рис. 3.5), полученной с осциллографа, подключенного к простому переключателю.

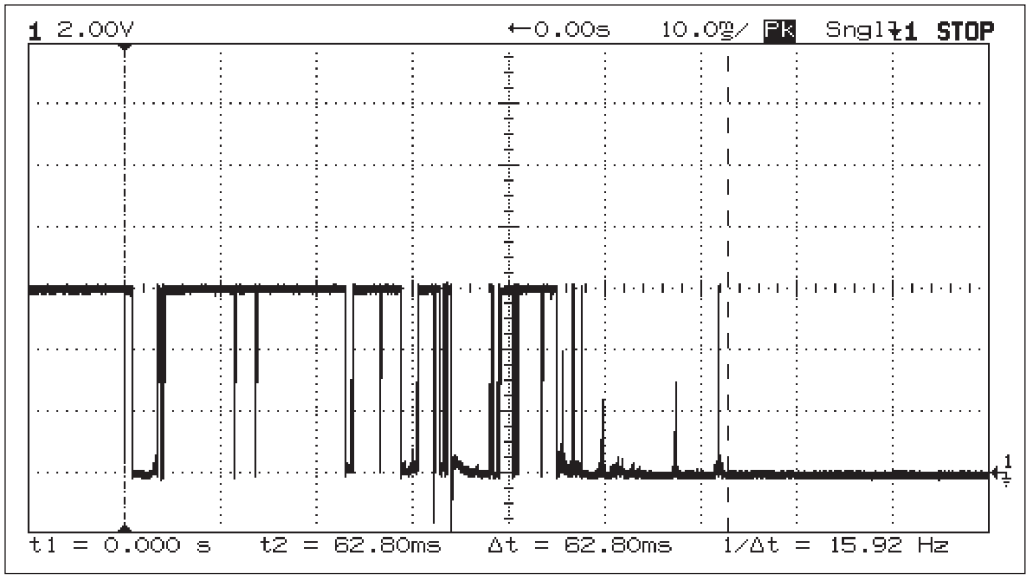

Рис. 3.5

В результате этого явления сигнал, поступающий на контакты интерфейса GPIO одноплатного компьютера, будет многократно изменяться в течение нескольких миллисекунд (или больше, что ухудшает ситуацию). Таким образом, выполнение действий любого типа, зависящих от этих изменений входного сигнала на переключателе, приведет к большим проблемам, поскольку невозможно правильно различить требуемое изменение состояния переключателя и чрезвычайно нежелательный быстрый дребезг контактов в течение наблюдаемого интервала времени.

Существует возможность устранения дребезга контактов аппаратным или программным способом. Программное решение подразумевает запуск таймера в момент первого изменения состояния переключателя. В основу заложено предположение о том, что после истечения определенного интервала времени (в миллисекундах) переключатель находится в стабильном состоянии, которое можно безопасно определить (считать). Недостатком этого метода является создание на систему дополнительной нагрузки, связанной с работой одного или нескольких таймеров, и/или задержка (пауза) в выполнении программы. Кроме того, обра-

ботка прерываний на входе переключателя требует запрещения прерываний во время работы таймера, что усложняет код приложения.

Аппаратное устранение дребезга контактов можно реализовать с использованием дискретных компонентов или с применением SR-триггеров (Set/Reset), состоящих из двух логических вентилей И-НЕ. Для рассматриваемого здесь приложения воспользуемся следующей электрической схемой, показанной на рис. 3.6, которая успешно работает с наиболее широко используемым типом однополюсных переключателей SPST (single-pole, single-throw).

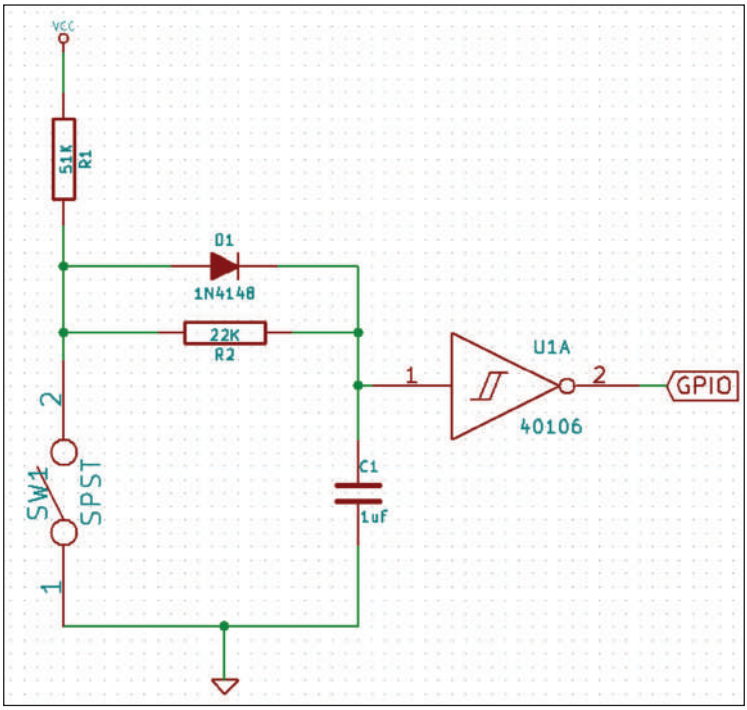

Рис. 3.6

Принцип работы схемы, показанной на рис. 3.6, состоит в том, что при разомкнутом контакте переключателя конденсатор заряжается через сопротивление R1 (и диод D1), усиливая уровень входного сигнала в контуре триггера Шмитта с обратной связью U1. В результате на контактном штырьке GPIO, соединенном с одноплатным компьютером, выходной сигнал триггера U1 считывается как низкий. Когда контакт переключателя замыкается, конденсатор разряжается в землю через сопротивление R2.

Процессы зарядки и разрядки конденсатора занимают некоторое время, то есть добавляется задержка, перед тем как на входе триггера U1 будет зарегистрировано изменение состояния. Скорости процессов зарядки и разрядки конденсатора определяются значениями сопротивлений R1 и R2 с помощью следующих формул:

- зарядка: $V(t) = V_S(1 - e^{-(t/RC)})$;
- разрядка: $V(t) = V_S e^{-(t/RC)}$.

Здесь $V(t)$ – напряжение в момент времени t (в секундах). V_S – исходное напряжение источника, а t – время в секундах, прошедшее после начала подачи напряжения от источника. R – сопротивление схемы в омах, C – емкость конденсатора в фарадах. Значение e – математическая постоянная, приблизительно равная 2,71828, известная как число Эйлера.

Для описания процессов зарядки и разрядки конденсаторов используется константа времени RC, обозначаемая греческой буквой τ (тау):

$$\tau = RC.$$

Эта формула определяет время, требуемое для заряда конденсатора до уровня 63,2 % (1τ), затем до уровня 86 % (2τ). Разрядка конденсатора на 1τ от состояния полной заряженности снижает заряд до 37 %, а разрядка на 2τ – до 13,5 %. Здесь важно отметить, что конденсатор никогда полностью не заряжается и не разряжается. Процесс зарядки или разрядки просто замедляется до такой степени, что становится почти незаметным.

С учетом значений, которые используются в схеме устранения дребезга контактов для рассматриваемого здесь примера, получим следующее значение константы времени для процесса зарядки:

$$0{,}051 = 51\,000 \times 0{,}000001.$$

Значение константы времени для процесса разрядки:

$$0{,}022 = 22\,000 \times 0{,}000001.$$

Это означает 51 и 22 миллисекунды соответственно.

Любой триггер Шмитта обладает так называемой петлей гистерезиса, означающей наличие двойного порогового значения. Это неизбежно добавляет мертвую зону в область выходного ответного сигнала в верхней и нижней частях, но итоговый выходной сигнал не изменится (рис. 3.7).

Гистерезис в триггере Шмитта обычно используется для удаления шума из входящего сигнала посредством установки в явной форме уровней триггера. Даже с учетом того, что уже используемая нами RC-схема должна отфильтровывать практически все шумы, добавление триггера Шмитта немного увеличивает уверенность в отсутствии любых отрицательных воздействий.

При условии доступности можно также воспользоваться функциональностью гистерезиса контактных штырьков интерфейса GPIO на одноплатном компьютере. Для рассматриваемого здесь проекта и выбранной схемы устранения дребезга контактов также необходимо инвертировать функциональное свойство микросхемы, чтобы получить ожидаемую ответную реакцию высокого/низкого уровня для подключенного переключателя вместо инвертирования смысла программного кода.

Плата расширения (HAT) для устранения дребезга контактов

Информация, предоставленная в предыдущем разделе, и описанная там же схема устранения дребезга контактов позволяют скомпоновать физический прототип такой платы, показанный на рис. 3.8.

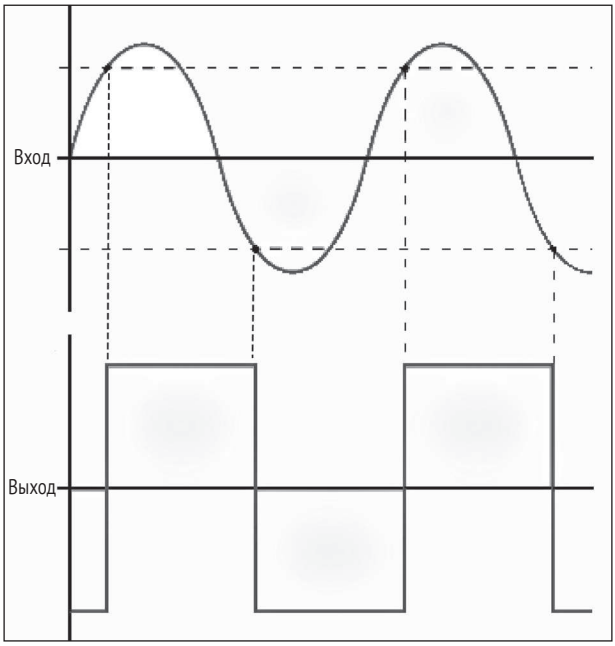

Рис. 3.7

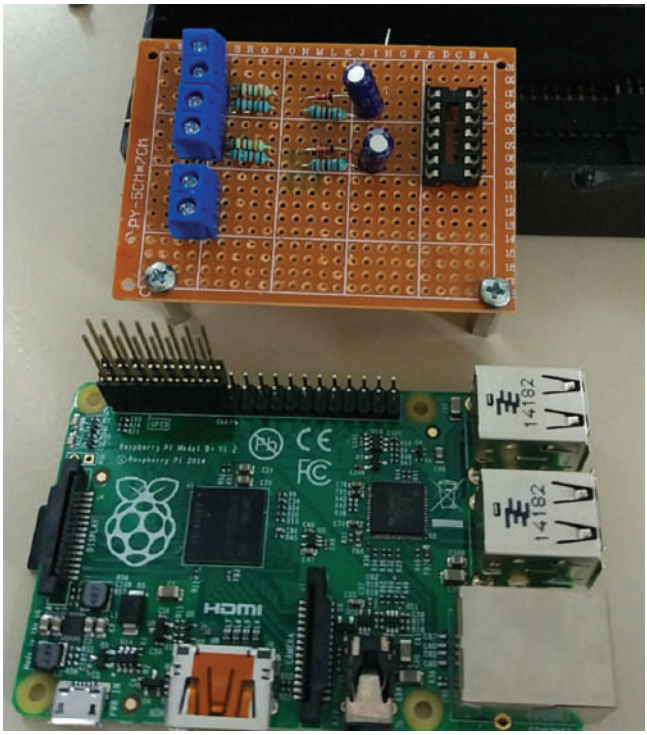

Рис. 3.8

На этом прототипе реализованы два канала устранения дребезга контактов для двух переключателей, необходимых в рассматриваемом проекте. Также добавлена клеммная колодка для подключения к источнику электропитания одноплатного компьютера. Это позволяет подавать электроэнергию на одноплатный компьютер через пятивольтовый многоштырьковый разъем вместо использования коннектора микро-USB на плате Raspberry Pi. При интеграции нескольких компонентов обычно проще запитать шины напрямую от источника электропитания через многоштырьковый разъем или его аналог, чем разбираться с подключением микро-USB.

Разумеется, этот прототип не является настоящей платой расширения HAT (Hardware Attached on Top), соответствующей спецификации сообщества Raspberry Pi Foundation, для которой должны быть выполнены следующие требования:

- необходим обязательный корректный блок памяти EEPROM, содержащий информацию о производителе, схему назначения контактов интерфейса GPIO и информацию об устройстве, подключаемом к контактам шины I2C `ID_SC` и `ID_SD` на одноплатном компьютере Paspberry Pi;
- обязателен современный 40-контактный (в форме гнезд – female) разъем интерфейса GPIO, обеспечивающий к тому же расстояние от SBC-платы не менее 8 мм;
- выполнение требований спецификации по механической части;
- если электропитание одноплатного компьютера обеспечивается через многоштырьковый разъем 5 В, то плата расширения HAT должна непрерывно поддерживать силу тока не менее 1,3 А.

С учетом требуемых I2C микросхем EEPROM (CAT24C32) и других дополнительных компонентов полная версия с использованием шести каналов, поддерживаемых шестиканальным триггером Шмитта с обратной связью IC (40106), должна выглядеть так, как показано на рис. 3.9.

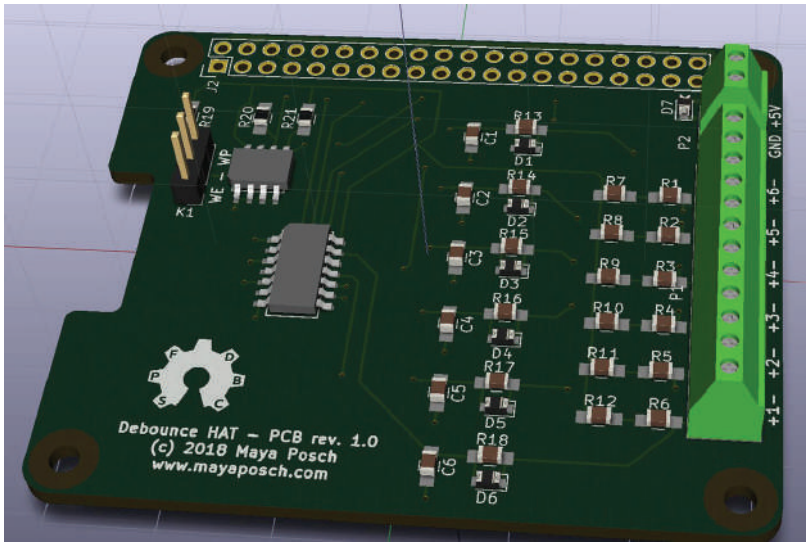

Рис. 3.9

Файлы для этого проекта в формате KiCad можно найти в учетной записи автора в репозитории GitHub **https://github.com/MayaPosch/DebounceHat**. При расширенном количестве каналов в систему достаточно просто добавляются дополнительные переключатели (коммутаторы), реле и иные элементы, позволяющие наблюдать за другими объектами, например за окнами, а также разнообразные сенсорные датчики, генерирующие сигналы высокого/низкого уровня.

Электропитание

Для рассматриваемого здесь проекта необходимо напряжение 5 В для платы Raspberry Pi и отдельное напряжение для источников света, включаемых и выключаемых с помощью реле. Поэтому выбран источник электропитания, способный обеспечить мощность, достаточную для одноплатного компьютера и для источников света. Для SBC достаточно силы тока 1–2 А, во втором случае все зависит от используемых световых устройств и требований к их обеспечению электропитанием.

Реализация

Служба наблюдения (мониторинга) может быть реализована как простой сервис `systemd`, то есть будет инициализироваться при запуске операционной системы. Таким образом, эта служба может контролироваться и перезапускаться с использованием всех обычных инструментальных средств systemd.

Для создаваемой службы должны быть удовлетворены следующие зависимости:

- POCO;
- WiringPi;
- libmosquittopp (и libmosquitto).

Библиотека libmosquitto (**https://mosquitto.org/man/libmosquitto-3.html**) используется для обеспечения поддержки протокола передачи сообщений MQTT. Библиотека libmosquittopp представляет собой обертку API libmosquitto, написанного на языке C, предоставляющую интерфейс на основе классов, упрощающий интеграцию библиотеки в проекты на языке C++.

Рабочая среда POCO (**https://pocoproject.org/**) – это набор API на языке C++ с высокой переносимостью, который обеспечивает полнофункциональность сетевых функций (включая поддержку протокола HTTP) для всех функций более низкого уровня. В рассматриваемом здесь проекте будет использоваться HTTP-сервер, поэтому необходима поддержка обработки его конфигурационных файлов.

Наконец, WiringPi (**http://wiringpi.com/**) – это фактический стандарт заголовка для доступа и использования функциональных возможностей разъема интерфейса GPIO на плате Paspberry Pi и совместимых системах. В нем реализованы API для обмена данными с устройствами I2C и UART, а также используется широтно-импульсная модуляция (PWM) и цифровые контакты. В рассматриваемом здесь проекте это позволит обмениваться данными с платой реле и с платой устранения дребезга контактов.

Текущая версия кода проекта доступна в репозитории автора на GitHub **https://github.com/MayaPosch/ClubStatusService**.

Начнем с рассмотрения основного файла проекта:

```
#include "listener.h"

#include <iostream>
#include <string>

using namespace std;

#include <Poco/Util/IniFileConfiguration.h>
#include <Poco/AutoPtr.h>
#include <Poco/Net/HTTPServer.h>

using namespace Poco::Util;
using namespace Poco;
using namespace Poco::Net;

#include "httprequestfactory.h"
#include "club.h"
```

Здесь в проект включаются заголовочные файлы, обеспечивающие основную функциональность стандартной библиотеки STL, а также поддержку HTTP-сервера и ini-файла из POCO. Заголовочный файл *listener.h* предназначен для класса MQTT. Указанные последними два заголовочных файла предназначены для поддержки работы HTTP-сервера и основной логики мониторинга соответственно.

```
int main(int argc, char* argv[]) {
          Club::log(LOG_INFO, "Starting ClubStatus server...");
          int rc;
          mosqpp::lib_init();

          Club::log(LOG_INFO, "Initialised C++ Mosquitto library.");

          string configFile;
          if (argc > 1) { configFile = argv[1]; }
          else { configFile = "config.ini"; }

          AutoPtr<IniFileConfiguration> config;
          try {
                config = new IniFileConfiguration(configFile);
          }
          catch (Poco::IOException &e) {
                Club::log(LOG_FATAL, "Main: I/O exception when opening configuration file: " +
                                     configFile + ". Aborting...");
                return 1;
          }

          string mqtt_host = config->getString("MQTT.host", "localhost");
          int mqtt_port = config->getInt("MQTT.port", 1883);
          string mqtt_user = config->getString("MQTT.user", "");
          string mqtt_pass = config->getString("MQTT.pass", "");
          string mqtt_topic = config->getString("MQTT.clubStatusTopic", "/public/clubstatus");
          bool relayactive = config->getBool("Relay.active", true);
          uint8_t relayaddress = config->getInt("Relay.address", 0x20);
```

В этом фрагменте инициализируется библиотека поддержки протокола MQTT (libmosquittopp) и делается попытка открыть файл конфигурации с использова-

нием пути и имени, заданного по умолчанию, если путь и имя не определены в параметрах командной строки.

Для POCO класс `IniFileConfiguration` используется для открытия и чтения файла конфигурации с генерацией исключения в том случае, если невозможно найти или открыть этот файл. POCO-тип `AutoPtr` равнозначен типу `unique_ptr` по стандарту C++11, позволяя создавать новый экземпляр в динамической памяти без необходимости выяснения его расположения в дальнейшем.

Далее считываются значения, необходимые для протокола MQTT и для обеспечения функциональности платы реле, которые определяют разумно подобранные характеристики по умолчанию:

```
Listener listener("ClubStatus", mqtt_host, mqtt_port, mqtt_user, mqtt_pass);

    Club::log(LOG_INFO, "Created listener, entering loop...");

    UInt16 port = config->getInt("HTTP.port", 80);
    HTTPServerParams* params = new HTTPServerParams;
    params->setMaxQueued(100);
    params->setMaxThreads(10);
    HTTPServer httpd(new RequestHandlerFactory, port, params);
    try {
          httpd.start();
    }
    catch (Poco::IOException &e) {
          Club::log(LOG_FATAL, "I/O Exception on HTTP server: port already in use?");
          return 1;
    }
    catch (...) {
          Club::log(LOG_FATAL, "Exception thrown for HTTP server start. Aborting.");
          return 1;
    }
```

В этом разделе инициализируется класс MQTT с передачей ему параметров, необходимых для установления соединения с MQTT-брокером. Далее считываются подробности конфигурации HTTP-сервера и создается новый экземпляр `HTTPServer`.

Экземпляр сервера конфигурируется с учетом переданного ему номера порта и некоторых ограничений по максимальному числу потоков, которые может использовать HTTP-сервер, а также по максимальному числу запросов на установление соединения, ожидающих своей очереди на обслуживание. Эти параметры важны для оптимизации производительности системы, поэтому подобный код необходим для систем с ограниченными ресурсами.

Новые клиентские запросы на установление соединения обрабатываются специализированным классом `RequestHandlerFactory`, код которого приведен ниже:

```
    Club::mqtt = &listener;
    Club::start(relayactive, relayaddress, mqtt_topic);

    while(1) {
          rc = listener.loop();
          if (rc){
              Club::log(LOG_ERROR, "Disconnected. Trying to
```

```
                                        reconnect...");
                listener.reconnect();
            }
        }

        mosqpp::lib_cleanup();
        httpd.stop();
        Club::stop();

        return 0;
}
```

Наконец, ссылка на созданный экземпляр класса `Listener` присваивается статическому члену `mqtt` класса `Club`. В дальнейшем это позволит существенно упростить использование объекта типа `Listener`, как мы увидим позже.

При вызове метода `start()` из класса `Club` начнется конфигурирование и мониторинг подключенного аппаратного оборудования, и это необходимо обеспечить в основной функции.

Далее следует вход в цикл для класса MQTT, гарантирующий постоянное соединение с MQTT-брокером. При выходе из цикла освобождаются ресурсы и останавливается работа HTTP-сервера и других объектов. Но поскольку цикл определен как бесконечный, строки кода после цикла никогда не будут выполнены в данной реализации.

Так как приведенная выше реализация предназначена для функционирования в непрерывном режиме 24/7, обеспечение завершения работы этого сервиса не является обязательным требованием. Относительно простым способом завершения работы может стать добавление обработчика сигнала, который при генерации прерывает выполнение цикла. Для упрощения примера данного проекта этот вариант не рассматривается.

Класс Listener

Ниже приведено объявление класса `Listener`:

```
class Listener : public mosqpp::mosquittopp {
          //
 public:
          Listener(string clientId, string host, int port, string user, string pass);
          ~Listener();

          void on_connect(int rc);
          void on_message(const struct mosquitto_message* message);
          void on_subscribe(int mid, int qos_count, const int* granted_qos);
          void sendMessage(string topic, string& message);
          void sendMessage(string& topic, char* message, int msgLength);
 };
```

Этот класс предоставляет простой API для обеспечения соединения с MQTT-брокером и отправки ему сообщений. Он является производным от класса `mosquittopp` и выполняет собственную реализацию ряда методов обратного вызова для обработки событий при установлении соединения по новым принятым сообщениям и подписки на публикации MQTT.

Рассмотрим подробнее реализацию данного класса:

```
#include "listener.h"

#include <iostream>

using namespace std;

Listener::Listener(string clientId, string host, int port, string user, string pass):
  mosquittopp(clientId.c_str()) {
          int keepalive = 60;
          username_pw_set(user.c_str(), pass.c_str());
          connect(host.c_str(), port, keepalive);
 }

 Listener::~Listener() {
          //
 }
```

В конструкторе выполняется присваивание специальной строки идентификации клиента MQTT с использованием конструктора класса `mosquittopp`. Определяется значение по умолчанию для интервала сохранения соединения, равное 60 с, то есть время, в течение которого соединение с MQTT-брокером остается открытым даже при отсутствии управляющих или любых других сообщений.

После определения имени пользователя и пароля устанавливается соединение с MQTT-брокером.

```
void Listener::on_connect(int rc) {
    cout << "Connected. Subscribing to topics...n";

          if (rc == 0) {
                // Подписка на требуемые публикации
                string topic = "/club/status";
                subscribe(0, topic.c_str(), 1);
          }
          else {
                cerr << "Connection failed. Aborting subscribing.n";
          }
 }
```

Эта функция обратного вызова активизируется при попытке соединения с MQTT-брокером. Проверяется значение `rc`, и если значение равно нулю (попытка успешна), то начинается процедура подписки на все требуемые публикации. В данном примере выполняется подписка только на одну публикацию `/club/status`. Если какие-либо другие MQTT-клиенты отправят сообщение в эту публикацию, то мы получим это сообщение с помощью следующей функции обратного вызова:

```
void Listener::on_message(const struct mosquitto_message* message) {
          string topic = message->topic;
          string payload = string((const char*) message->payload, message->payloadlen);

          if (topic == "/club/status") {
                string topic = "/club/status/response";
                char payload[] = { 0x01 };
```

```
                publish(0, topic.c_str(), 1, payload, 1); // QoS 1.
        }
}
```

В этой функции обратного вызова мы получаем структуру, содержащую тему публикации MQTT и само сообщение (полезную нагрузку). Затем тема публикации сравнивается со строками оформленных подписок, которые в данном случае представлены только строкой `/club/status`. При получении сообщения по этой теме подписки мы публикуем новое MQTT-сообщение также с указанием темы публикации и с соответствующей полезной нагрузкой. Самый последний параметр – значение качества обслуживания QoS (quality of service), для которого в этом примере установлен флаг, равный 1, – доставка, как минимум, однократная (deliver at least once). Это дает уверенность в том, что, как минимум, один из прочих MQTT-клиентов примет наше сообщение.

Полезная нагрузка MQTT-сообщения всегда представлена в бинарной форме, в данном примере это 1. Чтобы данное значение отображало состояние клубного зала (открыт или закрыт), необходимо интегрировать ответ из статического класса `Club`, который будет рассматриваться в следующем разделе.

Рассмотрим остальные функции (методы) класса `Listener`:

```
void Listener::on_subscribe(int mid, int qos_count, const int* granted_qos) {
        //
}

void Listener::sendMessage(string topic, string &message) {
        publish(0, topic.c_str(), message.length(), message.c_str(), true);
}

void Listener::sendMessage(string &topic, char* message, int msgLength) {
        publish(0, topic.c_str(), msgLength, message, true);
}
```

Функция обратного вызова для новой подписки здесь оставлена пустой, но могла бы использоваться для добавления возможности ведения журнала (logging) или аналогичной функциональности. Функция `sendMessage()` перегружена, что позволяет другим частям приложения также публиковать MQTT-сообщения.

Главная причина наличия этих двух различных функций состоит в том, что иногда проще использовать для отправки массив типа `char*`, например массив 8-битовых целых чисел как часть бинарного протокола, а в других случаях более удобной окажется строка в формате STL. Таким образом, мы пользуемся преимуществами обоих вариантов без преобразования форматов в любой точке кода, где возникает необходимость в отправке MQTT-сообщения.

Первый параметр метода `publish()` – идентификатор ID сообщения, специальное целое число, которое мы можем присвоить по своему усмотрению. Здесь определен идентификатор 0. Также используется флаг `retain` (последний параметр), для которого задано значение `true`. Истинное значение предполагает, что когда новый MQTT-клиент подписывается на публикацию, которую мы определили поддерживаемым сообщением, этот клиент всегда будет получать самое последнее сообщение, переданное в эту конкретную тему публикации.

Поскольку мы будем публиковать состояние клубного зала в некоторой теме публикации MQTT, желательно, чтобы самое последнее сообщение о состоянии

сохранялось MQTT-брокером, чтобы любой клиент, пользующийся этой информацией, немедленно получал бы текущее состояние непосредственно в момент установления соединения с брокером, а не ожидал бы очередного обновления состояния.

Класс Club

В заголовке для клуба объявляются классы, формирующие ядро проекта и ответственные за обработку входных данных, принимаемых от переключателей, за управление реле и за обновление состояния клубного зала:

```
#include <wiringPi.h>
#include <wiringPiI2C.h>
```

Сначала объясним предназначение включаемых файлов. Они добавляют в код поддержку основной функциональности интерфейса WiringPi GPIO, а также устройств I2C. В дальнейшем заголовки WiringPi можно включать в другие проекты, требующие поддержки функциональности последовательного интерфейса SPI, UART (последовательного), программной широтно-импульсной модуляции PWM, специализированной функциональности Raspberry Pi (Broadcom SoC) и т. п.

Определяются различные уровни ведения журнала (log) в форме перечисления (тип enum):

```
enum Log_level {
    LOG_FATAL = 1,
    LOG_ERROR = 2,
    LOG_WARNING = 3,
    LOG_INFO = 4,
    LOG_DEBUG = 5
 };
```

Предварительно объявляется класс `Listener`, так как он будет использоваться в реализации классов клуба, но включение соответствующего ему заголовочного файла в полном объеме пока не требуется:

```
class Listener;

class ClubUpdater : public Runnable {
          TimerCallback<ClubUpdater>* cb;
          uint8_t regDir0;
          uint8_t regOut0;
          int i2cHandle;
          Timer* timer;
          Mutex mutex;
          Mutex timerMutex;
          Condition timerCnd;
          bool powerTimerActive;
          bool powerTimerStarted;

 public:
          void run();
          void updateStatus();
          void writeRelayOutputs();
          void setPowerState(Timer &t);
};
```

Класс `ClubUpdater` отвечает за конфигурирование блока расширения интерфейса GPIO на основе I2C, который управляет реле, а также обрабатывает все обновления состояния помещения клуба. Экземпляр класса `Timer` из рабочей среды POCO используется для создания задержки при определении энергетического состояния реле, как мы увидим при дальнейшем рассмотрении реализации.

Этот класс является производным от POCO-класса `Runnable`, представляющего собой базовый класс, необходимый для POCO-класса `Thread`, являющегося оберткой для собственных потоков целевой системы.

Две переменные-члена типа `uint8_t` соответствуют двум регистрам устройства расширения интерфейса I2C GPIO и позволяют устанавливать направление и значение выходных контактов на этом устройстве, которое эффективно управляет подключенными реле.

```
class Club {
          static Thread updateThread;
          static ClubUpdater updater;

          static void lockISRCallback();
          static void statusISRCallback();

 public:
          static bool clubOff;
          static bool clubLocked;
          static bool powerOn;
          static Listener* mqtt;
          static bool relayActive;
          static uint8_t relayAddress;
          static string mqttTopic;      // Публикуемая тема подписки – текущие обновления
                                           состояния

          static Condition clubCnd;
          static Mutex clubCndMutex;
          static Mutex logMutex;
          static bool clubChanged ;
          static bool running;
          static bool clubIsClosed;
          static bool firstRun;
          static bool lockChanged;
          static bool statusChanged;
          static bool previousLockValue;
          static bool previousStatusValue;

          static bool start(bool relayactive, uint8_t relayaddress, string topic);
          static void stop();
          static void setRelay();
          static void log(Log_level level, string msg);
};
```

Класс `Club` можно интерпретировать как входную (обеспечивающую ввод данных) часть системы, выполняющую настройку и обработку прерываний. Кроме того, он действует как центральный (статический) класс, содержащий все переменные, определяющие состояние помещения клуба, как, например, состояние замка входной двери, состояние переключателя и состояние системы электропитания (клуб открыт или закрыт).

Этот класс определен как полностью статический, поэтому его можно свободно использовать в любых частях программы для получения информации о состоянии клубного зала.

Продолжим рассмотрение реализации:

```
#include "club.h"

#include <iostream>

using namespace std;

#include <Poco/NumberFormatter.h>

using namespace Poco;

#include "listener.h"
```

Здесь в программу включается заголовочный файл для класса `Listener`, чтобы появилась возможность его использования. Также включен заголовочный файл POCO-класса `NumberFormatter`, позволяющий форматировать целочисленные значения при записи их в журнал.

```
#define REG_INPUT_PORT0              0x00
#define REG_INPUT_PORT1              0x01
#define REG_OUTPUT_PORT0             0x02
#define REG_OUTPUT_PORT1             0x03
#define REG_POL_INV_PORT0            0x04
#define REG_POL_INV_PORT1            0x05
#define REG_CONF_PORT0               0x06
#define REG_CONG_PORT1               0x07
#define REG_OUT_DRV_STRENGTH_PORT0_L 0x40
#define REG_OUT_DRV_STRENGTH_PORT0_H 0x41
#define REG_OUT_DRV_STRENGTH_PORT1_L 0x42
#define REG_OUT_DRV_STRENGTH_PORT1_H 0x43
#define REG_INPUT_LATCH_PORT0        0x44
#define REG_INPUT_LATCH_PORT1        0x45
#define REG_PUD_EN_PORT0             0x46
#define REG_PUD_EN_PORT1             0x47
#define REG_PUD_SEL_PORT0            0x48
#define REG_PUD_SEL_PORT1            0x49
#define REG_INT_MASK_PORT0           0x4A
#define REG_INT_MASK_PORT1           0x4B
#define REG_INT_STATUS_PORT0         0x4C
#define REG_INT_STATUS_PORT1         0x4D
#define REG_OUTPUT_PORT_CONF         0x4F
```

Далее определяются все регистры целевого устройства расширения интерфейса GPIO NXP PCAL9535A. Даже если мы используем только два регистра из этого набора, рекомендуется всегда добавлять полный список регистров для упрощения дальнейшего расширения и изменения кода. Можно также использовать отдельный заголовочный файл для определения этих регистров, позволяющий без затруднений пользоваться различными устройствами расширения интерфейса GPIO без существенных изменений в исходном коде.

```
#define RELAY_POWER 0
#define RELAY_GREEN 1
```

```
#define RELAY_YELLOW 2
#define RELAY_RED 3
```

Здесь определяется функциональность, связанная с каждым реле, соответствующая конкретному выходному контакту на микросхеме расширения интерфейса GPIO. У нас имеется четыре реле, поэтому используются четыре контакта (штырька). Подключение производится к первой группе (из двух) из восьми контактов на микросхеме.

Разумеется, очень важно, чтобы все эти определения полностью соответствовали физическим соединениям с существующими реле. В зависимости от конкретного варианта практического использования можно сформировать следующую конфигурацию:

```
bool Club::clubOff;
bool Club::clubLocked;
bool Club::powerOn;
Thread Club::updateThread;
ClubUpdater Club::updater;
bool Club::relayActive;
uint8_t Club::relayAddress;
string Club::mqttTopic;
Listener* Club::mqtt = 0;

Condition Club::clubCnd;
Mutex Club::clubCndMutex;
Mutex Club::logMutex;
bool Club::clubChanged = false;
bool Club::running = false;
bool Club::clubIsClosed = true;
bool Club::firstRun = true;
bool Club::lockChanged = false;
bool Club::statusChanged = false;
bool Club::previousLockValue = false;
bool Club::previousStatusValue = false;
```

Поскольку `Club` является полностью статическим классом, все его члены инициализируются до перехода к реализации класса `ClubUpdater`.

```
void ClubUpdater::run() {
    regDir0 = 0x00;
    regOut0 = 0x00;
    Club::powerOn = false;
    powerTimerActive = false;
    powerTimerStarted = false;
    cb = new TimerCallback<ClubUpdater>(*this, &ClubUpdater::setPowerState);
    timer = new Timer(10 * 1000, 0);
```

При создании экземпляра этого класса вызывается его метод (функция) `run()`. Здесь устанавливается количество умолчаний. Для переменных регистров направления и вывода изначально устанавливаются нулевые значения. Для состояния электропитания клубного зала устанавливается значение `false`, логические переменные таймера электропитания также получают значение `false`, так как таймер электропитания пока не активен. Этот таймер используется для создания

задержки при подключении и отключении электропитания, как мы увидим немного ниже.

По умолчанию назначается задержка по этому таймеру величиной в десять секунд. Разумеется, эту настройку также можно изменить.

```
if (Club::relayActive) {
    Club::log(LOG_INFO, "ClubUpdater: Starting i2c relay device.");
    i2cHandle = wiringPiI2CSetup(Club::relayAddress);
    if (i2cHandle == -1) {
        Club::log(LOG_FATAL, string("ClubUpdater: error starting i2c relay device."));
        return;
    }

    wiringPiI2CWriteReg8(i2cHandle, REG_CONF_PORT0, 0x00);
    wiringPiI2CWriteReg8(i2cHandle, REG_OUTPUT_PORT0, 0x00);

    Club::log(LOG_INFO, "ClubUpdater: Finished configuring the i2c relay device's registers.");
}
```

Далее выполняется настройка устройства расширения интерфейса I2C GPIO. Для этого требуется адрес устройства I2C, который ранее был передан в класс `Club`. Функция настройки гарантирует, что по этому адресу на шине I2C действительно размещено активное устройство I2C. После настройки это устройство должно быть полностью готово к обмену данными. Но этот шаг можно пропустить, установив для переменной `relayActive` значение `false`. Это делается посредством установки соответствующего значения в файле конфигурации, что удобно при запуске комплексных интеграционных тестов всей системы без шины I2C или при отсутствии подключенного устройства.

После завершения этапа настройки присваиваются начальные значения регистрам направления и вывода для первой группы (банка). Оба значения записываются с указанием нулевых байтов, так что для всех восьми контактов, управляемых этими регистрами, устанавливается режим вывода и состояние, равное бинарному нулю (низкий уровень). Таким образом, все реле, соединенные с первыми четырьмя контактами, изначально отключены.

```
        updateStatus();

        Club::log(LOG_INFO, "ClubUpdater: Initial status update complete.");
        Club::log(LOG_INFO, "ClubUpdater: Entering waiting condition.");

        while (Club::running) {
              Club::clubCndMutex.lock();
              if (!Club::clubCnd.tryWait(Club::clubCndMutex, 60 * 1000)){
                    Club::clubCndMutex.unlock();
                    if (!Club::clubChanged) { continue; }
              }
              else {
                    Club::clubCndMutex.unlock();
              }

              updateStatus();
        }
}
```

После завершения всех вышеописанных этапов конфигурации выполняется первое обновление состояния клубного зала с использованием той же функции, которая будет вызываться и в дальнейшем при любых изменениях на входе. В результате проверяются все входные объекты, а для выходных объектов устанавливается соответствующее состояние.

Наконец, выполняется вход в цикл ожидания. Этот цикл управляется логической переменной `Club::running`, позволяющей прервать выполнение цикла с помощью обработчика сигнала или аналогичного механизма. Действительный режим ожидания обеспечивается с использованием условной переменной, то есть здесь ожидается либо завершение интервала тайм-аута, равного одной минуте (после чего происходит возврат в состояние ожидания после быстрой проверки), либо получение сигнала по одному из прерываний, которые будут настроены ниже для входных объектов.

Далее рассмотрим функцию, предназначенную для обновления состояния входных объектов.

```
void ClubUpdater::updateStatus() {
    Club::clubChanged = false;

    if (Club::lockChanged) {
          string state = (Club::clubLocked) ? "locked" : "unlocked";
          Club::log(LOG_INFO, string("ClubUpdater: lock status changed to ") + state);
          Club::lockChanged = false;

          if (Club::clubLocked == Club::previousLockValue) {
                Club::log(LOG_WARNING, string("ClubUpdater: lock interrupt triggered,
                                            but value hasn't changed. Aborting."));
                return;
          }

          Club::previousLockValue = Club::clubLocked;
    }
    else if (Club::statusChanged) {
          string state = (Club::clubOff) ? "off" : "on";
          Club::log(LOG_INFO, string("ClubUpdater: status switch status changed to ") + state);
          Club::statusChanged = false;

          if (Club::clubOff == Club::previousStatusValue) {
                Club::log(LOG_WARNING, string("ClubUpdater: status interrupt triggered,
                                              but value hasn't changed. Aborting."));
                return;
          }

          Club::previousStatusValue = Club::clubOff;
    }
    else if (Club::firstRun) {
          Club::log(LOG_INFO, string("ClubUpdater: starting initial update run."));
          Club::firstRun = false;
    }
    else {
          Club::log(LOG_ERROR, string("ClubUpdater: update triggered, but no change detected.
                                      Aborting."));
          return;
}
```

Сразу после входа в эту функцию обновления выполняется проверка логической переменной `Club::statusChanged` на равенство значению `false`, чтобы можно было снова установить ее через один из обработчиков прерываний.

После этого проверяется, что именно изменилось во входных объектах. Если сработал триггер переключателя замка, то для соответствующей ему логической переменной устанавливается значение `true` или переменная состояния переключателя, вероятно, была изменена. В этом случае переменная сбрасывается (переустанавливается) и вновь прочитанное значение сравнивается с самым последним известным значением для этого входного объекта.

Для проверки работоспособности мы игнорируем изменение триггера, если значение не изменилось. Это может произойти, если прерывание было сгенерировано из-за шума, например когда сигнальный провод переключателя расположен слишком близко к силовым линиям электропитания. Любые колебания характеристик силовой линии электропитания создают импульсные наводки в сигнальном проводе переключателя, поэтому возможна случайная генерация прерывания на контакте интерфейса GPIO. Это всего лишь один очевидный пример того, что может происходить в неидеальном физическом мире, который наглядно демонстрирует важность тщательного учета воздействия на надежность системы как аппаратного, так и программного обеспечения.

В дополнение к этой проверке событие фиксируется в журнале с использованием централизованного механизма журналирования и обновляется буферизованное значение входного объекта для возможности использования в следующей фазе выполнения.

Две последние ветви в конструкции `if/else` работают в начальной фазе выполнения, как и обработчик, заданный по умолчанию. При первоначальном запуске этой функции способом, рассмотренным выше, не генерируются никакие прерывания, поэтому очевидно, что необходимо добавить к двум вариантам третий для переключателей состояния и замка.

```
if (Club::clubIsClosed && !Club::clubOff) {
      Club::clubIsClosed = false;

      Club::log(LOG_INFO, string("ClubUpdater: Opening club."));

      Club::powerOn = true;
      try {
            if (!powerTimerStarted) {
                  timer->start(*cb);
                  powerTimerStarted = true;
            }
            else {
                  timer->stop();
                  timer->start(*cb);
            }
      }
      catch (Poco::IllegalStateException &e) {
            Club::log(LOG_ERROR, "ClubUpdater: IllegalStateException on timer start: " +
                              e.message());
            return;
      }
```

```
        catch (...) {
              Club::log(LOG_ERROR, "ClubUpdater: Unknown exception on timer start.");
              return;
        }

        powerTimerActive = true;

        Club::log(LOG_INFO, "ClubUpdater: Started power timer...");

        char msg = { '1' };
        Club::mqtt->sendMessage(Club::mqttTopic, &msg, 1);

        Club::log(LOG_DEBUG, "ClubUpdater: Sent MQTT message.");
  }
  else if (!Club::clubIsClosed && Club::clubOff) {
        Club::clubIsClosed = true;

        Club::log(LOG_INFO, string("ClubUpdater: Closing club."));

        Club::powerOn = false;
        try {
              if (!powerTimerStarted) {
                    timer->start(*cb);
                    powerTimerStarted = true;
              }
              else {
                    timer->stop();
                    timer->start(*cb);
              }
        }
        catch (Poco::IllegalStateException &e) {
              Club::log(LOG_ERROR, "ClubUpdater: IllegalStateException on timer start: " +
                                   e.message());
              return;
        }
        catch (...) {
              Club::log(LOG_ERROR, "ClubUpdater: Unknown exception on timer start.");
              return;
        }

        powerTimerActive = true;

        Club::log(LOG_INFO, "ClubUpdater: Started power timer...");

        char msg = { '0' };
        Club::mqtt->sendMessage(Club::mqttTopic, &msg, 1);

        Club::log(LOG_DEBUG, "ClubUpdater: Sent MQTT message.");
  }
```

Далее проверяется, необходимо ли изменение состояния клубного зала с закрытого на открытый или наоборот. Это определяется проверкой изменения значения логической переменной состояния помещения клуба `Club::clubOff` по сравнению со значением логической переменной `Club::clubIsClosed`, которая хранит самое последнее известное состояние.

По существу, если переключатель состояния изменился с «Вкл» на «Выкл» или наоборот, то это будет обнаружено, и начнется изменение, то есть переход в новое

состояние. Это означает, что будет запущен таймер электропитания, который позволит после предварительно определенной задержки включить или отключить подачу электропитания (не постоянную) в клубный зал.

POCO-класс `Timer` требует, чтобы таймер обязательно был остановлен до (повторного) запуска, если ранее он был активизирован. Поэтому необходимо добавить еще одну проверку.

Кроме того, используется ссылка на класс MQTT-клиента для передачи MQTT-брокеру сообщения об изменении состояния клубного зала. Сообщение содержит ASCII-символ 1 или 0. Это сообщение может использоваться для триггерного переключения других систем, которые могут обновлять в режиме онлайн состояние клубного зала или выполнять даже более сложные интеллектуальные операции.

Разумеется, полезное содержимое сообщения также может быть конфигурируемым.

Далее будет рассматриваться процедура изменения цветов индикатора состояния с учетом состояния элетропитания клубного зала. Для этого используется табл. 3.3.

Таблица 3.3

Цвет	Переключатель состояния	Переключатель замка	Состояние электропитания
Зеленый	Включен	Открыт	Включено
Желтый	Отключен	Открыт	Отключено
Красный	Отключен	Заперт	Отключено
Желтый и красный	Включен	Заперт	Включено

Ниже приведена реализация этой процедуры:

```
if (Club::clubOff) {
      Club::log(LOG_INFO, string("ClubUpdater: New lights, clubstatus off."));

      mutex.lock();
      string state = (Club::powerOn) ? "on" : "off";
      if (powerTimerActive) {
            Club::log(LOG_DEBUG, string("ClubUpdater: Power timer active,
                                 inverting power state from: ") + state);
            regOut0 = !Club::powerOn;
      }
      else {
            Club::log(LOG_DEBUG, string("ClubUpdater: Power timer not active,
                                 using current power state: ") + state);
            regOut0 = Club::powerOn;
      }

      if (Club::clubLocked) {
            Club::log(LOG_INFO, string("ClubUpdater: Red on."));
            regOut0 |= (1UL << RELAY_RED);
      }
      else {
            Club::log(LOG_INFO, string("ClubUpdater: Yellow on."));
            regOut0 |= (1UL << RELAY_YELLOW);
      }
```

```
        Club::log(LOG_DEBUG, "ClubUpdater: Changing output register to: 0x" +
                             NumberFormatter::formatHex(regOut0));

        writeRelayOutputs();
        mutex.unlock();
    }
```

В первую очередь проверяется состояние электропитания клубного зала, в результате чего мы получаем значение первого бита выходного регистра. Если таймер электропитания активен, необходимо инвертировать состояние электропитания, так как необходимо записать текущее состояние, а не будущее, которое сохранено в соответствующей логической переменной.

Если переключатель состояния клубного зала находится в позиции «Выкл», то состояние переключателя замка определяет окончательный цвет. При запертом клубном зале активизируется реле красного света, в противном случае срабатывает реле желтого света. Последний вариант соответствует промежуточному состоянию, когда клубный зал обесточен, но пока еще не заперт.

Здесь использование мьютекса (mutex) гарантирует, что запись в выходной регистр устройства I2C, а также обновление локальной переменной этого регистра выполняется синхронизированно.

```
    else {
            Club::log(LOG_INFO, string("ClubUpdater: New lights, clubstatus on."));

            mutex.lock();
            string state = (Club::powerOn) ? "on" : "off";
            if (powerTimerActive) {
                 Club::log(LOG_DEBUG, string("ClubUpdater: Power timer active,
                                             inverting power state from: ") + state);
                 regOut0 = !Club::powerOn; // Необходимо инвертировать значение,
            }                              если таймер активен
            else {
                  Club::log(LOG_DEBUG, string("ClubUpdater: Power timer not active,
                                             using current power state: ") + state);
                  regOut0 = Club::powerOn; // Используется текущее значение состояния
            }

            if (Club::clubLocked) {
                  Club::log(LOG_INFO, string("ClubUpdater: Yellow & Red on."));
                  regOut0 |= (1UL << RELAY_YELLOW);
                  regOut0 |= (1UL << RELAY_RED);
            }
            else {
                  Club::log(LOG_INFO, string("ClubUpdater: Green on."));
                  regOut0 |= (1UL << RELAY_GREEN);
            }
            Club::log(LOG_DEBUG, "ClubUpdater: Changing output register to: 0x" +
                                  NumberFormatter::formatHex(regOut0));

            writeRelayOutputs();
            mutex.unlock();
     }
}
```

Если переключатель состояния клубного зала находится в позиции «Вкл», то возможны два других варианта цвета: зеленый, если дверь в клубный зал не заперта и переключатель состояния также включен; если же переключатель состояния включен, но дверь заперта, то активизируются реле желтого и красного света.

После завершения формирования нового содержимого выходного регистра всегда используется функция `writeRelayOutputs()` для записи локально созданной версии в удаленное устройство и переключения реле в новое состояние.

```
void ClubUpdater::writeRelayOutputs() {
    wiringPiI2CWriteReg8(i2cHandle, REG_OUTPUT_PORT0, regOut0);
    Club::log(LOG_DEBUG, "ClubUpdater: Finished writing relay outputs with: 0x" +
                         NumberFormatter::formatHex(regOut0));
}
```

Эта функция очень проста, она использует API I2C WiringPi для записи одного 8-битового значения в выходной регистр подключенного устройства. Кроме того, записываемое значение регистрируется в журнале.

```
  void ClubUpdater::setPowerState(Timer &t) {
          Club::log(LOG_INFO, string("ClubUpdater: setPowerState called."));

          mutex.lock();
          if (Club::powerOn) { regOut0 |= (1UL << RELAY_POWER); }
          else { regOut0 &= ~(1UL << RELAY_POWER); }

          Club::log(LOG_DEBUG, "ClubUpdater: Writing relay with: 0x" +
                               NumberFormatter::formatHex(regOut0));

          writeRelayOutputs();

          powerTimerActive = false;
          mutex.unlock();
  }
```

В этой функции устанавливается состояние электропитания клубного зала, равное значению, содержащемуся в соответствующей логической переменной. При этом используется тот же мьютекс, который применялся при обновлении состояния реле, управляющих цветом, для клубного зала. Но здесь не создается содержимое выходного регистра с нуля, а вместо этого выборочно переключается первый бит этой переменной.

После изменения (переключения) данного бита запись в удаленное устройство выполняется обычным образом, вызывая переключение состояния электропитания в клубном зале.

Теперь рассмотрим реализацию самого класса `Club`, начиная с самой первой функции, которая вызывается для его инициализации.

```
bool Club::start(bool relayactive, uint8_t relayaddress, string topic) {
          Club::log(LOG_INFO, "Club: starting up...");

          relayActive = relayactive;
          relayAddress = relayaddress;
          mqttTopic = topic;

          wiringPiSetup();

          Club::log(LOG_INFO,  "Club: Finished wiringPi setup.");
```

```
        pinMode(0, INPUT);
        pinMode(7, INPUT);
        pullUpDnControl(0, PUD_DOWN);
        pullUpDnControl(7, PUD_DOWN);
        clubLocked = digitalRead(0);
        clubOff = !digitalRead(7);

        previousLockValue = clubLocked;
        previousStatusValue = clubOff;

        Club::log(LOG_INFO, "Club: Finished configuring pins.");

        wiringPiISR(0, INT_EDGE_BOTH, &lockISRCallback);
        wiringPiISR(7, INT_EDGE_BOTH, &statusISRCallback);

        Club::log(LOG_INFO, "Club: Configured interrupts.");

        running = true;

        updateThread.start(updater);

        Club::log(LOG_INFO, "Club: Started update thread.");

        return true;
}
```

С этой функции начинается вся система мониторинга клубного помещения, как мы видели выше, в точке входа в приложение. Функция принимает несколько параметров, позволяющих включать и отключать функциональность реле, определять адрес реле на шине I2C (если реле используются) и тему публикации MQTT, в которой будут публиковаться изменения состояния клубного зала.

После установки значений членов-переменных с использованием этих параметров начинается инициализация рабочей среды WiringPi. Существует множество разнообразных функций инициализации, предлагаемых рабочей средой WiringPi, которые в основном различаются по способу доступа к контактам интерфейса GPIO.

В рассматриваемом примере выбрана функция `wiringPiSetup()`, наиболее удобная для практического применения, поскольку использует виртуальные номера контактов, которые отображаются на контакты более низкого уровня Broadcom SoC. Главным преимуществом нумерации контактов WiringPi является то, что они остаются постоянными для различных вариаций и модификаций одноплатных компьютеров Raspberry Pi.

Как при использовании нумерации контактов Broadcom (BCM), так и при использовании физических позиций контактов в разъеме платы существует риск возникновения изменений в различных вариациях платы, но схема нумерации WiringPi способна компенсировать этот риск.

Для целей рассматриваемого здесь примера используются контакты одноплатного компьютера, указанные в табл. 3.4.

Таблица 3.4

	Переключатель замка	**Переключатель состояния**
Broadcom (BCM)	17	4
Физическая позиция	11	7
WiringPi	0	7

После инициализации библиотеки (рабочей среды) WiringPi устанавливается требуемый режим использования контактов, то есть оба указанных контакта будут работать как входные. Затем разрешается понижение напряжения на выходе этих контактов. Такой режим обеспечивает встроенное понижающее сопротивление на плате компьютера, которое всегда пытается понижать уровень входного сигнала (связанного с заземлением). Необходимость наличия работающего понижающего или повышающего сопротивления для входных (или выходных) контактов зависит от конкретных условий эксплуатации, особенно для подключенной электрической схемы.

Важно рассмотреть поведение подключенной электрической схемы. Если в схеме возникает явление «плавающего» значения в линии связи, то это может привести к нежелательному поведению входного контакта при случайных изменениях значения. Понижая или повышая уровень сигнала в линии, можно быть уверенным в том, что считываемый с контакта сигнал не является всего лишь шумом.

При таком режиме работы, установленном для используемых в проекте контактов, мы считываем с них первоначальные значения, что позволяет активизировать функцию обновления из класса `ClubUpdater` с передачей текущих значений в данный момент. Но прежде чем сделать это, сначала необходимо зарегистрировать методы прерываний для обоих контактов.

Обработчик прерываний представляет собой нечто большее, чем простой обратный вызов, активизируемый при наступлении определенного события на заданном контакте. Функция обработки прерываний (ISR) рабочей среды WiringPi принимает номер контакта, тип события и ссылку на функцию обработчика прерывания, которую необходимо использовать. Для типа события, выбранного в рассматриваемом примере, предусмотрен специализированный обработчик прерывания, активизируемый при каждом изменении значения на входном контакте – с высокого на низкое и наоборот. Это означает, что обработчик прерываний будет вызываться при переходе подключенного переключателя из состояния «Вкл» в состояние «Выкл» или из состояния «Выкл» в состояние «Вкл».

Далее инициализируется поток обновления с использованием экземпляра класса `ClubUpdater` и выполняется принудительная передача его в собственный поток.

```
void Club::stop() {
          running = false;
 }
```

Вызов этой функции позволит завершиться циклу в функции `run()` класса `ClubUpdater`, что, в свою очередь, завершит поток, выполняемый в ней, таким образом, остальная часть приложения завершит свою работу в безопасном режиме.

```
void Club::lockISRCallback() {
          clubLocked = digitalRead(0);
          lockChanged = true;

          clubChanged = true;
          clubCnd.signal();
}

void Club::statusISRCallback() {
          clubOff = !digitalRead(7);
```

```
        statusChanged = true;

        clubChanged = true;
        clubCnd.signal();
}
```

Оба используемых в этом примере обработчика прерываний достаточно просты. Когда ОС принимает сигнал о прерывании, она активизирует соответствующий обработчик, в котором считывается текущее значение входного контакта, и при необходимости это значение инвертируется. Для переменной `statusChanged` или `lockChanged` устанавливается значение `true`, чтобы сообщить функции обновления, какое именно из прерываний произошло.

То же самое делается для логической переменной `clubChanged` перед подачей сигнала условной переменной, которую ожидает цикл `run` класса `ClubUpdate`.

В завершающей части этого класса расположена функция ведения журнала (logging).

```
void Club::log(Log_level level, string msg) {
    logMutex.lock();
    switch (level) {
          case LOG_FATAL: {
                cerr << "FATAL:t" << msg << endl;
                string message = string("ClubStatus FATAL: ") + msg;
                if (mqtt) {
                      mqtt->sendMessage("/log/fatal", message);
                }
                break;
          }
          case LOG_ERROR: {
                cerr << "ERROR:t" << msg << endl;
                string message = string("ClubStatus ERROR: ") + msg;
                if (mqtt) {
                      mqtt->sendMessage("/log/error", message);
                }
                break;
          }
          case LOG_WARNING: {
                cerr << "WARNING:t" << msg << endl;
                string message = string("ClubStatus WARNING: ") + msg;
                if (mqtt) {
                      mqtt->sendMessage("/log/warning", message);
                }
                break;
          }
          case LOG_INFO: {
                cout << "INFO: t" << msg << endl;
                string message = string("ClubStatus INFO: ") + msg;
                if (mqtt) {
                      mqtt->sendMessage("/log/info", message);
                }
                break;
          }
```

```
        case LOG_DEBUG: {
              cout << "DEBUG:t" << msg << endl;
              string message = string("ClubStatus DEBUG: ") + msg;
              if (mqtt) {
                    mqtt->sendMessage("/log/debug", message);
              }
              break;
        }
        default:
              break;
    }

    logMutex.unlock();
}
```

Здесь используется еще один мьютекс для синхронизации процедур записи в журнал с перенаправлением в системный журнал (или в консоль) и для предотвращения параллельного доступа к классу MQTT, когда различные части приложения одновременно вызывают эту функцию. В дальнейшем можно будет видеть, что эта функция журналирования используется и в других классах.

С помощью данной функции можно вести журнал как локально (системный журнал), так и в удаленном режиме, используя для этого протокол MQTT.

Обработчик запросов HTTP

Когда POCO HTTP-сервер принимает новый запрос на установление соединения от клиента, он использует новый экземпляр класса `RequestHandlerFactory` для создания обработчика этого конкретного запроса. Так как это весьма простой класс, его реализация полностью размещена в заголовочном файле.

```
#include <Poco/Net/HTTPRequestHandlerFactory.h>
#include <Poco/Net/HTTPServerRequest.h>

using namespace Poco::Net;

#include "statushandler.h"
#include "datahandler.h"

class RequestHandlerFactory: public HTTPRequestHandlerFactory {
 public:
          RequestHandlerFactory() {}
          HTTPRequestHandler* createRequestHandler(const HTTPServerRequest& request) {
                if (request.getURI().compare(0, 12, "/clubstatus/") == 0) {
                     return new StatusHandler();
                }
                else { return new DataHandler(); }
          }
};
```

Этот класс не выполняет никаких особенных действий, кроме сравнения URL, который HTTP-сервер предоставил для определения типа обработчика, создания соответствующего экземпляра, после чего завершает свою работу. Здесь можно видеть, что если строка URL начинается с `/clubstatus`, то возвращается экземпляр обработчика состояния, который выполняет реализацию REST API.

Обработчиком по умолчанию является простой файловый сервер, который пытается интерпретировать запрос как имя файла. Более подробно об этом – в следующем разделе.

Обработчик состояния

Этот обработчик выполняет реализацию простого REST API и возвращает структуру формата JSON, содержащую текущее состояние помещения клуба. Такой обработчик может использоваться внешним приложением для вывода в реальном времени информации о системе, что может оказаться полезным для панели управления или для веб-сайта.

Реализация этого класса также полностью размещена в заголовочном файле, поскольку он достаточно прост.

```
#include <Poco/Net/HTTPRequestHandler.h>
#include <Poco/Net/HTTPServerResponse.h>
#include <Poco/Net/HTTPServerRequest.h>
#include <Poco/URI.h>

using namespace Poco;
using namespace Poco::Net;

#include "club.h"

class StatusHandler: public HTTPRequestHandler {
 public:
          void handleRequest(HTTPServerRequest& request, HTTPServerResponse& response)  {
                Club::log(LOG_INFO, "StatusHandler: Request from " +
                                    request.clientAddress().toString());
                URI uri(request.getURI());
                vector<string> parts;
                uri.getPathSegments(parts);

                response.setContentType("application/json");
                response.setChunkedTransferEncoding(true);

                if (parts.size() == 1) {
                      ostream& ostr = response.send();
                      ostr << "{ "clubstatus": " << !Club::clubOff << ",";
                      ostr << ""lock": " << Club::clubLocked << ",";
                      ostr << ""power": " << Club::powerOn << "";
                      ostr << "}";
                }
                else {
                      response.setStatus(HTTPResponse::HTTP_BAD_REQUEST);
                      ostream& ostr = response.send();
                      ostr << "{ "error": "Invalid request." }";
                }
          }
};
```

Здесь используется централизованная функция журналирования из класса `Club` для регистрации подробностей входящих запросов. В журнал записывается только IP-адрес клиента, но можно воспользоваться API-классом `HTTPServerRequest` из рабочей среды POCO для получения еще более подробной информации.

Далее из запроса извлекается URI и выделяется сегмент пути (path) URL в экземпляр вектора. После установки типа содержимого и передачи закодированных параметров настройки объекта ответа проверяется, действительно ли был получен ожидаемый вызов REST API. В этой точке формируется строка в формате JSON, извлекается информация о состоянии клубного зала из класса `Club`, которая возвращается в строке ответа.

В объект JSON включается информация об общем состоянии клубного зала с инвертированием соответствующей логической переменной, а также состояние замка и электропитания зала, при этом 1 означает, что замок закрыт или электропитание подается соответственно.

Если в пути URL имеются дополнительные сегменты, то он определяется как нераспознаваемый вызов API, следовательно, в ответе должно возвращаться сообщение об ошибке запроса HTTP 400 (Bad Request).

Обработчик данных

Обработчик данных вызывается при определении отсутствия вызова REST API механизмом обработчика запроса. Обработчик данных пытается найти заданный файл, считать его с диска и вернуть его содержимое вместе с корректными заголовками HTTP. Этот класс также реализован в соответствующем заголовочном файле.

```
#include <Poco/Net/HTTPRequestHandler.h>
#include <Poco/Net/HTTPServerResponse.h>
#include <Poco/Net/HTTPServerRequest.h>
#include <Poco/URI.h>
#include <Poco/File.h>

using namespace Poco::Net;
using namespace Poco;

class DataHandler: public HTTPRequestHandler {
 public:
    void handleRequest(HTTPServerRequest& request, HTTPServerResponse& response) {
          Club::log(LOG_INFO, "DataHandler: Request from " + request.clientAddress().toString());
          // Получение пути и проверка всех конечных точек для фильтрации
          URI uri(request.getURI());
          string path = uri.getPath();
          string fileroot = "htdocs";
          if (path.empty() || path == "/") { path = "/index.html"; }
          File file(fileroot + path);
          Club::log(LOG_INFO, "DataHandler: Request for " + file.path());
```

Здесь делается предположение о том, что любые требуемые файлы можно найти в подкаталоге, находящемся в том каталоге, из которого запускается этот сервис. Имя файла (и путь к нему) извлекается из запроса URL. Если путь пуст, то соответствующей переменной присваивается принятый по умолчанию служебный индексный файл.

```
          if (!file.exists() || file.isDirectory()) {
                response.setStatus(HTTPResponse::HTTP_NOT_FOUND);
                ostream& ostr = response.send();
                ostr << "File Not Found.";
```

```
            return;
        }

        string::size_type idx = path.rfind('.');
        string ext = "";
        if (idx != std::string::npos) {
            ext = path.substr(idx + 1);
        }

        string mime = "text/plain";
        if (ext == "html") { mime = "text/html"; }
        if (ext == "css") { mime = "text/css"; }
        else if (ext == "js") { mime = "application/javascript"; }
        else if (ext == "zip") { mime = "application/zip"; }
        else if (ext == "json") { mime = "application/json"; }
        else if (ext == "png") { mime = "image/png"; }
        else if (ext == "jpeg" || ext == "jpg") { mime = "image/jpeg"; }
        else if (ext == "gif") { mime = "image/gif"; }
        else if (ext == "svg") { mime = "image/svg"; }
```

Сначала проверяется корректность извлеченного пути к указанному файлу, а также тот факт, что он является обычным файлом, а не каталогом. Если результат проверки отрицательный, то возвращается ошибка HTTP 404 File Not Found.

После завершения проверки с положительным итогом выполняется попытка извлечения расширения файла из его путевого имени для определения специального типа MIME для этого файла. Если попытка неудачна, то по умолчанию используется тип MIME для обычного текста (plain text).

```
        try {
            response.sendFile(file.path(), mime);
        }
        catch (FileNotFoundException &e) {
            Club::log(LOG_ERROR, "DataHandler: File not found exception triggered...");
            cerr << e.displayText() << endl;

            response.setStatus(HTTPResponse::HTTP_NOT_FOUND);
            ostream& ostr = response.send();
            ostr << "File Not Found.";
            return;
        }
        catch (OpenFileException &e) {
            Club::log(LOG_ERROR, "DataHandler: Open file exception triggered: " +
                                 e.displayText());
            response.setStatus(HTTPResponse::HTTP_INTERNAL_SERVER_ERROR);
            ostream& ostr = response.send();
            ostr << "Internal Server Error. Couldn't open file.";
            return;
        }
    }
};
```

На завершающем этапе используется метод `sendFile()` объекта ответа для отправки файла клиенту вместе с типом MIME, определенным выше.

Кроме того, обрабатываются два исключения, которые может сгенерировать этот метод. Первое исключение возникает, если файл не может быть найден по какой-либо причине. В результате возвращается еще одна ошибка HTTP 404.

Если файл по какой-либо причине невозможно открыть, то возвращается внутренняя ошибка сервера HTTP 500 Internal Server Error, сопровождаемая текстом, определенным в обработчике этого исключения.

Конфигурация сервиса

В дистрибутиве Raspbian Linux для одноплатных компьютеров Raspberry Pi системные сервисы обычно управляются системным механизмом `systemd`. Для этого используется простой файл конфигурации, который в нашем примере мониторинга клубного зала выглядит приблизительно так:

```
[Unit]
Description=ClubStatus monitoring & control

[Service]
ExecStart=/home/user/clubstatus/clubstatus /home/user/clubstatus/config.ini
User=user
WorkingDirectory=/home/user/clubstatus
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Эта конфигурация сервиса определяет имя сервиса и тот факт, что сервис должен запускаться из каталога учетной записи пользователя `user`, а файл конфигурации для сервиса находится в том же каталоге. Устанавливается рабочий каталог для этого сервиса, также разрешается его автоматический перезапуск по истечении интервала в пять секунд, если по какой-либо причине сервис был остановлен.

Сервис инициализируется при запуске системы в тот момент, когда пользователь `user` получает возможность зарегистрироваться в системе (войти в систему). Таким образом, мы гарантируем, что сетевая и прочая функциональность уже будет приведена в полную готовность до запуска нашего сервиса. Если запустить сервис слишком рано, то возможен критический сбой из-за отсутствия требуемой функциональности системных сервисов, которые еще не были инициализированы.

Ниже показано содержимое INI-файла конфигурации:

```
[MQTT]
; URL и порт сервера MQTT
host = localhost
port = 1883

; Аутентификация
user = user
pass = password

; Тема состояния, в которой будут публиковаться изменения
clubStatusTopic = /my/topic
```

```
[HTTP]
port = 8080

[Relay]
; Подключена ли плата реле i2c. 0 (false) или 1 (true).
active = 0
; Адрес на шине i2c в десятичном или шестнадцатеричном формате
address = 0x20
```

Файл конфигурации разделен на три секции: MQTT, HTTP и Relay. В каждой секции определяются соответствующие переменные.

Для протокола MQTT назначаются предполагаемые параметры для установления соединения с MQTT-брокером, в том числе параметры аутентификации на основе пароля. Также создается тема, в которой будут публиковаться сообщения об обновлениях состояния клуба.

Секция HTTP содержит только номер прослушиваемого порта, а все прочие интерфейсы принимаются по умолчанию. При необходимости можно было бы создать сетевой интерфейс и соответствующую конфигурацию, а также обеспечить конфигурирование этой функции перед запуском сервера HTTP.

Последняя секция Relay позволяет включать и отключать функции платы реле, а также определяет адрес устройства на шине I2C, если эта функция используется.

Права доступа

Поскольку интерфейс GPIO и шина I2C интерпретируются как обычные устройства в системе Linux, они должны обладать собственным набором прав доступа к ним. Предполагая, что следует исключить запуск этого сервиса с правами суперпользователя root, необходимо добавить в систему учетную запись, от имени которой будет запускаться сервис с учетом соответствующих пользовательских групп `gpio` и `i2c`:

```
sudo usermod -a -G gpio user
sudo usermod -a -G i2c user
```

После этого необходимо перезапустить систему (или выйти из нее и снова зарегистрироваться), для того чтобы внесенные изменения вступили в силу. Теперь запуск сервиса должен выполняться без каких-либо проблем.

Окончательные результаты

После полного завершения процедуры конфигурирования и установки приложения и системного сервиса systemd на целевой одноплатный компьютер сервис, рассматриваемый в данном примере, будет автоматически запускаться и конфигурироваться. Чтобы завершить создание полноценной системы, можно установить в комплекте с подходящим источником электропитания в специальном коробе, в котором также будут проложены сигнальные провода от переключателей, сетевой кабель и т. п.

Одна из реализаций описанной здесь системы была установлена в хакерском клубе Entropia в городе Карлсруэ (Германия). В этом комплекте использовался настоящий светофор (приобретенный законно), установленный снаружи около

входной двери клуба. Для сигналов о состоянии в светофоре применялись 12-вольтовые LED-индикаторы. Одноплатный компьютер, плата реле, плата устранения дребезга контактов и источник электропитания (5 В и 12 В промышленный блок питания MeanWell) были размещены в едином деревянном корпусе, изготовленном методом лазерной резки (рис. 3.10).

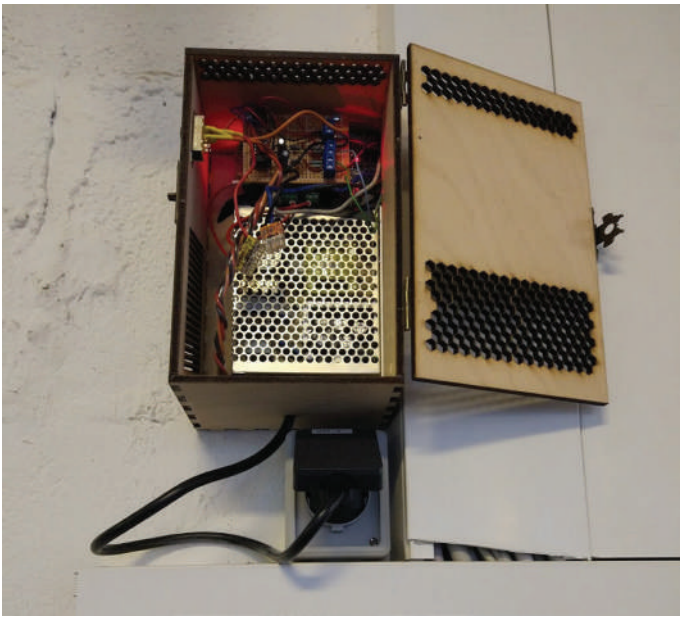

Рис. 3.10

Вы можете самостоятельно выбрать схему компоновки и размещения компонентов системы по своему усмотрению. Здесь главное – учитывать, что электронные компоненты должны быть надежно защищены от повреждений и случайных контактов, так как плата реле может срабатывать при изменении напряжения в электросети, возможно, при непосредственной близости проводки для блока питания.

Пример: простой медиа-плеер

Еще одним простым примером встроенной системы на основе одноплатного компьютера является медиаплеер, который может поддерживать функции обработки как аудио, так и аудиовизуальных (AV) форматов. Различие между системой на основе одноплатного компьютера, использующей для управления воспроизведением медиа ввод с обычной клавиатуры и мыши, и встроенным медиаплеером на основе одноплатного компьютера заключается в том, что в последнем случае система может использоваться только для этой единственной цели, а программное обеспечение и пользовательский интерфейс (как физический, так и программный) оптимизированы для применения в медиаплеере.

С этой целью должно быть разработано ПО внешнего звена вместе с физическим интерфейсом периферийных устройств, необходимых для обеспечения возможности управления медиаплеером. Это могут быть простые устройства, такие как группа переключателей, соединенных с контактами интерфейса GPIO, и обычный дисплей высокой четкости (HDMI) для вывода. Но можно воспользоваться и сенсорным экраном, хотя это потребует более сложной настройки драйвера.

Так как подобная система медиаплеера хранит файлы локально, необходимо использовать одноплатный компьютер, который поддерживает не только карту твердотельного (SD) диска, но и другие устройства хранения данных. На некоторых одноплатных компьютерах имеется разъем SATA, позволяющий подключить жесткий диск (HDD) с емкостью, значительно превышающей емкость SD-карт. Даже если подключить один из компактных жестких дисков с форм-фактором 2,5 дюйма, которые имеют приблизительно такие же размеры, как большинство распространенных одноплатных компьютеров, можно относительно дешево и просто получить в свое распоряжение несколько терабайтов для хранения медиафайлов.

Помимо требуемых устройств хранения данных, также необходимо наличие выхода цифрового видеосигнала, а кроме того, придется воспользоваться интерфейсом GPIO или USB-гнездами для кнопок пользовательского интерфейса.

Наиболее подходящей для этой цели является плата LeMaker Banana Pro, на которой размещена система на кристалле (SoC) H3 ARM, аппаратный разъем интерфейса SATA, поддержка сетевого соединения Gigabit Ethernet, а также полноразмерный выходной интерфейс HDMI с поддержкой декодирования 4K видео (рис. 3.11).

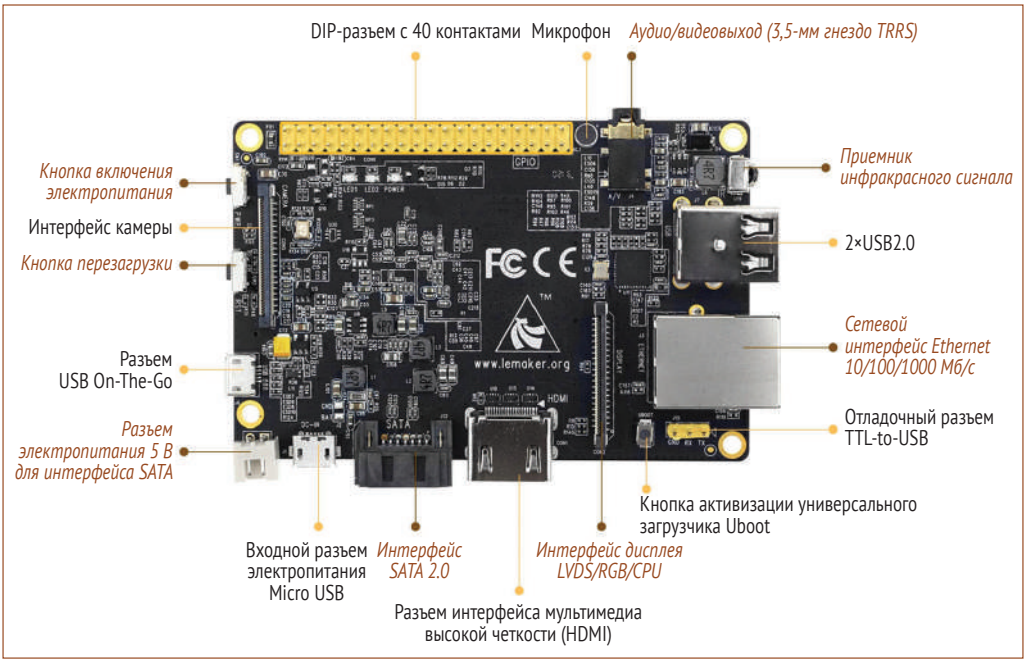

Рис. 3.11

После предварительной установки Armbian или аналогичных ОС на этот одноплатный компьютер можно начать установку и настройку приложения медиаплеера в системе, обеспечив его запуск вместе с запуском операционной системы и конфигурирование для загрузки плей-листа и отслеживания (прослушивания) событий на заданных контактах интерфейса GPIO. Эти контакты инфтерфейса GPIO должны быть соединены с соответствующими управляющими переключателями, позволяющими пользователю прокручивать плей-лист, запускать, останавливать и ставить на паузу воспроизведение элементов плей-листа.

Возможны и другие методы взаимодействия, например пульт управления с инфракрасным излучателем или на основе радиосигнала, каждый из которых имеет собственные достоинства и недостатки.

Мы продолжим работу по созданию такой системы медиаплеера и по превращению ее в информационно-развлекательную систему в следующих главах:

- главе 6 «Тестирование приложений, предназначенных для конкретных ОС»;
- главе 8 «Пример: информационно-развлекательная система на основе ОС Linux»;
- главе 10 «Разработка встроенных систем с использованием Qt».

Резюме

В этой главе рассматривались встроенные системы на основе специализированных ОС с описанием нескольких общедоступных операционных систем и наиболее существенных отличительных свойств, особенно характерных для операционных систем реального времени. Были описаны методы интегрирования периферийных устройств на примере часов реального времени в систему на основе одноплатного компьютера под управлением ОС Linux, методы использования модулей драйверов из пространства пользователя и пространства ядра, а также преимущества и недостатки этих методов.

На примере, приведенном во второй части данной главы, читатель получил достаточно глубокое представление о том, как преобразовать набор требований в полнофункциональную встроенную систему на основе ОС. Кроме того, читатель узнал, как в систему добавляются внешние периферийные устройства и как они используются с помощью средств ОС.

В следующей главе рассматривается разработка для встроенных систем с ограниченными ресурсами, в том числе для 8-битовых микроконтроллеров и для более продвинутых представителей этого семейства.

Глава 4

Встроенные системы с ограниченными ресурсами

Использование малых встроенных систем, таких как микроконтроллеры, означает работу с малым размером оперативной памяти (ОЗУ), с низкой мощностью центрального процессора и с малым объемом внешней памяти. В этой главе рассматриваются методы планирования и эффективного использования ограниченных ресурсов с учетом широкого диапазона доступных в настоящее время решений на основе микроконтроллеров и систем на кристалле (SoC), а именно следующие темы:

- выбор наиболее подходящего микроконтроллера для проекта;
- параллельный режим и управление памятью;
- добавление сенсоров, исполнительных устройств (актуаторов) и устройств доступа к сети;
- сравнение разработки непосредственно для аппаратуры с разработкой для ОС реального времени.

Общий обзор применения малых систем

Когда начинается разработка нового проекта, требующего использования по меньшей мере одного типа микроконтроллеров, задача выбора кажется невероятно сложной. В главе 1 мы убедились в наличии огромного выбора различных микроконтроллеров, даже если ограничиться только версиями и моделями, выпущенными в последнее время.

Вероятнее всего, начинать следует с определения требований к разрядности в битах, то есть выбрать 8-битовые, 16-битовые или 32-битовые микроконтроллеры либо учитывать количественные характеристики, такие как скорость таймера. Но метрики подобного рода иногда запутывают дело и часто не позволяют сузить диапазон при выборе конкретного продукта. Тогда основными определяющими категориями становятся доступность средств ввода/вывода и интегрируемых периферийных устройств, обеспечивающих удобную и надежную работу аппаратуры, и мощность средств обработки данных, соответствующая требованиям, определяемым в процессе проектирования и прогнозируемым для обеспечения производительности на протяжении всего жизненного цикла проектируемого продукта.

Необходимо получить более подробные ответы на следующие вопросы:

- периферийные устройства – какие периферийные устройства необходимы для взаимодействия с остальной системой?
- ЦПУ – какой уровень мощности ЦПУ необходим для выполнения кода приложения?
- операции с плавающей точкой – требуется ли аппаратное оборудование, поддерживающее выполнение операций с плавающей точкой?
- ПЗУ – какой объем ПЗУ необходим для хранения кода?
- ОЗУ – какой объем ОЗУ требуется для выполнения кода?
- электропитание и температурный режим – каковы требования к электропитанию и к соблюдению ограничений по температурному режиму?

Каждому семейству микроконтроллеров присущи свои сильные и слабые стороны, но одним из наиболее важных факторов при выборе одного из семейств микроконтроллеров является качество инструментальных средств разработки, ориентированных на конкретное семейство. В неформальных (разрабатываемых как хобби) и прочих некоммерческих проектах, возможно, в первую очередь оценивается масштаб сообщества и доступность свободных (бесплатных) средств разработки. В коммерческих проектах рассматривается степень и качество поддержки, ожидаемой от производителя микроконтроллера и, возможно, от третьих сторон.

Ключевым аспектом разработки встроенных систем является программирование и отладка непосредственно в разрабатываемой системе. Поскольку программирование и отладка связаны теснейшим образом, мы будем рассматривать соответствующие возможности интерфейса немного позже, чтобы можно было определить их соответствие нашим требованиям и ограничениям.

Широко распространенным мощным интерфейсом отладки фактически стал низкоуровневый стандарт IEEE 1149.1 Joint Test Action Group (JTAG), легко распознаваемый по сигналам, часто обозначаемым аббревиатурами TDI, TDO, TCK, TMS и TRST, определяющими соответственно именуемый Test Action Port (TAP). Стандарт постепенно расширялся и усовершенствовался до 1149.8, но не все версии применимы для цифровой логики, поэтому мы ограничимся использованием стандарта 1149.1 и сокращенной версии распределения контактов, описанной в стандарте 1149.7. Для текущих целей требуется лишь поддержка одного из полнофункциональных интерфейсов JTAG, SWD и UPDI.

При отладке систем на основе микроконтроллеров, в том числе и при отладке непосредственно на микросхеме, используются инструменты командной строки и интегрированные среды разработки (IDE), но эта тема будет более подробно рассматриваться в главе 7 «Тестирование платформ с ограниченными ресурсами».

Наконец, если планируется постоянный выпуск продукции, содержащей выбранную модель (или модели) микроконтроллера, в течение нескольких лет, чрезвычайно важно убедиться в доступности этого микроконтроллера (или совместимых с ним взаимозаменяемых моделей), по крайней мере, в течение планируемого периода. Производители с надежной репутацией предоставляют информацию о жизненном цикле своих продуктов как часть своей программы управления потоком производства и поставок. Оповещения публикуются заранее, за 1–2 года до фактического прекращения производства и поставок конкретной продукции, а кроме того, даются рекомендации по закупкам в течение оставшегося времени.

Для многих приложений практически невозможно не принять во внимание широкую распространенность и доступность той или иной микросхемы, мощность и простоту использования плат, совместимых с Arduino, пользующихся особой популярностью при проектировании на основе семейства микроконтроллеров AVR. Среди них можно выделить микропроцессоры ATmega – варианты mega168/328 и особенно mega1280/2560, – предоставляющие значительный объем мощностей для обработки данных, ПЗУ и ОЗУ для обеспечения функциональности высокого уровня и обработки входных данных, средства управления, контроля и телеметрии, а также разнообразный, но богатый набор периферийных устройств и интерфейсов GPIO.

Все эти аспекты обеспечивают чрезвычайную простоту прототипирования до перехода к какому-либо более определенному варианту при минимальных конкретных требованиях к характеристикам и (как можно надеяться) при более выгодной стоимости по ведомости основных материалов (BOM). Например, для такого прототипирования подходит плата ATmega2560 «MEGA», показанная на рис. 4.1, но мы будем подробно рассматривать и другие платы далее в этой главе вместе с рядом практических примеров разработки для платформы AVR.

Вообще говоря, рекомендуется выбрать несколько микроконтроллеров, которые могут использоваться в разрабатываемом проекте, определить платы разработки, установить их связь с остальными компонентами проектируемой системы (часто для этого требуется отдельная разработка или применение переходных плат – адаптеров) и начать разработку программного обеспечения для микроконтроллера, чтобы заставить все вышеперечисленное работать вместе.

Все больше и больше частей системы становятся полностью готовыми к эксплуатации, количество макетных плат и их основных компонентов будет сокращаться, пока не возникнет ситуация, в которой любой пользователь может начать работу с полностью завершенной компоновкой печатной платы (printed circuit board – PCB). Этот процесс пройдет множество итераций с устранением проблем, с добавлением функциональных возможностей в самый последний момент, наконец, система будет протестирована и оптимизирована как единое целое.

Микроконтроллеры в такой системе работают с аппаратными устройствами на физическом уровне, следовательно, требования часто определяются для объединенного комплекса аппаратного и программного обеспечения, потому что программное обеспечение взаимосвязано с функциональностью аппаратуры. Модульность аппаратных устройств – насущный вопрос в промышленности как в отношении небольших подключаемых печатных плат с минимальной дополнительной сложностью, добавляющих интерфейсы сенсорных датчиков или обмена данными в устройства, например контроллеры температуры и частотно-регулируемые приводы, так и в отношении полнофункциональных модулей с DIN-соединениями с общей последовательной шиной.

Пример: устройство управления лазерным резаком

Одним из самых быстрых и точных способов резки разнообразных материалов является использование мощных лазеров. Стоимость углекислого газа (CO_2) постоянно и резко снижается со временем, и это привело к массовой доступности дешевых углекислотных лазерных резаков (рис. 4.2).

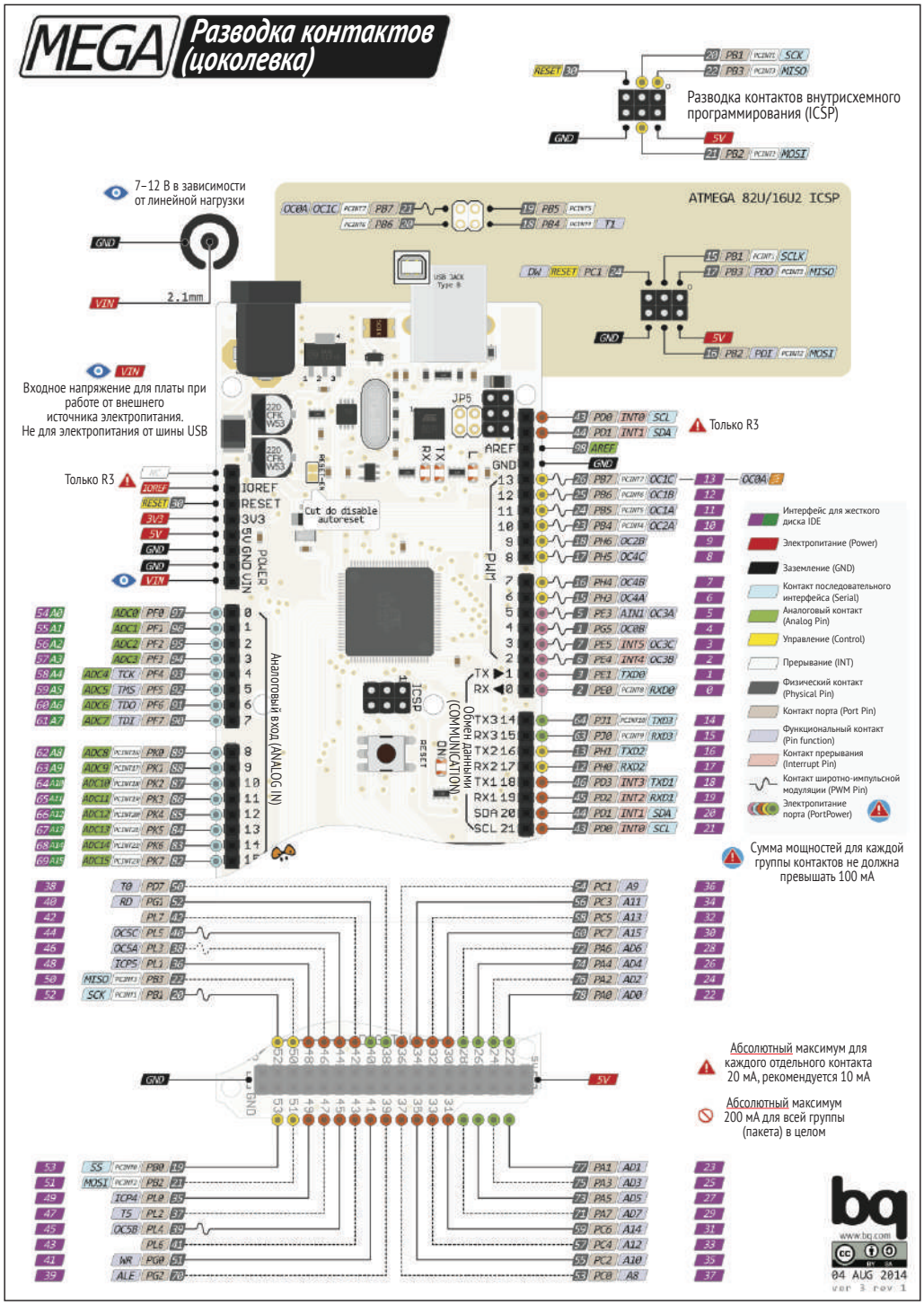
MEGA
Разводка контактов (цоколевка)
Разводка контактов внутрисхемного программирования (ICSP)
ATMEGA 82U/16U2 ICSP
7–12 В в зависимости от линейной нагрузки
2.1mm
Входное напряжение для платы при работе от внешнего источника электропитания. Не для электропитания от шины USB
Только R3
Cut do disable autoreset
Аналоговый вход (ANALOG IN)
Обмен данными (COMMUNICATION)
Интерфейс для жесткого диска IDE
Электропитание (Power)
Заземление (GND)
Контакт последовательного интерфейса (Serial)
Аналоговый контакт (Analog Pin)
Управление (Control)
Прерывание (INT)
Физический контакт (Physical Pin)
Контакт порта (Port Pin)
Функциональный контакт (Pin function)
Контакт прерывания (Interrupt Pin)
Контакт широтно-импульсной модуляции (PWM Pin)
Электропитание порта (PortPower)
Сумма мощностей для каждой группы контактов не должна превышать 100 мА
Абсолютный максимум для каждого отдельного контакта 20 мА, рекомендуется 10 мА
Абсолютный максимум 200 мА для всей группы (пакета) в целом
www.bq.com
04 AUG 2014
ver 3 rev 1

Рис. 4.1

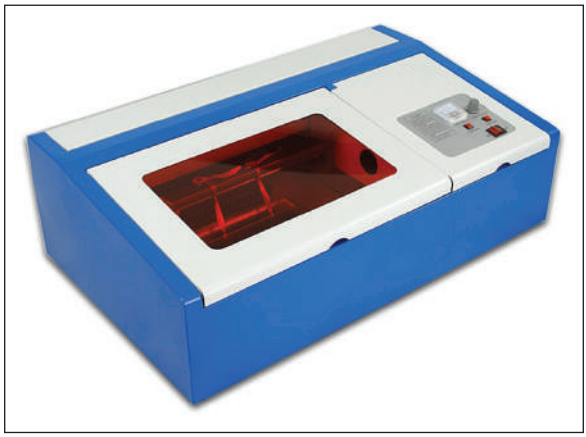

Рис. 4.2

С точки зрения удобства и безопасности использования можно работать только с основными простыми функциями лазерного резака и применять степпер, управляющий перемещением режущей головки по рабочей поверхности, но этого недостаточно. Кроме того, многие из дешевых лазерных резаков, имеющихся в общедоступной продаже, не обладают функциями, обеспечивающими безопасное и удобное использование.

Функциональная спецификация

Для создания полностью завершенного продукта необходимо добавить систему управления, которая использует сенсоры и исполнительные устройства (актуаторы) для наблюдения и управления состоянием машины, постоянно обеспечивая безопасность работы и отключение лазерного луча при необходимости. Это означает организацию защищенного доступа к каждой из трех частей, обозначенных на рис. 4.3.

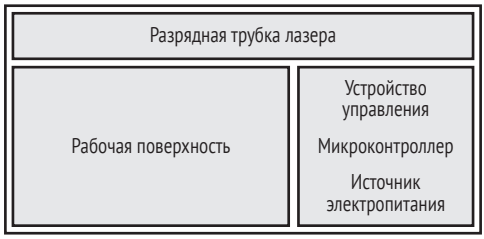

Рис. 4.3

Режущий луч обычно генерируется углекислотным лазером, то есть одним из типов газовых лазеров, изобретенных в 1964 году. Необходимо высокое напряжение и подача переменного тока, чтобы перевести молекулы газа в возбужденное состояние в канале разрядной трубки для создания активной режущей среды. В результате создается когерентный луч длинноволнового инфракрасного излу-

чения (long-wavelength infrared – LWIR или IR-C) с длиной волны 9.4 или 10.6 мкм.

Одна из характеристик длинноволнового инфракрасного излучения – его интенсивное поглощение многими материалами, поэтому его можно использовать для гравировки, резки и даже хирургических операций на тончайших тканях, поскольку вода в биологических тканях эффективно абсорбируется лучом лазера. Поэтому понятно, почему даже короткое облучение человеческой кожи лучом углекислотного лазера чрезвычайно опасно.

Для обеспечения безопасности излучение лазера должно быть локализовано в изолированном закрытом корпусе во время обычной рабочей операции. Кроме того, необходимо обеспечить отключение источника электропитания лазера и закрытие затвора луча, а лучше – сочетание обеих этих мер при отказе любой изолирующей блокировки или при нарушении какого-либо другого условия обеспечения безопасности.

Например, должны поддерживаться определенные температурные режимы: большинство углекислотных лазеров содержат газоразрядную трубку с водяным охлаждением, которая может быстро лопнуть или изогнуться при нарушении режима охлаждения. Более того, в процессе резки возникают раздражающие или ядовитые испарения, которые необходимо постоянно удалять из рабочего корпуса, чтобы они не загрязняли оптические устройства и не попали в окружающую среду после открытия крышки корпуса.

Эти требования определяют необходимость постоянного наблюдения за охлаждающим водяным потоком и за температурой, а также за воздушным вентилирующим потоком и за (аэродинамическим) сопротивлением воздушному потоку на вентиляционном фильтре (перепад давления (аэродинамическое сопротивление) с учетом массового расхода воздуха).

Наконец, следует также обеспечить удобство использования лазерного резака и избежать необходимости «применения устройства собственного изготовления» в процессе специализированного для этой конкретной задачи проектного решения с последующим его преобразованием и загрузкой на плату управления перемещением степпера через интерфейс USB. Вместо этого нужно загрузить проектное решение с карты твердотельного SD-диска или с USB-устройства и использовать простой LCD-монитор и кнопки для установки параметров и опций.

Проектные требования

С учетом требований, определенных в предыдущем разделе, можно сформировать список функциональных характеристик для системы управления:

- безопасность оператора:
  - переключатели взаимоблокировок на панелях доступа (закрыты, когда панель закрыта);
  - механизм блокировки (механическая блокировка панели доступа; избыточное, дополнительное требование);
  - экстренная остановка;
- охлаждение лазера:
  - реле насоса (подкачки);
  - сенсорный датчик температуры в резервуаре для воды (мощность охлаждения, температура на впускном клапане);

  - сенсорный датчик температуры охлаждения вытяжного клапана (температура кожуха (внешнего корпуса));
  - сенсорный датчик потока (скорость водяного потока; избыточное, дополнительное требование);
- воздушная вытяжка:
  - реле вентилятора;
  - состояние воздушного фильтра (сенсорный датчик перепада давления);
  - скорость (вращения) вентилятора (число оборотов в минуту);
- модуль лазера:
  - реле регулирования мощности лазера;
  - затвор луча лазера (избыточное, дополнительное требование);
- пользовательский интерфейс:
  - индикаторы экстренного оповещения:
  - панель взаимоблокировок;
  - состояние воздушного фильтра;
  - состояние вентилятора;
  - состояние насоса подкачки;
  - температура воды;
  - светодиодные (LED-) индикаторы:
  - состояние готовности к работе;
  - запуск;
  - рабочий режим;
  - экстренная остановка;
  - режим охлаждения;
- обмен данными:
  - обмен данными через интерфейс USB с платой степпера (UART);
  - управление движением: генерация инструкций для мотора степпера;
  - считывание файлов с карты твердотельного SD-диска/USB-накопителя;
  - доступ к файлам через Ethernet/Wi-Fi;
  - коммуникатор ближней бесконтактной связи (NFC) для идентификации пользователей.

Варианты выбора реализации

В начале этой главы было отмечено, что микроконтроллеры средней категории в настоящее время способны предоставить ресурсы для обеспечения большинства или даже всех перечисленных выше проектных требований. Один из самых важных вопросов: на что будут потрачены наши деньги – на аппаратные компоненты или на разработку программного обеспечения? Отбросив все несущественные детали, рассмотрим подробнее три возможных варианта решения:

- одна плата микропроцессора AVR средней категории (ATmega2560);
- плата микропроцессора Cortex-M3 более высокой категории (SAM3X8E);
- комбинация платы микропроцессора средней категории и одноплатного компьютера с операционной системой.

Проектным требованиям достаточно точно соответствует вариант с использованием платы Arduino Mega (ATmega2560), так как требования в первых пяти разделах в минимальной степени зависят от скорости центрального процессора.

Основным критерием является количество цифровых контактов ввода и вывода и нескольких аналоговых контактов в зависимости от конкретных сенсорных устройств, которые будут применяться, или, в большей мере, от используемых интерфейсов периферийных устройств (например, для сенсорных датчиков давления микроэлектромеханических систем (MEMS)).

Трудности начинаются при обеспечении функции управления движением с учетом взаимодействия (и обмена данными) с предшествующими ей в списке функциональными характеристиками, когда вдруг возникает необходимость преобразования файла масштабируемой векторной графики (.svg) в последовательность команд степпера. Это комплексная проблема передачи данных, синтаксического разбора (парсинга) файла, генерации файлового пути и того, что в сфере робототехники называется инверсной кинематикой. Обмен данными через USB также может стать проблематичным при выборе 8-битового микроконтроллера, главным образом потому, что периоды пиковой нагрузки процессора совпадают с тайм-аутами для конечных точек обмена данными с USB-устройствами или для обработки регистра RX-буфера UART.

Ключ к решению – точное знание того, когда следует менять режим работы механизмов. Для управления движением фактор времени является наиважнейшим, поскольку управление движением связано с инерцией в реальном физическом мире. Кроме того, существуют ограничения в плане обработки и пропускной способности (производительности) ресурсов выбранного контроллера для реализации управления и передачи данных, буферизации и в конечном итоге собственно для обработки и генерации выходных данных. В качестве общего шаблона более продвинутые внутренние или внешние периферийные устройства могут смягчить требования по времени для обработки событий и транзакций оперативной памяти, сокращая издержки на переключение контекста и обработку. Ниже приведен неполный список таких условий:

- простой универсальный асинхронный приемопередатчик (UART) требует получения и сохранения каждого байта через RX Complete (RXC). Сбой в этом процессе приводит к потере данных, о чем сообщает флаг DOR. Некоторые контроллеры, такие как семейство ATmega8u2 – ATmega32u4, обеспечивают встроенное аппаратное управление потоком через линии RTS/CTS, которое может регулировать отправку данных преобразователями USB-UART, такими как PL2303 и FT232, заставляя их выполнять буферизацию данных до тех пор, пока универсальный блок цифровой индикации (UDR) снова не станет пустым;
- специализированный USB-хост периферийных устройств, такой как MAX3421, подключается через последовательный интерфейс SPI и эффективно исключает требования по времени для USB при интеграции устройства массовой памяти;
- буферизация внешней схемы UART, периферийных устройств сетевого обмена данными, по существу, выполняется в программном обеспечении из-за сложности стека соответствующего уровня. Для Ethernet вполне подходящим решением является W5500;
- иногда имеет смысл добавить еще один менее мощный микроконтроллер, который независимо обрабатывает ввод/вывод и выполняет генерацию шаблона при реализации интерфейса по нашему выбору, – например, по-

следовательный или параллельный. Это уже предусмотрено в случае применения некоторых плат Arduino, оснащенных ATmega16u2 для последовательного преобразования USB.

Требование наличия коммуникатора ближней бесконтактной связи (NFC – near-field communication; это подмножество радиочастотной идентификации RFID – radio frequency identification) необходимо для предотвращения неавторизованного использования лазерного резака. Это создает самую большую нагрузку, но не из-за обмена данными с самим NFC-коммуникатором, а из-за увеличения размера кода и дополнительных требований к производительности ЦПУ для криптографической обработки сертификатов в зависимости от выбранного уровня защиты. Также необходимо обеспечить защищенное место хранения сертификатов, что обычно повышает уровень требований к характеристикам микроконтроллера.

Теперь можно рассмотреть более продвинутые возможности. Более простой микропроцессор ATmega2560 остается удачным вариантом с учетом наличия большого количества контактов GPIO и возможности чтения SD-карт через интерфейс SPI, а также с учетом возможности обмена данными с внешней интегрируемой микросхемой Ethernet. Но задачи управления движением с большим объемом вычислений и интенсивным использованием памяти, а также список функций NFC-коммуникатора, вероятнее всего, приведут к перегрузке этого микроконтроллера или к чрезмерно сложным «оптимизированным» решениям с последующим ухудшением качества сопровождения при попытках их реализации.

Замена микроконтроллера на ARM Cortex-M3, как это сделано на макетной плате Arduino Due, с большой вероятностью устранит все вышеперечисленные узкие места и проблемы. При этом сохраняется большое количество контактов GPIO, как и в варианте ATmega2560, но существенно увеличивается производительность центрального процессора. Шаблоны управления степпером могут генерироваться непосредственно микроконтроллером, который также обеспечивает встроенную поддержку USB вместе с другими продвинутыми периферийными устройствами/интерфейсами (USART, SPI, I2C и HSCMI), а кроме того, поддерживает прямой доступ к памяти (direct memory access – DMA).

Простой NFC-коммуникатор-считыватель может подключаться через UART, SPI или I2C. При таком выборе проектного решения формируется система, общая схема которой показана на рис. 4.4.

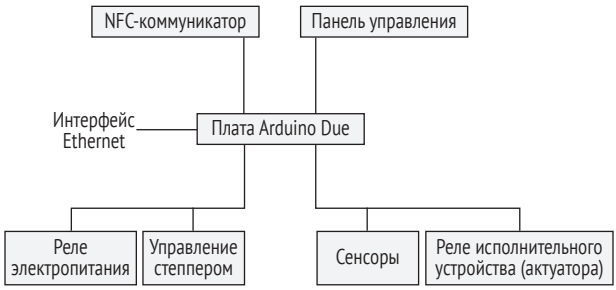

Рис. 4.4

В третьем варианте реализации с применением одноплатного компьютера снова используется микроконтроллер ATmega2560 в сочетании с одноплатным компьютером с низким энергопотреблением, управляемым операционной системой. Одноплатный компьютер должен выполнять любые задачи с интенсивным использованием ЦПУ, обеспечивать соединения по Ethernet и Wi-Fi, поддерживать USB (хост) и т. д. Кроме того, одноплатный компьютер должен обмениваться данными с платой микроконтроллера ATmega через UART, возможно, с добавлением цифрового изолятора или схемы сдвига уровня между платами для приведения в соответствие логических уровней 3,3 В (одноплатный компьютер) и 5 В TTL (ATmega).

Выбор варианта с объединением одноплатного компьютера и микроконтроллера значительно повышает сложность программного обеспечения, но лишь немного изменяет организацию системы с точки зрения аппаратных устройств. Общая схема системы показана на рис. 4.5.

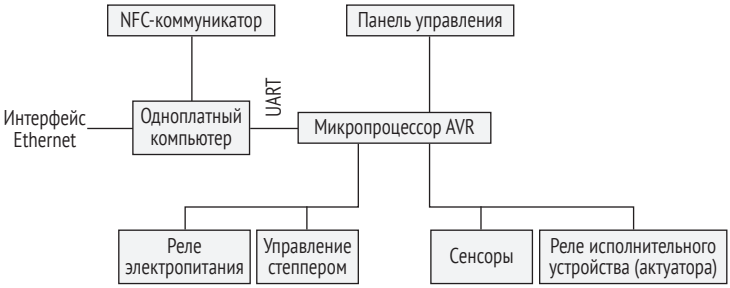

Рис. 4.5

Как и в большинстве процессов разработки, существует лишь несколько абсолютных ответов и множество возможных решений, в достаточной степени соответствующих функциональным требованиям после достижения компромиссов между энергопотреблением, сложностью и требованиями к удобству сопровождения, влияющих на выбор окончательного проектного решения.

В рассматриваемом здесь практическом примере можно выбрать решение на основе одного высокопроизводительного микроконтроллера или комбинацию из двух плат. Вероятнее всего, это потребует одинакового объема трудозатрат для выполнения требований. Одним из основных различий является то, что решение на основе ОС добавляет необходимость регулярного обновления версий операционной системы. Поэтому система должна иметь постоянное сетевое соединение и управляться полноценной ОС, тогда как встроенные контроллеры Ethernet с упрощенным аппаратным стеком TCP/IP и собственными устройствами памяти выглядят более надежными и проверенными.

Вариант на основе Cortex-M3 (или с более производительным микроконтроллером Cortex-M4) потребует написания специализированного программного кода, поэтому вряд ли возникнут какие-либо общие вопросы по безопасности, которые можно решить без затруднений. Проблема сопровождения не решена, но программный код будет достаточно компактным, то есть доступным для чтения и проверки в полном объеме. Единственный недостаток – отсутствие на плате

Arduino Due контактов RMII (независимого от среды передачи интерфейса) для подключения внешней интегральной схемы физического уровня Ethernet PHY, препятствующей использованию внутренней схемы Ethernet MAC.

В приведенном ниже списке объединены все соображения из предыдущей части главы, но теперь с учетом комбинации микроконтроллера ATmega2560 и одноплатного компьютера и соответствующего приложения обязанности распределяются следующим образом:

- периферийные устройства: на стороне микроконтроллера в основном необходим интерфейс GPIO, некоторые аналоговые (ADC) входные интерфейсы, Ethernet, USB, а также SPI и/или I2C;
- ЦПУ: требуемая производительность микроконтроллера критична по времени, но второстепенна, за исключением необходимости обработки вектора элементов пути для преобразования их в инструкции степпера. На стороне одноплатного компьютера может потребоваться интеллектуальная обработка, чтобы сформировать очередь из достаточного количества команд для выполнения на стороне микропроцессора, поэтому критичное по времени взаимодействие исключается;
- операции с плавающей точкой: алгоритм преобразования инструкций степпера в микроконтроллере выполняется значительно быстрее, если имеется аппаратная поддержка операций с плавающей точкой. Для значений размеров (длины) и времени можно обеспечить поддержку арифметики с фиксированной точкой, ослабляющую это требование;
- ПЗУ: полный код для микроконтроллера, вероятнее всего, потребует лишь нескольких килобайтов памяти, поскольку он не слишком сложен. Размер кода для одноплатного компьютера больше на несколько порядков из-за обращений к библиотекам более высокого уровня для обеспечения требуемой функциональности. В соответствующем масштабе увеличиваются и требования к объему памяти с произвольным доступом (массовой памяти) и мощностям обработки;
- ОЗУ: для микроконтроллера вполне достаточно нескольких килобайтов памяти типа SRAM. Алгоритм преобразования инструкций степпера может потребовать модификаций для соответствия ограничениям SRAM с учетом требований к буферизации и обработке данных. В наихудшей ситуации можно уменьшить размер буферов;
- электропитание и регулирование температуры: для системы электропитания лазерного резака и системы охлаждения не требуется больших затрат электроэнергии и не устанавливаются какие-либо жесткие ограничения температурного режима. Секция, содержащая систему управления, также включает основной источник электропитания и уже оборудована охлаждающим вентилятором соответствующего размера.

Здесь важно отметить, что хотя мы осознаем сложность задачи и требований к ее реализации в достаточной степени для того, чтобы сделать выводы, обеспечивающие выбор аппаратных компонентов, подробности достижения этих целей остаются прерогативой разработчика программного обеспечения.

Например, можно было бы определить собственные структуры данных и форматы и реализовать ориентированную на оборудование генерацию пути и управ-

ление движением или адаптировать промежуточный формат G-code (RS-274), апробированный и широко применяемый в течение нескольких десятилетий в приложениях ЧПУ, хорошо подходящий для генерации команд управления движением. G-code широко распространен при работе с аппаратурой, особенно при трехмерной печати FDM 3D.

Заслуживает внимания полноценная реализация управления движением на основе G-code с открытым исходным кодом – GRBL, которую лучше представят ее авторы:

> *«Grbl – бесплатное высокопроизводительное программное обеспечение с открытым исходным кодом для управления движением машин, которые перемещаются, делают вещи или перемещают вещи. Это ПО работает на обычной плате Arduino. Если движущиеся и движущие механизмы сформировали промышленность, то Grbl должен стать промышленным стандартом».*
>
> *– https://github.com/gnea/grbl*

Вероятно, следует добавить функции остановки и экстренной аварийной остановки при различных нарушениях режима безопасности. Отклонения от нормального температурного режима или загрязнение фильтра должны просто остановить работу лазерного резака и разрешить возобновление работы после устранения возникших проблем, но блокировка при открытии крышки корпуса непременно должна приводить к немедленному отключению луча лазера, даже без завершения выполнения текущей команды, определяющей перемещение по сегменту пути.

Выбор способа обеспечения модульности для задачи управления движением и генерация G-code для нее дает преимущества в плане доступности апробированных реализаций, позволяющих с легкостью добавлять полезные функции, такие как ручное управление для настройки и калибровки, а также для возможности тестирования с использованием предварительно сгенерированных, доступных для чтения человеком кодов на стороне машины в качестве мер по контролю и проверке выходных данных при интерпретации файла и алгоритмов генерации пути.

При наличии списка требований после завершения этапа начального проектирования и при глубоком понимании способов достижения установленных целей следующим шагом должно быть приобретение макетной платы (или нескольких плат) с выбранным микроконтроллером и/или системой на кристалле со всеми необходимыми периферийными устройствами, чтобы можно было приступить к разработке специализированного программного обеспечения и сборке системы.

Полная реализация системы управления машиной (механизмом), описанной в рассматриваемом здесь примере, выходит за рамки тематики данной книги, тем не менее глубокое понимание процесса разработки для различных вариаций целевого микроконтроллера и одноплатного компьютера потребуется в оставшейся части текущей главы, а также в главе 6 «Тестирование приложений, предназначенных для конкретных ОС», в главе 8 «Пример: информационно-развлекательная система на основе ОС Linux» и в главе 11 «Разработка для гибридных систем SoC/FPGA».

Интегрированные среды разработки и рабочие среды для встроенных систем

Разработка приложений для систем на кристалле становится все больше похожей на разработку для рабочих сред настольных компьютеров и серверов, но, как мы видели в предыдущей главе, разработка для микроконтроллера требует гораздо более глубоких знаний аппаратуры, для которой ведется разработка, иногда с точностью до отдельных битов, устанавливаемых в конкретном регистре.

Существует несколько рабочих сред (фреймворков – frameworks), пытающихся абстрагироваться от подробностей подобного рода для конкретных семейств микроконтроллеров, чтобы обеспечить возможность разработки для обобщенного API без излишнего беспокойства о том, как это будет реализовано для конкретного микроконтроллера. В этой группе рабочая среда Arduino является самым широко известным из свободно распространяемых промышленных приложений, хотя существует также множество коммерческих рабочих сред, сертифицированных для промышленной эксплуатации.

Рабочие среды, такие как Advanced Software Framework (ASF) для микроконтроллеров AVR и SAM, могут использоваться вместе с разнообразными интегрированными средами разработки (integrated development environment – IDE), включая Atmel Studio, Keil µVision и IAR Embedded Workbench.

Неполный список самых распространенных интегрированных сред разработки встроенных систем приведен в табл. 4.1.

Таблица 4.1

Наименование	Компания (производитель)	Лицензия	Поддерживаемые платформы	Примечания
Atmel Studio	Microchip	Проприетарная	AVR, SAM (ARM Cortex-M)	Изначально разработана компанией Atmel, затем приобретена компанией Microchip
µVision	Keil (ARM)	Проприетарная	ARM Cortex-M, 166, 8051, 251	Часть инструментального комплекта Microcontroller Development Kit (MDK)
Embedded Workbench	IAR	Проприетарная	ARM Cortex-M, 8051, MSP430, AVR, Coldfire, STM8, H8, SuperH, и т. д.	Отдельные интегрированные среды разработки для каждой архитектуры микроконтроллеров
MPLAB X	Microchip	Проприетарная	PIC, AVR	В качестве основы используется ориентированная на язык Java интегрированная среда разработки NetBeans
Arduino	Arduino	GPLv2	Некоторые микроконтроллеры AVR и SAM (поддержка постоянно расширяется)	Интегрированная среда разработки на основе языка Java. Поддерживает только собственный диалект языка C

Главная цель любой интегрированной среды разработки – включение полного рабочего потока в конкретное приложение: от написания исходного кода до программирования микроконтроллера и его памяти с использованием скомпилированного кода и отладки приложения во время его работы на целевой платформе.

Использование полнофункциональной интегрированной среды разработки в любом случае является вопросом личных предпочтений разработчика. Все основные функции остаются доступными и при работе в простом редакторе текста, и при использовании инструментов командной строки, хотя такие рабочие среды, как ASF, созданы для более глубокой интеграции с интегрированными средами разработки.

Одним из основных преимуществ широкоизвестной рабочей среды Arduino является наличие в ней более или менее стандартизированного API для разнообразных периферийных устройств микроконтроллеров и прочей функциональности, которая поддерживается для постоянно растущего числа архитектур микроконтроллеров. В совокупности с открытостью исходного кода рабочей среды это делает Arduino весьма привлекательным вариантом выбора для нового проекта. Особенно интересен вариант применения Arduino для прототипирования, поскольку для API этой рабочей среды написано множество библиотек и драйверов.

К сожалению, интегрированная среда разработки Arduino ориентирована исключительно на упрощенный диалект языка программирования C. В этом диалекте отсутствуют основные (стандартные) библиотеки, широко используемые в C++. Но есть возможность включения этих библиотек непосредственно в проекты встроенных систем на языке C++, как мы увидим далее в текущей главе.

Программирование микроконтроллеров

После завершения процедуры компиляции кода для целевого микроконтроллера бинарный образ должен быть записан в память микроконтроллера, чтобы стали возможными его выполнение и отладка. В этом разделе рассматриваются различные способы записи в память микроконтроллера. В настоящее время только промышленное программирование выполняется с тестовыми сокетами или в лучшем случае на уровне пластины кристалла до того, как заготовка микросхемы соединяется с рамкой с выводными контактами и заключается в корпус. При этом сразу предполагается наличие легко удаляемых и заменяемых внешних частей микроконтроллера для (многократного) программирования.

Количество опций (часто специализированных и зависящих от конкретного производителя), существующих для внутреннего программирования микросхемы, различается по используемым при этом периферийным устройствам и по воздействию на устройства памяти.

Таким образом, исходный «чистый» микроконтроллер часто требует программирования с применением внешнего программирующего адаптера. В основном это работа по настройке внешних контактов микроконтроллера с целью обеспечения входа в режим программирования, после чего микроконтроллер принимает поток данных, содержащий новый образ ПЗУ.

Другой широко используемый вариант – добавление в первую секцию ПЗУ загрузчика (boot loader), который позволяет микроконтроллеру программировать

себя. Загрузчик при запуске проверяет необходимость переключения в режим программирования или продолжения загрузки существующей программы, размещенной после секции загрузчика.

Программирование памяти и отладка устройства

Внешние программирующие адаптеры часто используют узкоспециализированные интерфейсы и связанные с ними протоколы, позволяющие программировать и отлаживать конкретное целевое устройство. Протоколы, обеспечивающие программирование микроконтроллеров, перечислены в табл. 4.2.

Таблица 4.2

Наименование	Число контактов	Функции	Описание
SPI (ISP)	4	Программирование	Последовательный интерфейс периферийных устройств (Serial Peripherial Interface – SPI), используемый для более старых микропроцессоров AVR для доступа к режиму последовательного программирования (In-circuit Serial Programming – ISP)
JTAG	5	Программирование, отладка, разграничение	Специализированный промышленный стандарт для поддержки программирования и отладки непосредственно на микросхеме. Поддерживается на устройствах AVR ATxmega
UPDI	1	Программирование, отладка	Универсальный интерфейс программирования и отладки (Unified Programming and Debug Interface – UPDI), используемый для более новых микроконтроллеров AVR, включая устройства ATtiny. Однопроводной интерфейс, предшественником которого был двухпроводной интерфейс PDI для устройств ATxmega
HVPP/HVSP	17/5	Программирование	Параллельное программирование устройств высокого напряжения (High Voltage Parallel Programming – HVPP) / Последовательное программирование устройств высокого напряжения (High Voltage Serial Programming – HVSP). Режим программирования AVR с использованием напряжения 12 В на контакте перезагрузки и прямой доступ к контактам 8+. Игнорирует любые настройки внутренних мостов (предохранителей) и прочие параметры и опции конфигурации. Используется в основном для промышленного программирования в фабричных условиях и для восстановления
TPI	3	Программирование	Специализированный интерфейс программирования Tiny Programming Interface (TPI), используемый для некоторых устройств ATtiny AVR. В этих устройствах слишком мало контактов для применения протоколов HVPP и HVSP

Таблица 4.2 *(окончание)*

Наименование	Число контактов	Функции	Описание
SWD	3	Программирование, отладка, разграничение	Последовательная шина отладки (Serial Wire Debug – SWD). Протокол аналогичен протоколу JTAG с сокращенным числом контактов на двух линиях, но использует функции ARM Debug Interface, позволяющие внешнему подключенному отладчику получить полное управление шиной с доступом к памяти и периферийным устройствам микроконтроллера

Микроконтроллеры ARM в основном поддерживают протокол JTAG как главное средство программирования и отладки. В 8-битовых микроконтроллерах протокол JTAG менее распространен, главным образом из-за сложности его требований.

В некоторых случаях микроконтроллеры AVR предлагают протокол программирования непосредственно в системе (In-System Programming – ISP) через последовательный интерфейс периферийных устройств (SPI) в дополнение к режимам программирования высокого напряжения. При входе в режим программирования требуется сохранение низкого уровня сигнала на контакте перезагрузки во время процедуры программирования и верификации. После завершения цикла программирования контакт перезагрузки разблокируется и стробируется.

Одно из требований протокола ISP – установка в микроконтроллере соответствующего режима (бит SPIEN fuse), позволяющего использовать интерфейс программирования непосредственно в системе. Если этот бит не установлен, то устройство не будет реагировать на сигналы по линиям SPI. Если протокол JTAG недоступен, но его использование разрешено посредством установки бита JTAGEN fuse, то для восстановления и перепрограммирования микросхемы доступны только протоколы HVPP и HVSP. В последнем случае неиспользуемый набор контактов и источник электропитания 12 В не обязательно интегрировать непосредственно в схему платы.

Физические соединения, требуемые для большинства последовательных интерфейсов программирования, достаточно просты, даже если микроконтроллер уже был интегрирован в электрическую схему, как показано на рис. 4.6.

Здесь внешний осциллятор не обязателен, если существует внутренний. Линии PDI, PDO и SCK соответствуют аналогичным линиям SPI. Линия перезагрузки (Reset) удерживается в активном состоянии (но с низким уровнем сигнала) во время процедуры программирования. После подключения к микроконтроллеру таким способом можно свободно выполнять запись в его флеш-память, EEPROM и конфигурационные мосты.

Для более новых устройств AVR предназначен универсальный интерфейс программирования и отладки (Unified Programming and Debug Interface – UPDI), использующий только одну линию (провод) (в дополнение к линиям электропитания и заземления) для соединения с целевым микроконтроллером и обеспечения поддержки режимов программирования и отладки.

Этот интерфейс упрощает предыдущую схему подключения к микроконтроллеру, как показано на рис. 4.7.

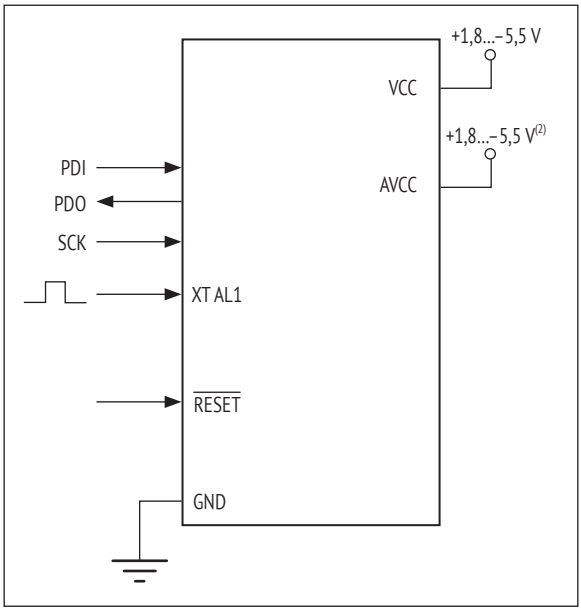

Рис. 4.6

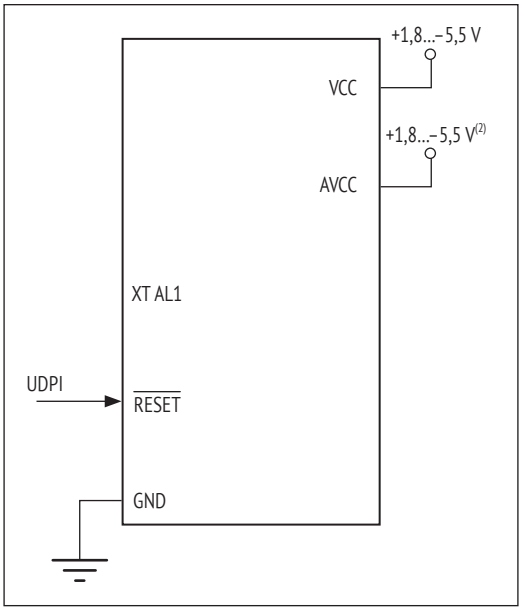

Рис. 4.7

Сравните со схемой подключения JTAG (IEEE 1149.1) для микроконтроллера ATxmega (если подключение разрешено), показанной на рис. 4.8.

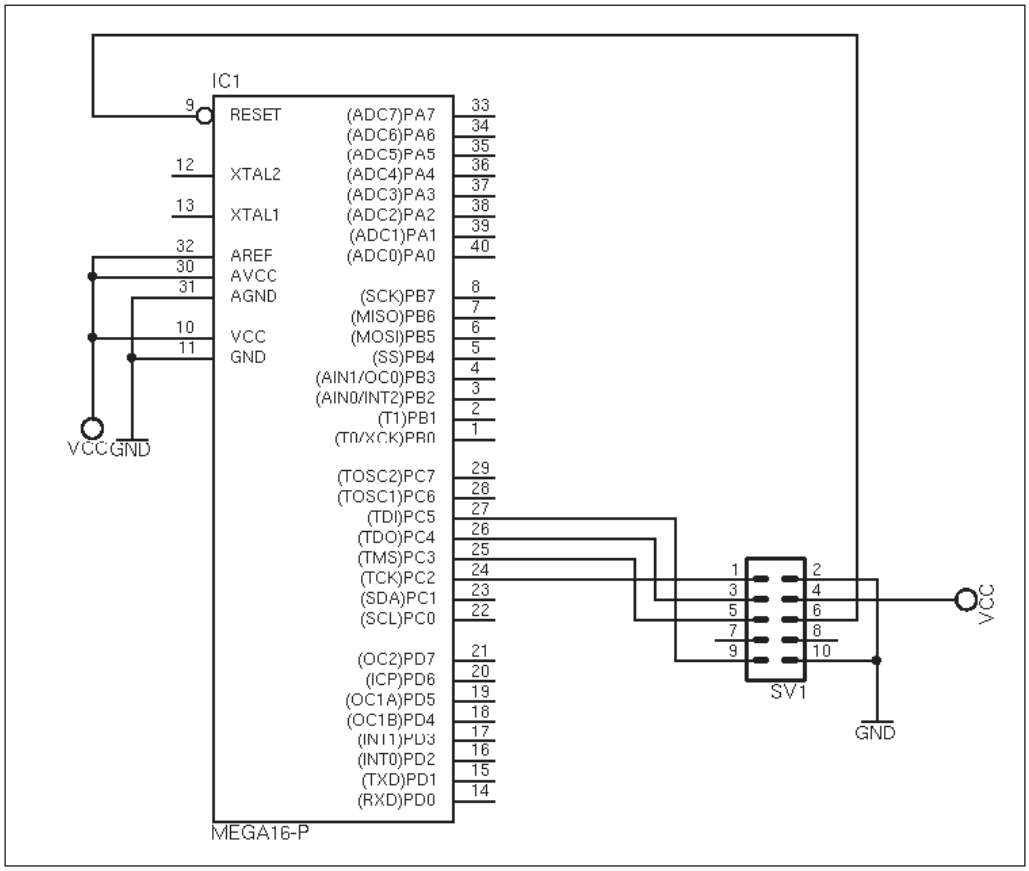

Рис. 4.8

Сокращенное количество контактов по стандарту JTAG (IEEE 1149), реализованное в микроконтроллерах ATxmega, требует наличия только одного таймера TCKC, одной линии данных TMSC и поэтому называется Compact JTAG. Из этих интерфейсов лишь в UPDI сохраняются требования наименьшего числа соединений с целевым устройством. Кроме этого различия, поддерживаемые обоими интерфейсами функции одинаковы для микроконтроллеров AVR.

Для других систем, использующих интерфейс JTAG для программирования и отладки, не существует стандартных схем подключения. Каждый производитель применяет собственный вариант реализации подключения: от 2×5 контактов (Altera, AVR) до 2×10 контактов (ARM) или единый 8-контактный коннектор (Lattice).

Поскольку JTAG в большей степени является стандартом протокола, чем спецификацией физического подключения, специфические подробности следует искать в документации на конкретную целевую платформу.

Загрузчик

Загрузчик (boot loader) был введен как небольшое вспомогательное приложение, использующее любой существующий интерфейс (например, UART или Ethernet) для предоставления возможности самопрограммирования. В микроконтроллерах AVR секция загрузчика размером от 256 байт до 4 Кб может быть зарезервирована во флеш-памяти устройства. Этот код может выполнять любое количество задач, определенных пользователем: от настройки последовательной линии связи с удаленной системой до загрузки из удаленного бинарного образа через интерфейс Ethernet с использованием среды начальной загрузки устройства PXE (Preboot eXecution Environment).

Как и ядро, загрузчик AVR не отличается от любого другого AVR-приложения, за исключением того, что при компиляции добавляется один дополнительный флаг для установки адреса начального байта бинарного образа загрузчика:

```
--section-start=.text=0x1800
```

Этот адрес заменяется на соответствующий адрес конкретного микроконтроллера, используемого в проекте (для AVR адрес зависит от настройки флагов BOOTSZ и применяемого контроллера; см. табл. спецификации по конфигурации загрузки Boot Size Configuration: Boot Reset Address, где, например, адрес рестарта загрузчика установлен равным 0xC00 в словах, а начало секции загрузчика определено в байтах). Это гарантирует, что код загрузчика будет записан в правильную локацию в ПЗУ микроконтроллера. Запись кода загрузчика в ПЗУ почти всегда выполняется по протоколу программирования непосредственно в системе (ISP).

В микропроцессорах AVR флеш-ПЗУ делится на две секции: без контроля (чтением) во время записи (no-read-while-write – NRWW) (для большей, если не всей части пространства памяти приложения) и с контролем (чтением) во время записи (read-while-write – RWW). Это означает, что секция RWW может быть стерта и перезаписана с соблюдением безопасности без какого-либо воздействия на операцию ЦПУ. Именно поэтому загрузчик размещается в секции NRWW, и поэтому не так-то просто обеспечить самообновление загрузчика.

Еще одна важная деталь – загрузчик к тому же не имеет возможности обновлять предохранительные мосты, которые устанавливают различные флаги в микроконтроллере. Для изменения этой ситуации требуется внешнее программирование устройства.

После программирования и ввода в микроконтроллер загрузчика в большинстве случаев необходимо установить флаги микроконтроллера, которые позволят процессору узнать о том, что загрузчик установлен. Для AVR это флаги BOOTSZ и BOOTRST.

Управление памятью

Система памяти (массовой и оперативной) микроконтроллеров состоит из множества компонентов. Существует секция постоянного запоминающего устройства (read-only memory – ROM), которая записывается только один раз во время про-

граммирования микросхемы и обычно не может быть изменена самим микроконтроллером, как мы видели в предыдущем разделе.

Кроме того, микроконтроллер может содержать бит постоянного устройства хранения данных в форме EEPROM или аналогичного устройства. Наконец, в микроконтроллере имеются регистры и оперативная память с произвольным доступом (random-access memory – RAM). В результате получается обобщенная схема организации памяти микроконтроллера, показанная на рис. 4.9.

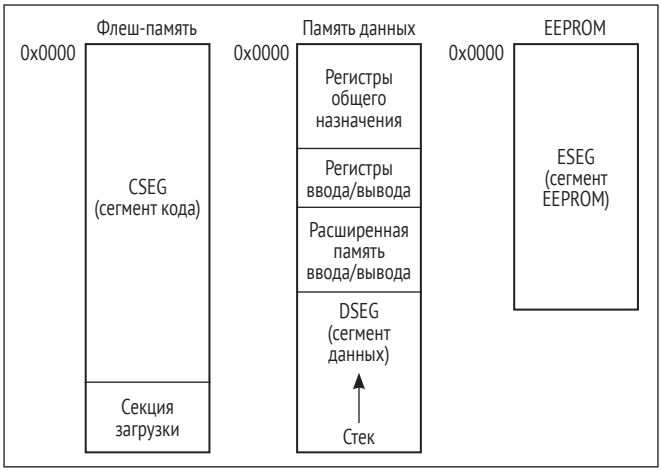

Рис. 4.9

В микроконтроллерах общепринятым решением является использование модифицированной гарвардской архитектуры (Harvard architecture) (разделение памяти программы и данных на одном архитектурном уровне, в общем случае с отдельными шинами данных). Например, в архитектуре AVR память программы расположена в ПЗУ, которое для ATmega2560 использует собственную шину соединения с ядром ЦПУ, как можно видеть на блок-схеме этого микроконтроллера в главе 1 (рис. 1.13).

Главным преимуществом наличия отдельных шин для различных пространств памяти является возможность раздельной адресации при обращении к этим пространствам, позволяющая более эффективно использовать ограниченное адресное пространство, доступное в 8-битовом процессоре (с размером адреса 1 и 2 байта). Кроме того, это позволяет организовать параллельный доступ к данным, пока ЦПУ занято в другом пространстве памяти, что улучшает оптимизацию использования доступных ресурсов.

При размещении памяти данных в SRAM в дальнейшем можно освободить ее для использования по своему усмотрению. Но при этом необходим, как мимимум, стек для нормальной работы программы. В зависимости от остающегося свободным объема памяти SRAM в микроконтроллере можно также добавить общедоступную память (heap – «кучу»). Приложения умеренной сложности можно реализовать только с использованием стека и статически распределенной памяти, не применяя функциональные возможности языка высокого уровня, которые генерируют код с динамическим распределением памяти из «кучи».

Стек и динамически распределяемая память

Необходимость организации стека в программируемом микроконтроллере зависит от уровня, на котором предполагается выполнение программирования. При использовании среды времени выполнения языка C (C-runtime) (в AVR: `avr-libc`) именно во время выполнения будет инициализирован стек и другие детали, если разрешить линкеру (редактору связей) разместить «чистый» (необработанный) код в секциях init, например, с помощью следующей директивы:

```
__attribute__ ((naked, used, section (".init3")))
```

Выполнение этой секции будет предшествовать выполнению любого кода приложения.

Стандартная схема распределения ОЗУ в микроконтроллере AVR начинается с сегмента переменных `.data` в самом начале ОЗУ, далее следует сегмент `.bss`. Стек начинается с противоположной стороны ОЗУ и растет к началу. Остается пространство между концом сегмента `.bss` и концом стека, как показано на рис. 4.10.

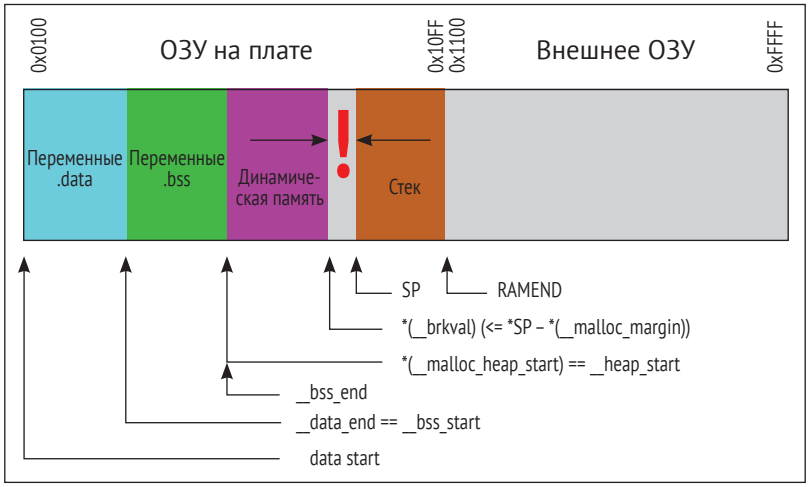

Рис. 4.10

Поскольку величина роста стека зависит от глубины вызовов функций в работающем приложении, трудно предсказать размер остающегося свободным доступного пространства. Некоторые микроконтроллеры позволяют использовать внешнюю дополнительную память ПЗУ, которую можно определить как возможную локацию для размещения динамической памяти («кучи»), как показано на рис. 4.11.

В библиотеке AVR Libc имеется реализация функции распределения динамической памяти `malloc()`, оптимизированной для архитектуры AVR. Используя эту функцию, при необходимости можно также реализовать функциональность собственных операторов new и delete, поскольку в комплекте инструментов AVR реализация этих операторов отсутствует.

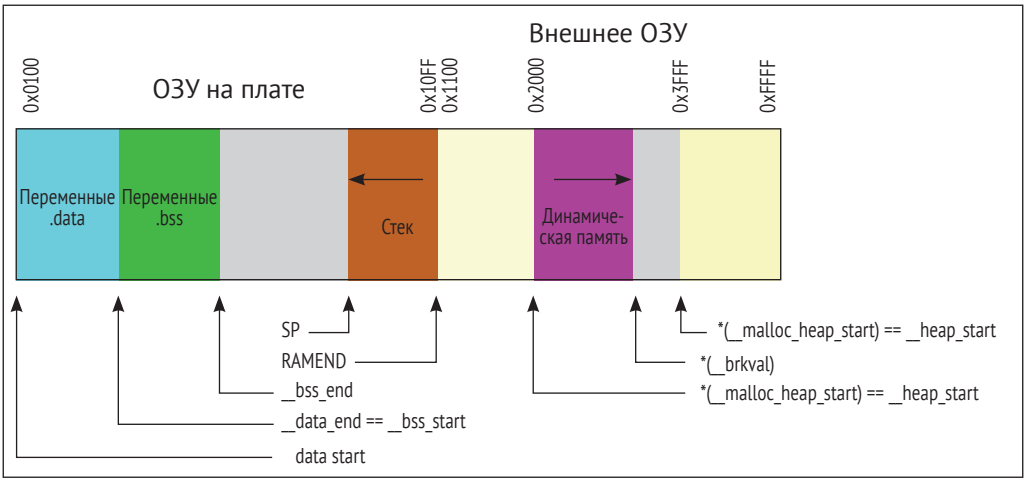

Рис. 4.11

Чтобы использовать внешнюю память микроконтроллера AVR для организации динамически распределяемой «кучи», необходимо обеспечить обязательную инициализацию внешней памяти, после которой соответствующее адресное пространство становится доступным функции `malloc()`. Начало и конец этого пространства динамически распределяемой памяти определяются следующими глобальными переменными:

```
char * __malloc_heap_start
char * __malloc_heap_end
```

В документации на AVR содержится следующая рекомендация по настройке динамически распределяемой памяти:

> *«Если динамически распределяемую память необходимо переместить во внешнее ОЗУ, то глобальная переменная* `__malloc_heap_end` *должна быть определена соответствующим образом. Это можно сделать либо во время выполнения, записав значение непосредственно в эту переменную, либо это происходит автоматически во время компиляции и связывания, посредством присваивания требуемого значения символу* `__heap_end`*».*

Прерывания, ESP8266 IRAM_ATTR

На обычном настольном компьютере или на сервере бинарный образ приложения должен быть полностью загружен в ОЗУ. Но в микроконтроллерах часто оставляют в ПЗУ как можно большее количество инструкций программы до тех пор, пока они не потребуются для выполнения. Это означает, что большинство инструкций приложения не могут быть выполнены немедленно, а сначала должны быть извлечены из ПЗУ, прежде чем ЦПУ микроконтроллера сможет получить их через шину инструкций для выполнения.

В микроконтроллерах AVR каждое возможное прерывание определяется в таблице вектора, записанной в ПЗУ. При этом предлагаются либо обработчики по умолчанию для каждого типа прерывания, либо версия, определенная пользователем. Для обозначения (маркировки) программы прерывания используется атрибут `__attribute__((signal))` или макро `ISR()`:

```
#include <avr/interrupt.h>

ISR(ADC_vect) {
        // пользовательский код
}
```

Эта макрокоманда обрабатывает детали регистрации прерывания. Необходимо лишь задать имя и определить функцию для обработчика прерывания. В дальнейшем функция обработчика будет вызываться через таблицу вектора прерываний.

В микроконтроллере ESP8266 (и в следующей модели этой линейки ESP32) можно пометить функцию обработки прерываний специальным атрибутом `IRAM_ATTR`. В отличие от AVR, микроконтроллер ESP8266 не имеет встроенного ПЗУ, но использует последовательный интерфейс периферийных устройств SPI для загрузки любых инструкций в ОЗУ. Очевидно, что это достаточно медленная процедура.

Пример использования этого атрибута для пометки обработчика прерываний приведен ниже:

```
void IRAM_ATTR MotionModule::interruptHandler() {
          int val = digitalRead(pin);
          if (val == HIGH) { motion = true; }
          else { motion = false; }
 }
```

Здесь показан обработчик прерываний, связанный с сигналом от детектора движения, подключенного к контакту ввода. Как и любой правильно написанный обработчик прерываний, он достаточно прост и предназначен для быстрого выполнения перед возвратом в обычный рабочий поток приложения.

Размещение данного обработчика в ПЗУ означает, что эта подпрограмма не будет почти мгновенно реагировать на изменение состояния вывода сенсора детектора движения. Еще хуже то, что обработчику требуется больше времени для завершения, следовательно, возникает задержка выполнения остального кода приложения.

Можно устранить описанную выше проблему, пометив функцию обработчика атрибутом `IRAM_ATTR`, тогда весь обработчик полностью будет находиться в ОЗУ, когда это необходимо. И всей системе не придется ждать возврата требуемых данных по шине SPI, прежде чем она сможет продолжить выполнение.

Может показаться, что этот тип атрибута следует использовать умеренно, так как большинство микроконтроллеров имеют больший объем ПЗУ, чем ОЗУ. Микроконтроллер ESP8266 имеет 64 Кб ОЗУ для выполнения кода с возможностью дополнения несколькими мегабайтами внешней флеш-памяти (ПЗУ).

При компиляции кода компилятор поместит инструкции, помеченные атрибутом `IRAM_ATTR`, в специальную секцию, поэтому микроконтроллер узнает о необходимости загрузки помеченного кода в ОЗУ.

Параллельный режим выполнения

Микроконтроллеры представляют собой системы с одним ядром, за редкими исключениями. Обычно многозадачность не требуется. Существует один поток выполнения с таймерами и прерываниями, добавляющими асинхронные методы обработки.

Атомарные (неделимые) операции, в основном поддерживаемые компиляторами и микроконтроллерами AVR, не являются исключением. Необходимость атомарных блоков инструкций может возникать в описанных ниже случаях. Следует помнить, что, несмотря на наличие нескольких исключений (инструкция MOVW для копирования пары регистров и косвенная адресация через указатели X, Y, Z), инструкции в 8-битовой архитектуре в общем случае работают только с 8-битовыми значениями:

- 16-битовая переменная считывается побайтово в основной функции и обновляется в ISR;
- 32-битовая переменная считывается, изменяется и после этого возвращается для сохранения либо в основную функцию, либо в ISR, в то время как другая подпрограмма может попытаться получить доступ к этой переменной;
- выполнение блока кода критично по времени (программная эмуляция последовательного соединения bit-banging, запрещение протокола JTAG).

Простой пример, иллюстрирующий первый случай, взят из документации на библиотеку AVR libc:

```
#include <cinttypes>
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/atomic.h>

volatile uint16_t ctr;

ISR(TIMER1_OVF_vect) {
   ctr--;
}

int main() {
         ctr = 0x200;
         start_timer();
         sei();
         uint16_t ctr_copy;
         do {
               ATOMIC_BLOCK(ATOMIC_FORCEON)
               {
                     ctr_copy = ctr;
               }
         }
         while (ctr_copy != 0);

         return 0;
}
```

В приведенном выше коде 16-битовое целое значение изменяется в обработчике прерываний, тогда как основная функция (`main`) копирует это значение в локальную переменную. Функция `sei()` (SEt global Interrupt flag – установка глобального флага прерывания) вызывается для обеспечения сохранения регистра прерывания в известном состоянии. Ключевое слово `volatile` сообщает компилятору, что эта переменная и методы доступа к ней не должны оптимизироваться каким-либо способом.

Поскольку в код включен заголовочный файл поддержки атомарности AVR, можно воспользоваться макрокомандами `ATOMIC_BLOCK` и `ATOMIC_FORCEON`. Эти макрокоманды создают секцию кода, которая непременно будет выполнена автоматически, без какого-либо воздействия со стороны обработчиков прерываний и прочих подпрограмм. Параметр, передаваемый в макро `ATOMIC_BLOCK`, принудительно устанавливает для глобального флага состояния прерывания значение enabled (разрешено).

Так как этот флаг установлен в состояние, существующее перед началом выполнения атомарного блока, нет необходимости сохранять предыдущее состояние флага. Это позволяет экономить ресурсы.

Как было отмечено выше, микроконтроллеры в основном являются одноядерными системами с ограниченной поддержкой функций многозадачности и многопоточности. Для правильной организации многопоточности и многозадачности необходимо выполнять переключения контекста, при которых сохраняется не только указатель стека текущей выполняемой задачи, но также состояние всех регистров и прочих важных объектов.

Несмотря на возможность запуска нескольких потоков и задач на одном микроконтроллере, при использовании 8-битовых микроконтроллеров, таких как AVR и PIC (8-битовая линейка), организация такого режима, вероятнее всего, себя не оправдает и потребует существенного объема трудозатрат.

На более мощных микроконтроллерах (таких как ESP8255 и ARM Cortex-M) можно установить ОС реального времени (ОСРВ – RTOS), в которых реализовано требуемое переключение контекста и не нужны дополнительные трудозатраты на организацию режима параллельного выполнения. ОС реального времени будут рассматриваться несколько позже в этой главе.

Разработка для AVR с использованием Nodate

Компания Microchip предоставляет бинарную версию инструментального комплекта GCC для разработки для микроконтроллеров AVR. Во время написания этой книги самой последней являлась версия AVR-GCC 3.6.1, содержащая набор компиляторов GCC версии 5.4.0, обеспечивающая полную поддержку стандарта C++14 и ограниченную поддержку стандарта C++17.

Пользоваться этим инструментальным комплектом достаточно просто. Можно загрузить его с веб-сайта компании Microchip, распаковать загруженный архивный файл в каталог с соответствующим именем и добавить имя каталога, содержащего выполняемые файлы GCC, в системный путь поиска (переменная среды `PATH`). После этого можно компилировать приложения для микроконтроллеров AVR. На некоторых платформах инструментальный комплект AVR также доступен через менеджер пакетов, что еще более упрощает процесс установки.

Следует отметить, что после установки инструментального комплекта AVR GCC отсутствует доступная стандартная библиотека C++ STL. Поэтому разработчик ограничен только возможностями языка C++, поддерживаемыми компилятором GCC. В документе Microchip AVR FAQ приведены следующие замечания:

- очевидно, что недоступны связанные с C++ стандартные функции, классы и шаблоны классов;
- не реализованы операторы `new` и `delete`. Попытка их использования приведет к выводу предупреждения от линкера (редактора связей) о неопределенных внешних ссылках (вероятно, это может быть исправлено);
- некоторые из представленных в комплекте включаемых файлов не являются безопасными для C++, то есть для них необходима обертка в форме `extern"C" { ... }` (это также должно быть обязательно исправлено);
- исключения не поддерживаются. Поскольку исключения по умолчанию разрешены во внешнем интерфейсе C++, необходимо явно отключить их с помощью флага компилятора `-fno-exceptions`. В противном случае линкер будет выдавать предупреждения о неопределенной внешней ссылке `__gxx_personality_sj0`.

При отсутствии реализации библиотеки libstdc++, которая должна содержать функции и возможности STL, можно добавить поддержку этой функциональности, только воспользовавшись реализацией от независимых сторон. Подобные реализации включают версии, обеспечивающие полную поддержку STL, а кроме того, существуют упрощенные реализации, не следующие стандарту STL API. Примером такой упрощенной реализации является ядро Arduino AVR, предоставляющее классы String и Vector, похожие на аналоги из библиотеки STL, но с некоторыми ограничениями и отличиями.

Быстро развивающейся альтернативой инструментальному комплекту AVR GCC является LLVM, мощная рабочая среда и комплект (фреймворк) компиляторов с недавно добавленной экспериментальной поддержкой AVR, которая в ближайшем будущем должна обеспечить создание бинарных образов для микроконтроллеров AVR с полной поддержкой функциональности STL через внешний интерфейс Clang (поддержка C/C++).

На рис. 4.12 показана обобщенная схема состояния разработки LLVM, демонстрирующая общую концепцию LLVM и особое внимание, уделяемое промежуточному представлению кода (intermediate representation – IR).

К сожалению, для семейства микроконтроллеров PIC, несмотря на то что оно также принадлежит компании Microchip и во многом похоже на семейство AVR, в настоящее время не существует доступного компилятора C++ от компании Microchip. Рекомендуется переход к семейству микроконтроллеров PIC32 (на основе MIPS).

Вводная информация о Nodate

Вы можете предпочесть использование одной из интегрированных сред разработки, рассмотренных ранее в этой главе, но вряд ли это поспособствует тщательному изучению самого процесса разработки для микроконтроллера AVR. Именно поэтому мы рассмотрим простое приложение, разрабатываемое для платы ATmega2560, которая использует модифицированную версию ядра Arduino AVR под названием Nodate (**https://github.com/MayaPosch/Nodate**). Эта рабочая среда ре-

структурирует исходное ядро, позволяя использовать его как обычную библиотеку C++ вместо ограниченного синтаксического анализатора (парсера) и внешнего интерфейса Arduino C-диалекта.

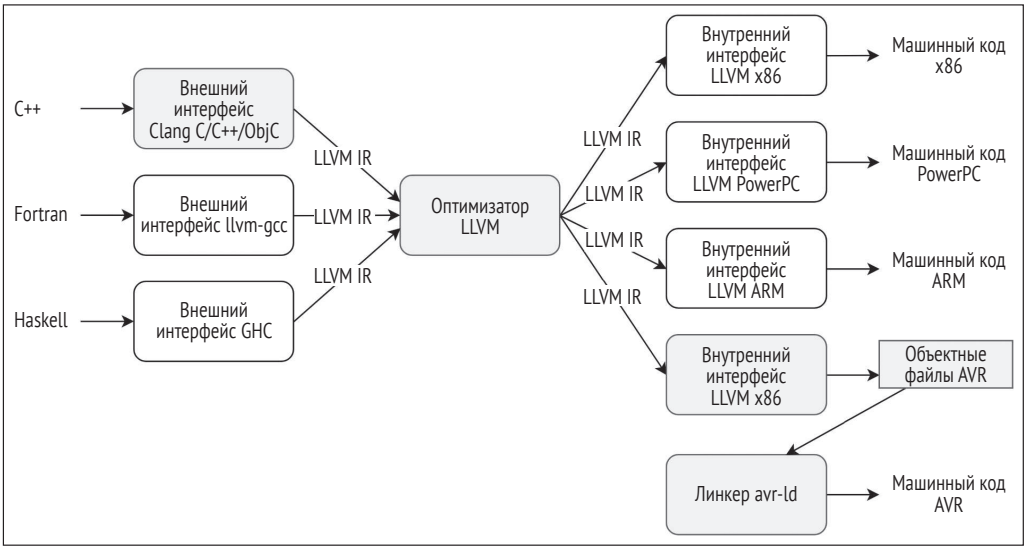

Рис. 4.12

Установка Nodate достаточно проста: необходимо загрузить и разместить Nodate в подходящей локации своей системы и установить для системной переменной `NODATE_HOME` значение, указывающее на корневой каталог установленного фреймворка Nodate. После этого можно воспользоваться одним из примеров приложений как основой для нового проекта.

Пример: инструмент тестирования интегральной микросхемы CMOS

Рассмотрим более полноценный пример проекта, реализующего инструментальное средство тестирования интегральной микросхемы (integrated circuit – IC) для микросхем 5В. В дополнение к проверке контактов GPIO на микросхеме в этом проекте также предусмотрено считывание описания микросхемы и тестирование программы (в форме таблицы логики) с карты твердотельного SD-диска через интерфейс SPI. Управление со стороны пользователя добавляется в форме интерфейса командной строки на основе последовательного интерфейса.

Сначала рассмотрим *Makefile* для этого проекта Nodate. Соответствующий файл расположен в корневом каталоге проекта.

```
ARCH ?= avr

# Предварительная установка типа платы.
BOARD ?= arduino_mega_2560
```

```
# Определение имени выходного файла (ELF и Hex).
OUTPUT := sdinfo

# Добавление файлов, включаемых при компиляции, в соответствующие переменные.
APP_CPP_FILES = $(wildcard src/*.cpp)
APP_C_FILES = $(wildcard src/*.c)

#
# --- Конец списка переменных, редактируемых пользователем --- #
#

# Включаемые файлы Nodate. Требуется обязательная установка переменной среды NODATE_HOME.
APPFOLDER=$(CURDIR)
export

all:
    $(MAKE) -C $(NODATE_HOME)

flash:
    $(MAKE) -C $(NODATE_HOME) flash

clean:
    $(MAKE) -C $(NODATE_HOME) clean
```

В первой строке определяется целевая архитектура, поскольку Nodate можно использовать и для других типов микроконтроллеров. В данном примере определяется архитектура AVR.

Далее устанавливаются предварительные параметры для макетной платы Arduino Mega 2560. В Nodate существует ряд подобных параметров, которые определяют подробные характеристики платы. Для Arduino Mega 2560 получаем следующие предварительные параметры:

```
MCU := atmega2560
PROGRAMMER := wiring
VARIANT := mega # тип платы "Arduino Mega"
```

Если предварительные параметры платы не определены, то необходимо определить эти переменные в Makefile проекта и выбрать существующее значение для каждой переменной, которые в отдельности определяются в собственном Makefile внутри подкаталогов Nodate AVR. В другом варианте можно добавить собственный файл определения микроконтроллера (`MCU`), программатора (`PROGRAMMER`) и версию схемы контактов (`VARIANT`) в Nodate вместе с новым комплектом предварительных параметров и использовать эти данные.

После завершения подготовки Makefile можно приступить к реализации основной функции.

```
#include <wiring.h>
#include <SPI.h>
#include <SD.h>

#include "serialcomm.h"
```

Заголовочный файл *wiring.h* предоставляет доступ ко всей функциональности, связанной с интерфейсом GPIO. Кроме того, включаются заголовочные файлы для шины интерфейса SPI, для устройства чтения SD-карт и специализированный

класс, представляющий собой обертку для последовательного интерфейса. Этот класс будет рассмотрен более подробно немного позже.

```
int main () {
    init();
    initVariant();

    Serial.begin(9600);

    SPI.begin();
```

Сразу после входа в основную функцию инициализируется функциональность интерфейса GPIO с помощью вызова функции `init()`. Следующий вызов функции загружает конфигурацию контактов для используемой в этом проекте конкретной целевой платы (переменная `VARIANT` выше или в Makefile предварительных параметров для этой платы).

После этого инициализируется работа первого последовательного порта со скоростью 9600 бод, затем активизируется шина интерфейса SPI, наконец, выводится информационное сообщение, как показано ниже:

```
   Serial.println("Initializing SD card...");

    if (!SD.begin(53)) {
          Serial.println("Initialization failed!");
          while (1);
    }

    Serial.println("initialization done.");

    Serial.println("Commands: index, chip");
    Serial.print("> ");
```

Предполагается, что к этому моменту к плате Mega уже подключена SD-карта, содержащая список доступных микросхем, которые можно тестировать. Здесь контакт 53 – это контакт аппаратного выбора микросхемы через интерфейс SPI. Этот контакт удобно расположен рядом с остальными контактами интерфейса SPI на плате.

Также предполагается, что все подключения платы выполнены корректно и SD-карту можно считывать без проблем. Далее выводится промпт командной строки на экран консоли.

```
          while (1) {
                String cmd;
                while (!SerialComm::readLine(cmd)) { }

                if (cmd == "index") { readIndex(); }
                else if (cmd == "chip") { readChipConfig(); }
                else { Serial.println("Unknown command.");      }

                Serial.print("> ");
          }

          return 0;
 }
```

Этот цикл просто ожидает ввода, принимаемого на последовательном входном интерфейсе, после чего производится попытка выполнения принятой команды.

Функция, вызываемая для чтения ввода через последовательный интерфейс, блокируется, возвращаясь в активное состояние, только если принят символ перехода на новую строку (пользователь нажал клавишу **Enter**) или если внутренний буфер переполнен, а символ перехода на новую строку не получен. В последнем случае ввод просто отбрасывается (игнорируется) и выполняется очередная попытка чтения из последовательного ввода. На этом реализация функции `main()` завершается.

Рассмотрим подробнее содержимое заголовочного файла для класса `SerialComm`.

```
#include <HardwareSerial.h>      // UART.

static const int CHARBUFFERSIZE 64

class SerialComm {
          static char charbuff[CHARBUFFERSIZE];
public:
          static bool readLine(String &str);
};
```

Этот заголовочный файл включается для поддержки подключения аппаратного последовательного интерфейса. Он обеспечивает доступ к периферийному устройству более низкого уровня UART. Сам класс полностью статический с определением максимального размера буфера символов и функции для чтения строки из последовательного входного интерфейса.

Далее следует его реализация:

```
#include "serialcomm.h"

char SerialComm::charbuff[CHARBUFFERSIZE];

bool SerialComm::readLine(String &str) {
          int index = 0;

          while (1) {
                while (Serial.available() == 0) { }

                char rc = Serial.read();
                Serial.print(rc);

                if (rc == '\n') {
                      charbuff[index] = 0;
                      str = charbuff;
                      return true;
                }

                if (rc >= 0x20 || rc == ' ') {
                      charbuff[index++] = rc;
                      if (index > CHARBUFFERSIZE) {
                            return false;
                      }
                }
          }

          return false;
}
```

В цикле while сначала выполняется вход во вложенный цикл, который работает до тех пор, пока не закончатся символы, считываемые из буфера последовательного входного интерфейса. Таким способом реализовано блокирующее чтение.

Поскольку пользователь должен видеть то, что он вводит с клавиатуры, в следующем блоке кода отображаются все считываемые символы. Далее выполняется проверка приема символа перехода на новую строку. Если этот символ введен, то добавляется завершающий нулевой байт в локальный буфер и его содержимое считывается в экземпляр класса String по предоставленной ссылке. После этого выполняется выход из функции с возвратом значения true.

Здесь можно добавить небольшое усовершенствование – поддержка функции удаления предыдущего символа с возвратом на одну позицию (backspace), чтобы пользователь мог удалять символы в буфере чтения клавишей **Backspace**. Для этого необходимо добавить вариант обработки соответствующего управляющего символа (ASCII 0x8 – backspace), при вводе которого удаляется самый последний символ в буфере, а на удаленном терминале удаляется последний видимый символ.

Если символ перехода на новую строку пока не обнаружен, то выполняется переход к следующей секции. Здесь проверяется прием допустимого корректного символа, интерпретируемого как ASCII 0x20 или пробел. Если прием пробела подтвержден, то продолжается добавление новых символов в буфер и в конце секции выполняется проверка достижения конца буфера чтения (исчерпания его размера). Если размер буфера исчерпан, то возвращается значение false, сообщающее, что буфер полон, но символ перехода на новую строку не обнаружен.

Функции-обработчики readIndex() и readChipConfig() предназначены для выполнения вводимых команд index и chip соответственно.

```
void readIndex() {
          File sdFile = SD.open("chips.idx");
          if (!sdFile) {
                Serial.println("Failed to open IC index file.");
                Serial.println("Please check SD card and try again.");
                while(1);
          }

          Serial.println("Available chips:");
          while (sdFile.available()) {
                Serial.write(sdFile.read());
          }

          sdFile.close();
}
```

Эта функция интенсивно использует класс SD и связанный с ним класс File из библиотеки Arduino SD card. По существу, здесь открывается файл списка микросхем на SD-карте, обеспечивая получение корректной управляющей характеристики (объекта) файла, затем выполняется чтение и вывод каждой строки файла. Это обычный текстовый файл, в котором наименование каждой микросхемы размещено в отдельной строке.

В конце кода обработчика после завершения чтения с SD-карты управляющий файловый объект закрывается с помощью функции sdFile.close(). Код реализации функции-обработчика readChipHandler() аналогичный, но несколько более длинный.

Практическое использование

Для демонстрации работы рассматриваемого примера протестируем простую интегральную микросхему HEF4001 IC (4000 CMOS series Quad 2- Input OR Gate), подключаемую к нашей плате. Для выполнения тестирования необходимо добавить на SD-карту файл, содержащий описание теста и управляющие данные для этой микросхемы. Содержимое файла теста *4001.ic* приведено ниже, оно предназначено для синтаксического разбора (парсинга) и выполнения соответствующих тестов программой, код которой будет рассмотрен во всех подробностях.

```
HEF4001B
Quad 2-input NOR gate.
A1-A2: 22-27, Vss: GND, 3A-4B: 28-33, Vdd: 5V
22:0,23:0=24:1
22:0,23:1=24:0
22:1,23:0=24:0
22:1,23:1=24:0
26:0,27:0=25:1
26:0,27:1=25:0
26:1,27:0=25:0
26:1,27:1=25:0
28:0,29:0=30:1
28:0,29:1=30:0
28:1,29:0=30:0
28:1,29:1=30:0
33:0,32:0=31:1
33:0,32:1=31:0
33:1,32:0=31:0
33:1,32:1=31:0
```

Первые три строки выводятся без изменений, как мы видели ранее, а остальные строки определяют отдельные варианты тестов. Строки тестов сформированы в соответствии со следующим форматом:

```
<pin>:<value>,[..,]<pin>:<value>=<pin>:<value>
```

Этот файл сохранен под именем *4001.ic*, а файл *index.idx* обновлен (введена новая строка «4001») на SD-карте. Для поддержки других интегральных микросхем нужно просто повторить этот шаблон с соответствующими тестовыми последовательностями, сохранить его в файле и внести имя этого файла в индекс. Ниже показан код обработчика конфигурации микросхемы, который также начинает процедуру тестирования:

```
void readChipConfig() {
          Serial.println("Chip name?");
          Serial.print("> ");
          String chip;
          while (!SerialComm::readLine(chip)) { }
```

Сначала пользователю предлагается ввести наименование микросхемы в том виде, в котором она указана в списке, выведенном командой `index`.

```
          File sdFile = SD.open(chip + ".ic");
          if (!sdFile) {
```

```
        Serial.println("Failed to open IC file.");
        Serial.println("Please check SD card and try again.");
        return;
    }

    String name = sdFile.readStringUntil('\n');
    String desc = sdFile.readStringUntil('\n');
```

Мы пытаемся открыть файл, содержащий подробности о заданной микросхеме, далее выполняется чтение содержимого этого файла, начиная с названия и описания тестируемой микросхемы.

```
Serial.println("Found IC:");
Serial.println("Name: " + name);
Serial.println("Description: " + desc);

String pins = sdFile.readStringUntil('\n');
Serial.println(pins);
```

После вывода наименования и описания выбранной микросхемы считывается строка, содержащая инструкции по ее подключению к разъемам на тестирующей плате Mega.

```
Serial.println("Type 'start' and press <enter> to start test.");
Serial.print("> ");
String conf;
while (!SerialComm::readLine(conf)) { }
if (conf != "start") {
      Serial.println("Aborting test.");
      return;
}
```

Запрашивается подтверждение пользователя на начало тестирования микросхемы. Любая команда, кроме `start`, отменяет тестирование и возвращает пользователя в главный управляющий цикл.

После ввода команды `start` начинается тестирование.

```
int result_pin, result_val;
while (sdFile.available()) {
      // Чтение строки в следующем формате:
      // <pin>:<value>, [..,]<pin>:<value>=<pin>:<value>
      pins = sdFile.readStringUntil('=');
      result_pin = sdFile.readStringUntil(':').toInt();
      result_val = sdFile.readStringUntil('\n').toInt();
      Serial.print("Result pin: ");
      Serial.print(result_pin);
      Serial.print(", expecting: ");
      Serial.println(result_val);
      Serial.print("\n");

      pinMode(result_pin, INPUT);
```

На первом этапе считывается следующая строка файла микросхемы, которая должна содержать инструкции первого теста. Первая секция содержит код настройки входных контактов, а после считанного знака равенства расположен раз-

дел, содержащий данные о контакте вывода, и соответствующее ожидаемое значение для этого теста.

Также выводится номер разъема платы, с которым соединен контакт результата, и его ожидаемое значение. Далее контакт результата определяется как контакт ввода, чтобы получить возможность чтения с него после завершения теста.

```
        int pin;
        bool val;
        int idx = 0;
        unsigned int pos = 0;
        while ((idx = pins.indexOf(':', pos)) > 0) {
              int pin = pins.substring(pos, idx).toInt();
              pos = idx + 1; // Переход к символу после двоеточия.

              bool val = false
              if ((idx = pins.indexOf(",", pos)) > 0) {
                    val = pins.substring(pos, idx).toInt();
                    pos = idx + 1;
              }
              else {
                    val = pins.substring(pos).toInt();
              }

              Serial.print("Setting pin ");
              Serial.print(pin);
              Serial.print(" to ");
              Serial.println(val);
              Serial.print("\n");
              pinMode(pin, OUTPUT);
              digitalWrite(pin, val);
        }
```

Для действительного выполнения теста используется первая строка `String`, прочитанная из файла для этого теста, выполняется ее синтаксический разбор (парсинг) и извлекаются значения для входных контактов. Для каждого контакта сначала мы получаем его номер, затем – соответствующее значение (`0` или `1`).

В консоли выводятся считанные номера контактов и значения последовательного интерфейса вывода до установки их в режим вывода. Затем в каждый контакт записывается тестовое значение, как показано ниже:

```
        delay(10);

        int res_val = digitalRead(result_pin);
        if (res_val != result_val) {
              Serial.print("Error: got value ");
              Serial.print(res_val);
              Serial.println(" on the output.");
              Serial.print("\n");
        }
        else {
              Serial.println("Pass.");
        }
  }
```

```
        sdFile.close();
}
```

После выхода из внутреннего цикла все входные значения будут уже установлены. Необходимо лишь немного подождать для полной уверенности в том, что тестируемая микросхема получила достаточно времени для установки новых выходных значений до очередной попытки считывания результирующего значения на выходном контакте.

Проверка микросхемы – это просто чтение результирующего значения с выходного контакта и последующее сравнение полученного значения с ожидаемым значением. Затем результат сравнения выводится на последовательное устройство вывода.

После завершения процедуры тестирования файл микросхемы закрывается и выполняется возврат в главный управляющий цикл для ожидания следующих инструкций.

После записи программы во флеш-память на плате Mega и установления соединения с последовательным портом платы мы получаем следующий результат:

```
Initializing SD card...
initialization done.
Commands: index, chip
> index
```

После запуска программы выводится сообщение о том, что SD-карта найдена и успешно инициализирована. Теперь можно считывать данные с SD-карты. Также выводится список доступных команд. Далее выполняется команда `index` для получения списка доступных для тестирования микросхем:

```
Available chips:
4001
> chip
Chip name?
> 4001
Found IC:
Name: HEF4001B
Description: Quad 2-input NOR gate.
A1-A2: 22-27, Vss: GND, 3A-4B: 28-33, Vdd: 5V
Type ‘start’ and press <enter> to start test.
> start
```

Получив только одну доступную для тестирования микросхему, выполняем команду `chip` для входа в меню выбора микросхемы, после чего вводим специальное наименование микросхемы.

После ввода наименования загружается файл, предварительно помещенный на SD-карту, и из него выводятся первые три строки. Затем следует цикл ожидания, предоставляющий время для подключения микросхемы в соответствии с номерами разъемов на плате Mega и схемой назначения контактов для тестируемой микросхемы по предоставленной спецификации.

После проверки правильности всех необходимых соединений вводится команда `start` и подтверждение начала тестирования. Запускается тест:

```
Result pin: 24, expecting: 1
Setting pin 22 to 0
Setting pin 23 to 0
Pass.
Result pin: 24, expecting: 0
Setting pin 22 to 0
Setting pin 23 to 1
Pass.
Result pin: 24, expecting: 0
Setting pin 22 to 1
Setting pin 23 to 0
[...]
Result pin: 31, expecting: 0
Setting pin 33 to 1
Setting pin 32 to 0
Pass.
Result pin: 31, expecting: 0
Setting pin 33 to 1
Setting pin 32 to 1
Pass.
>
```

Для каждого из четырех одинаковых вентилей OR на микросхеме выполняется проход по одной и той же таблице истинности для тестирования каждой входной комбинации. Эта конкретная микросхема успешно прошла тестирование и может использоваться в любом проекте с гарантией безопасности.

Описанный выше способ проверки устройств может быть полезным для тестирования любого типа микросхем пятивольтового уровня, в том числе логических микросхем 74 и 4000. Также возможна адаптация проекта для использования PWM, ADC и других контактов для тестирования микросхем, которые не являются строго цифровыми в отношении их элементов ввода и вывода.

Разработка для ESP8266 с использованием Sming

Для разработок на основе ESP8266 не существует инструментальных средств, официально предоставляемых производителем (Espressif), кроме «голого железа» и комплекта разработки ПО (SDK) на основе ОСРВ. Поэтому проекты с открытым исходным кодом с использованием Arduino предоставляют более удобную для разработчика рабочую среду для создания приложений. Для Arduino на основе ESP8266 альтернативой применения C++ является Sming (**https://github.com/SmingHub/Sming**), совместимая с Arduino рабочая среда, похожая на Nodate для AVR, описанную в предыдущем разделе.

В главе 5 «Пример: монитор влажности почвы с использованием протокола Wi-Fi» будет подробно рассматриваться процесс разработки с использованием этой рабочей среды для микроконтроллера ESP8266.

Разработка для микроконтроллеров ARM

Процесс разработки для платформы микроконтроллеров ARM незначительно отличается от разработки для микроконтроллеров AVR, за исключением гораздо лучшей поддержки C++. Поэтому для разработки предлагается богатый выбор инструментальных комплектов, как мы видели в начале текущей главы в списке наиболее известных и распространенных интегрированных сред разработки. Кроме того, список доступных для Cortex-M ОСРВ гораздо шире, чем для AVR и ESP8266.

Использование свободных компиляторов с открытым исходным кодом, в том числе GCC и LLVM для ориентации на широкий спектр архитектур микроконтроллеров ARM (на основе Cortex-M и аналогичных), обеспечивает полную свободу выбора и простой доступ к полнофункциональной библиотеке C++ STL (хотя некоторые предпочитают не пользоваться исключениями).

При разработке непосредственно для аппаратуры микроконтроллеров Cortex-M может возникать необходимость добавления следующего флага линкера для создания простых заглушек для некоторых функций, обычно предоставляемых операционной системой:

```
-specs=nosys.specs
```

Микроконтроллеры ARM могут показаться менее привлекательными из-за гораздо меньшей стандартизации плат и микроконтроллеров по сравнению с большей степенью стандартизации AVR в форме плат Arduino. Несмотря на то что сообщество Arduino в свое время выпустило плату Arduino Due на основе микроконтроллера SAM3X8E Cortex-M3, тем не менее эта плата использовала тот же форм-фактор и приблизительно ту же схему контактов (только с применением 3,3-вольтового ввода/вывода вместо 5 В), что и плата Arduino Mega на основе ATmega2560.

Из-за такого проектного решения бóльшая часть функциональности микроконтроллера нарушается и становится недоступной, если разработчик не обладает хорошими навыками работы с паяльником и тонкими проводами. Эта функциональность включает Ethernet-соединение, десять (цифровых) контактов GPIO и т. д. Подобная недоступность контактов встречается и при использовании платы Arduino Mega (ATmega2560), но для упомянутого выше микроконтроллера Cortex-M это становится еще более заметным.

В результате эти варианты позиционируются как платы для макетирования и протипирования, других очевидных обобщенных вариантов нет. Может возникнуть соблазн воспользоваться относительно дешевыми и богатыми функциональными платами прототипирования, например предлагаемыми компанией STMicroelectronics для собственной линейки микроконтроллеров на базе Cortex-M.

Использование операционной системы реального времени

При ограниченных ресурсах, доступных на условном усредненном микроконтроллере и абсолютно понятном цикле процесса в приложениях, работающих на

микроконтроллерах, трудно подобрать вариант использования операционной системы реального времени (ОСРВ) для таких микроконтроллеров. Если разработчик должен обеспечить сложный режим управления ресурсами и задачами, то применение ОСРВ становится более подходящим вариантом для экономии времени на разработку.

Таким образом, основным преимуществом использования ОСРВ является устранение необходимости заново «изобретать колесо». Тем не менее решение о применении ОСРВ должно приниматься отдельно для каждого конкретного случая. В большинстве проектов необходимость интеграции ОСРВ в комплект разработки более приемлема, чем менее реальные подходы, при которых скорее возрастает нагрузка на систему, нежели упрощается общее решение.

Например, в проектах, где разработчик пытается найти баланс использования времени процессора и прочих системных ресурсов между различными интерфейсами обмена и хранения данных, а также пользовательским интерфейсом, применение ОСРВ определенно имеет смысл.

Как мы уже видели в этой главе, при разработке встроенных систем в основном используется простой цикл (главный цикл) в совокупности с некоторым количеством прерываний для обработки задач реального времени. При совместном использовании данных функцией обработки прерывания и главным циклом разработчик несет полную ответственность за безопасность и корректность обработки данных.

Именно для этого случая ОСРВ может предоставить планировщик и даже возможность выполнения задач (процессов), полностью изолированных друг от друга (особенно для микропроцессоров, в которых имеется блок управления памятью (memory management unit – MMU). В многоядерных микроконтроллерах ОСРВ позволяет с легкостью эффективно задействовать все ядра без разработки собственного механизма планирования.

Вместе с тем использование ОСРВ дает не только набор преимуществ. Даже если не учитывать возрастающие требования к размерам ПЗУ и ОЗУ при внедрении ОСРВ в проект, придется считаться с фундаментальным изменением некоторых системных взаимодействий и, возможно, (парадоксальным) увеличением задержек при обработке прерываний.

Поэтому, несмотря на присутствие в названии ОСРВ словосочетания «реального времени», очень трудно получить более «реально временную» систему, чем при использовании простого цикла выполнения и небольшого набора прерываний. Таким образом, о преимуществах ОСРВ нельзя делать каких-либо всеобъемлющих умозаключений, особенно если доступна библиотека поддержки или рабочая среда для программирования непосредственно для аппаратуры (например, совместимая с Arduino рабочая среда, рассмотренная в этой главе), то есть пригодная для прототипирования и разработки промышленного уровня на том же уровне сложности, что и объединение нескольких существующих библиотек.

Резюме

В этой главе рассматривались методы выбора подходящего микроконтроллера для нового проекта, а также способы добавления периферийных устройств и определения требований к Ethernet-интерфейсу и последовательному интерфейсу в про-

екте. Мы узнали, как распределяется память в различных микроконтроллерах, как работать со стеком и с динамически распределяемой памятью (кучей). В конце главы был рассмотрен пример проекта AVR, методы разработки для других архитектур микроконтроллеров и возможные варианты использования ОСРВ.

Предполагается, что после изучения этой главы читатель способен обосновать выбор микроконтроллера на основе набора проектных требований и его преимущества по сравнению с другими микроконтроллерами. Также предполагается, что теперь читатель может реализовать простые проекты с использованием UART и прочих периферийных устройств и хорошо понимает принципы управления памятью и практическое применение механизма прерываний.

В следующей главе будет подробно рассматриваться разработка для микроконтроллера ESP8266 в форме проекта встроенной системы отслеживания уровня влажности почвы и управления водяным насосом, подключающимся при необходимости.

Глава 5

Пример: монитор влажности почвы с использованием протокола Wi-Fi

Уход за комнатными растениями – не самое простое занятие. Пример проекта в этой главе продемонстрирует, как создать монитор влажности почвы с использованием протокола Wi-Fi с функциями исполнительного устройства (актуатора) для насоса или аналогичного устройства, такого как запорный клапан (вентиль) и резервуар для воды с естественным (гравитационным) напором. Используя встроенный веб-сервер, мы получим возможность создания пользовательского интерфейса на основе браузера для наблюдения за состоянием растений и для реализации функций системы управления. Кроме того, можно интегрировать этот проект в более крупную систему, используя REST API на основе протокола HTTP.

В этой главе рассматриваются следующие темы:

- программирование микроконтроллера ESP8266;
- подключение сенсоров (датчиков) и исполнительных устройств (актуаторов) к микроконтроллеру ESP8266;
- реализация HTTP-сервера на этой платформе;
- разработка пользовательского веб-интерфейса для мониторинга и управления;
- интеграция рассматриваемого проекта в более крупную сеть.

Уход за растениями

Для поддержания растений в жизнеспособном состоянии необходимы следующие условия:

- питательные вещества;
- свет;
- вода.

Для выполнения двух первых условий обычно высаживают растения в почву, богатую питательными веществами, и помещают их в хорошо освещаемое место соответственно. После выполнения этих условий главной проблемой сохранения жизнеспособности растений обычно становится третье условие, поскольку необходимо обеспечивать водой растения ежедневно.

Но для этого нужно не просто поливать растения, а обеспечить достаточную, но не избыточную влажность почвы. Излишнее количество воды в почве отрицательно воздействует на способность поглощения растением кислорода через корни. В результате при избыточной влажности почвы растение вянет и погибает.

С другой стороны, при слишком малой влажности почвы растение не получает достаточного количества воды для компенсации испарения через листья, а также не может получать питательные вещества через корни. И в этом случае растение вянет и погибает.

При ручном поливе растений обычно используется приблизительная оценка количества воды, необходимого растению, с неточной проверкой влажности верхнего слоя почвы с помощью пальцев. Такой способ дает слишком мало информации о количестве воды, действительно имеющемся вблизи от корней растения, то есть в гораздо более глубоком слое почвы.

Для более точного измерения влажности почвы используются методы, описанные в табл. 5.1.

У всех перечисленных в табл. 5.1 сенсоров есть свои преимущества и недостатки. При использовании гипсового блока и тензиометра обеспечено значительное удобство сопровождения: в первом случае достаточно того, что гипс сохраняет способность растворяться в воде, при этом не отменяется калибровка. Во втором случае самым главным фактором является то, что герметическая заглушка (крышка) должна оставаться воздухонепроницаемой и не позволять воздуху проникать в трубку. Даже небольшая щель в заглушке сразу делает вакуумный сенсор бесполезным.

Другим важным фактором является стоимость. Сенсорные датчики FDR и TDR могут давать весьма точные показания, но они очень дорого стоят. Поэтому при необходимости просто поэкспериментировать с сенсорами влажности почвы обычно предпочитают пользоваться датчиками сопротивления или емкостными зондами. Главный недостаток датчика первого типа проявляется в течение месяца эксплуатации – коррозия.

Когда два электрода погружаются в раствор, содержащий ионы, а на один из электродов подается электрический ток, обычная химическая реакция приводит к тому, что один из электродов быстро корродирует (теряет материал), поэтому такая схема не может эксплуатироваться долго. Кроме того, почва загрязняется молекулами металла. Применение переменного тока вместо постоянного на одном электроде может несколько снизить эффект коррозии, но проблема не решается окончательно.

Среди сенсоров, сохраняющих баланс между дешевизной и достаточной точностью измерения влажности почвы, только емкостной датчик (зонд) удовлетворяет всем требованиям. Его точность вполне приемлема для измерений и сравнений с высокой чувствительностью (после калибровки), он не подвержен воздействию почвенной влаги и сам никак не влияет на окружающую почву.

Таблица 5.1

Тип	Основной принцип	Описание
Гипсовый блок	Сопротивление	Вода поглощается гипсом, часть ее рассеивается, создавая ток между двумя электродами. По значению сопротивления можно определить влажность почвы
Тензиометр	Вакуум	В пустой трубке имеется вакуумметр на одном конце и пористая крышка на другом конце, позволяющая воде свободно проникать внутрь и просачиваться наружу. Вода, попавшая в трубку из почвы, увеличивает показания вакуумного сенсора, свидетельствуя о затрудненном извлечении влаги из почвы растениями (всасывающая сила)
Емкостный датчик (зонд)	Рефлектометрия частотных интервалов (Frequency Domain Reflectometry – FDR)	Используется диэлектрическая константа между двумя металлическими электродами (погруженными в почву) в электрической схеме осциллятора для измерения изменений этой константы из-за колебаний уровня влажности. Позволяет определять уровень влажности
Микро-волновый сенсор	Рефлектометрия временных интервалов (Time Domain Reflectometry – TDR)	Измеряется время, требуемое для прохождения микроволнового сигнала к концу параллельных пробников (датчиков) и обратно. Разность во времени прохождения сигнала зависит от диэлектрической константы почвы. Таким образом измеряется уровень влажности
Устройство ThetaProbe	Радиочастотный амплитудный импеданс	Радиосигнал в форме 100 МГц синусоидальной волны передается между четырьмя датчиками, размещенными внутри цилиндрического участка почвы. Изменение импеданса синусоидальной волны используется для вычисления концентрации воды в почве
Резистивный датчик	Сопротивление	Аналогично использованию гипсового блока, за исключением применения электродов. Таким образом оценивается только наличие воды (и ее проводимость), а не всасывающая сила почвы

Для реальной подпитки растений водой необходимо определить способ подачи ее правильного количества. В основном это зависит от масштаба системы, которая определяет выбор способа подачи воды. Для орошения большого поля можно воспользоваться насосом с крыльчаткой (импеллером), способным распылять десятки литров воды в минуту.

Для одного растения, вероятнее всего, потребуется возможность подачи не более нескольких сотен миллилитров воды в минуту. В этом случае вполне достаточным будет использование перистальтического насоса. Это тот тип насоса, который можно применять в лабораторных и медицинских приложениях, когда необходимо обеспечить подачу небольшого объема жидкости с высокой точностью.

Предлагаемое решение

Для сохранения простоты мы будем создавать систему, обслуживающую одно растение. Это обеспечит бóльшую свободу в отношении ее размещения, и при дальнейшем создании отдельных систем для каждого растения не имеет значения,

располагается ли комплекс системы и растения на подоконнике, на столе или на балконе (террасе).

В дополнение к измерению уровня влажности почвы необходима также возможность автоматической подачи воды к растению в соответствии с установленными уровнями сигналов. Кроме того, необходимо наблюдать за этим процессом. Такие условия требуют некоторого типа сетевого доступа, причем предпочтителен беспроводной доступ, чтобы избавиться от любых проводов, кроме кабеля электропитания.

Микроконтроллер ESP8266 выглядит наиболее подходящим для этой задачи, а для разработки и отладки системы вполне подходит макетная плата NodeMCU. На ней будет размещен сенсорный датчик для измерения влажности почвы и перистальтический насос.

Установив соединение с IP-адресом системы ESP8266 с помощью веб-браузера, можно отслеживать текущее состояние системы, в том числе уровень влажности почвы и некоторые другие дополнительные характеристики. Конфигурирование системы и многие последующие операции будут выполняться с применением широко используемого компактного бинарного протокола MQTT, при этом система также будет публиковать свое состояние, так что можно будет считывать ее данные в базу данных для мониторинга и анализа.

При таком подходе в дальнейшем можно также создать внутренний сервис для объединения нескольких подобных (сетевых) узлов в единую связанную систему с централизованным контролем и управлением. Более подробно эта тема будет рассматриваться в главе 9 «Пример: мониторинг и управление внутренним микроклиматом в здании».

Аппаратура

Идеальное решение должно включать наиболее точный сенсорный датчик, но без ущерба банковскому счету. Это означает, что предпочтительнее воспользоваться емкостным датчиком, описанным выше в этой главе. Можно приобрести емкостные сенсорные датчики всего лишь за несколько евро или долларов для реализации простого проектного решения на основе интегральной микросхемы таймера 555 (рис. 5.1).

Необходимо просто погрузить датчик в почву так, чтобы электронные компоненты остались снаружи, затем подключить источник электропитания и присоединить кабель, связывающий датчик и аналогово-цифровой преобразователь микроконтроллера.

Большинство доступных перистальтических насосов требует напряжения 12 В. Таким образом, необходим источник электропитания, обеспечивающий напряжение 5 В и 12 В, или так называемый повышающий преобразователь напря-

жения с 5 В до 12 В. Кроме того, в любом случае необходим какой-либо способ включения и отключения насоса. При использовании повышающего преобразователя можно задействовать его контакт Enable (разрешить) для подачи выходного сигнала о включении и отключении через контакт интерфейса GPIO на микроконтроллере.

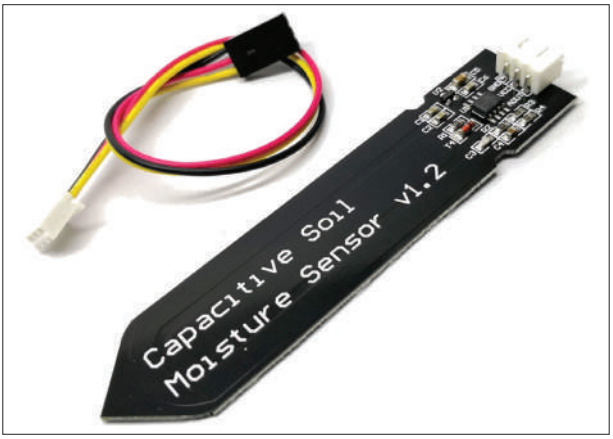

Рис. 5.1

Для прототипирования можно воспользоваться одним из таких модулей повышающих преобразователей напряжения с 5 В до 12 В на основе ступенчатого импульсного регулятора (рис. 5.2).

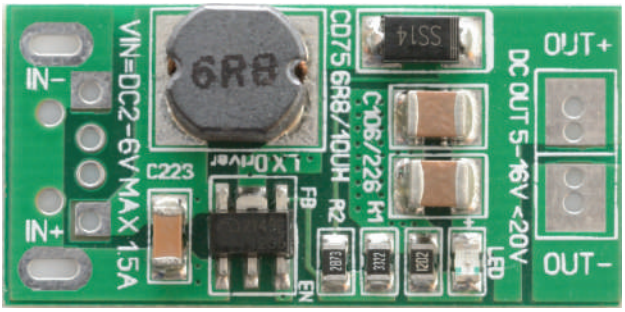

Рис. 5.2

На этом модуле нет выходного контакта Enable, но можно без затруднений припаять провод прямо к контакту EN на плате, как показано на рис. 5.3.

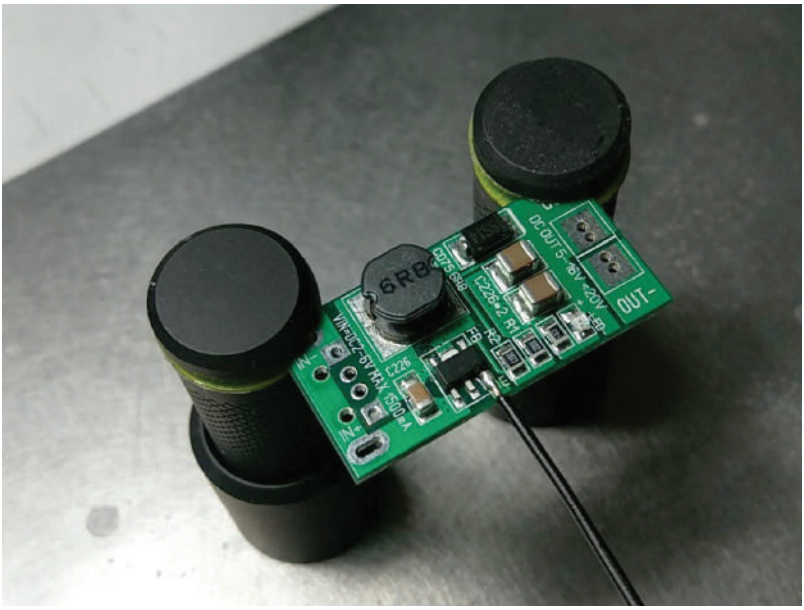

Рис. 5.3

После этого выходные контакты модуля повышающего преобразователя напряжения соединяются с контактами перистальтического насоса (рис. 5.4).

Рис. 5.4

Также необходимы трубки подходящего диаметра для соединения с резервуаром для воды и с растением. Сам насос может вращаться (подавать воду) в любом направлении. Поскольку по существу он состоит из набора роликовых приводов

в корпусе, подающих воду внутрь, любая сторона насоса может быть вводом или выводом.

Необходима обязательная предварительная проверка направления потока жидкости с помощью двух контейнеров и небольшого количества воды и пометка выявленного направления на поверхности корпуса насоса вместе с положительным и отрицательным контактами соединения, используемого при проверке.

В дополнение к описанным выше компонентам также потребуется RGB LED-индикатор, подключенный для визуального отображения получаемых сигналов. Воспользуемся модулем APA102 RGB LED, который соединяется с микроконтроллером ESP8266 через шину SPI, как показано на рис. 5.5.

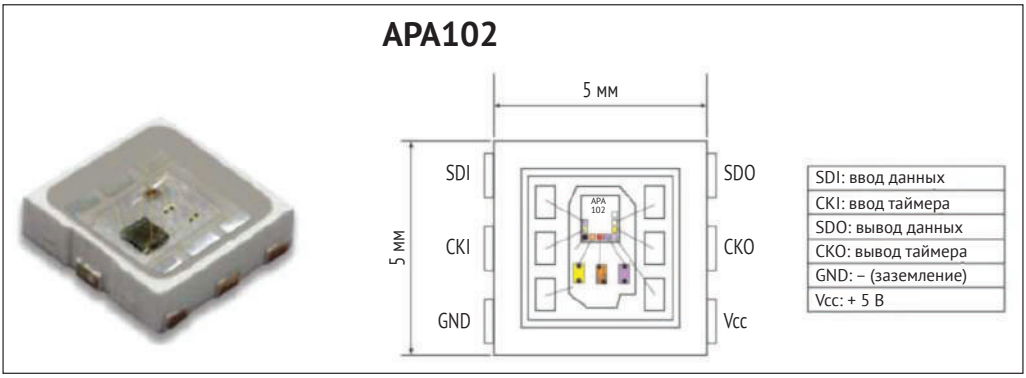

Рис. 5.5

Можно использовать один источник электропитания, обеспечивающий напряжение 5 В и силу тока 1 А и более, а также позволяющий регулировать внезапные скачки напряжения от повышающего преобразователя при каждом включении насоса.

На рис. 5.6 показана общая блок-схема проектируемой системы в целом.

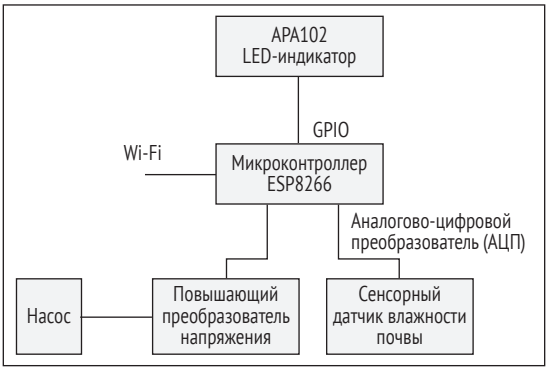

Рис. 5.6

Специализированное программное обеспечение

Для рассматриваемого здесь проекта предусмотрена реализация модуля, содержащего некоторое специализированное программное обеспечение (ПО), которое будет использоваться в главе 9 «Пример: мониторинг и управление внутренним микроклиматом в здании». Таким образом, в текущей главе рассматриваются исключительно те части, которые характерны только для данного модуля водоснабжения растения.

Перед началом разработки специализированного ПО необходимо установить и настроить среду разработки. Предлагается установка комплекта ESP8266 SDK и рабочей среды Sming.

Настройка рабочей среды Sming

Среда разработки для микроконтроллера ESP8266 на основе Sming может использоваться в ОС Linux, Windows и macOS. Возможно, вы предпочитаете отдельно разрабатываемую ветвь Sming, которая при использовании в ОС Linux (или в виртуальной машине Linux, или в подсистеме WSL (Windows Subsysem for Linux) ОС Windows 10) является самым простым вариантом, настоятельно рекомендуемым к практическому применению. В ОС Linux рекомендуется установка в каталог /opt для согласования с руководством для начинающих Sming.

Это краткое руководство для начинающих (при использовании ОС Linux) можно найти по адресу: **https://github.com/SmingHub/Sming/wiki/Linux-Quickstart**.

В Linux можно воспользоваться комплектом Open SDK для ESP8266, который предоставляет официальный Espressif (не для ОСРВ) SDK и заменяет все компоненты без открытого исходного кода на доступные альтернативные варианты с открытым исходным кодом. Установка выполняется следующим образом:

```
git clone --recursive https://github.com/pfalcon/esp-open-sdk.git
cd esp-open-sdk
make VENDOR_SDK=1.5.4 STANDALONE=y
```

Это позволит получить текущую версию исходного кода Open SDK и скомпилировать ее с указанием целевой версии 1.5.4 официального комплекта SDK. Уже существует версия 2.0 этого комплекта, но в ней могут оставаться некоторые проблемы с совместимостью с рабочей средой Sming. Использование версии 1.5.4 предлагает почти ту же функциональность, но с использованием тщательно протестированного надежного кода. Разумеется, со временем ситуация изменится, поэтому внимательно следите за официальной документацией Sming, чтобы вовремя получать инструкции по обновлению.

Установка параметра `STANDALONE` означает, что SDK будет скомпонован как отдельный независимо устанавливаемый экземпляр SDK и комплекта инструментов без внешних зависимостей. Это рекомендуемая опция для использования в совокупности с рабочей средой Sming.

Установка рабочей среды Sming выполняется так же просто:

```
git clone https://github.com/SmingHub/Sming.git
cd Sming
make
```

Здесь выполняется компиляция и сборка рабочей среды Sming. При добавлении новых библиотек в каталог *Libraries* рабочей среды Sming необходимо повторно выполнить последний шаг (`make`), чтобы скомпоновать и установить новый экземпляр совместно используемой библиотеки Sming.

Для рассматриваемого здесь проекта скопируйте подкаталоги из каталога libs проекта ПО для данной главы в каталог Sming/Sming/Libraries до компиляции рабочей среды Sming, иначе компиляция кода проекта не будет выполнена.

Компиляция рабочей среды Sming выполняется еще и с поддержкой протокола SSL. Для этого требуется установка параметра `ENABLE_SSL=1` в файле *Makefile*. Это позволит обеспечить при компиляции встроенную непосредственно в код библиотеки Sming поддержку шифрования на основе протокола axTLS.

После завершения всех вышеописанных шагов необходимо установить `esptool.py` и `esptool2`. В каталоге */opt* выполните следующие команды для получения `esptool`:

```
wget https://github.com/themadinventor/esptool/archive/master.zip
unzip master.zip
mv esptool-master esp-open-sdk/esptool
```

Файл *esptool.py* – это скрипт на языке Python, который позволяет обмениваться данными с ПЗУ SPI, являющимся частью каждого модуля ESP8266. Это способ передачи нашего кода в ПЗУ микроконтроллера. Это инструментальное средство автоматически используется рабочей средой Sming:

```
cd  $ESP_HOME
git clone https://github.com/raburton/esptool2
cd esptool2
make
```

Утилита `esptool2` представляет собой альтернативу набору скриптов в официальном комплекте SDK, выполняющему преобразование выходных данных линкера в формат ПЗУ, в котором бинарный код можно записать в ESP8266. Эта утилита автоматически вызывается средой Sming при компиляции приложения.

Наконец, предполагая, что комплект SDK и Sming уже установлены в каталог */opt*, можно добавить следующие глобальные переменные среды и отредактировать системную переменную `PATH`:

```
export ESP_HOME=/opt/esp-open-sdk
export SMING_HOME=/opt/Sming/Sming
export PATH=$PATH:$ESP_HOME/esptool2
export PATH=$PATH:$ESP_HOME/xtensa-lx106-elf/bin
```

В последней строке добавляется путь к бинарным файлам комплекта инструментов, которые потребуются при отладке приложений для микроконтроллера ESP8266, как мы увидим в главе 7 «Тестирование платформ с ограниченными ресурсами». После этого можно приступать к разработке с использованием рабочей среды Sming и создавать образы ПЗУ, которые записываются непосредственно в микроконтроллер.

Код модуля для растения (plant)

В этом разделе будет рассматриваться простой исходный код для нашего проекта, начиная с основного модуля (ядра) `OtaCore`, и далее – класс `BaseModule`, в котором регистрируются все специализированные программные модули. В конце раздела будет показан класс `PlantModule`, содержащий бизнес-логику (логику предметной области) для выполнения проектных требований, которые были описаны выше в этой главе.

Следует отметить, что для этого проекта разрешено использование опций управления загрузкой rBoot и rBoot Big Flash в общем проектном файле *Makefile*. Это позволяет создать четыре блока размером в 1 Мб в 4 Мб ПЗУ, доступных в соответствующем модуле ESP8266 (это все модули ESP-12E/F), из которых два используются для размещения образов специализированного ПО, а остальные два – для хранения файлов (с использованием файловой системы SPIFFS).

Затем загрузчик rBoot записывается в самое начало ПЗУ, чтобы при каждой загрузке он начинал работать в первую очередь. Только один из слотов специализированного ПО является активным в любой произвольный момент времени. В этой схеме настройки очень удобна и полезна возможность выполнения обновлений по воздуху (over the air – OTA) посредством простой записи нового образа специализированного ПО в неактивный слот спец-ПО, последующей смены активного слота и перезапуска микроконтроллера. Если загрузчик rBoot не может загрузить новый образ специализированного ПО, то он «откатывается» к другому слоту, содержащему работающую версию специализированного ПО, из которого было выполнено обновление по воздуху.

Файл Makefile-user.mk

В корневом каталоге проекта расположен файл *Makefile*. Он содержит ряд параметров настройки, которые можно изменить по своему усмотрению для конкретного проекта. Некоторые из этих параметров кратко описаны в табл. 5.2.

Таблица 5.2

Имя параметра	Описание
`COM_PORT`	Если соединение с платой всегда устанавливается через один и тот же последовательный порт, то можно жестко зафиксировать его здесь, чтобы не вводить вручную лишние данные
`SPI_MODE`	Это настройка режима последовательного интерфейса SPI, используемого для записи образов специализированного ПО в SPI ПЗУ. В режиме `dio` работают только две линии данных (`SD_D0, D1`), в режиме `qio` – четыре линии (`SD_D0-3`). Не во всех ПЗУ SPI подключены все четыре линии данных. Режим `qio` быстрее, но режим `dio` должен работать всегда
`RBOOT_ENABLED`	При установке значения 1 разрешается поддержка загрузчика rBoot. В нашем проекте необходимо, чтобы поддержка rBoot была разрешена
`RBOOT_BIG_FLASH`	При доступном размере ПЗУ 4 Мб разрешается использование всего объема памяти. В нашем проекте это также необходимо
`RBOOT_TWO_ROMS`	Этот параметр устанавливается, если необходимо разместить два образа специализированного ПО на отдельной микросхеме ПЗУ размером 1 Мб. Этот параметр можно применять к некоторым модулям и вариантам микроконтроллеров ESP8266

Таблица 5.2 *(окончание)*

Имя параметра	Описание
SPI_SIZE	Здесь устанавливается размер SPI ПЗУ микросхемы (для рассматриваемого проекта – 4 Мб)
SPIFF_FILES	Локация каталога, содержащего файлы, которые будут помещены в образ SPIFFS ПЗУ, записываемого в микроконтроллер
SPIFFS_SIZE	Размер создаваемого образа SPIFFS ПЗУ. Стандартный размер 64 Кб, но можно использовать до 1 Мб, если необходимо, но при размере ПЗУ 4 МБ и установленном (разрешенном) параметре RBOOT_BIG_FLASH
WIFI_SSID	Идентификатор SSID для сети Wi-Fi, с которой необходимо установить соединение
WIFI_PWD	Пароль для сети Wi-Fi
MQTT_HOST	URL или IP-адрес используемого сервера (брокера) MQTT
ENABLE_SSL	Разрешение поддержки протокола SSL, скомпилированного непосредственно в рабочую среду Sming для возможности использования специализированным ПО зашифрованных по протоколу TLS соединений с брокером MQTT
MQTT_PORT	Порт брокера MQTT. Зависит от разрешения использования протокола SSL
USE_MQTT_PASSWORD	Устанавливается значение true, если необходимо установление соединения с брокером MQTT с предъявлением имени пользователя и пароля
MQTT_USERNAME	Имя пользователя брокера MQTT, если требуется
MQTT_PWD	Пароль пользователя брокера MQTT, если требуется
MQTT_PREFIX	Префикс, который можно (но необязательно) добавлять перед каждой темой публикации MQTT, используемый специализированным ПО при необходимости. Префикс должен завершаться символом слеша, чтобы не оставаться пустым
OTA_URL	Жестко закодированный URL, который будет использоваться специализированным ПО при каждом запросе на обновление по воздуху (OTA)

Среди указанных в табл. 5.2 самыми важными являются параметры настройки Wi-Fi, MQTT и обновления по воздуху (OTA), так как они позволяют приложению установить соединение с сетью и брокером MQTT, а также получать обновления специализированного ПО без необходимости прошивки микроконтроллера через его последовательный интерфейс.

Основной модуль

Основной файл исходного кода, в котором расположена точка входа в приложение, не содержит никаких неожиданностей:

```
#include "ota_core.h"
void onInit() {
    //
}
void init() {
        OtaCore::init(onInit);
}
```

С помощью класса OtaCore, содержащего основную логику приложения, вызывается его статическая функция инициализации с предоставлением функции обратного вызова, если потребуется выполнение какого-либо дальнейшего фраг-

мента логики после завершения основным классом настройки сети, протокола MQTT и прочей функциональности.

Класс OtaCore

В этом классе выполняется настройка всей основной функциональности сетевой среды для специализированных функциональных модулей. Кроме того, обеспечивается поддержка полезных вспомогательных функций для журналирования и работы протокола MQTT. В этом классе также содержится основной механизм обработки команд, принимаемых по протоколу MQTT.

```
#include <user_config.h>
#include <SmingCore/SmingCore.h>
```

Здесь включаются два заголовочных файла, необходимых для использования рабочей среды Sming – для поддержки SDK (`user_config.h`) и для поддержки самой среды Sming (`SmingCore.h`). Также определяется количество директив препроцессора для использования упрощенного сетевого стека с открытым исходным кодом LwIP (Lightweight IP stack) и для устранения некоторых проблем в официальном комплекте SDK.

Также следует отметить использование заголовочного файла `esp_cplusplus.h`, включаемого косвенным путем. В его исходном коде реализованы функции (операторы) `new` и `delete`, а также некоторые обработчики, связанные с функциональностью рассматриваемого класса, как, например, `vtables` при использовании виртуальных классов. Тем самым обеспечивается совместимость с библиотекой STL.

```
enum {
          LOG_ERROR = 0,
          LOG_WARNING,
          LOG_INFO,
          LOG_DEBUG,
          LOG_TRACE,
          LOG_XTRACE
};

enum ESP8266_pins {
          ESP8266_gpio00 = 0x00001,     // Flash
          ESP8266_gpio01 = 0x00002,     // TXD 0
          ESP8266_gpio02 = 0x00004,     // TXD 1
          ESP8266_gpio03 = 0x00008,     // RXD 0
          ESP8266_gpio04 = 0x00010,     //
          ESP8266_gpio05 = 0x00020,     //
          ESP8266_gpio09 = 0x00040,     // SDD2 (QDIO Flash)
          ESP8266_gpio10 = 0x00080,     // SDD3 (QDIO Flash)
          ESP8266_gpio12 = 0x00100,     // HMISO (SDO)
          ESP8266_gpio13 = 0x00200,     // HMOSI (SDI)
          ESP8266_gpio14 = 0x00400,     // SCK
          ESP8266_gpio15 = 0x00800,     // HCS
          ESP8266_gpio16 = 0x01000,     // User, Wake
          ESP8266_mosi = 0x02000,
          ESP8266_miso = 0x04000,
          ESP8266_sclk = 0x08000,
          ESP8266_cs = 0x10000
};
```

Приведенные выше два перечисления определяют уровни журналирования, контакты интерфейса GPIO и прочие контакты микроконтроллера ESP8266, которые можно использовать. Значения в перечислении контактов ESP8266 соответствуют позициям в битовой маске.

```
#define SCL_PIN 5
#define SDA_PIN 4
```

Здесь определяются жестко фиксированные контакты для шины I2C. Они соответствуют контактам 4 и 5 интерфейса GPIO, также помеченным как D1 и D2 на платах NodeMCU. Главная причина предварительного определения этих контактов заключается в том, что это два из нескольких безопасных контактов микроконтроллера ESP8266.

Многие контакты микроконтроллера ESP8266 будут менять уровни во время запуска перед стабилизацией, и это может привести к непредсказуемому и нежелательному поведению любого подключенного периферийного устройства.

```
typedef void (*topicCallback)(String);
typedef void (*onInitCallback)();
```

Определяются два указателя на функции: один должен использоваться функциональными модулями при необходимости регистрации темы публикации по протоколу MQTT вместе с функцией обратного вызова; другой указатель представляет собой обратный вызов, который мы видели ранее в основной функции (main).

```
class OtaCore {
        static Timer procTimer;
        static rBootHttpUpdate* otaUpdater;
        static MqttClient* mqtt;
        static String MAC;
        static HashMap<String, topicCallback>* topicCallbacks;
        static HardwareSerial Serial1;
        static String location;
        static String version;
        static int sclPin;
        static int sdaPin;
        static bool i2c_active;
        static bool spi_active;
        static uint32 esp8266_pins;

        static void otaUpdate();
        static void otaUpdate_CallBack(rBootHttpUpdate& update, bool result);
        static void startMqttClient();
        static void checkMQTTDisconnect(TcpClient& client, bool flag);
        static void connectOk(IPAddress ip, IPAddress mask, IPAddress gateway);
        static void connectFail(String ssid, uint8_t ssidLength, uint8_t *bssid,
                                uint8_t reason);
        static void onMqttReceived(String topic, String message);
        static void updateModules(uint32 input);
        static bool mapGpioToBit(int pin, ESP8266_pins &addr);

public:
        static bool init(onInitCallback cb);
        static bool registerTopic(String topic, topicCallback cb);
```

```
        static bool deregisterTopic(String topic);
        static bool publish(String topic, String message, int qos = 1);
        static void log(int level, String msg);
        static String getMAC() { return OtaCore::MAC; }
        static String getLocation() { return OtaCore::location; }
        static bool starti2c();
        static bool startSPI();
        static bool claimPin(ESP8266_pins pin);
        static bool claimPin(int pin);
        static bool releasePin(ESP8266_pins pin);
        static bool releasePin(int pin);
};
```

Это объявление класса `OtaCore` само по себе дает определенное общее представление о предоставляемой им функциональности. В первую очередь следует отметить, что класс полностью статический. Это обеспечивает немедленную инициализацию функциональности класса в самом начале работы специализированного ПО, а также доступность в глобальном контексте без необходимости создания конкретных экземпляров этого класса.

Здесь также наблюдается первое использование типа `uint32`, который вместе с другими целочисленными типами определен способом, похожим на определение в заголовочном файле `cstdint`.

Далее рассматриваем реализацию:

```
#include <ota_core.h>

#include "base_module.h"

#define SPI_SCLK 14
#define SPI_MOSI 13
#define SPI_MISO 12
#define SPI_CS 15

Timer OtaCore::procTimer;
rBootHttpUpdate* OtaCore::otaUpdater = 0;
MqttClient* OtaCore::mqtt = 0;
String OtaCore::MAC;
HashMap<String, topicCallback>* OtaCore::topicCallbacks = new
HashMap<String, topicCallback>();
HardwareSerial OtaCore::Serial1(UART_ID_1); // UART 0 - 'Serial'.
String OtaCore::location;
String OtaCore::version = VERSION;
int OtaCore::sclPin = SCL_PIN; // по умолчанию
int OtaCore::sdaPin = SDA_PIN; // по умолчанию
bool OtaCore::i2c_active = false;
bool OtaCore::spi_active = false;
uint32 OtaCore::esp8266_pins = 0x0;
```

Сюда включен заголовочный файл класса `BaseModule`, чтобы в дальнейшем можно было вызвать его функцию инициализации после завершения настройки основной функциональности. Здесь также инициализируются статические члены класса с присваиванием ряда значений по умолчанию, где это уместно.

Следует отметить инициализацию объекта второго последовательного интерфейса в дополнение к экземпляру объекта `Serial` по умолчанию. Эти объекты со-

ответствуют первой (UART0, Serial) и второй (UART1, Serial1) схемам UART в микроконтроллере ESP8266.

В более старых версиях Sming файловые функции, связанные с SPIFFS, испытывали затруднения при работе с бинарными данными (из-за предположения о внутреннем представлении строк с завершающим нулевым символом), поэтому были добавлены приведенные ниже дополнительные альтернативные функции. Их имена представляют собой слегка измененную версию имен исходных функций для предотвращения конфликта имен.

Поскольку сертификаты TLS и прочие файлы с бинарными данными, хранимые в SPIFFS, должны корректно записываться, считываться и обрабатываться специализированным ПО, здесь необходимо было найти некий компромисс.

```
String getFileContent(const String fileName) {
         file_t file = fileOpen(fileName.c_str(), eFO_ReadOnly);

         fileSeek(file, 0, eSO_FileEnd);
         int size = fileTell(file);
         if (size <= 0)    {
                fileClose(file);
                return "";
         }

         fileSeek(file, 0, eSO_FileStart);
         char* buffer = new char[size + 1];
         buffer[size] = 0;
         fileRead(file, buffer, size);
         fileClose(file);
         String res(buffer, size);
         delete[] buffer;
         return res;
}
```

Эта функция считывает все содержимое заданного файла в экземпляр строки `String` и возвращает полученную строку.

```
void setFileContent(const String &fileName, const String &content) {
          file_t file = fileOpen(fileName.c_str(), eFO_CreateNewAlways | eFO_WriteOnly);
          fileWrite(file, content.c_str(), content.length());
          fileClose(file);
}
```

Эта функция заменяет существующее содержимое заданного файла на новые данные, представленные в экземпляре строки `String`.

```
bool readIntoFileBuffer(const String filename, char* &buffer, unsigned int &size) {
         file_t file = fileOpen(filename.c_str(), eFO_ReadOnly);

         fileSeek(file, 0, eSO_FileEnd);
         size = fileTell(file);
         if (size == 0)    {
                fileClose(file);
                return true;
         }

         fileSeek(file, 0, eSO_FileStart);
```

```
        buffer = new char[size + 1];
        buffer[size] = 0;
        fileRead(file, buffer, size);
        fileClose(file);
        return true;
}
```

Эта функция похожа на функцию `getFileContent()`, но возвращает простой буфер символов вместо экземпляра строки `String`. Она в основном используется для считывания данных сертификата, которые передаются в библиотеку TLS, написанную на языке C (библиотека axTLS), когда преобразование в экземпляр `String` становится неэффективным с учетом копирования, особенно если сертификаты имеют размер в несколько килобайтов.

Функция инициализации класса `OtaCore`:

```
bool OtaCore::init(onInitCallback cb) {
        Serial.begin(9600);

        Serial1.begin(SERIAL_BAUD_RATE);
        Serial1.systemDebugOutput(true);
```

Сначала инициализируются две схемы UART (последовательные интерфейсы) на плате NodeMCU. Хотя в микроконтроллере ESP8266 официально объявлено о наличии двух схем UART, вторая схема состоит только из линии выходного сигнала TX (по умолчанию GPIO 2). Поэтому необходимо оставить свободной первую схему UART для применения приложениями, требующими монопольного использования последовательной линии, например некоторыми сенсорными датчиками.

Таким образом, первая схема UART (`Serial`) инициализируется так, чтобы в дальнейшем можно было использовать ее в функциональных модулях, тогда как вторая схема UART (Serial1) инициализируется значением скорости по умолчанию 115 200 бод вместе с системным отладочным выводом (стек WiFi/IP и т. п.), также направляемым на эту последовательную выходную линию. Следовательно, этот второй последовательный интерфейс будет использоваться исключительно для вывода информации в журнал.

```
  BaseModule::init();
```

Далее инициализируется статический класс `BaseModule`. При этом все функциональные модули, задействованные в данном специализированном ПО, должны быть зарегистрированы, что позволяет активизировать их в дальнейшем.

```
    int slot = rboot_get_current_rom();
    u32_t offset;
    if (slot == 0) { offset = 0x100000; }
    else { offset = 0x300000; }
    spiffs_mount_manual(offset, 65536);
```

Автоматическое монтирование файловой системы SPIFFS при использовании загрузчика rBoot не выполняется в более старых версиях Sming, поэтому необходимо монтирование вручную. Для этого мы получаем от загрузчика rBoot текущий слот специализированного ПО, используя который, можно выбрать правильное смещение либо в начале второго мегабайта ПЗУ, либо в начале четвертого мегабайта.

После определения требуемого смещения используется функция ручного монтирования SPIFFS с передачей в нее значения выбранного смещения и размера секции SPIFFS. Теперь можно пользоваться созданным хранилищем для чтения и записи данных.

```
Serial1.printf("\r\nSDK: v%s\r\n", system_get_sdk_version());
Serial1.printf("Free Heap: %d\r\n", system_get_free_heap_size());
Serial1.printf("CPU Frequency: %d MHz\r\n", system_get_cpu_freq());
Serial1.printf("System Chip ID: %x\r\n", system_get_chip_id());
Serial1.printf("SPI Flash ID: %x\r\n", spi_flash_get_id());
```

Далее некоторая системная информация выводится в последовательную выходную линию. Здесь содержится версия скомпилированного комплекта ESP8266 SDK, текущий размер свободной динамической памяти (кучи), частота ЦПУ, 32-битовый идентификатор микроконтроллера и идентификатор микросхемы ПЗУ SPI.

```
mqtt = new MqttClient(MQTT_HOST, MQTT_PORT, onMqttReceived);
```

В динамической памяти создается новый экземпляр клиента MQTT с указанием функции обратного вызова, обращение к которой будет выполняться при получении нового сообщения. Значения хоста и порта брокера MQTT подставляются препроцессором на основе данных, добавленных в пользовательский Makefile текущего проекта.

```
        Serial1.printf("\r\nCurrently running rom %d.\r\n", slot);

        WifiStation.enable(true);
        WifiStation.config(WIFI_SSID, WIFI_PWD);
        WifiStation.connect();
        WifiAccessPoint.enable(false);

       WifiEvents.onStationGotIP(OtaCore::connectOk);
       WifiEvents.onStationDisconnect(OtaCore::connectFail);

         (*cb)();
}
```

На заключительном этапе инициализации выводится информация о текущем слоте специализированного ПО, из которого выполняется запуск приложения, затем разрешается работа клиента Wi-Fi при запрещении функциональности беспроводной точки доступа WAP (wireless access point). Клиенту Wi-Fi сообщается о соединении с Wi-Fi SSID с использованием учетных данных, предварительно определенных в Makefile.

Наконец, определяются обработчики успешно установленного Wi-Fi-соединения и неудачных попыток соединения перед обращением к функции обратного вызова, которая была передана как параметр.

После обновления по воздуху (OTA) специализированного ПО выполняется обращение к следующей функции обратного вызова:

```
void OtaCore::otaUpdate_CallBack(rBootHttpUpdate& update, bool result) {
        OtaCore::log(LOG_INFO, "In OTA callback...");
        if (result == true) { // успешно
              uint8 slot = rboot_get_current_rom();
              if (slot == 0) { slot = 1; } else { slot = 0; }
```

```
                Serial1.printf("Firmware updated, rebooting to ROM slot %d...\r\n", slot);
                OtaCore::log(LOG_INFO, "Firmware updated, restarting...");
                rboot_set_current_rom(slot);
                System.restart();
        }
        else {
                OtaCore::log(LOG_ERROR, "Firmware update failed.");
        }
}
```

В этом обратном вызове изменяется активный слот ПЗУ, если обновление по воздуху завершилось успешно, после чего выполняется перезагрузка системы. В противном случае в журнал записывается сообщение об ошибке обновления и перезагрузка не выполняется.

Далее следует несколько функций, связанных с поддержкой протокола MQTT.

```
bool OtaCore::registerTopic(String topic, topicCallback cb) {
        OtaCore::mqtt->subscribe(topic);
        (*topicCallbacks)[topic] = cb;
        return true;
}

bool OtaCore::deregisterTopic(String topic) {
        OtaCore::mqtt->unsubscribe(topic);
        if (topicCallbacks->contains(topic)) {
                topicCallbacks->remove(topic);
        }
        return true;
}
```

Эти функции позволяют функциональным модулям соответственно регистрироваться и отменять регистрацию заголовка публикации MQTT с использованием ранее определенной функции обратного вызова. Брокер MQTT вызывается с запросом на подписку или на отмену подписки, а экземпляр `HashMap` обновляется соответствующим образом.

```
bool OtaCore::publish(String topic, String message, int qos /* = 1 */) {
        OtaCore::mqtt->publishWithQoS(topic, message, qos);
        return true;
}
```

Каждый функциональный модуль может публиковать MQTT-сообщение в любой теме публикации, используя эту функцию. Параметр качества обслуживания QoS (Quality of Service) определяет режим публикации. По умолчанию сообщения публикуются в режиме сохранения (retain), то есть брокер будет сохранять самое последнее опубликованное сообщение для конкретной темы.

Точка входа в режим функциональности обновления по воздуху (OTA) расположена в следующей функции:

```
void OtaCore::otaUpdate() {
        OtaCore::log(LOG_INFO, "Updating firmware from URL: " + String(OTA_URL));

        if (otaUpdater) { delete otaUpdater; }
        otaUpdater = new rBootHttpUpdate();
        rboot_config bootconf = rboot_get_config();
```

```
        uint8 slot = bootconf.current_rom;
        if (slot == 0) { slot = 1; } else { slot = 0; }

        otaUpdater->addItem(bootconf.roms[slot], OTA_URL + MAC);

        otaUpdater->setCallback(OtaCore::otaUpdate_CallBack);
        otaUpdater->start();
}
```

Для обновления по воздуху необходимо создать чистый экземпляр `rBootHttpUpdate`. Затем нужно сконфигурировать этот экземпляр с помощью подробных данных текущего слота специализированного ПО. Для этого извлекается конфигурация из загрузчика rBoot и по ней определяется номер текущего слота специализированного ПО. Полученная информация используется для определения номера другого слота специализированного ПО, необходимого для функции обновления по воздуху.

В рассматриваемом примере выполняется конфигурирование только для обновления слота специализированного ПО, но можно также обновлять секцию SPIFFS для другого слота тем же способом. Специализированное ПО передается по протоколу HTTP с жестко заданного URL, определенного ранее. MAC-адрес микроконтроллера ESP8266 присоединяется как суффикс к концу запроса в форме строкового параметра, поэтому сервер обновлений точно знает, какой образ специализированного ПО соответствует данной системе.

После определения функции обратного вызова, которая рассматривалась выше, начинается процедура обновления.

```
void OtaCore::checkMQTTDisconnect(TcpClient& client, bool flag) {
        if (flag == true) { Serial1.println("MQTT Broker disconnected.");
        }
        else {
              String tHost = MQTT_HOST;
              Serial1.println("MQTT Broker " + tHost + " unreachable."); }
        procTimer.initializeMs(2 * 1000,
OtaCore::startMqttClient).start();
}
```

Здесь определяется обработчик разрыва соединения по протоколу MQTT. Он вызывается, когда соединение с брокером MQTT аварийно закрывается, так что можно попытаться восстановить соединение после двухсекундной задержки.

Для параметра `flag` устанавливается значение `true`, если ранее соединение было установлено, или значение `false`, если неудачно завершилась начальная попытка установления соединения с брокером MQTT (нет доступа к сети, некорректный адрес и т. п.).

Следующая функция предназначена для конфигурирования и запуска MQTT-клиента.

```
void OtaCore::startMqttClient() {
        procTimer.stop();
        if (!mqtt->setWill("last/will", "The connection from this device is lost:(", 1,
            true)) {
              debugf("Unable to set the last will and testament.
                            Most probably there is not enough memory on the device.");
        }
```

Таймер procTimer останавливается, если он запущен в случае вызова из таймера восстановления соединения. Далее устанавливается так называемая «последняя воля и завещание» (last will and testament – LWT) для этого устройства, позволяющая определить сообщение, которое брокер MQTT опубликует при потере соединения с клиентом (то есть с нами).

Также определяются три различных пути выполнения, но скомпилирован будет только один из них, в зависимости от применения или неприменения протокола TLS (SSL), процедуры регистрации (входа) с комбинацией имя пользователя/пароль или анонимного доступа:

```
#ifdef ENABLE_SSL
        mqtt->connect(MAC, MQTT_USERNAME, MQTT_PWD, true);
        mqtt->addSslOptions(SSL_SERVER_VERIFY_LATER);

        Serial1.printf("Free Heap: %d\r\n", system_get_free_heap_size());

        if (!fileExist("esp8266.client.crt.binary")) {
              Serial1.println("SSL CRT file is missing: esp8266.client.crt.binary.");
              return;
        }
        else if (!fileExist("esp8266.client.key.binary")) {
              Serial1.println("SSL key file is missing: esp8266.client.key.binary.");
              return;
        }

        unsigned int crtLength, keyLength;
        char* crtFile;
        char* keyFile;
        readIntoFileBuffer("esp8266.client.crt.binary", crtFile, crtLength);
        readIntoFileBuffer("esp8266.client.key.binary", keyFile, keyLength);
        Serial1.printf("keyLength: %d, crtLength: %d.\n", keyLength, crtLength);
        Serial1.printf("Free Heap: %d\r\n", system_get_free_heap_size());
        if (crtLength < 1 || keyLength < 1) {
              Serial1.println("Failed to open certificate and/or key file.");
              return;
        }

        mqtt->setSslClientKeyCert((const uint8_t*) keyFile, keyLength, (const uint8_t*)
                                  crtFile, crtLength, 0, true);
        delete[] keyFile;
        delete[] crtFile;

        Serial1.printf("Free Heap: %d\r\n", system_get_free_heap_size());
```

При наличии сертификатов TLS соединение с брокером MQTT устанавливается с использованием физического адреса MAC как идентификатора клиента, затем разрешается применение протокола SSL для этого соединения. Размер доступного пространства динамической памяти (кучи) выводится через последовательную линию журналирования для целей отладки. Обычно в этой точке программы должно оставаться свободным около 25 Кб ОЗУ, то есть вполне достаточное пространство для хранения сертификата и ключа в памяти, а также для буферов RX и TX для процедуры обмена рукопожатиями по протоколу TLS, если эта процедура сконфигурирована в конечной точке SSL-соединения. Вполне приемлемый

размер при использовании опции размера фрагмента SSL. Более подробно этот вопрос будет рассматриваться в главе 9 «Пример: мониторинг и управление внутренним микроклиматом в здании».

Далее считываются файлы сертификата в DER-кодировке (бинарный) и ключа из файловой системы SPIFFS. Эти файлы имеют постоянные имена. Для каждого файла выводится его размер, а также текущий размер свободной динамической памяти. Если какой-либо из этих файлов имеет нулевой размер, то попытка чтения считается неудачной, и попытка установления соединения отменяется.

При удачном считывании файлов данные ключа и сертификата используются для установления соединения по протоколу MQTT, что должно привести к успешной процедуре обмена рукопожатиями и установлению зашифрованного соединения с брокером MQTT.

После удаления файлов данных ключа и сертификата выводится текущий размер свободной динамической памяти для проверки успешности очистки.

```
#elif defined USE_MQTT_PASSWORD
         mqtt->connect(MAC, MQTT_USERNAME, MQTT_PWD);
```

При использовании имени и пароля по протоколу MQTT для регистрации (входа) в брокере необходимо вызвать предыдущую функцию в экземпляре клиента MQTT с передачей в нее MAC-адреса как идентификатора клиента, а также имени пользователя и пароля.

```
#else
         mqtt->connect(MAC);
#endif
```

Если разрешен анонимный доступ, то просто устанавливается соединение с брокером и передается MAC-адрес как идентификатор клиента.

```
         mqtt->setCompleteDelegate(checkMQTTDisconnect);

         mqtt->subscribe(MQTT_PREFIX"upgrade");
         mqtt->subscribe(MQTT_PREFIX"presence/tell");
         mqtt->subscribe(MQTT_PREFIX"presence/ping");
         mqtt->subscribe(MQTT_PREFIX"presence/restart/#");
         mqtt->subscribe(MQTT_PREFIX"cc/" + MAC);

         delay(100);

         mqtt->publish(MQTT_PREFIX"cc/config", MAC);
}
```

Здесь в первую очередь устанавливается обработчик разрыва MQTT-соединения. Затем оформляется подписка на ряд тем публикаций, на которые предполагается отвечать (реагировать). Все эти темы связаны с управлением функциональностью используемого специализированного ПО, позволяющего системе обрабатывать запросы и выполнять операции конфигурации по протоколу MQTT.

После подписки создается небольшой (100 мс) интервал ожидания, чтобы предоставить брокеру время на обработку заявленных подписок, прежде чем мы начнем публиковать сообщения в централизованной теме оповещения, используя собственный MAC-адрес, чтобы позволить всем заинтересованным клиентам и серверам узнать о том, что данная система перешла в режим онлайн.

Далее следуют обработчики соединения по протоколу Wi-Fi.

```
void OtaCore::connectOk(IPAddress ip, IPAddress mask, IPAddress gateway) {
          Serial1.println("I'm CONNECTED. IP: " + ip.toString());

          MAC = WifiStation.getMAC();
          Serial1.printf("MAC: %s.\n", MAC.c_str());

          if (fileExist("location.txt")) {
                location = getFileContent("location.txt");
          }
          else {
                location = MAC;
          }
          if (fileExist("config.txt")) {
                String configStr = getFileContent("config.txt");
                uint32 config;
                configStr.getBytes((unsigned char*) &config, sizeof(uint32), 0);
                updateModules(config);
          }
          startMqttClient();
}
```

Этот обработчик вызывается при успешном установлении соединения с сетью Wi-Fi, сконфигурированной с использованием предоставленных учетных данных. После установления соединения копия MAC-адреса (`MAC`) сохраняется в памяти как уникальный идентификатор клиента.

Специализированное ПО также обеспечивает сохранение определяемой пользователем строки как наше местоположение или другой аналогичный идентификатор. Если такой идентификатор был определен предварительно, то он загружается из файловой системы SPIFFS и используется, в противном случае строкой определения местоположения является MAC-адрес (`MAC`).

Далее 32-битовая маска, определяющая конфигурацию функциональных модулей, загружается из файловой системы SPIFFS, если существует соответствующий файл. При отсутствии файла битовой маски все функциональные модули в начале работы остаются неактивизированными. Если файл битовой маски считан успешно, то маска передается в функцию `updateModules()` и заданные модули будут активизированы.

```
void OtaCore::connectFail(String ssid, uint8_t ssidLength, uint8_t* bssid, uint8_t reason) {
          Serial1.println("I'm NOT CONNECTED. Need help :(");
          debugf("Disconnected from %s. Reason: %d", ssid.c_str(), reason);

          WDT.alive();

          WifiEvents.onStationGotIP(OtaCore::connectOk);
          WifiEvents.onStationDisconnect(OtaCore::connectFail);
}
```

Если попытка соединения с сетью Wi-Fi завершилась неудачно, то этот факт регистрируется в журнале, затем сторожевому таймеру микроконтроллера сообщается о том, что мы остаемся в нормальном рабочем режиме, чтобы предотвратить программную перезагрузку, прежде чем будет выполнена повторная попытка соединения с сетью.

На этом все функции инициализации завершаются. Далее следуют функции, используемые в нормальном рабочем режиме, начиная с обработчика MQTT-сообщений.

```
void OtaCore::onMqttReceived(String topic, String message) {
        Serial1.print(topic);
        Serial1.print(":\n");
        Serial1.println(message);

        log(LOG_DEBUG, topic + " - " + message);

        if (topic == MQTT_PREFIX"upgrade" && message == MAC) {
               otaUpdate();
        }
        else if (topic == MQTT_PREFIX"presence/tell") {
               mqtt->publish(MQTT_PREFIX"presence/response", MAC);
        }
        else if (topic == MQTT_PREFIX"presence/ping") {
              mqtt->publish(MQTT_PREFIX"presence/pong", MAC);
        }
        else if (topic == MQTT_PREFIX"presence/restart" && message == MAC) {
              System.restart();
        }
        else if (topic == MQTT_PREFIX"presence/restart/all") {
              System.restart();
        }
```

Этот обратный вызов регистрируется при первоначальном создании экземпляра MQTT-клиента. Каждый раз, когда в теме, на которую мы подписаны в брокере, появляется новое сообщение, мы получаем оповещение об этом, а функция обратного вызова принимает строку, содержащую тему и другую строку, в которой содержится само сообщение (полезная нагрузка).

Можно сравнить эту тему с темами, в которых мы зарегистрированы, и выполнить требуемую операцию: либо обновление по воздуху (если в сообщении указан наш MAC-адрес), либо ответ на запрос ping, то есть ответ pong с указанием нашего MAC-адреса, либо перезагрузка системы.

Следующая тема определяет более часто выполняемые операции сопровождения и обслуживания, позволяя конфигурировать активные функциональные модули, устанавливать строку местоположения (локации) и запрашивать текущее состояние системы. Формат полезной нагрузки сообщения определяет командную строку, за которой следует точка с запятой, и далее строка собственно полезной нагрузки:

```
else if (topic == MQTT_PREFIX"cc/" + MAC) {
       int chAt = message.indexOf(';');
       String cmd = message.substring(0, chAt);
       ++chAt;

       String msg(((char*) &message[chAt]), (message.length() - chAt));

       log(LOG_DEBUG, msg);

       Serial1.printf("Command: %s, Message: ", cmd.c_str());
       Serial1.println(msg);
```

Мы начинаем с извлечения командной строки из строки полезной нагрузки, используя простой метод поиска подстроки. Затем считывается оставшаяся строка полезной нагрузки с учетом того, что она является бинарной строкой. Для этого используется длина оставшейся строки, а в качестве начальной позиции принимается символ, следующий непосредственно за точкой с запятой.

После извлечения командной строки и полезной нагрузки сообщения можно узнать, что именно требуется сделать:

```
if (cmd == "mod") {
      if (msg.length() != 4) {
            Serial1.printf("Payload size wasn't 4 bytes: %d\n", msg.length());
            return;
      }

      uint32 input;
      msg.getBytes((unsigned char*) &input, sizeof(uint32), 0);
      String byteStr;
      byteStr = "Received new configuration: ";
      byteStr += input;
      log(LOG_DEBUG, byteStr);
      updateModules(input);
 }
```

Эта команда определяет, какие функциональные модули должны активизироваться. В сопровождающей полезной нагрузке содержится беззнаковое 32-битовое целое число, представляющее битовую маску, для которой выполняется проверка, позволяющая убедиться в том, что действительно получены четыре байта этой маски.

В полученной битовой маске каждый бит соответствует модулю, как показано в табл. 5.3.

Таблица 5.3

Позиция (номер) бита	Имя функционального модуля
0x01	THPModule
0x02	CO2Module
0x04	JuraModule
0x08	JuraTermModule
0x10	MotionModule
0x20	PwmModule
0x40	IOModule
0x80	SwitchModule
0x100	PlantModule

Модули CO2, Jura и JuraTerm взаимоисключающие, так как все три модуля используют первую схему UART (`Serial`). Если в битовой маске указаны два или три этих модуля, то будет активизирован только первый (по порядку) модуль, а остальные игнорируются. Подробное описание и разбор этих модулей приведены в главе 9 «Пример: мониторинг и управление внутренним микроклиматом в здании».

После получения новой конфигурационной битовой маски мы передаем ее в функцию `updateModules()`.

```
    else if (cmd == "loc") {
           if (msg.length() < 1) { return; }
           if (location != msg) {
                 location = msg;
                 fileSetContent("location.txt", location);
           }
     }
```

С помощью этой команды устанавливается новая строка локации, если она отличается от текущей, и сохраняется в соответствующем файле в SPIFFS, чтобы сохранить ее после перезагрузки.

```
        else if (cmd == "mod_active") {
              uint32 active_mods = BaseModule::activeMods();
              if (active_mods == 0) {
                    mqtt->publish(MQTT_PREFIX"cc/response", MAC + ";0");
                    return;
              }
              mqtt->publish(MQTT_PREFIX"cc/response", MAC + ";" +
                            String((const char*) &active_mods, 4));
        }
        else if (cmd == "version") {
              mqtt->publish(MQTT_PREFIX"cc/response", MAC + ";" + version);
        }
        else if (cmd == "upgrade") {
              otaUpdate();
        }
  }
```

Команда `mod_active` возвращает текущую битовую маску для активных функциональных модулей, команда `version` выводит версию специализированного ПО, команда `upgrade` активизирует процедуру обновления по воздуху.

```
        else {
              if (topicCallbacks->contains(topic)) {
                    (*((*topicCallbacks)[topic]))(message);
               }
        }
}
```

Самая последняя часть в блоке `if…else` проверяет, не находится ли заданная тема в нашем списке обратных вызовов для функциональных модулей. Если тема найдена в этом списке, то выполняется соответствующий обратный вызов с передачей в него строки MQTT-сообщения.

Разумеется, это означает, что только один функциональный модуль может зарегистрироваться в конкретной теме. Поскольку подразумевается, что каждый модуль работает в собственной подтеме MQTT для разделения потока сообщений, в общем случае это не является проблемой.

```
void OtaCore::updateModules(uint32 input) {
        Serial1.printf("Input: %x, Active: %x.\n", input, BaseModule::activeMods());
```

```
        BaseModule::newConfig(input);
        if (BaseModule::activeMods() != input) {
              String content(((char*) &input), 4);
              setFileContent("config.txt", content);
        }
}
```

Эта функция весьма проста. В основном она служит для перехода к классу `BaseModule`, но, кроме того, обеспечивает своевременное сохранение файла конфигурации в файловой системе SPIFFS, записывая новую битовую маску сразу после ее изменения.

Необходимо строго контролировать и запрещать ненужные записи в файловую систему SPIFFS, так как хранилище на основе флеш-памяти имеет ограниченное количество циклов записи. Жесткое ограничение количества циклов записи может значительно продлить жизненный цикл аппаратуры, а также снизить общую нагрузку на систему.

```
bool OtaCore::mapGpioToBit(int pin, ESP8266_pins &addr) {
         switch (pin) {
               case 0:
                     addr = ESP8266_gpio00;
                     break;
               case 1:
                     addr = ESP8266_gpio01;
                     break;
               case 2:
                     addr = ESP8266_gpio02;
                     break;
               case 3:
                     addr = ESP8266_gpio03;
                     break;
               case 4:
                     addr = ESP8266_gpio04;
                     break;
               case 5:
                     addr = ESP8266_gpio05;
                     break;
               case 9:
                     addr = ESP8266_gpio09;
                     break;
               case 10:
                     addr = ESP8266_gpio10;
                     break;
               case 12:
                     addr = ESP8266_gpio12;
                     break;
               case 13:
                     addr = ESP8266_gpio13;
                     break;
               case 14:
                     addr = ESP8266_gpio14;
                     break;
               case 15:
```

```
                addr = ESP8266_gpio15;
                break;
            case 16:
                addr = ESP8266_gpio16;
                break;
            default:
                log(LOG_ERROR, "Invalid pin number specified: " + String(pin));
                return false;
        };

        return true;
}
```

Эта функция устанавливает связь номера каждого конкретного контакта используемого интерфейса GPIO с соответствующей позицией во внутренней битовой маске. При этом используется перечисление, определенное в заголовочном файле для этого класса. Применяя такое отображение, можно устанавливать состояние «используется / не используется» контактов GPIO модуля ESP8266 с помощью единственного значения типа `uint32`.

```
void OtaCore::log(int level, String msg) {
        String out(lvl);
        out += " - " + msg;
        Serial1.println(out);
        mqtt->publish(MQTT_PREFIX"log/all", OtaCore::MAC + ";" + out);
}
```

В методе ведения журнала добавляется уровень журналирования к строке сообщения перед записью ее в последовательный вывод, а также публикация этого сообщения в соответствии с протоколом MQTT. Здесь публикация выполняется в одной теме, но можно улучшить функцию ведения журнала, размещая сообщения в различных темах в зависимости от заданного уровня журналирования.

Имеет смысл внимательно прослеживать зависимость от того, для какого типа внутреннего компонента настроено наблюдение (прослушивание), и процесс вывода в журнал из систем ESP8266, использующих это специализированное ПО.

```
bool OtaCore::starti2c() {
        if (i2c_active) { return true; }

        if (!claimPin(sdaPin)) { return false; }
        if (!claimPin(sclPin)) { return false; }

        Wire.pins(sdaPin, sclPin);
        pinMode(sclPin, OUTPUT);
        for (int i = 0; i < 8; ++i) {
            digitalWrite(sclPin, HIGH);
            delayMicroseconds(3);
            digitalWrite(sclPin, LOW);
            delayMicroseconds(3);
        }

        pinMode(sclPin, INPUT);

        Wire.begin();
        i2c_active = true;
}
```

Эта функция инициализирует работу шины I2C, если шина еще не начала функционировать. Выполняется попытка регистрации контактов, предназначенных для использования шиной I2C. Если контакты доступны, то линия таймера (SCL) устанавливается в режим вывода и первый импульс повторяется восемь раз для «размораживания» всех I2C-устройств на этой шине.

После передачи импульсов по линии таймера активизируются все контакты шины I2C и отмечается активное состояние шины.

Устройства I2C могут оставаться в «замороженном» состоянии, если электропитание на микроконтроллер подается циклически, а на устройства I2C электропитание не подается и они остаются в неопределенном состоянии. При таком режиме пульсации необходимо гарантировать, что система в конечном итоге не перейдет в нерабочее состояние, поэтому требуется ручное вмешательство:

```
bool OtaCore::startSPI() {
    if (spi_active) { return true; }
    if (!claimPin(SPI_SCLK)) { return false; }
    if (!claimPin(SPI_MOSI)) { return false; }
    if (!claimPin(SPI_MISO)) { return false; }
    if (!claimPin(SPI_CS)) { return false; }
    SPI.begin();
    spi_active = true;
 }
```

Инициализация шины последовательного интерфейса SPI похожа на инициализацию шины I2C, но в данном случае отсутствует механизм восстановления.

```
bool OtaCore::claimPin(int pin) {
          ESP8266_pins addr;
          if (!mapGpioToBit(pin, addr)) { return false; }

          return claimPin(addr);
    }

bool OtaCore::claimPin(ESP8266_pins pin) {
          if (esp8266_pins & pin) {
                log(LOG_ERROR, "Attempting to claim an already claimed pin:" + String(pin));
                log(LOG_DEBUG, String("Current claimed pins: ") + String(esp8266_pins));
                return false;
          }

          log(LOG_INFO, "Claiming pin position: " + String(pin));

          esp8266_pins |= pin;

          log(LOG_DEBUG, String("Claimed pin configuration: ") + String(esp8266_pins));

          return true;
}
```

Эта перегружаемая функция используется для регистрации контакта GPIO функциональным модулем перед началом его работы, чтобы исключить попытку двух модулей одновременно воспользоваться одними и теми же контактами. Первая версия принимает номер контакта (GPIO) и использует функцию установления связи (отображения), рассмотренную выше, для получения адреса (позиции)

соответствующего бита в битовой маске esp8266_pins, прежде чем передать его во вторую версию функции.

В этой функции перечисление, определяющее контакты, используется для выполнения сравнения с помощью побитовой операции AND. Если бит еще не был установлен, то он устанавливается, и возвращается значение true. В противном случае функция возвращает значение false, то есть вызывающий модуль оповещается о невозможности продолжения его инициализации.

```
bool OtaCore::releasePin(int pin) {
          ESP8266_pins addr;
          if (!mapGpioToBit(pin, addr)) { return false; }

          return releasePin(addr);
}

bool OtaCore::releasePin(ESP8266_pins pin) {
          if (!(esp8266_pins & pin)) {
                log(LOG_ERROR, "Attempting to release a pin which has not been set: " +
                                String(pin));
                return false;
          }

          esp8266_pins &= ~pin;

          log(LOG_INFO, "Released pin position: " + String(pin));
          log(LOG_DEBUG, String("Claimed pin configuration: ") + String(esp8266_pins));

          return true;
 }
```

Эта перегружаемая функция используется для освобождения контакта GPIO, когда функциональный модуль завершает свою работу. Функция работает аналогично функции регистрации контакта. Здесь также используется функция установления связи (отображения) для получения адреса (позиции) бита в маске в первой версии. Затем вторая версия выполняет побитовую операцию AND для проверки текущего состояния бита. Если бит действительно установлен, то он сбрасывается в состояние off с помощью комбинации оператора присваивания и побитового оператора AND.

Класс BaseModule

Этот класс содержит логику регистрации и отслеживания текущего состояния (активен / не активен) каждого функционального модуля. Содержимое заголовочного файла для этого класса приведено ниже.

```
#include "ota_core.h"

enum ModuleIndex {
   MOD_IDX_TEMPERATURE_HUMIDITY = 0,
   MOD_IDX_CO2,
   MOD_IDX_JURA,
   MOD_IDX_JURATERM,
   MOD_IDX_MOTION,
   MOD_IDX_PWM,
   MOD_IDX_IO,
```

```
    MOD_IDX_SWITCH,
    MOD_IDX_PLANT
};

typedef bool (*modStart)();
typedef bool (*modShutdown)();
```

Включение заголовочного файла класса `OtaCore` позволяет воспользоваться функцией ведения журнала. Здесь создается еще одно перечисление, определяющее соответствие каждого конкретного функционального модуля определенному биту в битовой маске функциональных модулей (`active_mods`).

Далее определяются указатели на функции, предназначенные соответственно для запуска и завершения работы функционального модуля. Сами функции будут определяться каждым конкретным функциональным модулем при регистрации.

```
#include "thp_module.h"
#include "jura_module.h"
#include "juraterm_module.h"
#include "co2_module.h"
#include "motion_module.h"
#include "pwm_module.h"
#include "io_module.h"
#include "switch_module.h"
#include "plant_module.h"
```

Это заголовочные файлы для всех функциональных модулей, входящих в текущий комплект специализированного ПО. Поскольку для рассматриваемого проекта нам необходим только модуль растения (plant), можно было бы закомментировать все заголовочные файлы для остальных модулей, а также блоки их инициализации в функции инициализации этого класса.

В любом случае это не должно повлиять на итоговый образ специализированного ПО, за исключением того, что мы не сможем воспользоваться закомментированными модулями, как если бы они не существовали вовсе.

Далее следует объявление самого класса:

```
class BaseModule {
        struct SubModule {
              modStart start;
              modShutdown shutdown;
              ModuleIndex index;
              uint32 bitmask;
              bool started;
        };

        static SubModule modules[32];
        static uint32 active_mods;
        static bool initialized;
        static uint8 modcount;

public:
        static void init();
        static bool registerModule(ModuleIndex index, modStart start,
                                   modShutdown shutdown);
```

```
        static bool newConfig(uint32 config);
        static uint32 activeMods() { return active_mods; }
};
```

Каждый функциональный модуль имеет внутреннее представление в форме экземпляра класса SubModule, который более подробно будет описан ниже в определении класса.

```
#include "base_module.h"

BaseModule::SubModule BaseModule::modules[32];
uint32 BaseModule::active_mods = 0x0;
bool BaseModule::initialized = false;
uint8 BaseModule::modcount = 0;
```

Поскольку это статический класс, сразу инициализируются переменные-члены класса. Здесь же создается массив с размером, достаточным для размещения 32 экземпляров SubModule и соответствующим полному размеру битовой маски. В текущий момент активных модулей нет, поэтому переменные-члены инициализируются нулями и логическими значениями false.

```
void BaseModule::init() {
    CO2Module::initialize();
    IOModule::initialize();
    JuraModule::initialize();
    JuraTermModule::initialize();
    MotionModule::initialize();
    PlantModule::initialize();
    PwmModule::initialize();
    SwitchModule::initialize();
    THPModule::initialize();
}
```

При вызове этой функции в классе OtaCore (см. предыдущий раздел) начиналась процедура регистрации функциональных модулей, определенных здесь. Если выборочно удалить или закомментировать некоторые модули в этой функции, то можно исключить их из итогового образа специализированного ПО. Модули, вызовы которых остаются в этой функции, будут, в свою очередь, вызывать следующую функцию, чтобы зарегистрироваться:

```
bool BaseModule::registerModule(ModuleIndex index, modStart start, modShutdown shutdown) {
        if (!initialized) {
              for (uint8 i = 0; i < 32; i++) {
                    modules[i].start = 0;
                    modules[i].shutdown = 0;
                    modules[i].index = index;
                    modules[i].bitmask = (1 << i);
                    modules[i].started = false;
              }

              initialized = true;
        }

        if (modules[index].start) {
              return false;
```

```
        }

        modules[index].start = start;
        modules[index].shutdown = shutdown;
        ++modcount;

        return true;
}
```

Первый функциональный модуль, который вызывает эту функцию, запускает процедуру инициализации массива `SubModule`, устанавливая для всех его элементов нейтральные значения, а также содавая битовую маску для соответствующей (своей) позиции в массиве. Это позволяет обновить битовую маску `active_mods`, как мы увидим несколько позже.

После инициализации массива проверяется, был ли уже зарегистрирован какой-либо модуль в этой позиции массива. Если обнаружена регистрация модуля, то возвращается значение `false`. Если позиция свободна, то для текущего модуля регистрируются указатели на функции запуска и завершения работы, потом счетчик активных модулей увеличивается на единицу и возвращается значение `true`.

```
bool BaseModule::newConfig(uint32 config) {
    OtaCore::log(LOG_DEBUG, String("Mod count: ") + String(modcount));
    uint32 new_config = config ^ active_mods;
    if (new_config == 0x0) {
        OtaCore::log(LOG_INFO, "New configuration was 0x0. No change.");
        return true;
    }
    OtaCore::log(LOG_INFO, "New configuration: " + new_config);
    for (uint8 i = 0; i < 32; ++i) {
        if (new_config & (1 << i)) {
            OtaCore::log(LOG_DEBUG, String("Toggling module: ") + String(i));
            if (modules[i].started) {
                if ((modules[i]).shutdown()) {
                    modules[i].started = false;
                    active_mods ^= modules[i].bitmask;
                }
                else {
                    OtaCore::log(LOG_ERROR, "Failed to shutdown module.");
                    return false;
                }
            }
            else {
                if ((modules[i].start) && (modules[i]).start()) {
                    modules[i].started = true;
                    active_mods |= modules[i].bitmask;
                }
                else {
                    OtaCore::log(LOG_ERROR, "Failed to start module.");
                    return false;
                }
            }
        }
```

```
    }
    return true;
}
```

Входным параметром для этой функции является битовая маска, извлеченная из содержимого полезной нагрузки в объекте `OtaCore`. Здесь используется логическая операция исключающего или `XOR` для наложения на битовую маску активных модулей, чтобы получить новую маску, учитывающую все сделанные изменения. Если результат нулевой, то обе сравниваемые маски идентичны, поэтому никаких дальнейших действий не требуется и выполняется выход из функции.

Полученная таким способом битовая маска типа `uint32` показывает, какие модули должны быть подключены или отключены. Для этого проверяется каждый отдельный бит маски. Если бит равен `1`, то есть установлен (оператор `AND` возвращает ненулевое значение), выполняется проверка существования модуля на этой позиции в массиве, а также проверяется, не был ли этот модуль активизирован ранее.

Если модуль уже активизирован, то выполняется попытка остановить его. Если функция `shutdown()` этого модуля завершена успешно (возвращено значение `true`), то соответствующий бит в маске `active_mods` переключается (сбрасывается) для обновления его состояния. Если же модуль до этого не был активизирован и зарегистрирован в этой позиции массива, то выполняется попытка его запуска, а при успешном запуске обновляется соответствующий бит состояния в маске активных модулей.

Проверяется также, был ли зарегистрирован обратный вызов запускаемой функции, чтобы убедиться в том, что не будет случайно выполнен вызов некорректно зарегистрированного модуля, который может привести к аварийному завершению работы всей системы.

Класс PlantModule

В этом разделе мы подробно рассмотрим код поддержки нижнего уровня, который упрощает написание новых модулей, выполняя всю необходимую работу. Единственная тема, которой мы до сих пор еще не уделили внимания, – модуль действительно рабочих операций, то есть код, непосредственно реализующий проект, рассматриваемый в данной главе.

Здесь рассматривается последняя часть задачи, а именно класс `PlantModule`.

```
#include "base_module.h"
#include <Libraries/APA102/apa102.h>

#define PLANT_GPIO_PIN 5
#define NUM_APA102 1

class PlantModule {
        static int pin;
        static Timer timer;
        static uint16 humidityTrigger;
        static String publishTopic;
        static HttpServer server;
        static APA102* LED;
        static void onRequest(HttpRequest& request, HttpResponse& response);
```

```
public:
        static bool initialize();
        static bool start();
        static bool shutdown();
        static void readSensor();
        static void commandCallback(String message);
};
```

В этом объявлении класса следует отметить включение заголовочного файла библиотеки APA102. Это простая библиотека, позволяющая управлять цветом и яркостью отображения данных на LED-индикаторах APA102 RGB (с полным цветовым спектром) по шине SPI.

Здесь также определяется контакт, который будет использоваться для переключения режима работы перистальтического насоса (GPIO 5), и количество подключенных модулей APA102 LED (один). При необходимости можно добавить несколько LED-индикаторов APA102 и просто обновить определение счетчика соответствующим образом.

Далее следует реализация класса.

```
#include "plant_module.h"

int PlantModule::pin = PLANT_GPIO_PIN;
Timer PlantModule::timer;
uint16 PlantModule::humidityTrigger = 530;
String PlantModule::publishTopic;
HttpServer PlantModule::server;
APA102* PlantModule::LED = 0;

enum {
        PLANT_SOIL_MOISTURE = 0x01,
        PLANT_SET_TRIGGER = 0x02,
        PLANT_TRIGGER = 0x04
};
```

В этом блоке кода инициализируются статические члены класса, устанавливается контакт GPIO и определяется начальное значение сенсора, при котором должен переключаться режим работы насоса. Это переключающее (триггерное) значение должно быть обновлено в соответствии с результатами калибровки применяемого в данном проекте сенсорного датчика.

Также определяется перечисление, содержащее допустимые для этого модуля команды, которые могут быть переданы с использованием протокола MQTT.

```
bool PlantModule::initialize() {
         BaseModule::registerModule(MOD_IDX_PLANT, PlantModule::start, PlantModule::shutdown);
}
```

Это функция инициализации, выполняемая при обращении к объекту `BaseModule` при запуске системы. Можно видеть, что функция активизирует регистрацию самого этого модуля с предварительно заданными значениями, в том числе с функциями обратного вызова для запуска и останова модуля.

```
bool PlantModule::start() {
        OtaCore::log(LOG_INFO, "Plant Module starting...");

        if (!OtaCore::claimPin(pin)) { return false; }
```

```
        publishTopic = MQTT_PREFIX + "plant/response/" + OtaCore::getLocation();
        OtaCore::registerTopic(MQTT_PREFIX + String("plants/") +
                               OtaCore::getLocation(), PlantModule::commandCallback);

        pinMode(pin, OUTPUT);

        server.listen(80);
        server.setDefaultHandler(PlantModule::onRequest);

        LED = new APA102(NUM_APA102);
        LED->setBrightness(15);
        LED->clear();
        LED->setAllPixel(0, 255, 0);
        LED->show();

        timer.initializeMs(60000, PlantModule::readSensor).start();
        return true;
}
```

После начала работы модуля выполняется попытка объявления контакта, который предполагается использовать для включения/отключения насоса, а также регистрируется обратный вызов для темы MQTT, чтобы получить возможность приема команд с помощью обратного вызова обработчика команд. Кроме того, здесь же определяется тема, в которой мы будем отвечать после обработки команды.

Устанавливается режим вывода (`OUTPUT`) для объявленного контакта, затем запускается сервер HTTP с указанием номера порта 80 и регистрацией простого обработчика клиентских запросов. Далее создается новый экземпляр класса `APA102`, который используется для настройки подключенного LED-индикатора для подсветки зеленым цветом приблизительно в половину полной яркости.

Запускается таймер, с помощью которого будет активизироваться операция считывания показаний сенсорного датчика почвы каждую минуту.

```
bool PlantModule::shutdown() {
        if (!OtaCore::releasePin(pin)) { return false; }

        server.shutdown();

        if (LED) {
              delete LED;
              LED = 0;
        }

        OtaCore::deregisterTopic(MQTT_PREFIX + String("plants/") + OtaCore::getLocation());

        timer.stop();
        return true;
}
```

При завершении работы этого модуля освобождается ранее зарегистрированный контакт, останавливается веб-сервер, удаляется экземпляр класса RGB LED (с проверкой действительной необходимости такого удаления), отменяется регистрация темы MQTT и останавливается таймер сенсорного датчика.

```
void PlantModule::commandCallback(String message) {
        OtaCore::log(LOG_DEBUG, "Plant command: " + message);
```

```
        if (message.length() < 1) { return; }
        int index = 0;
        uint8 cmd = *((uint8*) &message[index++]);

        if (cmd == PLANT_SOIL_MOISTURE) {
              readSensor();
        }
        else if (cmd == PLANT_SET_TRIGGER) {
               if (message.length() != 3) { return; }
              uint16 payload = *((uint16*) &message[index]);
              index += 2;
              humidityTrigger = payload;
        }
        else if (cmd == PLANT_TRIGGER) {
              OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" +
                               String(((char*) &humidityTrigger), 2));
        }
}
```

Обращение к этой функции обратного вызова происходит при публикации сообщения MQTT в зарегистрированной теме. В этих сообщениях мы предполагаем обнаружить однобайтовое значение (тип `uint8`), определяющее конкретную команду (до восьми возможных команд). Для этого модуля ранее были определены три возможные команды, которые описаны в табл. 5.4.

Таблица 5.4

Команда	Описание	Полезная нагрузка	Возвращаемое значение
0x01	Получить влажность почвы	–	0xXXXX
0x02	Установить уровень триггера включения/отключения	uint16 (новый уровень триггера)	-
0x04	Получить текущий уровень триггера	–	0xXXXX

Каждая команда из табл. 5.4 возвращает запрошенное значение, если это возможно.

После проверки того факта, что полученное сообщение содержит по меньшей мере один байт, извлекается первый байт и выполняется попытка интерпретации его как команды. При установке нового уровня триггера из сообщения также извлекается новое значение типа `uint16`, если подтверждена корректность сформированного сообщения.

Последней определена функция, в которой происходит все, для чего был создан наш проект.

```
void PlantModule::readSensor() {
    int16_t val = 0;
    val = analogRead(A0); // вызывает system_adc_read().
    String response = OtaCore::getLocation() + ";" + val;
    OtaCore::publish(MQTT_PREFIX"nsa/plant/moisture_raw", response);
```

Сначала считывается текущее значение сенсорного датчика из аналогового ввода ESP8266 и публикуется в теме MQTT, предназначенной для этого.

```
        if (val >= humidityTrigger) {
               digitalWrite(pin, HIGH);

               LED->setBrightness(31);
               LED->setAllPixel(0, 0, 255);
               LED->show();

               for (int i = 0; i < 10; ++i) {
                     LED->directWrite(0, 0, 255, 25);
                     delay(200);
                     LED->directWrite(0, 0, 255, 18);
                     delay(200);
                     LED->directWrite(0, 0, 255, 12);
                     delay(200);
                     LED->directWrite(0, 0, 255, 5);
                     delay(200);
                     LED->directWrite(0, 0, 255, 31);
                     delay(200);
               }

               digitalWrite(pin, LOW);
         }
}
```

Во время калибровки одного из прототипов с сенсорным датчиком влажности почвы было выяснено, что значение для абсолютно сухого сенсора (находящегося в воздухе) приблизительно равнялось 766. Тот же сенсорный датчик, полностью погруженный в воду, показал значение 379. По этим значениям можно сделать вывод о том, что 60 % влажности должно приблизительно соответствовать значение 533, которое принимается в качестве начального значения, устанавливаемого во время инициализации статических членов класса. Идеальная точка переключения и целевой уровень влажности почвы, разумеется, зависит от типа почвы и конкретного растения.

При достижении заданного уровня переключения устанавливается выходной контакт, соединенный с контактом Enable (разрешить) повышающего преобразователя для повышения напряжения. Это активизирует вывод преобразователя, что, в свою очередь, приводит к включению насоса. Запланировано разрешение работы насоса в течение десяти секунд.

В этот интервал времени устанавливается синий цвет LED-индикатора, при этом каждую секунду яркость снижается со 100 % почти до нуля, затем яркость снова повышается до 100 % с созданием эффекта пульсации.

После этого для выходного контакта преобразователя устанавливается режим низкого напряжения, при котором насос отключается, и происходит переход в режим ожидания очередного считывания показаний сенсорного датчика влажности почвы.

```
void PlantModule::onRequest(HttpRequest& request, HttpResponse& response) {
         TemplateFileStream* tmpl = new TemplateFileStream("index.html");
         TemplateVariables& vars = tmpl->variables();
         int16_t val = analogRead(A0);
         int8_t perc = 100 - ((val - 379) / 3.87);
         vars["raw_value"] = String(val);
```

```
        vars["percentage"] = String(perc);
        response.sendTemplate(tmpl);
}
```

В завершающей части расположен обработчик запросов к веб-серверу. Он считывает файл шаблонов из файловой системы SPIFFS (подробности описаны в следующем разделе), извлекает список переменных из этого файла, затем выполняет чтение текущего значения сенсорного датчика.

С использованием этого значения вычисляется текущий процент влажности почвы, после чего исходное и вычисленное значения передаются в две переменные извлеченного шаблона, который возвращается как результат обработки запроса.

Файл Index.html

Для работы веб-сервера в модуле PlantModule необходимо добавить в файловую систему SPIFFS следующий файл шаблона Index.html:

```
<!DOCTYPE html>
<html>
<head>
        <title>Plant soil moisture readings</title>
   </head>
   <body>
        Current value: {raw_value}<br>
        Percentage: {percentage}%
</body>
</html>
```

Компиляция и запись в ПЗУ

После завершения написания кода для рассматриваемого приложения можно скомпилировать его с помощью одной команды, выполненной в корневом каталоге проекта:

```
make
```

После завершения процедуры компиляции и сборки в каталоге *out* содержатся бинарные файлы, в том числе и образы ПЗУ. Так как мы используем и загрузчик rBoot, и файловые системы SPIFFS, всего в подкаталоге *firmware* находится три образа ПЗУ.

Теперь можно установить соединение с модулем ESP8266 в форме платы NodeMCU или одной из множества альтернатив и пометить последовательный порт, который будет использоваться для подключения. В ОС Windows это должно быть нечто, похожее на `COM3`, в системах Linux адаптеры между USB и последовательным портом обычно регистрируются как `/dev/ttyUSB0` или похожим образом.

Если последовательный порт (`COM_PORT`) не был определен в пользовательском *Makefile*, необходимо явно определить его при записи образа в модуль ESP8266:

```
make flash COM_PORT=/dev/ttyUSB0
```

После выполнения этой команды мы должны увидеть вывод утилиты `esptool.py`, подтверждающий, что она установила соединение с ПЗУ ESP8266 и начала запись образов ПЗУ.

Когда процесс записи завершится, будет выполнен перезапуск микроконтроллера с загрузкой нового образа специализированного ПО. После загрузки система будет ожидать команд пользователя для конфигурирования.

Первоначальное конфигурирование

Как уже было отмечено ранее в этой главе, рассматриваемое здесь специализированное ПО предназначено для конфигурирования и сопровождения по протоколу MQTT. Поэтому требуется доступность брокера MQTT. Широко распространенным брокером MQTT является Mosquitto (**http://mosquitto.org/**). Поскольку это упрощенный сервер, его можно установить на настольную систему, на небольшой одноплатный компьютер, в виртуальную машину и т. д.

В дополнение к брокеру и микроконтроллеру ESP8266 под управлением специализированного ПО также необходим специализированный клиент для взаимодействия со специализированным ПО. Поскольку используются бинарные протоколы, выбор несколько ограничен, так как наиболее часто применяемые в настольных системах MQTT-клиенты работают с текстовыми сообщениями. Можно применить подход с публикацией бинарных сообщений, воспользовавшись MQTT-клиентом, входящим в комплект Mosquitto, и работать с шестнадцатеричным форматом ввода инструмента командной строки **echo** для передачи бинарных данных как потока, который должен публиковаться инструментальным средством клиента.

По этой причине автор разработала программу нового настольного MQTT-клиента (на основе языка C++ и библиотеки Qt), предназначенного для использования и отладки бинарных протоколов на основе MQTT: **https://github.com/Maya-Posch/MQTTCute**.

Теперь все три компонента готовы к работе – микроконтроллер ESP8266 с ПО, разработанным в рассматриваемом проекте, брокер MQTT и настольный клиент. Можно скомпоновать всю систему мониторинга и водоснабжения растения и передать в нее команду, активизирующую модуль Plant.

При наблюдении за появлением сообщений в теме cc/config мы должны увидеть отчет о наличии микроконтроллера ESP8266 в виде публикуемого им MAC-адреса. Это сообщение также можно увидеть, подключив последовательный адаптер USB-TTL к контакту последовательного вывода журналирования (`D4` на плате NodeMCU). Наблюдая вывод на последовательной консоли, мы увидим IP-адрес и MAC-адрес системы.

После создания новой темы в формате `cc/<MAC>` можно публиковать команды, передаваемые специализированному ПО, например:

```
log;plant001
```

Эта команда определяет имя локации системы как `plant001`.

При использовании клиента MQTTCute можно применять бинарный ввод в стиле команды echo в шестнадцатеричном формате для активизации модуля растения Plant:

```
mod;\x00\x01\x00\x00
```

Здесь команда `mod` передается в специализированное ПО вместе с битовой маской `0x100`. По этой команде модуль Plant должен быть инициализирован и запущен в работу. Поскольку существует и строка локации, и конфигурация, нет необходимости повторять этот шаг, если только не выполняется процедура обновления по воздуху, когда новому специализированному ПО потребуется пустая файловая система SPIFFS, если только мы не записываем один и тот же образ SPIFFS в оба слота в ПЗУ.

Можно было бы дополнить код процедуры обновления по воздуху (OTA) для загрузки образа SPIFFS ПЗУ в дополнение к образу специализированного ПО, но это приводит к усложнению возможной перезаписи существующих в файловой системе SPIFFS файлов.

После выполнения всех вышеописанных действий мы должны получить готовую к работе систему мониторинга и водоснабжения комнатного растения.

Использование системы

Можно использовать измеряемые значения и сохранять их в базе данных, подписавшись на тему `nsa/plant/moisture_raw`. Точку переключения можно отрегулировать, отправив новую команду в тему `plant/<location string>`.

К веб-серверу на устройстве можно получить доступ по его IP-адресу, который определяется из данных, выводимых в последовательной консоли, как описано в предыдущем разделе, или при просмотре активных IP-адресов в сетевом маршрутизаторе.

Открыв этот IP-адрес в браузере, мы должны увидеть HTML-шаблон, заполненный текущими значениями.

Дальнейшие действия

Необходимо выполнить следующие условия:

- можно усовершенствовать систему, дополнив ее реализацией профилей водоснабжения растения с учетом засушливых периодов или подсистемой регулирования для различных типов почвы. Также можно добавить новые режимы RGB LED-индикаторов, чтобы полностью задействовать все доступные варианты цветов подсветки;
- всю аппаратуру можно встроить в объемлющую систему, чтобы сделать ее невидимой или, наоборот, сделать ее более видимой;
- можно расширить веб-интерфейс, предоставив возможность управления точкой переключения непосредственно из браузера вместо подачи команд с помощью MQTT-клиента;
- в дополнение к сенсорному датчику влажности можно добавить сенсорный датчик освещенности, датчик температуры и т. д. для измерения большего количества факторов, влияющих на жизнеспособность растения;
- можно автоматизировать применение (жидких) удобрений для растения.

Сложности

Затруднение может возникнуть в работе аналого-цифрового преобразователя (ADC) микроконтроллера ESP8266 с платами NodeMCU, так как на этих платах первый зарезервированный контакт (RSV) расположен в непосредственной близости от контакта ADC, напрямую соединенного с входным контактом ADC модуля ESP8266. Может возникнуть проблема, связанная с электростатическим разрядом неизолированного контакта. По существу, это разряд с высоким напряжением, но низкой силой тока, направленный в микроконтроллер. Добавление небольшого конденсатора к зарезервированному контакту RSV для его заземления может снизить риск.

Очевидно, что рассматриваемая здесь система не способна защитить растения от вредителей и паразитов. Поэтому даже после автоматизации водоснабжения вы не должны оставлять без внимания свои растения. Регулярное наблюдение не только за растениями, но за самой системой с целью своевременного выявления возникающих проблем (расстыковка и смещение трубок и прочих компонентов аппаратуры, непредвиденные воздействия домашних животных и т. п.) остается весьма важной задачей.

Резюме

В этой главе рассматривалась реализация теории в простом проекте на основе микроконтроллера ESP8266 с упрощенными требованиями к функциональному проектному решению в форме гибкого специализированного ПО и набора вариантов ввода/вывода, используя которые, мы гарантировали, что обслуживаемое растение получает именно то количество воды, которое необходимо для сохранения его жизнеспособности. Также рассматривалась процедура настройки среды разработки для микроконтроллера ESP8266.

После изучения этой главы читатель должен приобрести навыки создания проектов для ESP8266, программирования микроконтроллеров с использованием нового специализированного ПО и получить прочные знания о преимуществах и ограничениях этой платформы разработки.

В следующей главе будет рассматриваться методика тестирования встроенного ПО, написанного для систем на кристаллах и других, более крупных встроенных платформ.

Часть II

ТЕСТИРОВАНИЕ, МОНИТОРИНГ

В этой части подробно рассматривается правильно организованный рабочий поток разработки для разнообразных встроенных платформ, в том числе стратегии тестирования, и важность написания переносимого кода.

Часть II включает следующие главы:

- главу 6 «Тестирование приложений, предназначенных для конкретных ОС»;
- главу 7 «Тестирование платформ с ограниченными ресурсами»;
- главу 8 «Пример: информационно-развлекательная система на основе ОС Linux»;
- главу 9 «Пример: мониторинг и управление микроклиматом в здании».

Глава 6

Тестирование приложений, предназначенных для конкретных ОС

Часто встроенные системы используют более или менее обычные операционные системы (ОС), которые в большинстве случаев весьма похожи на настольные ОС в плане среды времени выполнения и инструментальных средств, особенно если применяется встроенная система Linux. Но различия в производительности и уровне доступа встроенной аппаратуры по сравнению с персональными компьютерами делают важным определение методики выполнения разработки и тестирования, а также интеграции ее в конкретный рабочий поток.

В этой главе рассматриваются следующие темы:

- разработка кода, переносимого между платформами;
- отладка и тестирование кроссплатформного кода в ОС Linux;
- эффективное использование кросс-компиляторов;
- создание системы сборки, поддерживающей несколько целей.

Почему следует избегать разработки на реальной аппаратуре

Одним из самых значительных преимуществ разработки на основе ОС на таких платформах, как встроенная система Linux, является тот факт, что такая система очень похожа на обычный экземпляр Linux, установленный на десктопной системе. Особенно при использовании дистрибутивов Linux на основе Debian (Armbian, Raspbian и др.) для систем на кристалле доступны практически те же инструментальные средства, полноценный менеджер пакетов, набор компиляторов и библиотек, которые устанавливаются буквально одной-двумя командами.

Но в то же время это представляет собой большую проблему.

Можно писать код, копировать его на одноплатный компьютер, компилировать его там, выполнять тесты, вносить изменения, после чего повторять весь процесс снова. Или можно даже писать код на самом одноплатном компьютере, в сущности используя его как конкретную платформу разработки.

Главные причины, по которым так поступать не следует, перечислены ниже:

- современные персональные компьютеры намного быстрее;
- тестирование на реальной аппаратуре не должно выполняться до перехода разработки в завершающую стадию;
- автоматизированное комплексное (интегрированное) тестирование выполнять гораздо труднее (на реальной аппаратуре).

Первый пункт вполне очевиден. Зачем ждать целую минуту при компиляции в системе на кристалле с одноядерным или двухъядерным процессором ARM, если весь процесс компиляции и связывания объектов (сборки) занимает десять секунд и даже меньше на относительно современном многоядерном многопоточном процессоре с тактовой частотой 3 ГГц и более и доступен комплект инструментов поддержки многоядерной компиляции?

То есть вместо полуминутного или более длительного ожидания того момента, когда, наконец, можно будет запустить новый тест или начать сеанс отладки, все это можно делать практически моментально.

Следующие два пункта не столь однозначны. Может показаться, что тестирование на реальной аппаратуре имеет свои преимущества, но при таком подходе возникают и сложности. Правильная работа аппаратуры зависит от ряда внешних факторов, включающих электропитание, проводку от источников электропитания, периферийные устройства и сигнальные интерфейсы. Проблемы могут возникать из-за таких факторов, как электромагнитные помехи, которые ослабляют и зашумляют сигналы, а также прерывания, срабатывающие из-за электромагнитной связи контуров.

Пример электромагнитной связи контуров можно было наблюдать при разработке проекта управления состоянием клубного помещения в главе 3. В этом проекте сигнальный провод для переключателей был проложен рядом с проводкой электропитания переменного тока 230 В. Из-за этого в сигнальном проводе возникали индуцированные импульсы, что приводило к ложным прерываниям при отсутствии действительных соответствующих событий.

Все эти потенциальные проблемы, связанные с аппаратурой, подтверждают, что такой подход к тестированию нельзя считать детерминированным и соответствующим нашим требованиям. При подобном подходе разработка проекта займет больше времени, чем планировалось, а отладка станет сложнее из-за противоречивых и недетерминированных результатов тестов.

Другим отрицательным фактором разработки на реальной аппаратуре является существенное усложнение автоматизированного тестирования. Причина в том, что мы не можем использовать какой-либо кластер сборки общего назначения и, например, среду тестирования на основе виртуальной машины в ОС Linux, так же как и широко используемые сервисы непрерывной интеграции (continuous integration – CI).

Вместо этого придется каким-то образом интегрировать одноплатный компьютер или подобную аппаратуру в систему непрерывной интеграции, выполняя кросс-компиляцию и копируя бинарные файлы на одноплатный компьютер для запуска теста, или выполнять компиляцию на самом одноплатном компьютере, что опять возвращает нас к первому пункту.

В следующих разделах будут рассматриваться подходы и методы, позволяющие сделать процесс разработки встроенных систем на основе Linux максимально удобным и безболезненным, а начнем мы с кросс-компиляции.

Кросс-компиляция для одноплатных компьютеров

Процесс компиляции предполагает превращение файлов исходного кода в некоторый промежуточный формат, после чего этот формат можно использовать для ориентации на конкретную целевую архитектуру ЦПУ. Это означает, что компилировать приложения для одноплатного компьютера не обязательно на самом этом компьютере, это можно сделать на полноценном настольном компьютере, предназначенном для разработки.

Чтобы выполнить компиляцию для одноплатного компьютера, например для Raspberry Pi (системы на кристалле на основе Broadcom Cortex-A ARM), необходимо сначала установить комплект инструментов `arm-linux-gnueabihf`, предназначенный для целевой архитектуры ARM с поддержкой аппаратных операций с плавающей точкой, позволяющий создавать бинарные файлы, совместимые с ОС Linux.

В системе Linux на основе дистрибутива Debian весь комплект инструментов можно установить с помощью следующих команд:

```
sudo apt install build-essential
sudo apt install g++-arm-linux-gnueabihf
sudo apt install gdb-multiarch
```

Первая команда устанавливает штатный комплект компиляторов GCC для рабочей системы (если до этого он не был установлен) вместе со всеми необходимыми инструментами и утилитами, включая `make`, `libtool`, `flex` и т. п. Вторая команда устанавливает сам пакет кросс-компилятора. Третий устанавливаемый пакет – это версия отладчика GDB, поддерживающая различные архитектуры, которая потребуется позже для выполнения удаленной процедуры отладки на реальной аппаратуре, а также для анализа дампов ядра (памяти) после аварийных завершений разрабатываемого приложения.

Теперь можно использовать компилятор g++ для целевого одноплатного компьютера, если ввести его полное имя в командной строке:

```
arm-linux-gnueabihf-g++
```

Для проверки правильности установки комплекта инструментов можно выполнить следующую команду, выводящую версию компилятора и другую информацию о нем:

```
arm-linux-gnueabihf-g++ -v
```

Кроме того, может потребоваться установление связи с некоторыми совместно используемыми динамическими библиотеками, которые существуют на целевой системе. Для этого можно скопировать все содержимое каталогов */lib* и */usr* и включить их как часть корневого каталога системы для компилятора:

```
mkdir ~/raspberry/sysroot
scp -r pi@Pi-system:/lib ~/raspberry/sysroot
scp -r pi@Pi-system:/usr ~/raspberry/sysroot
```

Здесь `Pi-system` – это IP-адрес или сетевое имя системы Raspberry Pi либо другой используемой в разработке системы. После этого можно сообщить компиляторам GCC о необходимости использования этих каталогов вместо стандартных путей. Для этого применяется флаг `sysroot`:

```
--sysroot=dir
```

Здесь `dir` обозначает каталог, в который были скопированы подкаталоги */lib* и */usr*, то есть в рассматриваемом примере это каталог *~/raspberry/sysroot*.

В другом варианте можно скопировать только требуемые заголовочные и библиотечные файлы и добавить их как часть дерева исходного кода. Какой вариант окажется проще и удобнее, в основном определяется зависимостями конкретного проекта.

Для проекта управления состоянием клубного помещения требуется весьма небольшое количество заголовочных и библиотечных файлов для WiringPi, а также для проекта POCO и его зависимостей. Можно без труда определить необходимые зависимости и скопировать требуемые заголовочные и библиотечные файлы, отсутствующие в комплекте инструментов, установленном ранее. Если файлов слишком много, то гораздо проще полностью скопировать каталоги из ОС одноплатного компьютера.

Кроме метода с использованием каталога sysroot, можно также явно определить пути к совместно используемым динамическим библиотекам, которые необходимы для связывания с разрабатываемым кодом. Разумеется, такой подход обладает собственными достоинствами и недостатками.

Комплексный тест для сервиса управления состоянием клубного помещения

Для тестирования сервиса управления состоянием клубного помещения на обычной настольной системе Linux (или macOS, или Windows), прежде чем начать кросс-компиляцию и тестирование на реальной аппаратуре, был написан простой комплексный тест, использующий имитацию периферийных устройств GPIO и I2C.

В дереве исходного кода для проекта, рассматриваемого в главе 3, файлы для этих периферийных устройств располагались в подкаталоге *wiring*.

Сначала рассмотрим содержимое заголовочного файла *wiringPi.h*:

```
#include <Poco/Timer.h>

#define  INPUT              0
#define  OUTPUT             1
#define  PWM_OUTPUT         2
#define  GPIO_CLOCK         3
#define  SOFT_PWM_OUTPUT    4
#define  SOFT_TONE_OUTPUT   5
#define  PWM_TONE_OUTPUT    6
```

В этот файл включен заголовок из рабочей среды POCO, позволяющий в дальнейшем без затруднений создать экземпляр таймера. Далее определяются все возможные режимы контакта в соответствии с определениями реального заголовочного файла WiringPi.

```
#define  LOW               0
#define  HIGH              1

#define  PUD_OFF           0
#define  PUD_DOWN          1
#define  PUD_UP            2

#define  INT_EDGE_SETUP    0
#define  INT_EDGE_FALLING  1
#define  INT_EDGE_RISING   2
#define  INT_EDGE_BOTH     3
```

Здесь определяются дополнительные режимы контактов, в том числе уровни цифрового ввода, возможные состояния повышения и понижения нагрузки на контактах, а также вероятные типы прерываний, определяющие триггер или несколько триггеров для генерации прерываний.

```
typedef void (*ISRCB)(void);
```

Определяется формат указателя на функцию обратного вызова прерывания.

Теперь рассмотрим класс `WiringTimer`:

```
class WiringTimer {
    Poco::Timer* wiringTimer;
    Poco::TimerCallback<WiringTimer>* cb;
    uint8_t triggerCnt;

 public:
    ISRCB isrcb_0;
    ISRCB isrcb_7;
    bool isr_0_set;
    bool isr_7_set;
    WiringTimer();
    ~WiringTimer();
    void start();
    void trigger(Poco::Timer &t);
};
```

Этот класс является составной частью реализации имитации аппаратного интерфейса GPIO. Его главная задача – непрерывное отслеживание регистрации двух интересующих нас перерываний и генерация их через постоянные интервалы времени с использованием таймера, как мы увидим в дальнейшем.

```
int wiringPiSetup();
void pinMode(int pin, int mode);
void pullUpDnControl(int pin, int pud);
int digitalRead(int pin);
int wiringPiISR(int pin, int mode, void (*function)(void));
```

Далее определяются стандартные функции WiringPi до перехода к реализации.

```
#include "wiringPi.h"

#include <fstream>
#include <memory>

WiringTimer::WiringTimer() {
   triggerCnt = 0;
   isrcb_0 = 0;
   isrcb_7 = 0;
   isr_0_set = false;
   isr_7_set = false;

   wiringTimer = new Poco::Timer(10 * 1000, 10 * 1000);
   cb = new Poco::TimerCallback<WiringTimer>(*this, &WiringTimer::trigger);
}
```

В приведенном выше конструкторе класса устанавливаются значения по умолчанию перед созданием экземпляра таймера, выполняется его конфигурирование для обращения к ранее определенной функции обратного вызова через каждые десять секунд после начальной 10-секундной задержки.

```
WiringTimer::~WiringTimer() {
   delete wiringTimer;
   delete cb;
}
```

Деструктор удаляет экземпляр таймера и соответствующую функцию обратного вызова.

```
void WiringTimer::start() {
   wiringTimer->start(*cb);
}
```

Это функция действительного запуска таймера.

```
void WiringTimer::trigger(Poco::Timer &t) {
    if (triggerCnt == 0) {
          char val = 0x00;
          std::ofstream PIN0VAL;
          PIN0VAL.open("pin0val", std::ios_base::binary | std::ios_base::trunc);
          PIN0VAL.put(val);
          PIN0VAL.close();

          isrcb_0();

          ++triggerCnt;
    }
    else if (triggerCnt == 1) {
          char val = 0x01;
          std::ofstream PIN7VAL;
          PIN7VAL.open("pin7val", std::ios_base::binary | std::ios_base::trunc);
          PIN7VAL.put(val);
          PIN7VAL.close();

          isrcb_7();

          ++triggerCnt;
```

```
    }
    else if (triggerCnt == 2) {
          char val = 0x00;
          std::ofstream PIN7VAL;
          PIN7VAL.open("pin7val", std::ios_base::binary | std::ios_base::trunc);
          PIN7VAL.put(val);
          PIN7VAL.close();

          isrcb_7();

          ++triggerCnt;
    }
    else if (triggerCnt == 3) {
          char val = 0x01;
          std::ofstream PIN0VAL;
          PIN0VAL.open("pin0val", std::ios_base::binary | std::ios_base::trunc);
          PIN0VAL.put(val);
          PIN0VAL.close();

          isrcb_0();

          triggerCnt = 0;
    }
}
```

Последней в этом классе определена функция обратного вызова для таймера. Она отслеживает количество собственных обратных вызовов и устанавливает соответствующий уровень для контакта в форме значения в файле, записываемом на диск.

После начального интервала задержки первый триггер установит для переключателя блокировки значение `false`, второй триггер переключает состояние в `true`, третий возвращает значение `false`, наконец, четвертый триггер снова устанавливает для переключателя блокировки значение `true`, до переустановки счетчика, после чего цикл начинается снова.

```
namespace Wiring {
   std::unique_ptr<WiringTimer> wt;
   bool initialized = false;
}
```

Добавляется глобальное пространство имен, в котором содержится экземпляр указателя `unique_ptr` для экземпляра класса `WiringTimer` вместе с индикатором состояния инициализации.

```
int wiringPiSetup() {
    char val = 0x01;
    std::ofstream PIN0VAL;
    std::ofstream PIN7VAL;
    PIN0VAL.open("pin0val", std::ios_base::binary | std::ios_base::trunc);
    PIN7VAL.open("pin7val", std::ios_base::binary | std::ios_base::trunc);
    PIN0VAL.put(val);
    val = 0x00;
    PIN7VAL.put(val);
    PIN0VAL.close();
```

```
    PIN7VAL.close();

    Wiring::wt = std::make_unique<WiringTimer>();
    Wiring::initialized = true;

    return 0;
 }
```

Функция настройки WiringPi используется для записи на диск значений по умолчанию для имитируемых входных сигналов на контактах GPIO. Также создается указатель на экземпляр класса `WiringTimer`.

```
void pinMode(int pin, int mode) {
    //

    return;
 }

void pullUpDnControl(int pin, int pud) {
    //

    return;
 }
```

Поскольку создаваемая здесь имитационная реализация определяет поведение контактов, можно игнорировать любые входные сигналы в этих функциях. Для целей тестирования можно было бы добавить условный оператор контроля для проверки факта вызова этих функций в требуемое время с соответствующими параметрами настройки.

```
int digitalRead(int pin) {
    if (pin == 0) {
          std::ifstream PIN0VAL;
          PIN0VAL.open("pin0val", std::ios_base::binary);
          int val = PIN0VAL.get();
          PIN0VAL.close();

          return val;
    }
    else if (pin == 7) {
          std::ifstream PIN7VAL;
          PIN7VAL.open("pin7val", std::ios_base::binary);
          int val = PIN7VAL.get();
          PIN7VAL.close();

          return val;
    }

    return 0;
 }
```

При считывании значения с одного из двух имитируемых контактов открывается соответствующий файл и считывается его содержимое, то есть значение 1 или 0, установленное функцией настройки или функцией обратного вызова.

```
// Это значение возвращается в вызывающую функцию.

 int wiringPiISR(int pin, int mode, void (*function)(void)) {
```

```
    if (!Wiring::initialized) {
          return 1;
    }

    if (pin == 0) {
          Wiring::wt->isrcb_0 = function;
          Wiring::wt->isr_0_set = true;
    }
    else if (pin == 7) {
          Wiring::wt->isrcb_7 = function;
          Wiring::wt->isr_7_set = true;
    }

    if (Wiring::wt->isr_0_set && Wiring::wt->isr_7_set) {
          Wiring::wt->start();
    }

    return 0;
}
```

Эта функция используется для регистрации прерывания и соответствующей ему функции обратного вызова. После начальной проверки факта инициализации имитируемых контактов функцией настройки продолжается регистрация прерывания для одного или двух заданных контактов.

После установки прерывания для обоих контактов запускается таймер, который, в свою очередь, начинает генерировать события для обратных функций прерывания.

Далее следует имитация шины I2C.

```
int wiringPiI2CSetup(const int devId);
int wiringPiI2CWriteReg8(int fd, int reg, int data);
```

Здесь необходимы только две функции: функция настройки и простая функция записи в однобайтовый регистр.

Реализация приведена ниже.

```
#include "wiringPiI2C.h"

#include "../club.h"

#include <Poco/NumberFormatter.h>

using namespace Poco;

int wiringPiI2CSetup(const int devId) {
   Club::log(LOG_INFO, "wiringPiI2CSetup: setting up device ID: 0x" +
                       NumberFormatter::formatHex(devId));
   return 0;
}
```

В функции настройки в журнале фиксируется идентификатор запрашиваемого устройства (адрес шины I2C) и возвращается стандартный обработчик для этого устройства. Здесь используется функция `log()` из класса `Club` для интеграции имитации в остальной исходный код.

```
int wiringPiI2CWriteReg8(int fd, int reg, int data) {
    Club::log(LOG_INFO, "wiringPiI2CWriteReg8: Device handle 0x" +
```

```
                        NumberFormatter::formatHex(fd) +
                       ", Register 0x" + NumberFormatter::formatHex(reg) +
                       " set to: 0x" + NumberFormatter::formatHex(data));
    return 0;
}
```

Поскольку код, который будет вызывать эту функцию, не ожидает ответа, после простого подтверждения приема данных можно просто записать в журнал принятые данные и прочие сопутствующие подробности. Класс `NumberFormatter` из рабочей среды POCO используется для форматирования целочисленных данных в виде шестнадцатеричных значений для согласования с форматом, используемым в этом приложении.

Теперь можно скомпилировать проект с помощью следующей команды:

```
make TEST=1
```

Запуск приложения (под управлением отладчика GDB, чтобы видеть, когда создаются новые потоки) дает следующий результат:

```
Starting ClubStatus server...
 Initialised C++ Mosquitto library.
 Created listener, entering loop...
 [New Thread 0x7ffff49c9700 (LWP 35462)]
 [New Thread 0x7ffff41c8700 (LWP 35463)]
 [New Thread 0x7ffff39c7700 (LWP 35464)]
 Initialised the HTTP server.
 INFO:       Club: starting up...
 INFO:       Club: Finished wiringPi setup.
 INFO:       Club: Finished configuring pins.
 INFO:       Club: Configured interrupts.
 [New Thread 0x7ffff31c6700 (LWP 35465)]
 INFO:       Club: Started update thread.
 Connected. Subscribing to topics...
 INFO:       ClubUpdater: Starting i2c relay device.
 INFO:       wiringPiI2CSetup: setting up device ID: 0x20
 INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x6 set to: 0x0
 INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x2 set to: 0x0
 INFO:       ClubUpdater: Finished configuring the i2c relay device's registers.
```

В этот момент система сконфигурирована с установкой всех прерываний, а устройство I2C сконфигурировано приложением. Таймер запущен, начался отсчет первоначального интервала.

```
INFO:       ClubUpdater: starting initial update run.
INFO:       ClubUpdater: New lights, clubstatus off.
DEBUG:      ClubUpdater: Power timer not active, using current power state: off
INFO:       ClubUpdater: Red on.
DEBUG:      ClubUpdater: Changing output register to: 0x8
INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x2 set to: 0x8
DEBUG:      ClubUpdater: Finished writing relay outputs with: 0x8
INFO:       ClubUpdater: Initial status update complete.
```

Считано начальное состояние контактов GPIO, и для обоих переключателей определена позиция `off` (отключен), поэтому активизируется красный цвет на светофоре посредством записи соответствующей позиции в регистр.

```
INFO:       ClubUpdater: Entering waiting condition.
INFO:       ClubUpdater: lock status changed to unlocked
INFO:       ClubUpdater: New lights, clubstatus off.
DEBUG:      ClubUpdater: Power timer not active, using current power state: off
INFO:       ClubUpdater: Yellow on.
DEBUG:      ClubUpdater: Changing output register to: 0x4
INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x2 set to: 0x4
DEBUG:      ClubUpdater: Finished writing relay outputs with: 0x4
INFO:       ClubUpdater: status switch status changed to on
INFO:       ClubUpdater: Opening club.
INFO:       ClubUpdater: Started power timer...
DEBUG:      ClubUpdater: Sent MQTT message.
INFO:       ClubUpdater: New lights, clubstatus on.
DEBUG:      ClubUpdater: Power timer active, inverting power state from: on
INFO:       ClubUpdater: Green on.
DEBUG:      ClubUpdater: Changing output register to: 0x2
INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x2 set to: 0x2
DEBUG:      ClubUpdater: Finished writing relay outputs with: 0x2
INFO:       ClubUpdater: status switch status changed to off
INFO:       ClubUpdater: Closing club.
INFO:       ClubUpdater: Started timer.
INFO:       ClubUpdater: Started power timer...
DEBUG:      ClubUpdater: Sent MQTT message.
INFO:       ClubUpdater: New lights, clubstatus off.
DEBUG:      ClubUpdater: Power timer active, inverting power state from: off
INFO:       ClubUpdater: Yellow on.
DEBUG:      ClubUpdater: Changing output register to: 0x5
INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x2 set to: 0x5
DEBUG:      ClubUpdater: Finished writing relay outputs with: 0x5
INFO:       ClubUpdater: setPowerState called.
DEBUG:      ClubUpdater: Writing relay with: 0x4
INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x2 set to: 0x4
DEBUG:      ClubUpdater: Finished writing relay outputs with: 0x4
DEBUG:      ClubUpdater: Written relay outputs.
DEBUG:      ClubUpdater: Finished setPowerState.
INFO:       ClubUpdater: lock status changed to locked
INFO:       ClubUpdater: New lights, clubstatus off.
DEBUG:      ClubUpdater: Power timer not active, using current power state: off
INFO:       ClubUpdater: Red on.
DEBUG:      ClubUpdater: Changing output register to: 0x8
INFO:       wiringPiI2CWriteReg8: Device handle 0x0, Register 0x2 set to: 0x8
DEBUG:      ClubUpdater: Finished writing relay outputs with: 0x8
```

Далее таймер начинает регулярно повторяющуюся генерацию обращений к функции обратного вызова, что приводит к переходам в различные состояния. Это позволяет убедиться в правильности функционирования кода приложения в целом.

После этого можно приступить к реализации более сложных вариантов тестирования, возможно, даже с применением скриптов, организующих тесты во встроенной среде выполнения языков Lua, Python и т. п.

Имитация или реальная аппаратура

При имитации крупных блоков кода и аппаратных периферийных устройств возникает естественный вопрос: насколько реальными являются результаты такой имитации. Разумеется, в комплексном тесте необходимо охватить максимально возможное количество вариантов поведения системы в условиях реальной эксплуатации, прежде чем мы перейдем к тестированию настоящей целевой системы.

Если необходимо знать, какие варианты тестирования должны быть покрыты создаваемой имитацией, то следует внимательно рассмотреть проектные требования (что именно должно управляться), а также ситуации и входные сигналы, которые могут возникать в процессе реальной эксплуатации.

Для этого необходимо проанализировать код нижнего уровня, чтобы узнать, какие ситуации могут возникать, и решить, какие из этих ситуаций являются приемлемыми для нас.

Для рассмотренной выше имитации WiringPi при беглом просмотре исходного кода реализации библиотеки становится ясно, насколько упрощен код по сравнению с версией для реальной целевой системы.

Рассматривая основную функцию настройки WiringPi, мы видим, что она выполняет следующие действия:

- точно определяет модель платы и системы на кристалле, чтобы получить схему размещения контактов GPIO;
- открывает устройство ОС Linux для установления соответствия размещения в памяти контактов GPIO;
- устанавливает смещения в памяти для устройства GPIO и использует функцию `mmap()` для установления отображения в памяти конкретных периферийных устройств, таких как PWM, таймер и GPIO.

Вместо игнорирования вызовов функции `pinMode()` рассматриваемая реализация выполняет следующие действия:

- соответствующим образом настраивает аппаратный регистр направления GPIO в системе на кристалле (для установки режима ввода/вывода);
- активизирует устройство широтно-импульсной модуляции (PWM), программную версию PWM или режим Tone на контакте (по запросу); вспомогательные функции устанавливают соответствующие регистры.

Процесс продолжается на стороне шины I2C, где реализация функции настройки выглядит следующим образом:

```
int wiringPiI2CSetup (const int devId) {
   int rev;
   const char *device;
   rev = piGpioLayout();
   if (rev == 1) {
         device = "/dev/i2c-0";
   }
```

```
   else {
         device = "/dev/i2c-1";
   }
   return wiringPiI2CSetupInterface (device, devId);
}
```

Основное отличие от имитационной реализации состоит в том, что предполагается наличие периферийного устройства I2C в файловой системе, размещенной в памяти ОС, а выбор определяется конкретной версией платы.

Последняя вызываемая функция пытается открыть это устройство, так как в Linux и в подобных ОС каждое устройство представлено обычным файлом, который можно открыть и получить соответствующий дескриптор этого файла. Файловый дескриптор – это идентификатор, возвращаемый как результат выполнения функции.

```
int wiringPiI2CSetupInterface (const char *device, int devId) {
   int fd;
   if ((fd = open (device, O_RDWR)) < 0) {
         return wiringPiFailure (WPI_ALMOST, "Unable to open I2C device: %s\n", strerror
(errno));
   }
   if (ioctl (fd, I2C_SLAVE, devId) < 0) {
         return wiringPiFailure (WPI_ALMOST, "Unable to select I2C device: %s\n", strerror
(errno));
   }
   return fd;
}
```

После открытия файла устройства I2C системная функция ОС Linux `ioctl()` используется для передачи данных в периферийное устройство I2C. В данном случае необходимо использовать адрес подчиненного устройства I2C. При успешном завершении операции мы получаем неотрицательный ответ и возвращаем целое число, сооответствующее дескриптору файла `fd`.

Запись и чтение на шине I2C также выполняются с использованием системного вызова `ioctl()`, как можно видеть в том же файле исходного кода:

```
static inline int i2c_smbus_access (int fd, char rw, uint8_t command, int size,
                                    union i2c_smbus_data *data) {
   struct i2c_smbus_ioctl_data args;
   args.read_write = rw;
   args.command    = command;
   args.size       = size;
   args.data       = data;
   return ioctl(fd, I2C_SMBUS, &args);
}
```

Эта встроенная (inline) функция вызывается при каждой операции доступа к шине I2C. Поскольку требуемое устройство I2C уже выбрано, можно просто указать идентификатор периферийного устройства I2C и начать процедуру передачи полезной нагрузки в это устройство.

Здесь тип `i2c_smbus_data` – это простое объединение (union), поддерживающее различные размеры возвращаемого значения (при выполнении операции чтения):

```
union i2c_smbus_data {
   uint8_t byte;
   uint16_t word;
   uint8_t block[I2C_SMBUS_BLOCK_MAX + 2];
};
```

Здесь наиболее ярко проявляется преимущество использования абстрактного прикладного программного интерфейса (API). Без него исходный код был бы перегружен многочисленными вызовами низкого уровня, которые гораздо труднее имитировать. Кроме того, очевидно, что существует множество условий, которые также должны быть протестированы: отсутствие подчиненного устройства I2C, ошибки чтения/записи на шине I2C в результате непредвиденного поведения, непредусмотренные входные сигналы на контактах GPIO, включая прерывания контактов, которые уже были отмечены в начале этой главы.

Разумеется, все вероятные варианты и сценарии запланировать невозможно, но необходимо задокументировать все действительно возможные ситуации и включить их в имитационную реализацию, чтобы сделать доступными при комплексном и регрессионном тестировании и при отладке.

Тестирование с использованием Valgrind

Valgrind – это самый известный и широко используемый комплект инструментов с открытым исходным кодом для анализа и профилирования поведения всех компонентов приложения от кеша и динамической памяти до выявления утечек памяти и потенциальных проблем в многопоточном режиме. Valgrind работает в тесном сотрудничестве с используемой операционной системой, так как в зависимости от применяемого конкретного инструмента должен перехватывать любое действие от операций распределения памяти до инструкций, связанных с многопоточностью, и т. п. Поэтому полная поддержка Valgrind возможна только на 64-битовых архитектурах Linux x86_64.

Использование Valgrind на других поддерживаемых платформах (Linux x86, PowerPC, ARM, S390, MIPS, ARM, а также Solaris и macOS) вполне возможно, но главной целевой платформой разработки проекта Valgrind является x86_64/Linux, что делает ее наилучшей платформой для профилирования и отладки. Целевая разработка для прочих платформ также будет обеспечена в будущем.

Полный список и обзор поддерживаемых в настоящее время платформ можно найти на официальном сайте Valgrind **http://valgrind.org/info/platforms.html**.

Одним из наиболее привлекательных свойств Valgrind является то, что ни один из его инструментов не требует внесения каких бы то ни было изменений в исходный код и в итоговые бинарные файлы. Поэтому Valgrind чрезвычайно просто интегрируется в любой существующий рабочий поток, включая системы автоматизированного тестирования и интеграции.

В Windows-системах также доступны такие инструментальные средства, как Dr. Memory (**http://drmemory.org/**), которые могут выполнять, как мимимум, профилирование поведения, связанного с использованием памяти. В этот инструментальный комплект также включено средство Dr Fuzz – инструмент, способный многократно вызывать функции с разнообразными входными данными, что может оказаться весьма полезным при комплексном тестировании.

Применяя комплексный тест, подобный рассмотренному в предыдущем разделе, можно без каких-либо ограничений провести полный анализ поведения кода, используя комфортную среду настольного персонального компьютера. Поскольку все инструменты из комплекта Valgrind существенно замедляют выполнение кода (в 10–100 раз), возможность проведения большей части отладки и профилирования на быстрой системе означает значительную экономию времени до перехода к тестированию на целевой аппаратуре.

Из наиболее часто используемых инструментальных средств Memcheck, Helgrind и DRD весьма полезны для выявления проблем с распределением памяти и организации многопоточности. После прогона кода через эти три инструмента в совокупности с применением всеобъемлющего комплексного теста, обеспечивающего полное покрытие кода, можно переходить к профилированию и оптимизации.

Для профилирования кода используется инструмент Callgrind, позволяющий обнаружить те места в коде, которые требуют наибольшего времени для выполнения. Затем применяется Massif для профилирования операций распределения динамической памяти. С помощью информации, извлеченной из этих данных, можно вносить в код изменения, улучшающие общую стратегию во всех случаях выделения и освобождения памяти. Эти инструменты также помогают определить места, где имеет смысл применить кеш-память для повторного использования ресурсов вместо многократных операций выделения/освобождения памяти.

В завершение выполняется еще один цикл использования Memcheck, Helgrind и DRD для полной уверенности в том, что внесенные изменения не привели к снижению эффективности. После успешного завершения этого цикла код развертывается на целевой системе, и оценивается качество его выполнения.

Если целевая система работает под управлением ОС Linux или другой поддерживаемой ОС, можно также использовать Valgrind для дополнительной проверки и тестирования. В зависимости от конкретной платформы (ОС и архитектура ЦПУ) могут существовать некоторые ограничения в применении данной версии Valgrind. Возможны ошибки типа «недопустимая инструкция» в тех случаях, когда в инструментальном средстве отсутствует реализация какой-либо инструкции ЦПУ, из-за чего Valgrind не может продолжить работу.

При расширении комплексного теста для использования одноплатного компьютера вместо локального процесса можно настроить систему непрерывной интеграции так, чтобы в дополнение к тестам в локальном процессе выполнялось тестирование и на реальной аппаратуре с учетом ограничений аппаратной платформы по сравнению с системой на основе Linux86_64, в основном используемой в процессе тестирования.

Многоцелевая система сборки

Системы кросс-компиляции и многоцелевой сборки – термины, которые выглядят слишком сложными и слегка пугающими для большинства людей, главным образом из-за того, что связаны с представлением о сверхсложных скриптах сборки, которые требуют тайных мистических заклинаний для выполнения необходимых операций. В этом разделе мы рассмотрим простую систему сборки на основе

Makefile, которая действительно использовалась в коммерческих проектах, ориентированных на несколько вариантов целевой аппаратуры.

Один из факторов, обеспечивающих удобство использования системы сборки, – это возможность полной настройки компиляции с минимальными трудозатратами и наличие централизованного пункта управления всеми аспектами сборки проекта или его составных частей в совокупности с созданием и выполнением тестов.

Поэтому мы используем в самом верхнем пункте проекта единственный Makefile, который обрабатывает все основные компоненты, включая определение платформы, для которой предназначено создаваемое приложение. Здесь допущено единственное упрощение: предполагается использование Unix-подобной среды с MSYS2 или Cygwin в ОС Windows или Linux, BSD, OS X/macOS и подобных систем с собственной командной оболочкой (shell). Тем не менее возможна также адаптация к среде Microsoft Visual Studio, Intel Compiler Collection (ICC) и другим компиляторам, при условии что они предоставляют требуемые инструментальные средства.

Самым главным компонентом системы сборки являются простые файлы Makefile, в которых определяются подробности и характеристики конкретной целевой платформы, например для стандартной системы Linux, работающей на аппаратуре x86_x64:

```
TARGET_OS = linux
TARGET_ARCH = x86_64

export CC = gcc
export CXX = g++
export CPP = cpp
export AR = ar
export LD = g++
export STRIP = strip
export OBJCOPY = objcopy

PLATFORM_FLAGS = -D__PLATFORM_LINUX__ -D_LARGEFILE64_SOURCE -D __LINUX__ STD_FLAGS = \
                 $(PLATFORM_FLAGS) -Og -g3 -Wall -c -fmessage-length=0 \
                 -ffunction-sections -fdata-sections -DPOCO_HAVE_GCC_ATOMICS \
                 -DPOCO_UTIL_NO_XMLCONFIGURATION -DPOCO_HAVE_FD_EPOLL
STD_CFLAGS = $(STD_FLAGS)
STD_CXXFLAGS = -std=c++11 $(STD_FLAGS)
STD_LDFLAGS = -L $(TOP)/build/$(TARGET)/libboost/lib \
                         -L $(TOP)/build/$(TARGET)/poco/lib \
                         -Wl,--gc-sections
STD_INCLUDE = -I. -I $(TOP)/build/$(TARGET)/libboost/include \
                         -I $(TOP)/build/$(TARGET)/poco/include \
                         -I $(TOP)/extern/boost-1.58.0
STD_LIBDIRS = $(STD_LDFLAGS)
STD_LIBS = -ldl -lrt -lboost_system -lssl -lcrypto -lpthread
```

Здесь можно определить имена инструментов командной строки, которые будут использоваться при компиляции, создании архивных файлов, удалении отладочных символов из бинарных файлов и т. д. Система сборки использует название целевой ОС и архитектуры для раздельного хранения создаваемых

бинарных файлов, поэтому можно пользоваться одним и тем же деревом исходного кода для создания бинарных файлов для всех целевых платформ за один проход.

Здесь мы видим, как флаги, передаваемые компилятору и линкеру, разделяются по различным категориям: флаги, зависимые от платформы, общие (стандартные) флаги, наконец, флаги, специально предназначенные для компилятора C и C++. Флаги для компилятора C необходимы, если в дерево исходного кода включены внешние зависимости, написанные на C. Такие зависимости собраны в подкаталоге *extern*, и немного позже мы рассмотрим их более подробно.

Файл такого типа тщательно специализируется и настраивается для соответствия конкретному проекту: добавляются требуемые включаемые файлы, библиотеки и флаги компиляции. В файле, рассматриваемом в нашем примере, можно видеть использование в проекте библиотек POCO и Boost, а также OpenSSL, адаптирующей библиотеку POCO для целевой платформы.

Теперь рассмотрим начальную секцию конфигурационного файла для macOS:

```
TARGET_OS = osx
TARGET_ARCH = x86_64

export CC = clang
export CXX = clang++
export CPP = cpp
export AR = ar
export LD = clang++
export STRIP = strip
export OBJCOPY = objcopy
```

Остальная часть файла почти та же самая, но в начальной секции мы видим удачный пример обобщения вызываемого инструментального средства. Несмотря на то что Clang поддерживает те же флаги, что и GCC, эти инструменты вызываются по-разному. При таком подходе мы просто записываем другие имена один раз в самом начале файла, а вся остальная часть файла продолжает корректно работать.

Аналогичная ситуация и для Linux на целевой платформе ARM, где настраивается кросс-компиляция для этой платформы:

```
TARGET_OS = linux
TARGET_ARCH = armv7
TOOLCHAIN_NAME = arm-linux-gnueabihf

export CC = $(TOOLCHAIN_NAME)-gcc
export CXX = $(TOOLCHAIN_NAME)-g++
export AR = $(TOOLCHAIN_NAME)-ar
export LD = $(TOOLCHAIN_NAME)-g++
export STRIP = $(TOOLCHAIN_NAME)-strip
export OBJCOPY = $(TOOLCHAIN_NAME)-objcopy
```

Здесь наблюдается определение другого инструментального комплекта кросс-компиляции для платформ ARM Linux, рассматриваемых ранее в этой главе. Для сокращения кода, вводимого вручную, основное имя определяется один раз в начале файла, и это позволяет с легкостью изменить его при необходимости. Кроме того, это наглядно демонстрирует гибкость и универсальность Makefile. Подклю-

чив немного творческого мышления, мы можем создать набор шаблонов, которые обобщают включение комплектов инструментов в простой Makefile, включаемый в основной Makefile в зависимости от определений в Makefile конкретной платформы (или в другом конфигурационном файле). Это весьма гибкое и универсальное решение.

Далее рассмотрим содержимое основного Makefile, расположенного в корневом каталоге проекта.

```
ifndef TARGET
 $(error TARGET parameter not provided.)
endif
```

Поскольку невозможно заранее узнать, какую платформу пользователь выбрал для проекта, требуется явное указание цели с именем платформы, задаваемым как значение, например `linux-x86_x64`.

```
export TOP := $(CURDIR)
export TARGET
```

Позже потребуется узнать текущий каталог в локальной файловой системе, поэтому можно определить абсолютные пути. Здесь для этого используется стандартная переменная **make**, которая экспортируется как наша собственная переменная среды вместе с именем цели сборки.

```
UNAME := $(shell uname)
ifeq ($(UNAME), Linux)
export HOST = linux
else
export HOST = win32
export FILE_EXT = .exe
endif
```

Используя утилиту командной строки `uname`, можно узнать имя текущей операционной системы, если командная оболочка поддерживает такую команду, возвращающую, например, `Linux` в ОС Linux или `Darwin` в macOS. В ОС Windows (без MSYS2 или Cygwin) такой команды нет, поэтому будет выполнена вторая часть оператора `if/else`.

Этот оператор можно расширить для поддержки большего количества ОС в зависимости от того, для каких систем будет предусматриваться сборка. В рассматриваемом примере оператор `if/else` используется только для определения наличия или отсутствия файлового расширения (.exe для Windows) для создаваемого выполняемого файла.

```
ifeq ($(HOST), linux)
export MKDIR   = mkdir -p
export RM      = rm -rf
export CP      = cp -RL
else
export MKDIR   = mkdir -p
export RM      = rm -rf
export CP      = cp -RL
endif
```

В этом операторе `if/else` устанавливаются команды для часто применяемых операций с файлами. Поскольку рассматривается упрощенный пример, здесь предполагается использование MSYS2 или аналогичной реализации командной оболочки bash в Windows.

Концепцию обобщения можно развивать и дальше, выделив определение инструментов командной строки для работы с файлами для каждой ОС в отдельный набор файлов Makefile, которые также включаются как часть настроек для каждой конкретной ОС.

```
include Makefile.$(TARGET)

export TARGET_OS
export TARGET_ARCH
export TOOLCHAIN_NAME
```

Здесь используется подставляемое значение параметра `TARGET`, передаваемое в Makefile для включения соответствующего конфигурационного файла. После экспортирования некоторых подробностей из этого файла мы получаем полностью сконфигурированную систему сборки.

```
all: extern-$(TARGET) core

extern:
    $(MAKE) -C ./extern $(LIBRARY)

extern-$(TARGET):
    $(MAKE) -C ./extern all-$(TARGET)

core:
    $(MAKE) -C ./Core

clean: clean-core clean-extern

clean-extern:
    $(MAKE) -C ./extern clean-$(TARGET)

clean-core:
    $(MAKE) -C ./Core clean

.PHONY: all clean core extern clean-extern clean-core extern-$(TARGET)
```

Используя этот Makefile, можно выбрать компиляцию всего проекта в целом, или только компиляцию зависимостей, или только компиляцию ядра (core) проекта. Кроме того, можно скомпилировать лишь заданную внешнюю зависимость и ничего больше.

Наконец, можно очистить (то есть удалить все скомпилированные бинарные и вспомогательные файлы) ядро проекта, зависимости или весь проект.

Этот Makefile самого верхнего уровня главным образом предназначен для управления файлами Makefile более низких уровней. Следующие два файла Makefile размещены в подкаталогах *Core* и *extern*. Makefile Core просто компилирует ядро проекта.

```
include ../Makefile.$(TARGET)

OUTPUT := CoreProject

INCLUDE = $(STD_INCLUDE)
```

```
LIBDIRS = $(STD_LIBDIRS)

include ../version
VERSIONINFO = -D__VERSION="\"$(VERSION)\""
```

В первой строке включается конфигурационный файл для целевой платформы, чтобы обеспечить доступ ко всем его определениям. Можно было бы экспортировать эти определения непосредственно в главном Makefile, но при используемом здесь подходе мы получаем гораздо бóльшую свободу в настройке системы сборки.

Определяется имя создаваемого выходного бинарного файла, далее выполняются некоторые небольшие вспомогательные задачи, в том числе открытие файла *version* (с использованием синтаксиса Makefile) в корневом каталоге проекта. Этот файл содержит номер версии исходного кода, используемого при сборке. Номер версии готовится к передаче в компилятор как определение препроцессора.

```
ifdef RELEASE
TIMESTAMP = \
$(shell date --date=@`git show -s --format=%ct$(RELEASE)^{commit}` -u +%Y-%m-%dT%H:%M:%SZ)
else ifdef GITTIME
TIMESTAMP = $(shell date --date=@`git show -s --format=%ct` -u +%Y-%m-%dT%H:%M:%SZ)
TS_SAFE = _$(shell date --date=@`git show -s --format=%ct` -u +%Y-%m-%dT%H%M%SZ)
else
TIMESTAMP = $(shell date -u +%Y-%m-%dT%H:%M:%SZ)
TS_SAFE = _$(shell date -u +%Y-%m-%dT%H%M%SZ)
endif
```

В этой секции Makefile снова используется командная оболочка bash или совместимая с ней: выполняется команда `date` для создания метки времени для сборки. Формат зависит от параметра, который был передан в главный Makefile. Если создается эксплуатационная версия (релиз), то метка времени берется из репозитория Git с именем тега коммита Git, используемого для извлечения метки времени этого коммита перед форматированием соответствующего тега.

Если метка `GITTIME` передана как параметр, то используется самый последний коммит Git. В противном случае применяется текущая дата и время (UTC).

Этот фрагмент кода предназначен для решения одной из проблем, возникающей при наличии большого количества тестовых и интеграционных сборок: отслеживания даты и времени создания каждой такой сборки и номера версии исходного кода. Эту секцию можно адаптировать к любой другой системе управления версиями файлов, если она поддерживает функции извлечения файлов с заданными метками времени.

Здесь же создается вторая метка времени. Это немного измененный формат метки времени, которая присоединяется к создаваемому бинарному файлу, за исключением указания режима релиза.

```
CFLAGS = $(STD_CFLAGS) $(INCLUDE) $(VERSIONINFO) -
D__TIMESTAMP="\"$(TIMESTAMP)\""
CXXFLAGS = $(STD_CXXFLAGS) $(INCLUDE) $(VERSIONINFO) -
D__TIMESTAMP="\"$(TIMESTAMP)\""

OBJROOT := $(TOP)/build/$(TARGET)/obj
CPP_SOURCES := $(wildcard *.cpp)
```

```
CPP_OBJECTS := $(addprefix $(OBJROOT)/,$(CPP_SOURCES:.cpp=.o))
OBJECTS := $(CPP_OBJECTS)
```

Определяются флаги, которые необходимо передать компилятору, в том числе номер версии и метка времени, передаваемые как директивы препроцессора.

Далее собираются все файлы исходного кода в текущем каталоге проекта и определяется каталог для вывода объектных файлов. В рассматриваемом примере объектные файлы будут записываться в подкаталог, расположенный в корневом каталоге проекта, с дальнейшим разделением по цели компиляции.

```
.PHONY: all clean

all: makedirs $(CPP_OBJECTS) $(C_OBJECTS)
$(TOP)/build/bin/$(TARGET)/$(OUTPUT)_$(VERSION)_$(TARGET)$(TS_SAFE)
makedirs:
   $(MKDIR) $(TOP)/build/bin/$(TARGET)
   $(MKDIR) $(OBJROOT)
   $(OBJROOT)/%.o: %.cpp
   $(CXX) -o $@ $< $(CXXFLAGS)
```

Эта часть почти одинакова во всех файлах Makefile. Здесь указана цель `all` (все), а также операция создания каталогов в файловой системе, если они не существуют. Далее набор файлов исходного кода в следующей целевой ветви компилируется в соответствии с заданной конфигурацией, а созданные объектные файлы выводятся в соответствующий каталог.

```
$(TOP)/build/bin/$(TARGET)/$(OUTPUT)_$(VERSION)_$(TARGET)$(TS_SAFE):
$(OBJECTS)
   $(LD) -o $@ $(OBJECTS) $(LIBDIRS) $(LIBS)
   $(CP) $@ $@.debug
ifeq ($(TARGET_OS), osx)
   $(STRIP) -S $@
else
   $(STRIP) -S --strip-unneeded $@
endif
```

После создания всех объектных файлов из файлов исходного кода необходимо объединить (связать) их. Для этого предназначена приведенная выше секция Makefile. Здесь также можно видеть, куда в конечном итоге попадают бинарные файлы: в подкаталог *bin*, расположенный в каталоге сборки *build* проекта.

Вызывается линкер (редактор связей) и создается копия итогового бинарного файла с суффиксом *.debug*, обозначающим версию, содержащую отладочную информацию. Затем из оригинала бинарного файла удаляются отладочные символы и прочая ненужная информация, чтобы получить компактный бинарный файл для копирования на удаленную тестовую систему. Копия этого бинарного файла со всей отладочной информацией необходима для анализа дампов памяти и для выполнения отладки в удаленном режиме.

Здесь применен небольшой трюк, использующий недокументированный флаг командной строки линкера Clang и требующий реализации отдельного особого варианта. При кросс-платформенной компиляции и выполнении связанных с ней задач, вероятнее всего, придется иметь дело с такими мелкими деталями, кото-

рые усложняют написание универсальной системы сборки, которая всегда сохраняет работоспособность.

```
clean:
   $(RM) $(CPP_OBJECTS)
   $(RM) $(C_OBJECTS)
```

На завершающем этапе удаляются все сгенерированные объектные файлы.

Второй вспомогательный Makefile в подкаталоге *extern* также заслуживает внимания, поскольку управляет всеми зависимостями более низкого уровня.

```
ifndef TARGET
$(error TARGET parameter not provided.)
endif

all: libboost poco

all-linux-%:
   $(MAKE) libboost poco

all-qnx-%:
   $(MAKE) libboost poco
all-osx-%:
   $(MAKE) libboost poco
all-windows:
   $(MAKE) libboost poco
```

Здесь интересной функциональной возможностью является механизм выбора конкретной зависимости на основе заданной целевой платформы. Если имеются зависимости, которые не должны компилироваться для заданной платформы, то их можно пропустить с помощью этого механизма. Эта функциональная возможность также позволяет напрямую указывать файлу Makefile необходимость компиляции всех зависимостей для конкретной платформы. В рассматриваемом здесь примере допускается указание зависимостей для Linux, QNX, OS X/macOS и Windows без учета архитектуры.

```
libboost:
   cd boost-1.58.0 && $(MAKE)
poco:
   cd poco-1.7.4 && $(MAKE)
```

В строках реальных целей явно вызывается другой Makefile, расположенный на верхнем уровне проекта зависимостей, который, в свою очередь, компилирует эти зависимости и добавляет их в каталог сборки, где ими может воспользоваться Makefile для ядра Core.

Разумеется, можно также скомпилировать проект непосредственно из этого Makefile, используя существующую систему сборки, как показано в варианте для OpenSSL:

```
openssl:
   $(MKDIR) $(TOP)/build/$(TARGET)/openssl
   $(MKDIR) $(TOP)/build/$(TARGET)/openssl/include
   $(MKDIR) $(TOP)/build/$(TARGET)/openssl/lib
   cd openssl-1.0.2 && ./Configure --openssldir="$(TOP)/build/$(TARGET)/openssl" shared \
      os/compiler:$(TOOLCHAIN_NAME):$(OPENSSL_PARAMS) && \
```

```
    $(MAKE) build_libs
  $(CP) openssl-1.0.2/include $(TOP)/build/$(TARGET)/openssl
  $(CP) openssl-1.0.2/libcrypto.a $(TOP)/build/$(TARGET)/openssl/lib/.
  $(CP) openssl-1.0.2/libssl.a $(TOP)/build/$(TARGET)/openssl/lib/.
```

Этот код выполняет все обычные этапы ручной сборки библиотеки OpenSSL, после чего копирует созданные бинарные файлы в соответствующие целевые подкаталоги.

Одна из существенных проблем в кроссплатформенных системах сборки заключается в том, что широко используемые инструменты GNU, такие как Autoconf, значительно замедляют работу операционных систем, подобных Windows, потому что инициализируют множество процессов при выполнении сотен тестов. Даже в ОС Linux этот процесс может продолжаться достаточно долго. Это весьма неприятная ситуация, особенно если необходимо выполнять процесс сборки несколько раз в день.

Идеальным вариантом является единственный Makefile, в котором предварительно определено все необходимое в точно известном состоянии, поэтому нет необходимости в проверке существования библиотек и прочих требуемых компонентов. Это одна из причин, по которой исходный код библиотеки POCO добавляется в конкретный проект и создается простой файл для компиляции:

```
include ../../Makefile.$(TARGET)

all: poco-foundation poco-json poco-net poco-util

poco-foundation:
   cd Foundation && $(MAKE)
poco-json:
   cd JSON && $(MAKE)
poco-net:
   cd Net && $(MAKE)
poco-util:
   cd Util && $(MAKE)
clean:
   cd Foundation && $(MAKE) clean
   cd JSON && $(MAKE) clean
   cd Net && $(MAKE) clean
   cd Util && $(MAKE) clean
```

Затем этот Makefile вызывает отдельные файлы Makefile для каждого модуля, как показано в следующем примере:

```
include ../../../Makefile.$(TARGET)

OUTPUT = libPocoNet.a
INCLUDE = $(STD_INCLUDE) -Iinclude
CFLAGS = $(STD_CFLAGS) $(INCLUDE)
OBJROOT = $(TOP)/extern/poco-1.7.4/Net/$(TARGET)
INCLOUT = $(TOP)/build/$(TARGET)/poco
SOURCES := $(wildcard src/*.cpp)
HEADERS := $(addprefix $(INCLOUT)/,$(wildcard include/Poco/Net/*.h))

OBJECTS := $(addprefix $(OBJROOT)/,$(notdir $(SOURCES:.cpp=.o)))
```

```
all: makedir $(OBJECTS) $(TOP)/build/$(TARGET)/poco/lib/$(OUTPUT)
$(HEADERS)

$(OBJROOT)/%.o: src/%.cpp
   $(CC) -c -o $@ $< $(CFLAGS)
makedir:
   $(MKDIR) $(TARGET)
   $(MKDIR) $(TOP)/build/$(TARGET)/poco
   $(MKDIR) $(TOP)/build/$(TARGET)/poco/lib
   $(MKDIR) $(TOP)/build/$(TARGET)/poco/include
   $(MKDIR) $(TOP)/build/$(TARGET)/poco/include/Poco
   $(MKDIR) $(TOP)/build/$(TARGET)/poco/include/Poco/Net
$(INCLOUT)/%.h: %.h
   $(CP) $< $(INCLOUT)/$<

$(TOP)/build/$(TARGET)/poco/lib/$(OUTPUT): $(OBJECTS)
   -rm -f $@
   $(AR) rcs $@ $^
clean:
   $(RM) $(OBJECTS)
```

Этот Makefile компилирует полный модуль `Net` библиотеки POCO. По своей структуре Makefile похож на аналогичный файл для компиляции исходного кода проекта. Кроме компиляции объектных файлов, здесь также выполняется их упаковка в архивный формат, чтобы в дальнейшем устанавливать связь с ним. Архивный файл, как и заголовочные файлы, копируется в соответствующий каталог сборки.

Главная причина компиляции библиотеки POCO непосредственно в проекте – возможность специализированных оптимизаций и тонкой настройки, что недоступно при использовании предварительно скомпилированной библиотеки. При таком подходе очень легко управлять содержимым исходной системы сборки библиотеки, экспериментировать с разнообразными параметрами настройки, и этот метод работает даже в Windows.

Удаленное тестирование на реальной аппаратуре

После завершения всех этапов локального тестирования кода приложения и обоснованного предположения о том, что приложение должно работать на реальной аппаратуре, можно использовать систему кросс-компиляции и сборки для создания бинарного файла, предназначенного для работы на целевой системе.

Для этого можно просто скопировать итоговый бинарный файл и все вспомогательные файлы на целевую систему и проверить, работает ли приложение. Но более разумно использовать отладчик GDB. С помощью сервера сопровождения GDB, установленного на целевой системе Linux, можно установить соединение с отладчиком GDB с настольного компьютера через сеть или через последовательный интерфейс.

На одноплатном компьютере под управлением дистрибутива Linux на основе Debian сервер GDB устанавливается чрезвычайно просто:

```
sudo apt install gdbserver
```

Несмотря на название gdbserver, основной функцией этой программы является реализация удаленной заглушки (stub) для отладчика, работающего на хост-системе. Поэтому gdbserver представляет собой весьма упрощенную программу, которую легко реализовать для новых целей.

После установки необходимо убедиться в том, что gdbserver работоспособен, войдя в систему и запустив эту программу одним из нескольких способов. Для TCP-соединений по сети это можно сделать следующим образом:

```
gdbserver host:2345 <program> <parameters>
```

Другой способ – присоединение к активному работающему процессу:

```
gdbserver host:2345 --attach <PID>
```

В первом аргументе часть host определяет имя или IP-адрес хост-системы, с которой устанавливается соединение. В рассматриваемом здесь примере эта часть параметра игнорируется, поэтому его можно было бы оставить пустым. Во второй части первого аргумента (после двоеточия) должен указываться номер порта, который в данном примере не используется на целевой системе.

Еще один вариант – использование некоторого типа соединения через последовательный интерфейс:

```
gdbserver /dev/tty0 <program> <parameters>
gdbserver --attach /dev/tty0 <PID>
```

В момент запуска gdbserver он создает паузу в выполнении целевого приложения, если оно уже работает, позволяя установить соединение с отладчиком на хост-системе. В целевой системе можно запустить бинарный файл, из которого удалены отладочные символы. Но эти отладочные символы должны обязательно присутствовать в бинарном файле, который используется на стороне хоста.

```
$ gdb-multiarch <program>
(gdb) target remote <IP>:<port>
Remote debugging using <IP>:<port>
```

В этот момент отладочные символы должны быть загружены из бинарного файла, а кроме того, должны загружаться отладочные символы из всех зависимостей (если это необходимо). Соединение через последовательный интерфейс почти ничем не отличается, разве что вместо сетевого адреса и номера порта используется путевое имя последовательного интерфейса. Скорость передачи данных baud rate при соединении через последовательный интерфейс (если не используется скорость по умолчанию 9600 бод) определяется как параметр отладчика GDB при его запуске:

```
$ gdb-multiarch -baud <baud rate> <program>
```

После передачи отладчику GDB всех подробностей об удаленной цели мы должны увидеть обычный интерфейс командной строки отладчика, позволяющий пошагово выполнять, анализировать и отлаживать программу, как если бы она работала локально на настольной системе.

Как уже было отмечено ранее, мы используем отладчик gdb-multiarch, поскольку эта версия поддерживает различные архитектуры. Это очень удобно, так как сам

отладчик, вероятнее всего, будет работать на системе x86_64, тогда как тестируемый одноплатный компьютер может иметь архитектуру ARM, MIPS или x86 (i686).

В дополнение к работе приложения непосредственно с `gdbserver` также можно запускать `gdbserver` в режиме ожидания соединения с отладчиком:

```
gdbserver --multi <host>:<port>
```

Вариант для соединения через последовательный интерфейс:

```
gdbserver --multi <serial port>
```

Затем устанавливается соединение с этой удаленной целью, как показано ниже:

```
$ gdb-multiarch <program>
(gdb) target extended-remote <remote IP>:<port>
(gdb) set remote exec-file <remote file path>
(gdb) run
```

Здесь также можно видеть появление интерфейса командной строки отладчика GDB с загрузкой бинарного файла программы и на целевой системе, и на хосте.

Большим преимуществом такого метода является тот факт, что `gdbserver` не завершает работу, когда выполняется выход из отлаживаемого приложения. Кроме того, такой режим позволяет одновременно отлаживать несколько различных приложений на одной целевой системе, если целевая система поддерживает одновременное выполнение нескольких программ.

Резюме

В этой главе рассматривались методы разработки и тестирования встроенных приложений на основе ОС. Вы узнали, как установить и использовать комплект инструментов для кросс-компиляции, как организовать отладку в удаленном режиме с помощью отладчика GDB, как написать систему сборки, позволяющую компилировать бинарные файлы для разнообразных целевых систем с минимальными трудозатратами при добавлении новой целевой системы.

Предполагается, что после изучения этой главы читатель получает возможность самостоятельно разрабатывать и отлаживать встроенные приложения для одноплатных компьютеров на основе Linux или подобных ОС наиболее эффективным методом.

В следующей главе рассматриваются методы разработки и тестирования приложения для платформ с более ограниченными ресурсами на основе микроконтроллеров.

Глава 7

Тестирование платформ с ограниченными ресурсами

Разработка для микроконтроллеров и аналогичных платформ с ограниченными ресурсами ведется главным образом на обычных настольных компьютерах, за исключением этапа тестирования и отладки. Вопрос в том, когда следует начинать тестирование на целевом физическом устройстве и в каких случаях разработчик должен обращаться к альтернативным средствам тестирования и отладки кода для ускорения разработки и сокращения трудозатрат на отладку.

В этой главе рассматриваются следующие темы:

- определение ресурсов, необходимых для конкретного специализированного кода;
- эффективное использование инструментальных средств на основе Linux для тестирования кроссплатформного кода;
- использование удаленной отладки;
- использование кросс-компиляторов;
- создание системы сборки, независмой от платформы.

Снижение степени износа оборудования

Во время разработки часто используют методику, по которой после каждого устранения проблемы в системе повторяется один и тот же цикл: изменение – компиляция – развертывание – тестирование. Такому многократному повторению цикла присущи следующие недостатки:

- процедура утомительна – постоянное ожидание результатов без ясного понимания, будет ли действительно устранена проблема после внесения очередных изменений на этот раз, не добавляет оптимизма;
- процедура непродуктивна – разработчик затрачивает слишком много времени на ожидание результатов, которые не потребовались бы при более глубоком и тщательном анализе возникшей проблемы;
- при многократном выполнении процедуры изнашивается аппаратура – после десятков соединений и разъединений разъемов, бесконечного количества записей и перезаписей одних и тех же секций ПЗУ микросхемы, сотен включений и отключений электропитания системы жизненный цикл

аппаратуры значительно сокращается, кроме того, возникают новые ошибки, а терпение разработчика истощается;

- кропотливое ручное тестирование аппаратуры утомительно – наилучший вариант для любой встроенной системы – возможность воспользоваться макетной платой, подключить все периферийные устройства и кабели, записать в ПЗУ разрабатываемое приложение, включить электропитание и наблюдать, как работает система. Любое отклонение от этого сценария приводит к разочарованиям и бесполезным тратам времени.

Поэтому весьма важно избегать упомянутых выше циклов в процессе разработки. Вопрос в том, как наиболее эффективно определить этап разработки кода для 8-битового микроконтроллера или более мощного 32-битового ARM микроконтроллера без каких-либо обращений к аппаратуре до завершающих стадий тестирования.

Планирование проектного решения

В главе 4 рассматривались методы выбора подходящего микроконтроллера для встроенной платформы. При проектировании специализированного ПО для микроконтроллера чрезвычайно важно учитывать не только требования к ресурсам для специализированного кода, но и простоту отладки.

Важным преимуществом применения языка C++ являются предлагаемые им абстракции, в том числе возможность разделения кода на логические классы, пространства имен и прочие абстракции, позволяющие с легкостью повторно использовать, тестировать и отлаживать код. Это важнейший аспект любого проектного решения, который непременно должен быть полностью реализован перед реальным воплощением данного проектного решения.

В зависимости от принятого проектного решения отладка может быть чрезвычайно простой, или неимоверно сложной, или занимать некоторое промежуточное положение. Если существует явное и очевидное разделение между всеми типами функциональности без утечек через API или аналогичных проблем, из-за которых возникают утечки внутренних закрытых данных, то создание различных версий основных классов для таких задач, как интеграция и модульное тестирование, будет весьма простым.

Само по себе использование классов и подобных объектов не гарантирует модульность проектного решения. Даже при таком подходе можно в конечном итоге прийти к передаче внутренних данных класса между различными классами, нарушая тем самым модульность. Если это происходит, общее проектное решение усложняется, так как уровень зависимостей возрастает с изменением структур и форматов данных. Это влечет за собой риск возникновения потенциальных проблем в любой точке приложения и потребует неформальных приемов и трюков при написании тестов и изменении реализации API как части более крупных комплексных тестов.

В главе 4 рассматривался выбор правильного микроконтроллера. Характеристики ОЗУ, ПЗУ и блока операций с плавающей точкой напрямую влияют на проектное решение. Как было отмечено в главе 2, важно понимать, как работает код, скомпилированный из написанного нами исходного кода. Это понимание позволяет

интуитивно чувствовать приблизительную стоимость ресурса для каждой строки исходного кода даже без пошагового прохода по сгенерированному машинному коду и благодаря этому чувству создавать точный временной счетчик цикла.

Здесь должно быть понятно, что перед выбором конкретного микроконтроллера необходимо иметь четко оформленную идею общего проектного решения и требования к ресурсам, поэтому чрезвычайно важно начать с создания правильно обоснованного проектного решения.

Системы сборки, независимые от платформы

В идеальном случае выбранное проектное решение и систему сборки можно использовать для создания целевого приложения на любой настольной платформе. Обычно главным условием становится доступность комплекта инструментальных средств и средств программирования для каждой платформы разработки. К счастью, для платформ микроконтроллеров на основе AVR и ARM доступен комплект инструментов на основе GCC, поэтому нет необходимости применять различные комплекты инструментов с разными соглашениями об именованиях, флагами и параметрами настройки.

Остается сложность непосредственного использования комплекта инструментов, следовательно, обеспечения удобства и эффективности работы программиста таким способом, который не требует знания операционной системы, на основе которой сформирован комплект инструментов.

В главе 6 рассматривалась многоцелевая система сборки, способная генерировать бинарные файлы для различных целевых платформ с минимальными трудозатратами для включения любой новой цели. Для целевого микроконтроллера потребуются только две новые цели:

- физический целевой микроконтроллер;
- локальная целевая ОС.

Здесь первая цель определена явно и недвусмысленно, поскольку мы выбрали микроконтроллер, соответствующий нашей целевой задаче. Если исключить непредвиденные ситуации, то мы будем использовать эту первую цель в течение всего процесса разработки. Кроме того, потребуется выполнение локального тестирования в среде разработки настольного компьютера. Это вторая цель.

Неплохо было бы иметь версию одного и того же или аналогичного комплекта инструментов для разработки на языке C++ в каждой настольной ОС, включенной в процесс разработки. К счастью, набор компиляторов GCC доступен практически на любой платформе. Кроме того, существует внешний интерфейс Clang C++ комплекта инструментов LLVM, использующий обычные флаги компиляторов GCC, что обеспечивает почти полную совместимость.

Достаточно сложную систему сборки, рассмотренную в главе 6, можно упростить, пользуясь только набором компиляторов GCC, что позволит работать с этим комплектом инструментов в операционных системах на основе Linux и BSD, а также в Windows (MinGW с MSYS2 или равноценной командной оболочкой) и в macOS (после установки комплекта GCC).

Для обеспечения полной совместимости при работе в macOS рекомендуется использование набора компиляторов GCC из-за небольших проблем в реализации

Clang. Одной из таких текущих проблем является, например, некорректная работа макроатрибута `__forceinline`, который может испортить большой блок кода, предполагающий использование компилятора GCC.

Использование кросс-компиляторов

Каждый инструментальный комплект компиляторов состоит из внешнего компонента, который работает с исходным кодом, и внутреннего компонента, который генерирует файлы в бинарном формате для целевой платформы. При этом нет причин, по которым внутренний компонент не смог бы работать на любой другой платформе, кроме целевой. В конечном итоге этот компонент всего лишь выполняет преобразование текстовых файлов в некоторую последовательность байтов.

Таким образом, кросс-компиляция является чрезвычайно важным функциональным свойством разработки, ориентированной на микроконтроллеры, поскольку компиляция непосредственно на самих микроконтроллерах весьма неэффективна в большинстве случаев. Но в процессе кросс-компиляции нет ничего загадочного и таинственного. При использовании комплекта инструментов на основе GCC или совместимого с GCC разработчик продолжает взаимодействие с теми же интерфейсами комплекта, просто отдельные инструменты обычно имеют префиксы в виде имен целевых платформ, чтобы отличать их от инструментов для других платформ. Например, вместо `g++` используется `arm-none-eabi-g++`.

Бинарные файлы генерируются в формате, соответствующем конкретной целевой платформе.

Локальная отладка и отладка на микросхеме

В главе 6 рассматривалась отладка приложения с использованием Valgrind и аналогичных инструментов, а также с применением отладчика GDB и инструментов этого семейства. При выполнении комплексных тестов на основе ОС для проектов с применением микроконтроллеров, таких как рассматриваемый в следующем разделе «Пример: комплексный тест ESP8266», можно использовать точно такие же методы профилирования и отладки кода, не беспокоясь о том, что этот же код будет работать на гораздо более медленной и более ограниченной платформе во время завершающего комплексного тестирования на реальной аппаратуре.

На завершающем этапе комплексного тестирования действительно возникает сложность: специализированное ПО, которое ранее тестировалось на быстрой настольной системе с применением Valgrind и прочих весьма мощных инструментов, теперь работает на более слабом микроконтроллере ATmega 16 МГц без возможности быстрого запуска кода с помощью инструмента Valgrind или в сеансе отладчика GDB.

На этом этапе неизбежно обнаружение ошибок и других проблем, поэтому необходимо подготовиться к действиям при возникновении подобных ситуаций. Часто применяют метод отладки на микросхеме (on-chip debugging – OCD), который можно выполнять с использованием любого отладочного интерфейса, предоставляемого микроконтроллером. Это может быть JTAG, DebugWire или

SWD, PDI или любой другой тип отладочного интерфейса. В главе 4 рассматривались некоторые из этих интерфейсов в контексте программирования микроконтроллеров.

Встроенные интегрированные среды разработки (IDE) обеспечивают возможность выполнения отладки на микросхеме прямо «из коробки», устанавливая соединение с целевой аппаратурой и обеспечивая создание контрольных точек останова, практически так же, как это делается в локальном процессе. Разумеется, еще возможно использование отладчика GDB из командной строки для выполнения тех же операций, а также применение таких программ, как OpenOCD (**http://openocd.org/**), предоставляющих интерфейс `gdbserver` для отладчика GDB с обеспечением взаимодействия с широким спектром разнообразных отладочных интерфейсов.

Пример: комплексный тест ESP8266

В этом примере проекта рассматривается реализация интерфейсов API для платформы Arduino из рабочей среды (фреймворка) Sming, который был подробно разобран в примере из главы 5. Цель этого проекта – создание реализации собственной рабочей среды для настольных операционных систем, позволяющей компилировать специализированное ПО в выполняемом формате и запускать его в локальной среде.

Кроме того, необходимо создать имитацию сенсоров и исполнительных устройств (актуаторов), с которыми специализированное ПО может устанавливать соединение для считывания данных среды и передачи данных в исполнительные устройства, как часть проекта по мониторингу, и управления микроклиматом в здании, который был кратко упомянут в главе 5 и будет подробно рассматриваться в главе 9 «Пример: мониторинг и управление внутренним микроклиматом в здании». Для этого также необходима централизованная служба, отслеживающая информацию подобного рода. При таком подходе можно организовать одновременную работу нескольких процессов специализированного ПО для имитации помещений со множеством устройств.

Причиной столь широкого масштаба имитации является отсутствие реальной физической аппаратуры. Без физической системы микроконтроллеров нет физических сенсоров, и эти сенсоры отсутствуют в реальном помещении. Следовательно, необходимо генерировать имитационные входные данные для сенсоров, а также имитировать эффект от действий любых исполнительных устройств. Такой подход обладает рядом собственных преимуществ.

Удобство и польза наличия описанной выше регулируемой возможности заключаются в том, что она позволяет проверять специализированное ПО не только как отдельную независимую систему, но также как часть более крупной системы, в которую она встраивается. В проекте мониторинга и управления микроклиматом здания это должно означать наличие отдельного компонента (узла), установленного в помещении здания, вместе с десятками и даже сотнями других компонентов (узлов), установленных в том же и в других помещениях на различных этажах здания, в совокупности с соответствующими вспомогательными скрытыми службами, работающими в одной и той же сетевой среде.

При наличии возможности такой крупномасштабной имитации можно тестировать не только правильность работы самого специализированного ПО, но также всей системы в целом. При этом можно применять различные типы специализированного ПО или различные его версии в сочетании с разнообразными сенсорами и исполнительными устройствами (для устройств кондиционирования воздуха, вентиляторов, кофе-машин, регуляторов, переключателей и т. д.). Кроме того, внутренние скрытые службы должны управлять компонентами (узлами) в соответствии с данными, передаваемыми самими этими компонентами.

В такой имитируемой среде здания можно будет сконфигурировать особые помещения с конкретными условиями внутренней среды, существующей в течение рабочего дня, когда люди входят, работают и покидают помещение. Это позволит определить воздействие различных уровней заполненности здания, внешних условий и т. д. Это также можно сделать с помощью специализированного ПО и внутренних служб, которые должны использоваться для конечной эксплуатационной системы. Хотя такой способ тестирования системы не устранит полностью все потенциальные проблемы, он должен, по меньшей мере, обеспечить проверку правильности функционирования программного компонента системы.

Поскольку встраиваемые системы по определению являются частью более крупной системы (основанной на аппаратуре), в полный комплексный тест будет включена реальная аппаратура или равнозначная ей. Таким образом, следует рассматривать приведенный здесь пример как комплексный тест ПО, выполняемый до развертывания специализированного ПО на целевой аппаратуре в физическом здании.

Процессы сервера имитации и конкретной версии специализированного ПО обладают собственной основной функцией и работают независимо друг от друга. Это позволяет наблюдать за функциональностью специализированного ПО с минимальными возможными помехами и обеспечивает создание правильного проектного решения. Для эффективного обмена информацией между этими двумя процессами используется библиотека удаленных вызовов процедур (remote procedure call – RPC), с помощью которой, собственно, и устанавливаются соединения между специализированным ПО и устройствами на основе I2C, SPI и UART в имитируемом помещении. В рассматриваемом примере используется RPC-библиотека NymphRPC, разработанная автором книги. Исходный код текущей версии библиотеки включен в исходный код рассматриваемого примера. Текущую версию библиотеки NymphRPC можно найти в репозитории Git **https://github.com/MayaPosch/NymphRPC**.

Сервер

Сначала рассмотрим реализацию сервера для комплексного теста. Его основной функцией является поддержка работы сервера удаленных вызовов процедур (RPC) и обработка состояния каждого сенсорного и исполнительного устройства, а также состояния помещений.

Основной файл *simulation.cpp* содержит код настройки конфигурации службы удаленных вызовов процедур и главный управляющий цикл, как показано в следующем фрагменте кода:

```
#include "config.h"
#include "building.h"
#include "nodes.h"
#include <nymph/nymph.h>
#include <thread>
#include <condition_variable>
#include <mutex>

std::condition_variable gCon;
std::mutex gMutex;
bool gPredicate = false;

void signal_handler(int signal) {
    gPredicate = true;
    gCon.notify_one();
}

void logFunction(int level, string logStr) {
    std::cout << level << " - " << logStr << endl;
}
```

Включаемые файлы в начале кода показывают основную структуру и зависимости. С их помощью здесь объявляется специализированный класс конфигурации, класс, определяющий здание, статический класс для узлов (компонентов). Кроме того, включены заголовочные файлы для поддержки многопоточности (доступны начиная со стандарта C++11) и для библиотеки MymphRPC.

Определена функция обработки сигналов для использования в дальнейшем в состоянии ожидания, чтобы обеспечить завершение работы сервера по простому управляющему сигналу. Также определена функция журналирования для использования сервером NymphRPC.

Далее определяется функция обратного вызова для сервера RPC, как показано ниже:

```
NymphMessage* getNewMac(int session, NymphMessage* msg, void* data) {
    NymphMessage* returnMsg = msg->getReplyMessage();
    std::string mac = Nodes::getMAC();
    Nodes::registerSession(mac, session);
    returnMsg->setResultValue(new NymphString(mac));
    return returnMsg;
}
```

Это функция инициализации, которую клиент вызывает на сервере. Она проверяет глобальный статический класс `Nodes` на наличие доступного MAC-адреса. Этот адрес недвусмысленно идентифицирует новый экземпляр узла тем же способом, которым устройство в сети должно идентифицироваться по своему уникальному MAC-адресу Ethernet. Это внутренняя функция, не требующая каких-либо изменений в специализированном ПО, так как передает возможность назначения MAC-адресов серверу вместо жесткого кодирования их в каком-либо другом месте. После присваивания нового MAC-адреса устанавливается его связь с идентификатором сеанса NymphRPC, так что в дальнейшем можно использовать MAC-адрес для поиска соответствующего идентификатора сеанса, а внутри этого сеанса – клиента для вызова событий, генерируемых имитируемыми устройствами.

Здесь же можно видеть основную сигнатуру функции обратного вызова NymphRPC, используемую в экземпляре сервера. Результат ее работы – возвращаемое сообщение, а в качестве параметров она принимает идентификатор сеанса, связанный с клиентом, с которым установлено соединение, сообщение, принятое от этого клиента, и некоторые данные, определенные пользователем.

```
NymphMessage* writeUart(int session, NymphMessage* msg, void* data) {
    NymphMessage* returnMsg = msg->getReplyMessage();

    std::string mac = ((NymphString*) msg->parameters()[0])->getValue();
    std::string bytes = ((NymphString*) msg->parameters()[1])->getValue();
    returnMsg->setResultValue(new NymphBoolean(Nodes::writeUart(mac, bytes)));
    return returnMsg;
}
```

Эта функция обратного вызова содержит реализацию способа записи в интерфейс UART узла в имитируемой системе, соответствующего любому подключенному имитируемому устройству.

Для поиска узла используется MAC-адрес, передаваемый вместе с набором байтов, который должен быть записан в соответствующей функции класса `Nodes`.

```
NymphMessage* writeSPI(int session, NymphMessage* msg, void* data) {
    NymphMessage* returnMsg = msg->getReplyMessage();
    std::string mac = ((NymphString*) msg->parameters()[0])->getValue();
    std::string bytes = ((NymphString*) msg->parameters()[1])->getValue();
    returnMsg->setResultValue(new NymphBoolean(Nodes::writeSPI(mac, bytes)));
    return returnMsg;
}

NymphMessage* readSPI(int session, NymphMessage* msg, void* data) {
    NymphMessage* returnMsg = msg->getReplyMessage();
    std::string mac = ((NymphString*) msg->parameters()[0])->getValue();
    returnMsg->setResultValue(new NymphString(Nodes::readSPI(mac)));
    return returnMsg;
}
```

Для шины SPI используется аналогичная система записи и чтения. MAC-адрес идентифицирует узел, а строка либо передается в шину, либо принимается из нее. Здесь имеется одно ограничение: предполагается наличие единственного SPI-устройства, так как не предусмотрен способ выбора другой линии CS (chip-select) шины SPI. Отдельный параметр CS должен быть передан, чтобы появилась возможность использования более одного SPI-устройства.

Рассмотрим следующий фрагмент кода:

```
NymphMessage* writeI2C(int session, NymphMessage* msg, void* data) {
    NymphMessage* returnMsg = msg->getReplyMessage();

    std::string mac = ((NymphString*) msg->parameters()[0])->getValue();
    int i2cAddress = ((NymphSint32*) msg->parameters()[1])->getValue();
    std::string bytes = ((NymphString*) msg->parameters()[2])->getValue();
    returnMsg->setResultValue(new NymphBoolean(Nodes::writeI2C(mac, i2cAddress, bytes)));
    return returnMsg;
}
```

```
NymphMessage* readI2C(int session, NymphMessage* msg, void* data) {
    NymphMessage* returnMsg = msg->getReplyMessage();

    std::string mac = ((NymphString*) msg->parameters()[0])->getValue();
    int i2cAddress = ((NymphSint32*) msg->parameters()[1])->getValue();
    int length = ((NymphSint32*) msg->parameters()[2])->getValue();
    returnMsg->setResultValue(new NymphString(Nodes::readI2C(mac, i2cAddress, length)));
    return returnMsg;
}
```

В версию функции обратного вызова для шины I2C передается адрес подчиненного I2C-устройства, позволяющий использовать более одного такого устройства.

Основная функция `main` регистрирует методы обратного вызова процедур, начинает процесс имитации, затем входит в режим условного ожидания.

```
int main() {
    Config config;
    config.load("config.cfg");
```

Сначала загружаются данные конфигурации имитационной системы, которые будут использоваться в следующем коде. Конфигурация определена в отдельном файле, загружаемом с помощью специализированного класса `Config`, который будет рассматриваться более подробно при изучении синтаксического анализатора (парсера) конфигурации.

```
    vector<NymphTypes> parameters;
    NymphMethod getNewMacFunction("getNewMac", parameters, NYMPH_STRING);
    getNewMacFunction.setCallback(getNewMac);
    NymphRemoteClient::registerMethod("getNewMac", getNewMacFunction);

    parameters.push_back(NYMPH_STRING);
    NymphMethod serialRxCallback("serialRxCallback", parameters, NYMPH_NULL);
    serialRxCallback.enableCallback();
    NymphRemoteClient::registerCallback("serialRxCallback", serialRxCallback);

    // string readI2C(string MAC, int i2cAddress, int length)
    parameters.push_back(NYMPH_SINT32);
    parameters.push_back(NYMPH_SINT32);
    NymphMethod readI2CFunction("readI2C", parameters, NYMPH_STRING);
    readI2CFunction.setCallback(readI2C);
    NymphRemoteClient::registerMethod("readI2C", readI2CFunction);

    // bool writeUart(string MAC, string bytes)
    parameters.clear();
    parameters.push_back(NYMPH_STRING);
    parameters.push_back(NYMPH_STRING);
    NymphMethod writeUartFunction("writeUart", parameters, NYMPH_BOOL);
    writeUartFunction.setCallback(writeUart);
    NymphRemoteClient::registerMethod("writeUart", writeUartFunction);

    // bool writeSPI(string MAC, string bytes)
    NymphMethod writeSPIFunction("writeSPI", parameters, NYMPH_BOOL);
    writeSPIFunction.setCallback(writeSPI);
    NymphRemoteClient::registerMethod("writeSPI", writeSPIFunction);

    // bool writeI2C(string MAC, int i2cAddress, string bytes)
```

```
    parameters.clear();
    parameters.push_back(NYMPH_STRING);
    parameters.push_back(NYMPH_SINT32);
    parameters.push_back(NYMPH_SINT32);
    NymphMethod writeI2CFunction("writeI2C", parameters, NYMPH_BOOL);
    writeI2CFunction.setCallback(writeI2C);
    NymphRemoteClient::registerMethod("writeI2C", writeI2CFunction);
```

В этом фрагменте кода выполняется регистрация методов, которые в дальнейшем планируется предоставить процессам клиентского узла. Это позволит им вызывать функции, определенные выше в этом же файле исходного кода. Для регистрации функции стороны сервера в NymphRPC необходимо определить типы параметров (в заданном порядке) и использовать их для определения нового экземпляра `NymphMethod`, который предоставляется вместе с этим списком типов параметров, именем функции и типом возвращаемого значения.

Затем эти экземпляры методов регистрируются в объекте `NymphRemoteClient`, являющемся классом самого верхнего уровня на стороне сервера NymphRPC.

```
    signal(SIGINT, signal_handler);

    NymphRemoteClient::start(4004);

    Building building(config);

    std::unique_lock<std::mutex> lock(gMutex);
    while (!gPredicate) {
          gCon.wait(lock);
    }

    NymphRemoteClient::shutdown();

    Thread::sleep(2000);

    return 0;
}
```

В завершающей части кода функции `main` устанавливается обработчик сигнала `SIGINT` (**Ctrl**+**C**). Сервер NymphRPC запускается на порте 4004 для всех интерфейсов. Далее создается экземпляр `Building` с передачей в него экземпляра конфигурации, загруженной ранее с помощью класса парсера конфигурации.

После этого начинается выполнение цикла, который проверяет, изменилось ли значение глобальной переменной `gPredicate` на `true` в том случае, если сработал обработчик сигнала, следовательно, для этой логической переменной было установлено значение `true`. Условная переменная используется, чтобы разрешить блокировку выполнения основного потока как можно быстрее после того, как обработчик сигнала сообщит об изменении значения этой переменной.

При наличии условной переменной, определяющей условие ожидания внутри цикла, гарантируется, что даже если условие ожидания нарушается ложным сигналом пробуждения, происходит возврат в состояние ожидания до следующего оповещения.

Наконец, если принят запрос на завершение работы сервера, то сервер NymphRPC останавливается и выдерживается дополнительная пауза в две секунды, чтобы все активные потоки могли корректно завершить свою работу. После этого завершает работу и основной сервер.

Теперь рассмотрим содержимое файла конфигурации *config.cfg*, загружаемого для этой системы имитации.

```
[Building]
floors=2

[Floor_1]
rooms=1,2

[Floor_2]
rooms=2,3

[Room_1]
; Определение конфигурации помещения.
; Для сенсоров и исполнительных устройств используется формат:
; <device_id>:<node_id>
nodes=1
devices=1:1

[Room_2]
nodes=2

[Room_3]
nodes=3

[Room_4]
nodes=4

[Node_1]
mac=600912760001
sensors=1

[Node_2]
mac=600912760002
sensors=1

[Node_3]
mac=600912760003
sensors=1

[Node_4]
mac=600912760004
sensors=1

[Device_1]
type=i2c
address=0x20
device=bme280

[Device_2]
type=spi
cs_gpio=1

[Device_3]
type=uart
uart=0
baud=9600
device=mh-z19
```

```
[Device_4]
type=uart
uart=0
baud=9600
device=jura
```

Здесь можно видеть, что в этом файле конфигурации используется стандартный формат INI-файлов. Конфигурация определяет здание с двумя этажами, на каждом этаже по две комнаты. В каждой комнате размещен один узел, на каждом узле имеется один сенсорный датчик BME280, подключенный к шине I2C.

Определяются и другие устройства, но в рассматриваемом примере они не используются.

Приведенный выше формат конфигурации обрабатывается синтаксическим анализатором (парсером) конфигурации, объявленным в заголовочном файле *config.h*. Рассмотрим исходный код его реализации.

```
#include <string>
#include <memory>
#include <sstream>
#include <iostream>
#include <type_traits>

#include <Poco/Util/IniFileConfiguration.h>
#include <Poco/AutoPtr.h>

using Poco::AutoPtr;
using namespace Poco::Util;

class Config {
   AutoPtr<IniFileConfiguration> parser;
public:
   Config();
   bool load(std::string filename);
   template<typename T>
   auto getValue(std::string key, T defaultValue) -> T {
         std::string value;
         try {
               value = parser->getRawString(key);
         }
         catch (Poco::NotFoundException &e) {
               return defaultValue;
         }
         // Преобразование значения в соответствующий выходной тип, если это возможно.
         std::stringstream ss;
         if (value[0] == '0' && value[1] == 'x') {
               value.erase(0, 2);
               ss << std::hex << value; // Считывается как шестнадцатеричное значение.
         }
         else {
               ss.str(value);
         }
         T retVal;
         if constexpr (std::is_same<T, std::string>::value) { retVal = ss.str(); }
```

```
        else { ss >> retVal; }
        return retVal;
    }
};
```

Здесь можно наблюдать интересное использование шаблонов, а также одно из их ограничений. Тип, передаваемый в шаблон, используется и для параметра по умолчанию, и для возвращаемого значения, позволяя шаблону выполнять приведение типа (cast) исходной «сырой» строки, полученной из файла конфигурации, в требуемый тип, но при этом также избегая возникновения проблемы неполного шаблона, поскольку тот же тип используется для возвращаемого этой функцией значения.

Из-за ограничения языка C++, согласно которому каждая функция с одинаковым именем непременно должна иметь отличающийся набор параметров, даже при различных возвращаемых значениях, мы обязательно должны использовать здесь значение по умолчанию для параметра, чтобы обойти эту проблему. Так как в большинстве случаев необходимо предоставлять значение по умолчанию для ключей, которые мы пытаемся считывать, здесь это не приводит к каким-либо затруднениям.

Далее выполняется сравнение типа с помощью метода `std::is_same`, чтобы убедиться в том, что если целевым возвращаемым типом является строка (string), то эта строка копируется непосредственно из потока `stringstream` вместо попытки ее преобразования с использованием форматированного вывода. Поскольку значения из INI-файла считываются с помощью механизма чтения POCO INI как необработанные («сырые») строки, здесь нет необходимости в каком-либо их преобразовании.

Код реализации синтаксического анализатора (парсера) в файле *config.cpp* имеет небольшой размер благодаря шаблонам, определенным в заголовочном файле.

```
#include "config.h"

Config::Config() {
    parser = new IniFileConfiguration();
}

bool Config::load(std::string filename) {
    try {
        parser->load(filename);
    }
    catch (...) {
        // Возникло исключение. Возвращается значение false.
        return false;
    }
    return true;
}
```

Здесь показана реализация метода, который действительно загружает конфигурацию из файла, имя которого передано в строке `filename`. В этой реализации создается экземпляр класса POCO `IniFileConfiguration` с предположением о том, что мы пытаемся выполнить синтаксический разбор INI-файла. Если по какой-либо причине загрузка файла конфигурации завершается аварийно, то возвращается ошибка (значение false).

В более усовершенствованной версии парсера можно было бы обеспечить поддержку различных типов конфигурационных файлов или даже исходных кодов с расширенной обработкой ошибок. Но для рассматриваемого примера более чем достаточен скромный формат INI.

Далее рассмотрим код класса `Building`.

```
#include <vector>
#include <string>

#include "floor.h"

class Building {
    std::vector<Floor> floors;
 public:
    Building(Config &cfg);
};
```

Поскольку в сервер имитации не были добавлены какие-либо расширенные дополнительные функциональные возможности, код объявления этого класса, как и код его реализации, приведенный ниже, не представляет особого интереса.

```
#include "building.h"
#include "floor.h"

Building::Building(Config &config) {
    int floor_count = config.getValue<int>("Building.floors", 0);
    for (int i = 0; i < floor_count; ++i) {
          Floor floor(i + 1, config); // Нумерация этажей начинается с 1.
          floors.push_back(floor);
    }
}
```

Здесь определения этажей считываются из файла, и для каждого этажа создается экземпляр класса `Floor`, добавляемый в массив. Эти экземпляры также получают ссылку на объект конфигурации.

Определение класса `Floor` не менее просто по той же причине – отсутствие расширенных функциональных возможностей.

```
#include <vector>
#include <cstdint>

#include "room.h"

class Floor {
    std::vector<Room> rooms;
 public:
    Floor(uint32_t level, Config &config);
};
```

Код реализации этого класса приведен ниже:

```
#include "floor.h"
#include "utility.h"

#include <string>

Floor::Floor(uint32_t level, Config &config) {
```

```
    std::string floor_cat = "Floor_" + std::to_string(level);
    std::string roomsStr = config.getValue<std::string>(floor_cat + ".rooms", 0);

    std::vector<std::string> room_ids;
    split_string(roomsStr, ',', room_ids);
    int room_count = room_ids.size();

    if (room_count > 0) {
          for (int i = 0; i < room_count; ++i) {
                Room room(std::stoi(room_ids.at(i)), config);
                rooms.push_back(room);
          }
    }
}
```

Следует отметить способ синтаксического разбора файла конфигурации по частям: по одному фрагменту для каждого отдельного класса, при этом каждый экземпляр класса обрабатывает только небольшую секцию конфигурации, которая определяется по заданному идентификатору.

Рассматриваются лишь помещения, определенные для этажа с заданным идентификатором. Извлекаются идентификаторы этих помещений, затем создаются соответствующие новые экземпляры классов. Экземпляр для каждого помещения копируется в вектор. В более продвинутой реализации сервера имитации здесь можно было бы, например, реализовать события для всего этажа в целом.

Во вспомогательном заголовочном файле *utility.h* определяется простой метод разделения строк, как показано в следующем фрагменте кода:

```
#include <string>
#include <vector>

void split_string(const std::string& str, char chr, std::vector<std::string>& vec);
```

Ниже приведена реализация этого метода:

```
#include "utility.h"

#include <algorithm>

void split_string(const std::string& str, char chr, std::vector<std::string>& vec) {
     std::string::const_iterator first = str.cbegin();
     std::string::const_iterator second = std::find(first + 1, str.cend(), chr);

     while (second != str.cend()) {
         vec.emplace_back(first, second);
         first = second;
         second = std::find(second + 1, str.cend(), chr);
     }

     vec.emplace_back(first, str.cend());
}
```

Эта простая функция принимает разделяющий символ и строку, которую разделяет на части с учетом заданного разделяющего символа. Затем разделенные части строки копируются в вектор в соответствии с их местоположением в исходной строке.

Теперь рассмотрим класс `Room`, объявленный в заголовочном файле *room.h*:

```
#include "node.h"
#include "devices/device.h"

#include <vector>
#include <map>
#include <cstdint>

class Room {
    std::map<std::string, Node> nodes;
    std::vector<Device> devices;
    std::shared_ptr<RoomState> state;

 public:
    Room(uint32_t type, Config &config);
};
```

Код реализации класса `Room`:

```
#include "room.h"

#include "utility.h"

Room::Room(uint32_t type, Config &config) {
    std::string room_cat = "Room_" + std::to_string(type);
    std::string nodeStr = config.getValue<std::string>(room_cat + ".nodes", "");

    state->setTemperature(24.3);
    state->setHumidity(51.2);
    std::string sensors;
    std::string actuators;
    std::string node_cat;
    if (!nodeStr.empty()) {
          std::vector<std::string> node_ids;
          split_string(nodeStr, ',', node_ids);
          int node_count = node_ids.size();

          for (int i = 0; i < node_count; ++i) {
                Node node(node_ids.at(i), config);
                node_cat = "Node_" + node_ids.at(i);
                nodes.insert(std::map<std::string, Node>::value_type(node_ids.at(i), node));
          }

          std::string devicesStr = config.getValue<std::string>(node_cat + ".devices", "");
          if (!devicesStr.empty()) {
                std::vector<std::string> device_ids;
                split_string(devicesStr, ':', device_ids);
                int device_count = device_ids.size();

                for (int i = 0; i < device_count; ++i) {
                      std::vector<std::string> device_data;
                      split_string(device_ids.at(i), ':', device_data);
                      if (device_data.size() != 2) {
                            // Incorrect data. Abort.
                            continue;
                      }
```

```
                    Device device(device_data[0], config, state);

                    nodes.at(device_data[1]).addDevice(std::move(device));
                    devices.push_back(device);
                }
            }
        }
}
```

Конструктор этого класса начинается с настройки начальных условий конкретного помещения, а именно определяются значения температуры и влажности. Далее считываются характеристики узлов и устройств по идентификатору этого помещения и создаются соответствующие экземпляры. Сначала берется список узлов для данного помещения, затем для каждого узла извлекается список устройств, и эта строка разделяется на отдельные идентификаторы устройств.

Для каждого идентификатора устройства создается экземпляр соответствующего класса, после чего созданный экземпляр добавляется в узел, использующий это устройство. Процесс завершается общей инициализацией сервера имитации.

Рассмотрим подробнее класс `Device`:

```
#include "config.h"
#include "types.h"

class Device {
    std::shared_ptr<RoomState> roomState;
    Connection connType;
    std::string device;
    std::string mac;
    int spi_cs;
    int i2c_address;
    int uart_baud;          // Скорость в бодах UART.
    int uart_dev;           // Периферийное устройство UART (0, 1 и т. д.)
    Config devConf;
    bool deviceState;
    uint8_t i2c_register;

    void send(std::string data);

 public:
    Device() { }
    Device(std::string id, Config &config, std::shared_ptr<RoomState> rs);
    void setMAC(std::string mac);
    Connection connectionType() { return connType; }
    int spiCS() { return spi_cs; }
    int i2cAddress() { return i2c_address; }

    bool write(std::string bytes);
    std::string read();
    std::string read(int length);
};
```

Код реализации этого класса:

```
#include "device.h"
#include "nodes.h"
```

```
Device::Device(std::string id, Config &config, std::shared_ptr<RoomState> rs) : roomState(rs),
               spi_cs(0) {
    std::string cat = "Device_" + id;
    std::string type = config.getValue<std::string>(cat + ".type", "");
    if (type == "spi") {
          connType = CONN_SPI;
          spi_cs = config.getValue<int>(cat + ".cs_gpio", 0);
          device = config.getValue<std::string>(cat + ".device", "");
    }
    else if (type == "i2c") {
          connType == CONN_I2C;
          i2c_address = config.getValue<int>(cat + ".address", 0);
          device = config.getValue<std::string>(cat + ".device", "");
    }
    else if (type == "uart") {
          connType == CONN_UART;
          uart_baud = config.getValue<int>(cat + ".baud", 0);
          uart_dev = config.getValue<int>(cat + ".uart", 0);
          device = config.getValue<std::string>(cat + ".device", "");
    }
    else {
          // Ошибка. Недопустимый тип.
    }
}
```

В конструкторе считывается информация для данного конкретного устройства с использованием переданного идентитификатора. В зависимости от типа устройства определяются специализированные ключи. Все ключи сохраняются в членах-переменных.

```
void Device::setMAC(std::string mac) {
    this->mac = mac;
}

// Метод вызывается, когда устройство (на основе UART) должно передавать данные.
void Device::send(std::string data) {
    Nodes::sendUart(mac, data);
}
```

После простого метода настройки (setter) MAC-адреса подключенного узла определяется метод, позволяющий генерировать события UART для активизации функции обратного вызова в процессе узла через метод RPC (как мы увидим в дальнейшем при рассмотрении реализации класса Nodes).

```
bool Device::write(std::string bytes) {
    if (!deviceState) { return false; }
    // Первый байт - содержимое регистра чтения/записи на шине I2C. Сохраняется как ссылка.
    if (connType == CONN_I2C && bytes.length() > 0) {
          i2c_register = bytes[0];
    }
    else if (connType == CONN_SPI) {
          // .
    }
```

```
    else if (connType == CONN_UART) {
          //
    }
    else { return false; }

    return true;
}
```

Выше приведено определение обобщенного метода для записи в устройство независимо от его типа. Здесь показана обработка только интерфейса I2C для получения регистра устройства, в которое должна выполняться запись.

```
std::string Device::read(int length) {
    if (!deviceState) { return std::string(); }

    switch (connType) {
          case CONN_SPI:
                return std::string();
                break;
          case CONN_I2C:
          {
                // Получение определенных значений из экземпляра состояния помещения.
                // Здесь жестко закодирован сенсор BME280.
                // Возвращаемое значение зависит от значения, установленного в регистре.
                uint8_t zero = 0x0;
                switch (i2c_register) {
                      case 0xFA: // Температура. MSB, LSB, XLSB.
                      {
                            std::string ret = std::to_string(roomState->getTemperature()); // MSB
                            ret.append(std::to_string(zero)); // LSB
                            ret.append(std::to_string(zero)); // XLSB
                            return ret;
                            break;
                      }
                      case 0xF7: // Давление. MSB, LSB, XLSB.
                      {
                            std::string ret = std::to_string(roomState->getPressure()); // MSB
                            ret.append(std::to_string(zero)); // LSB
                            ret.append(std::to_string(zero)); // XLSB
                            return ret;
                            break;
                      }
                      case 0xFD: // Влажность. MSB, LSB.
                      {
                            std::string ret = std::to_string(roomState->getHumidity()); // MSB
                            ret.append(std::to_string(zero)); // LSB
                            return ret;
                            break;
                      }
                      default:
                            return std::string();
                            break;
                }
```

```
                break;
            }
            case CONN_UART:
                //
                break;
            default:
                // Ошибка.
                return std::string();
        };

        return std::string();
}

std::string Device::read() {
    return read(0);
}
```

В группу методов `read` включена версия, определяющая параметр размера `length` для набора считываемых байтов, и версия без параметров, просто передающая ноль в первую версию метода. Параметр размера должен быть полезен для интерфейса UART, где для данных используется фиксированный размер буфера.

Для упрощения примера жестко закодирован ответ сенсора BME280, объединяющий измерительные устройства температуры, влажности и атмосферного давления. Проверяется значение регистра, которое было передано с помощью описанной выше команды `write`, затем возвращается соответствующее ему значение как результат считывания характеристик текущего помещения.

Можно подключить гораздо большее количество устройств, но для этого необходима соответствующая реализация в отдельных файлах конфигурации и создание специализированных классов вместо жесткого кодирования, как это сделано в рассматриваемом примере.

Специализированные типы для рассматриваемого приложения определены в заголовочном файле *types.h*, как показано в следующем фрагменте кода:

```
#include <memory>
#include <thread>
#include <mutex>

enum Connection {
    CONN_NC = 0,
    CONN_SPI = 1,
    CONN_I2C = 2,
    CONN_UART = 3
};

class RoomState {
    float temperature;      // Температура в помещении
    float humidity;         // Относительная влажность воздуха (0.00-100.00 %)
    uint16_t pressure;      // Атмосферное давление.
    std::mutex tmtx;
    std::mutex hmtx;
    std::mutex pmtx;

 public:
    RoomState() :
```

```
        temperature(0),
        humidity(0),
        pressure(1000) {
        //
    }

    float getTemperature() {
        std::lock_guard<std::mutex> lk(tmtx);
        return temperature;
    }

    void setTemperature(float t) {
        std::lock_guard<std::mutex> lk(tmtx);
        temperature = t;
    }

    float getHumidity() {
        std::lock_guard<std::mutex> lk(hmtx);
        return humidity;
    }

    void setHumidity(float h) {
        std::lock_guard<std::mutex> lk(hmtx);
        temperature = h;
    }

    float getPressure() {
        std::lock_guard<std::mutex> lk(pmtx);
        return pressure;
    }

    void setPressure(uint16_t p) {
        std::lock_guard<std::mutex> lk(pmtx);
        pressure = p;
    }
};
```

Здесь определено перечисление для различных типов соединения. Также приведено определение класса RoomState с реализацией простых методов get/set, в которых доступ к соответствующим значениям регулируется с помощью механизма мьютекса (взаимного исключения), обеспечивающего безопасную работу нескольких потоков, так как несколько узлов могут пытаться одновременно получить доступ к одним и тем же значениям, в то время как устройство в помещении пытается обновить эти же значения.

Теперь рассмотрим подробнее класс `Node`:

```
#include "config.h"
#include "devices/device.h"

#include <string>
#include <vector>
#include <map>

class Node {
    std::string mac;
    bool uart0_active;
```

```
    Device uart0;
    std::map<int, Device> i2c;
    std::map<int, Device> spi;
    std::vector<Device> devices;

 public:
    Node(std::string id, Config &config);
    bool addDevice(Device &&device);
    bool writeUart(std::string bytes);
    bool writeSPI(std::string bytes);
    std::string readSPI();
    bool writeI2C(int i2cAddress, std::string bytes);
    std::string readI2C(int i2cAddress, int length);
};
```

Код реализации этого класса:

```
#include "node.h"
#include "nodes.h"

#include <cstdlib>
#include <utility>

Node::Node(std::string id, Config &config) : uart0_active(false) {
    std::string node_cat = "Node_" + id;
    mac = config.getValue<std::string>(node_cat + ".mac", "");

    Nodes::addNode(mac, this);
    std::system("esp8266");
};
```

При создании новый экземпляр этого класса получает собственный MAC-адрес, добавляет его во внутреннюю локальную переменную и регистрирует его как класс `Node`. Новый экземпляр выполняемого узла (в рассматриваемом примере `esp8266`) запускается с помощью стандартного системного вызова, то есть операционная система инициализирует соответствующий новый процесс.

После начала работы нового процесса он устанавливает соединение с RPC-сервером и получает MAC-адрес с использованием функций обратного вызова, которые рассматривались ранее в этом разделе. После этого экземпляр класса `Node` и удаленный процесс работают как зеркальные образы друг друга.

```
bool Node::addDevice(Device &&device) {
    device.setMAC(mac);

    switch (device.connectionType()) {
          case CONN_SPI:
                spi.insert(std::pair<int, Device>(device.spiCS(), std::move(device)));
                break;
          case CONN_I2C:
                i2c.insert(std::pair<int, Device>(device.i2cAddress(), std::move(device)));
                break;
          case CONN_UART:
                uart0 = std::move(device);
                uart0_active = true;
                break;
```

```
            default:
                  // Ошибка.
                  break;
      }

      return true;
}
```

Когда класс `Room` размещает новое устройство в конкретном узле, этому устройству присваивается MAC-адрес как идентификатор принадлежности к данному узлу. После этого запрашивается тип интерфейса нового устройства, чтобы можно было добавить соответствующий интерфейс с учетом линии CS (выбор микросхемы) для SPI и адреса шины I2C.

Используя гибкую семантику, мы гарантируем, что не бездумно копируем тот же экземпляр класса устройства, а по существу передаем право владения исходному экземпляру, следовательно, повышаем эффективность.

```
bool Node::writeUart(std::string bytes) {
      if (!uart0_active) { return false; }

      uart0.write(bytes);

      return true;
}

bool Node::writeSPI(std::string bytes) {
      if (spi.size() == 1) {
            spi[0].write(bytes);
      }
      else {
            return false;
      }

      return true;
}

std::string Node::readSPI() {
      if (spi.size() == 1) {
            return spi[0].read();
      }
      else {
            return std::string();
      }
}

bool Node::writeI2C(int i2cAddress, std::string bytes) {
      if (i2c.find(i2cAddress) == i2c.end()) { return false; }

      i2c[i2cAddress].write(bytes);
      return true;
}

std::string Node::readI2C(int i2cAddress, int length) {
      if (i2c.count(i2cAddress) || length < 1) { return std::string(); }

      return i2c[i2cAddress].read(length);
}
```

Для обеспечения функций чтения и записи не требуется каких-то сверхусилий. Используя линию выбора микросхемы CS (SPI), адрес шины (I2C) или что-то другое (UART), мы узнаем, к какому типу устройства происходит доступ, и вызываем соответствующие методы.

Приведенный ниже класс `Nodes` объединяет все компоненты, описанные выше:

```
#include <map>
#include <string>
#include <queue>

class Node;

class Nodes {
    static Node* getNode(std::string mac);
    static std::map<std::string, Node*> nodes;
    static std::queue<std::string> macs;
    static std::map<std::string, int> sessions;

 public:
    static bool addNode(std::string mac, Node* node);
    static bool removeNode(std::string mac);
    static void registerSession(std::string mac, int session);
    static bool writeUart(std::string mac, std::string bytes);
    static bool sendUart(std::string mac, std::string bytes);
    static bool writeSPI(std::string mac, std::string bytes);
    static std::string readSPI(std::string mac);
    static bool writeI2C(std::string mac, int i2cAddress, std::string bytes);
    static std::string readI2C(std::string mac, int i2cAddress, int length);
    static void addMAC(std::string mac);
    static std::string getMAC();
 };
```

Код определения (реализации) этого класса:

```
#include "nodes.h"
#include "node.h"
#include <nymph/nymph.h>

// Статические инициализации.
std::map<std::string, Node*> Nodes::nodes;
std::queue<std::string> Nodes::macs;
std::map<std::string, int> Nodes::sessions;

Node* Nodes::getNode(std::string mac) {
    std::map<std::string, Node*>::iterator it;
    it = nodes.find(mac);
    if (it == nodes.end()) { return 0; }
    return it->second;
}

bool Nodes::addNode(std::string mac, Node* node) {
    std::pair<std::map<std::string, Node*>::iterator, bool> ret;
    ret = nodes.insert(std::pair<std::string, Node*>(mac, node));
    if (ret.second) { macs.push(mac); }
    return ret.second;
}
```

```
bool Nodes::removeNode(std::string mac) {
    std::map<std::string, Node*>::iterator it;
    it = nodes.find(mac);
    if (it == nodes.end()) { return false; }
    nodes.erase(it);
    return true;
}
```

Здесь показаны методы установки, добавления и удаления экземпляров класса Node.

```
void Nodes::registerSession(std::string mac, int session) {
    sessions.insert(std::pair<std::string, int>(mac, session));
}
```

Эта функция регистрирует идентификатор сеанса с новым MAC-адресом и функцией обратного вызова.

```
bool Nodes::writeUart(std::string mac, std::string bytes) {
    Node* node = getNode(mac);
    if (!node) { return false; }

    node->writeUart(bytes);

    return true;
}

bool Nodes::sendUart(std::string mac, std::string bytes) {
    std::map<std::string, int>::iterator it;
    it = sessions.find(mac);
    if (it == sessions.end()) { return false; }

    vector<NymphType*> values;
    values.push_back(new NymphString(bytes));
    string result;
    NymphBoolean* world = 0;
    if (!NymphRemoteClient::callCallback(it->second, "serialRxCallback", values, result)) {
          //
    }

    return true;
}

bool Nodes::writeSPI(std::string mac, std::string bytes) {
    Node* node = getNode(mac);
    if (!node) { return false; }

    node->writeSPI(bytes);

    return true;
}

std::string Nodes::readSPI(std::string mac) {
    Node* node = getNode(mac);
    if (!node) { return std::string(); }

    return node->readSPI();
}
```

```
bool Nodes::writeI2C(std::string mac, int i2cAddress, std::string bytes) {
    Node* node = getNode(mac);
    if (!node) { return false; }

    node->writeI2C(i2cAddress, bytes);

    return true;
}

std::string Nodes::readI2C(std::string mac, int i2cAddress, int length) {
    Node* node = getNode(mac);
    if (!node) { return std::string(); }

    return node->readI2C(i2cAddress, length);
}
```

Здесь показаны простые методы чтения и записи для различных интерфейсов, явно использующие MAC-адрес для определения экземпляра класса `Node`, чтобы вызвать соответствующий метод.

Следует отметить несколько более сложный метод `sendUart()`, который использует сервер NymphRPC для обращения к функции обратного вызова в процессе соответствующего узла для активизации собственной принимающей функции обратного вызова UART.

```
void Nodes::addMAC(std::string mac) {
    macs.push(mac);
}

std::string Nodes::getMAC() {
     if (macs.empty()) { return std::string(); }

     std::string val = macs.front();
     macs.pop();
     return val;
}
```

Завершают определение класса `Nodes` методы установки и получения (считывания) MAC-адреса для новых узлов.

Итак, мы имеем основу для полнофункционального сервера имитации и интеграции. В следующем разделе мы подробно рассмотрим специализированное ПО и клиентскую сторону системы, прежде чем перейти к полному объединению всех компонентов.

Makefile

Makefile для серверной части проекта выглядит следующим образом:

```
export TOP := $(CURDIR)

GPP = g++
GCC = gcc
MAKEDIR = mkdir -p
RM = rm

OUTPUT = bmac_server
INCLUDE = -I .
FLAGS := $(INCLUDE) -g3 -std=c++17 -U__STRICT_ANSI__
```

```
LIB := -lnymphrpc -lPocoNet -lPocoUtil -lPocoFoundation -lPocoJSON
CPPFLAGS := $(FLAGS)
CFLAGS := -g3
CPP_SOURCES := $(wildcard *.cpp) $(wildcard devices/*.cpp)
CPP_OBJECTS := $(addprefix obj/,$(notdir) $(CPP_SOURCES:.cpp=.o))

all: makedir $(C_OBJECTS) $(CPP_OBJECTS) bin/$(OUTPUT)

obj/%.o: %.cpp
    $(GPP) -c -o $@ $< $(CPPFLAGS)

bin/$(OUTPUT):
    -rm -f $@
    $(GPP) -o $@ $(C_OBJECTS) $(CPP_OBJECTS) $(LIB)

makedir:
    $(MAKEDIR) bin
    $(MAKEDIR) obj/devices

clean:
    $(RM) $(CPP_OBJECTS)
```

Это относительно простой Makefile, который не содержит каких-либо особых требований. Обрабатываются файлы исходного кода, определяются имена создаваемых объектных файлов, выполняется общая компиляция, после чего из полученных объектных файлов генерируется итоговый бинарный файл.

Узел

В этом разделе подробно рассматривается специализированное ПО для комплексного теста с адаптацией реализации Arduino API, используемой в рабочей среде Sming.

Здесь самым важным является тот факт, что нет никакой необходимости каклибо менять исходный код специализированного ПО. Изменения, отличающие тестовый код от исходного образа специализированного ПО для микроконтроллера ESP8266, планируется ввести только в API, с которыми взаимодействует наше приложение.

Это означает, что сначала необходимо определить API, с которыми взаимодействует наш код, и скорректировать их реализацию способом, поддерживаемым на целевой (настольной) платформе. Для специализированного ПО на основе ESP8266, например, это означает, что сторона сети Wi-Fi остается нереализованной, так как используется стек локальной сети операционной системы, следовательно, нет необходимости уделять внимание этим подробностям.

Интерфейсы I2C, SPI и UART также реализованы как явные заглушки, которые обращаются к соответствующим компонентам интерфейса RPC, рассмотренным в предыдущем разделе. Для клиента протокола MQTT можно воспользоваться библиотекой `emqtt`, представляющей часть рабочей среды Sming. Но сразу обнаруживается, что эта библиотека предназначена для использования во встроенных системах, где необходим код, отвечающий за ее соединение с сетевым стеком.

Наш код взаимодействует с API, предлагаемым классом `MqttClient` из рабочей среды Sming. Этот класс является производным от класса `TcpClient` и использует библиотеку `emqtt` для работы по протоколу MQTT. Следуя по иерархии исходно-

го кода, в конечном итоге мы приходим к классу соединения по протоколу TCP, прежде чем углубиться в исследование библиотеки сетевого стека LWIP на более низком уровне.

Чтобы сэкономить время и трудозатраты, проще всего воспользоваться другой библиотекой поддержки протокола MQTT, например клиентской библиотекой Mosquitto, работающей в настольных ОС, следовательно, использующей API сокетов, предоставляемый самой операционной системой. Эту библиотеку можно без затруднений связать с методами, предоставляемыми классом клиента MQTT из Sming.

Заголовочный файл для этого класса почти не изменяется, за исключением того, что добавляются включаемые файлы для использования библиотеки Mosquitto, как показано ниже:

```
class TcpClient;
#include "../Delegate.h"
#include "../../Wiring/WString.h"
#include "../../Wiring/WHashMap.h"
#include "libmosquitto/cpp/mosquittopp.h"
#include "URL.h"

typedef Delegate<void(String topic, String message)> MqttStringSubscriptionCallback;
typedef Delegate<void(uint16_t msgId, int type)> MqttMessageDeliveredCallback;
typedef Delegate<void(TcpClient& client, bool successful)> TcpClientCompleteDelegate;

class MqttClient;
class URL;

class MqttClient : public mosqpp::mosquittopp {
 public:
    MqttClient(bool autoDestruct = false);
    MqttClient(String serverHost, int serverPort, MqttStringSubscriptionCallback callback = NULL);
    virtual ~MqttClient();

    void setCallback(MqttStringSubscriptionCallback subscriptionCallback = NULL);

    void setCompleteDelegate(TcpClientCompleteDelegate completeCb);
    void setKeepAlive(int seconds);
    void setPingRepeatTime(int seconds);
    bool setWill(const String& topic, const String& message, int QoS, bool retained = false);
    bool connect(const URL& url, const String& uniqueClientName, uint32_t sslOptions = 0);
    bool connect(const String& clientName, bool useSsl = false, uint32_t sslOptions = 0);
    bool connect(const String& clientName, const String& username, const String& password,
                 bool useSsl = false, uint32_t sslOptions = 0);

    bool publish(String topic, String message, bool retained = false);
    bool publishWithQoS(String topic, String message, int QoS, bool retained = false,
                        MqttMessageDeliveredCallback onDelivery = NULL);

    bool subscribe(const String& topic);
    bool unsubscribe(const String& topic);

    void on_message(const struct mosquitto_message* message);

 protected:
    void debugPrintResponseType(int type, int len);
    static int staticSendPacket(void* userInfo, const void* buf, unsigned int count);
```

```
private:
    bool privateConnect(const String& clientName, const String& username,
                        const String& password, bool useSsl = false,
                        uint32_t sslOptions = 0);
    URL url;
    mosqpp::mosquittopp mqtt;
    int waitingSize;
    uint8_t buffer[MQTT_MAX_BUFFER_SIZE + 1];
    uint8_t* current;
    int posHeader;
    MqttStringSubscriptionCallback callback;
    TcpClientCompleteDelegate completed = nullptr;
    int keepAlive = 60;
    int pingRepeatTime = 20;
    unsigned long lastMessage = 0;
    HashMap<uint16_t, MqttMessageDeliveredCallback> onDeliveryQueue;
};
```

Здесь включен заголовочный файл для обертки на языке C++ для клиентской библиотеки Mosquitto той версии, которая используется в рассматриваемом примере проекта. Причина в том, что официальная версия этой библиотеки не поддерживает сборку с помощью MinGW.

После включения заголовочного файла библиотеки мы получаем класс, производный от класса клиента MQTT библиотеки Mosquitto.

Разумеется, необходимо полностью изменить реализацию класса клиента Sming MQTT, как показано в следующем фрагменте кода:

```
#include "MqttClient.h"
#include "../Clock.h"
#include <algorithm>
#include <cstring>

MqttClient::MqttClient(bool autoDestruct /* = false*/)
{
    memset(buffer, 0, MQTT_MAX_BUFFER_SIZE + 1);
    waitingSize = 0;
    posHeader = 0;
    current = NULL;

    mosqpp::lib_init();
}

MqttClient::MqttClient(String serverHost, int serverPort,
                       MqttStringSubscriptionCallback callback /* = NULL*/)
{
    url.Host = serverHost;
    url.Port = serverPort;
    this->callback = callback;
    waitingSize = 0;
    posHeader = 0;
    current = NULL;

    mosqpp::lib_init();
}
```

Конструктор просто инициализирует библиотеку Mosquitto, для которой не требуются какие-либо дополнительные входные данные.

```
MqttClient::~MqttClient() {
    mqtt.loop_stop();
    mosqpp::lib_cleanup();
}
```

В деструкторе (его код приведен выше) останавливается выполнение потока слушающего клиента MQTT, который был активизирован при соединении с брокером MQTT, и освобождаются ресурсы, используемые библиотекой.

```
void MqttClient::setCallback(MqttStringSubscriptionCallback callback) {
    this->callback = callback;
}

void MqttClient::setCompleteDelegate(TcpClientCompleteDelegate completeCb)
{
    completed = completeCb;
}

void MqttClient::setKeepAlive(int seconds) {
    keepAlive = seconds;
}

void MqttClient::setPingRepeatTime(int seconds) {
    if(pingRepeatTime > keepAlive) {
          pingRepeatTime = keepAlive;
    } else {
          pingRepeatTime = seconds;
    }
}

bool MqttClient::setWill(const String& topic, const String& message, int QoS,
                         bool retained /* = false*/)
{
    return mqtt.will_set(topic.c_str(), message.length(), message.c_str(), QoS, retained);
}
```

Не все из приведенных выше вспомогательных функций будут использоваться, но их код включен для полноты и завершенности реализации. Кроме того, трудно предвидеть, какие из этих функций могут потребоваться, поэтому зачастую лучше реализовать немного больше функций, чем необходимо, особенно если их размер невелик и время, затраченное на их написание, меньше, чем время для определения нужности или ненужности того или иного метода или функции.

```
bool MqttClient::connect(const URL& url, const String& clientName, uint32_t sslOptions) {
    this->url = url;
    if(!(url.Protocol == "mqtt" || url.Protocol == "mqtts")) {
          return false;
    }

    waitingSize = 0;
    posHeader = 0;
    current = NULL;
```

```
    bool useSsl = (url.Protocol == "mqtts");
    return privateConnect(clientName, url.User, url.Password, useSsl, sslOptions);
}

bool MqttClient::connect(const String& clientName, bool useSsl /* = false */,
                         uint32_t sslOptions /* = 0 */)
{
    return MqttClient::connect(clientName, "", "", useSsl, sslOptions);
}

bool MqttClient::connect(const String& clientName, const String& username,
                         const String& password, bool useSsl /* = false */,
                         uint32_t sslOptions /* = 0 */)
{
    return privateConnect(clientName, username, password, useSsl, sslOptions);
}
```

Методы `connect` остаются неизменными, поскольку все они используют один и тот же `private` (закрытый) метод класса для выполнения реальной операции соединения.

```
bool MqttClient::privateConnect(const String& clientName, const String& username,
                                const String& password, bool useSsl /* = false */,
                                uint32_t sslOptions /* = 0 */)
{
    if (clientName.length() > 0) {
          mqtt.reinitialise(clientName.c_str(), false);
    }

    if (username.length() > 0) {
          mqtt.username_pw_set(username.c_str(), password.c_str());
    }

    if (useSsl) {
          //
    }

    mqtt.connect(url.Host.c_str(), url.Port, keepAlive);
    mqtt.loop_start();
    return true;
}
```

Это первый фрагмент кода, в котором напрямую используется библиотека Mosquitto. Клиентский экземпляр реинициализируется либо без пароля, но с использованием TLS (анонимный доступ к брокеру), либо с паролем через TLS (здесь этот вариант не реализован, так как он не нужен).

В этом методе также инициализируется поток прослушивания для клиента MQTT, который будет обрабатывать все входящие сообщения, поэтому нет необходимости в дальнейшем уделять внимание данному аспекту процесса.

```
bool MqttClient::publish(String topic, String message, bool retained /* = false*/)
{
    int res = mqtt.publish(0, topic.c_str(), message.length(), message.c_str(), 0, retained);
    return res > 0;
}
```

```
bool MqttClient::publishWithQoS(String topic, String message, int QoS,
                                bool retained /* = false*/,
                                MqttMessageDeliveredCallback onDelivery /* = NULL */)
{
    int res = mqtt.publish(0, topic.c_str(), message.length(), message.c_str(), QoS, retained);
    return res > 0;
}
```

Реализация функциональности для публикации сообщений MQTT связывается непосредственно с методами библиотеки Mosquitto.

```
bool MqttClient::subscribe(const String& topic)
{
    int res = mqtt.subscribe(0, topic.c_str());
    return res > 0;
}

bool MqttClient::unsubscribe(const String& topic)
{
    int res = mqtt.unsubscribe(0, topic.c_str());
    return res > 0;
}
```

Операции подписки и отмены подписки также легко связываются с экземпляром клиента MQTT.

```
void MqttClient::on_message(const struct mosquitto_message* message)
{
    if (callback) {
          callback(String(message->topic), String((char*) message->payload,
                   message->payloadlen));
    }
}
```

Последним реализован метод обратного вызова `callback` для обработки события получения нового сообщения от брокера. Для каждого принимаемого сообщения вызывается зарегистрированный метод `callback` (из кода специализированного ПО) для извлечения из сообщения заголовка (темы) и полезной нагрузки (тела сообщения).

Это все, что необходимо для взаимодействия клиента MQTT со специализированным ПО. Далее необходимо реализовать все API, совместимые с настольной ОС.

Специализированное ПО использует следующие заголовочные файлы Sming.

```
#include <user_config.h>
#include <SmingCore/SmingCore.h>
```

Первый заголовочный файл определяет некоторые зависимые от платформы функциональные характеристики, поэтому его мы оставим как есть. Второй заголовочный файл добавляет все, что нам нужно.

Для проверки кода специализированного ПО на наличие зависимостей от API используются стандартные инструменты текстового поиска, чтобы найти все вызовы функций и отфильтровать те, которые обращаются к методам рабочей среды

Sming. После этого можно написать следующий заголовочный файл *SmingCore.h* с найденными зависимостями.

```
#include <cstdint>
#include <cstdio>
#include <string>
#include <iostream>
#include "wiring/WString.h"
#include "wiring/WVector.h"
#include "wiring/WHashMap.h"
#include "FileSystem.h"
#include "wiring/Stream.h"
#include "Delegate.h"
#include "Network/MqttClient.h"
#include "Timer.h"
#include "WConstants.h"
#include "Clock.h"

#include <nymph/nymph.h>
```

Сначала включаются заголовки стандартной библиотеки языка C и библиотеки STL, затем перечисляется ряд заголовочных файлов, определяющих остальные API, которые мы реализуем. Также используются заголовочные файлы, определяющие классы, которые применяются реализуемыми API, но не входят в состав специализированного ПО.

Такие классы, как, например, `Delegate`, достаточно абстрактны, чтобы можно было их использовать как есть. Как мы увидим в дальнейшем, классы `Filesystem` и `Timer` требуют небольшой доработки для наших целей. Ранее мы уже наблюдали внесение необходимых изменений в код клиента MQTT.

Разумеется, включен также заголовочный файл библиотеки NymphRPC для обеспечения обмена информацией со стороной сервера в комплексном тесте.

```
typedef uint8_t uint8;
typedef uint16_t uint16;
typedef uint32_t uint32;
typedef int8_t int8;
typedef int16_t int16;
typedef int32_t int32;
typedef uint32_t u32_t;
```

Для обеспечения совместимости необходимо определить группу типов, используемых в специализированном ПО. Они равнозначны типам из заголовка *cstdint* стандартной библиотеки языка C, поэтому можно воспользоваться простыми операторами `typedef`, как показано выше.

```
#define UART_ID_0 0 ///< Идентификатор UART 0
#define UART_ID_1 1 ///< Идентификатор UART 1
#define SERIAL_BAUD_RATE 115200

typedef Delegate<void(Stream& source, char arrivedChar, uint16_t availableCharsCount)>
                     StreamDataReceivedDelegate;

class SerialStream : public Stream
{
```

```
    //
 public:
    SerialStream();
    size_t write(uint8_t);
    int available();
    int read();
    void flush();
    int peek();
};

class HardwareSerial
{
    int uart;
    uint32_t baud;
    static StreamDataReceivedDelegate HWSDelegate;
    static std::string rxBuffer;

 public:
    HardwareSerial(const int uartPort);
    void begin(uint32_t baud = 9600);
    void systemDebugOutput(bool enable);
    void end();
    size_t printf(const char *fmt, ...);
    void print(String str);
    void println(String str);
    void println(const char* str);
    void println(int16_t ch);
    void setCallback(StreamDataReceivedDelegate dataReceivedDelegate);
    static void dataReceivedCallback(NymphMessage* msg, void* data);
    size_t write(const uint8_t* buffer, size_t size);
    size_t readBytes(char *buffer, size_t length);
};

extern HardwareSerial Serial;
```

Первым API с измененной реализацией является последовательное устройство на аппаратной основе. Поскольку это устройство обменивается информацией напрямую с виртуальным интерфейсом на сервере, необходимо просто предоставить здесь соответствующие методы с их определением в файле исходного кода, как мы увидим немного позже.

Также объявляется глобальный экземпляр класса этого последовательного объекта, точно так же, как это делает реализация исходной рабочей среды.

```
struct rboot_config {
    uint8 current_rom;
    uint32 roms[2];
};

int rboot_get_current_rom();
void rboot_set_current_rom(int slot);
rboot_config rboot_get_config();
class rBootHttpUpdate;
typedef Delegate<void(rBootHttpUpdate& client, bool result)> OtaUpdateDelegate;

class rBootHttpUpdate
```

```
{
    //
 public:
    void addItem(int offset, String firmwareFileUrl);
    void setCallback(OtaUpdateDelegate reqUpdateDelegate);
    void start();
};

void spiffs_mount_manual(u32_t offset, int count);
```

Для функциональности менеджера загрузки rboot и файловой системы SPIFFS не существует равнозначных объектов в настольной системе, поэтому они объявляются здесь (но, как вскоре станет ясно, они остаются простыми заглушками).

```
class StationClass
{
    String mac;
    bool enabled;

 public:
    void enable(bool enable);
    void enable(bool enable, bool save);
    bool config(const String& ssid, const String& password, bool autoConnectOnStartup = true,
                bool save = true);
    bool connect();
    String getMAC();

    static int handle;
};

extern StationClass WifiStation;

class AccessPointClass
{
    bool enabled;

 public:
    void enable(bool enable, bool save);
    void enable(bool enable);
};

extern AccessPointClass WifiAccessPoint;

class IPAddress {
    //
 public:
    String toString();
};

typedef Delegate<void(uint8_t[6], uint8_t)> AccessPointDisconnectDelegate;
typedef Delegate<void(String, uint8_t, uint8_t[6], uint8_t)> StationDisconnectDelegate;
typedef Delegate<void(IPAddress, IPAddress, IPAddress)> StationGotIPDelegate;

class WifiEventsClass
{
    //
 public:
```

```
    void onStationGotIP(StationGotIPDelegate delegateFunction);
    void onStationDisconnect(StationDisconnectDelegate delegateFunction);
};

extern WifiEventsClass WifiEvents;
```

На стороне сети необходимо представить экземпляры всех классов и соответствующую информацию, которая обычно используется для установления соединения с точкой доступа Wi-Fi и гарантии того, что соединение действительно установлено. Поскольку здесь функциональность Wi-Fi не тестируется, соответствующие методы почти не используются, но они необходимы для полноты и завершенности кода специализированного ПО и для удовлетворения требований компилятора.

```
void debugf(const char *fmt, ...);

class WDTClass {
    //
 public:
    void alive();
};

extern WDTClass WDT;
```

Затем объявляется функция вывода для режима отладки, а также класс сторожевого таймера.

```
class TwoWire
{
    uint8_t rxBufferIndex;
    std::string buffer;
    int i2cAddress;

 public:
    void pins(int sda, int scl);
    void begin();
    void beginTransmission(int address);
    size_t write(uint8_t data);
    size_t write(int data);
    size_t endTransmission();
    size_t requestFrom(int address, int length);
    int available();
    int read();
};

extern TwoWire Wire;

class SPISettings
{
    //
 public:
    //
};

class SPIClass {
    //
```

```
 public:
    void begin();
    void end();
    void beginTransaction(SPISettings mySettings);
    void endTransaction();
    void transfer(uint8* buffer, size_t numberBytes);
};

extern SPIClass SPI;
```

Объявляются два типа (класса) шин обмена данными, а также глобальные экземпляры обоих типов шин.

```
void pinMode(uint16_t pin, uint8_t mode);
void digitalWrite(uint16_t pin, uint8_t val);
uint8_t digitalRead(uint16_t pin);

uint16_t analogRead(uint16_t pin);
```

Эти функции необходимы, поскольку специализированное ПО содержит код, использующий контакты интерфейса GPIO и аналого-цифрового преобразователя (ADC).

```
String system_get_sdk_version();
int system_get_free_heap_size();
int system_get_cpu_freq();
int system_get_chip_id();
int spi_flash_get_id();

class SystemClass
{
    //
 public:
    void restart();
};

extern SystemClass System;

// --- TcpClient ---
class TcpClient
{
    //
 public:
    //
};

extern void init();
```

В завершающей части объявляются классы и функции, которые в основном нужны для выполнения требований компилятора. Они практически не используются для решения текущей задачи комплексного тестирования, тем не менее возможна их реализация для создания расширенных сценариев тестирования.

Далее рассмотрим реализацию всех объявленных выше функций.

```
#include "SmingCore.h"

#include <iostream>
```

```
#include <cstdio>
#include <cstdarg>

int StationClass::handle;
```

В этом модуле компиляции переменная `handle` объявляется как статическая. Она предназначена для хранения идентификатора дескриптора удаленного сервера с целью использования его для дальнейших операций после установления соединения с сервером RPC.

```
void logFunction(int level, string logStr) {
    std::cout << level << " - " << logStr << std::endl;
}
```

Как и в коде для стороны сервера, здесь определяется простая функция ведения журнала для использования с библиотекой NymphRPC.

```
void debugf(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    int written = vfprintf(stdout, fmt, ap);
    va_end(ap);
}
```

Реализуется простая функция вывода отладочной информации с использованием стиля форматирования строки языка C для полного соответствия сигнатуре функции.

```
StreamDataReceivedDelegate HardwareSerial::HWSDelegate = nullptr;
std::string HardwareSerial::rxBuffer;
HardwareSerial Serial(0);
```

Определяется последовательный делегированный обратный вызов и буфер приема данных последовательного интерфейса как статические объекты, так как предполагается наличие единственного интерфейса UART, способного принимать данные (RX). Это предположение верно для микроконтроллера ESP8266. Также создается один экземпляр класса `HardwareSerial` для интерфейса UART 0.

```
SerialStream::SerialStream() { }
size_t SerialStream::write(uint8_t) { return 1; }
int SerialStream::available() { return 0; }
int SerialStream::read() { return 0; }
void SerialStream::flush() { }
int SerialStream::peek() { return 0; }
```

Здесь класс `SerialStream` работает исключительно как заглушка. Поскольку методы этого объекта вообще не используются в нашем коде, все они остаются нереализованными.

```
HardwareSerial::HardwareSerial(const int uartPort) {
    uart = uartPort;
}

void HardwareSerial::begin(uint32_t baud/* = 9600*/) {
    this->baud = baud;
}
```

```
void HardwareSerial::systemDebugOutput(bool enable) { }
void HardwareSerial::end() { }

size_t HardwareSerial::printf(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    int written = vfprintf(stdout, fmt, ap);
    va_end(ap);
    return written;
}

void HardwareSerial::print(String str) {
    std::cout << str.c_str();
}

void HardwareSerial::println(String str) {
    std::cout << str.c_str() << std::endl;
}

void HardwareSerial::println(const char* str) {
    std::cout << str << std::endl;
}

void HardwareSerial::println(int16_t ch) {
    std::cout << std::hex << ch << std::endl;
}

void HardwareSerial::setCallback(StreamDataReceivedDelegate dataReceivedDelegate) {
    HWSDelegate = dataReceivedDelegate;
}
```

Большинство методов класса `HardwareSerial` достаточно просто реализованы как обычная запись в стандартный (системный) поток вывода или присваивание значения переменной. Некоторые методы не отличаются от оригиналов, даже в случае назначения функции делегирования обратного вызова в последнем методе этой группы, где код оригинала вызывается с обращением к API низкого уровня на основе языка C из комплекта разработки (SDK) для микроконтроллера ESP8266.

```
void HardwareSerial::dataReceivedCallback(NymphMessage* msg, void* data) {
    rxBuffer = ((NymphString*) msg->parameters()[0])->getValue();
    SerialStream stream;
    int length = rxBuffer.length();
    int i = 0;
    HWSDelegate(stream, rxBuffer[i], length - i);
}
```

Для приема сообщений интерфейса UART определяется функция обратного вызова NymphRPC, которая именно по этой причине определена как статическая. Поскольку в микроконтроллере ESP8266 имеется только один интерфейс UART, способный принимать данные, этого вполне достаточно.

При вызове этот метод считывает полезную нагрузку, принятую на UART, и вызывает функцию `callback`, ранее зарегистрированную в специализированном ПО.

```
size_t HardwareSerial::write(const uint8_t* buffer, size_t size) {
    vector<NymphType*> values;
    values.push_back(new NymphString(WifiStation.getMAC().c_str()));
```

```
    values.push_back(new NymphString(std::string((const char*) buffer, size)));
    NymphType* returnValue = 0;
    std::string result;
    if (!NymphRemoteServer::callMethod(StationClass::handle, "writeUart", values,
                                       returnValue, result)) {
          std::cout << "Error calling remote method: " << result << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return 0;
    }

    if (returnValue->type() != NYMPH_BOOL) {
          std::cout << "Return value wasn't a boolean. Type: " << returnValue->type()
                    << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return 0;
    }

    return size;
}
```

Запись в удаленный интерфейс UART выполняется с использованием вызова RPC. Для этого создается вектор библиотеки STL, который заполняется параметрами для передачи в правильном порядке. В рассматриваемом примере это MAC-адрес узла и данные, которые необходимо передать в удаленный интерфейс UART.

После этого используется дескриптор NymphRPC, полученный при установлении соединения с обращением к серверу RPC, и начинается интервал ожидания ответа от удаленной функции.

```
size_t HardwareSerial::readBytes(char* buffer, size_t length) {
    buffer = rxBuffer.data();
    return rxBuffer.length();
}
```

Считывание из интерфейса UART выполняется после приема данных UART с помощью приведенного выше метода так же, как это делается в оригинальном коде.

```
int rboot_get_current_rom() { return 0; }
void rboot_set_current_rom(int slot) { }
rboot_config rboot_get_config() {
    rboot_config cfg;
    cfg.current_rom = 0;
    cfg.roms[0] = 0x1000;
    cfg.roms[1] = 0x3000;
    return cfg;
}

void rBootHttpUpdate::addItem(int offset, String firmwareFileUrl) { }
void rBootHttpUpdate::setCallback(OtaUpdateDelegate reqUpdateDelegate) { }
void rBootHttpUpdate::start() { }

void spiffs_mount_manual(u32_t offset, int count) { }
```

Менеджер загрузки rboot и файловая система SPIFFS в комплексном тесте не используются, поэтому можно ограничиться возвратом безопасных значений, как показано в приведенном выше коде. Функциональность передачи по воздуху (over the air – OTA) можно было бы реализовать в зависимости от функциональных характеристик тестируемой системы.

```
StationClass WifiStation;

void StationClass::enable(bool enable) { enabled = enable; }
void StationClass::enable(bool enable, bool save) { enabled = enable; }
String StationClass::getMAC() { return mac; }

bool StationClass::config(const String& ssid, const String& password,
                          bool autoConnectOnStartup /* = true*/, bool save /* = true */) {
    //
    return true;
}
```

При отсутствии адаптера Wi-Fi, которым можно было бы воспользоваться напрямую, и при использовании только сетевых возможностей настольной ОС объект `WiFiStation` не применяет большинство своих методов, за исключением метода действительного установления соединения с сервером RPC.

```
bool StationClass::connect() {
    long timeout = 5000; // 5 секунд.
    NymphRemoteServer::init(logFunction, NYMPH_LOG_LEVEL_TRACE, timeout);
    std::string result;
    if (!NymphRemoteServer::connect("localhost", 4004, StationClass::handle, 0, result)) {
          cout << "Connecting to remote server failed: " << result << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return false;
    }

    vector<NymphType*> values;
    NymphType* returnValue = 0;
    if (!NymphRemoteServer::callMethod(StationClass::handle, "getNewMac", values,
                                       returnValue, result)) {
          std::cout << "Error calling remote method: " << result << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return false;
    }

    if (returnValue->type() != NYMPH_STRING) {
          std::cout << "Return value wasn't a string. Type: " << returnValue->type() <<
std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return false;
    }

    std::string macStr = ((NymphString*) returnValue)->getValue();
    mac = String(macStr.data(), macStr.length());

    delete returnValue;
```

```
    returnValue = 0;

    // Установка функции обратного вызова для последовательного интерфейса.
    NymphRemoteServer::registerCallback("serialRxCallback", HardwareSerial::dataReceivedCallb
ack,0);
    return true;
}
```

Это один из первых методов, вызываемых из специализированного ПО при попытке установления соединения с точкой доступа Wi-FI. Вместо соединения с точкой доступа Wi-Fi здесь этот метод используется для установления соединения с сервером RPC.

Инициализация библиотеки NymphRPC начинается с вызова метода инициализации класса `NymphRemoteServer`. Далее устанавливается соединение с сервером RPC с использованием жестко закодированной локации и номера порта. После успешного соединения с сервером RPC клиент получает список доступных методов сервера. В рассматриваемом здесь примере это все зарегистрированные методы, которые рассматривались в коде сервера имитации в предыдущем разделе.

После этого выполняется запрос на получение MAC-адреса клиента от сервера, проверка полученной строки и установка MAC-адреса для дальнейшего использования. Далее локально регистрируется функция обратного вызова для интерфейса UART с помощью библиотеки NymphRPC. Как мы уже видели в разделе сервера имитации, класс `Nodes` на сервере предполагает, что эта функция обратного вызова существует в программном коде клиента.

```
AccessPointClass WifiAccessPoint;

void AccessPointClass::enable(bool enable, bool save) {
    enabled = enable;
}

void AccessPointClass::enable(bool enable) {
    enabled = enable;
}

WifiEventsClass WifiEvents;

String IPAddress::toString() { return "192.168.0.32"; }

void WifiEventsClass::onStationGotIP(StationGotIPDelegate delegateFunction) {
    // Немедленное обращение к функции обратного вызова.
    IPAddress ip;
    delegateFunction(ip, ip, ip);
}

void WifiEventsClass::onStationDisconnect(StationDisconnectDelegate delegateFunction) {
    //
}

WDTClass WDT;

void WDTClass::alive() { }
```

Сетевая секция кода завершается несколькими классами-заглушками и классом сторожевого таймера, который может стать превосходным пунктом расширенного тестирования, включая тестирование программной перезагрузки для

долговременно выполняющегося кода. Разумеется, подобные расширенные тесты потребуют, чтобы код выполнялся с учетом производительности микроконтроллера ESP8266, то есть при тактовой частоте процессора ниже 100 МГц.

Следует отметить класс событий `WifiEventsClass`, в котором выполняется немедленное обращение к функции обратного вызова `callback` для успешного установления соединения с точкой доступа Wi-Fi или, по крайней мере, для имитации этого события. Без возникновения данного события специализированное ПО входит в бесконечный цикл ожидания.

```
void SPIClass::begin() { }
void SPIClass::end() { }
void SPIClass::beginTransaction(SPISettings mySettings) { }
void SPIClass::endTransaction() { }

void SPIClass::transfer(uint8* buffer, size_t numberBytes) {
    vector<NymphType*> values;
    values.push_back(new NymphString(WifiStation.getMAC().c_str()));
    values.push_back(new NymphString(std::string((char*) buffer, numberBytes)));
    NymphType* returnValue = 0;
    std::string result;
    if (!NymphRemoteServer::callMethod(StationClass::handle, "writeSPI", values,
                                       returnValue, result)) {
          std::cout << "Error calling remote method: " << result << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return;
    }

    if (returnValue->type() != NYMPH_BOOL) {
          std::cout << "Return value wasn't a boolean. Type: " << returnValue->type()
                    << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return;
    }
}

SPIClass SPI;
```

Для записи в шину SPI тоже вызывается RPC-метод на сервере, после завершения выполнения которого мы получаем ответ, как показано в приведенном выше коде. В рассматриваемом здесь примере проекта для упрощения функциональность чтения шины SPI не реализована.

```
void TwoWire::pins(int sda, int scl) { }
void TwoWire::begin() { }
void TwoWire::beginTransmission(int address) { i2cAddress = address; }

size_t TwoWire::write(uint8_t data) {
    vector<NymphType*> values;
    values.push_back(new NymphString(WifiStation.getMAC().c_str()));
    values.push_back(new NymphSint32(i2cAddress));
    values.push_back(new NymphString(std::to_string(data)));
    NymphType* returnValue = 0;
```

```
    std::string result;
    if (!NymphRemoteServer::callMethod(StationClass::handle, "writeI2C", values,
                                       returnValue, result)) {
          std::cout << "Error calling remote method: " << result << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return 0;
    }

    if (returnValue->type() != NYMPH_BOOL) {
          std::cout << "Return value wasn't a boolean. Type: " << returnValue->type()
                    << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return 0;
    }

    return 1;
}

size_t TwoWire::write(int data) {
    vector<NymphType*> values;
    values.push_back(new NymphString(WifiStation.getMAC().c_str()));
    values.push_back(new NymphSint32(i2cAddress));
    values.push_back(new NymphString(std::to_string(data)));
    NymphType* returnValue = 0;
    std::string result;
    if (!NymphRemoteServer::callMethod(StationClass::handle, "writeI2C", values,
                                       returnValue, result)) {
          std::cout << "Error calling remote method: " << result << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return 0;
    }

    if (returnValue->type() != NYMPH_BOOL) {
          std::cout << "Return value wasn't a boolean. Type: " << returnValue->type()
                    << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          return 0;
    }

    return 1;
}
```

После нескольких методов-заглушек класса шины I2C приведены определения методов записи `write` для этого класса. По существу, это одинаковые методы, использующие метод `remote` для передачи данных в имитируемую шину I2C на сервере.

```
size_t TwoWire::endTransmission() { return 0; }

size_t TwoWire::requestFrom(int address, int length) {
    write(address);
```

```
    vector<NymphType*> values;
    values.push_back(new NymphString(WifiStation.getMAC().c_str()));
    values.push_back(new NymphSint32(address));
    values.push_back(new NymphSint32(length));
    NymphType* returnValue = 0;
    std::string result;
    if (!NymphRemoteServer::callMethod(StationClass::handle, "readI2C", values,
                                       returnValue, result)) {
          std::cout << "Error calling remote method: " << result << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          exit(1);
    }

    if (returnValue->type() != NYMPH_STRING) {
          std::cout << "Return value wasn't a string. Type: " << returnValue->type()
                    << std::endl;
          NymphRemoteServer::disconnect(StationClass::handle, result);
          NymphRemoteServer::shutdown();
          exit(1);
    }

    rxBufferIndex = 0;
    buffer = ((NymphString*) returnValue)->getValue();
    return buffer.size();
}
```

Для чтения из шины I2C используется предыдущий метод, сначала записывающий требуемый адрес I2C, затем вызывающий RPC-функцию для чтения из имитируемого I2C-устройства, в котором должны находиться данные, доступные для чтения.

```
int TwoWire::available() {
    return buffer.length() - rxBufferIndex;
}

int TwoWire::read() {
    int value = -1;
    if (rxBufferIndex < buffer.length()) {
          value = buffer.at(rxBufferIndex);
          ++rxBufferIndex;
    }

    return value;
}

TwoWire Wire;
```

Функции чтения шины I2C по существу те же, что и в исходной реализации, поскольку они просто взаимодействуют с локальным буфером, как показано в приведенном выше фрагменте кода.

```
String system_get_sdk_version() { return "SIM_0.1"; }
int system_get_free_heap_size() { return 20000; }
int system_get_cpu_freq() { return 1200000; }
int system_get_chip_id() { return 42; }
```

```
int spi_flash_get_id() { return 42; }

void SystemClass::restart() { }

SystemClass System;
```

Это функции-заглушки, которые можно использовать для специализированных вариантов тестирования.

```
void pinMode(uint16_t pin, uint8_t mode) { }
void digitalWrite(uint16_t pin, uint8_t val) { }
uint8_t digitalRead(uint16_t pin) { return 1; }

uint16_t analogRead(uint16_t pin) { return 1000; }
```

Приведенные выше функции оставлены без реализации, но при необходимости можно реализовать функциональность контактов интерфейса GPIO и аналого-цифрового преобразователя ADC, соединенных с виртуальными контактами GPIO на стороне сервера для управления устройствами и записи данных, не используемых интерфейсами UART, SPI или I2C. То же самое относится и к функциональности устройства широтно-импульсной модуляции (PWM).

В завершающей части этого файла исходного кода расположена основная функция `main`.

```
int main() {
    // Начало работы образа специализированного ПО.
    init();
    return 0;
}
```

Как и в версии точки входа Sming, вызывается глобальная функция `init()` в адаптированном коде специализированного ПО, которая здесь служит в качестве точки входа. Предполагается, что в этой же функции `main` при необходимости можно выполнить различные типы инициализации.

Методы класса файловой системы реализованы с использованием объединения способов доступа к файлам в стиле языка C и операций с файловой системой в стиле C++17.

```
#include "FileSystem.h"
#include "../Wiring/WString.h"

#include <filesystem>
#include <iostream>
#include <fstream>

namespace fs = std::filesystem;

file_t fileOpen(const String& name, FileOpenFlags flags) {
    file_t res;

    if ((flags & eFO_CreateNewAlways) == eFO_CreateNewAlways) {
          if (fileExist(name)) {
                fileDelete(name);
          }

          flags = (FileOpenFlags)((int)flags & ~eFO_Truncate);
    }
```

```
    res = std::fopen(name.c_str(), "r+b");
    return res;
}
```

Для упрощения этого метода передаваемые флаги игнорируются, и файл всегда открывается в режиме неограниченного чтения и записи (обработку полного комплекта флагов следует реализовать, если это каким-то образом влияет на выполнение комплексного теста).

```
void fileClose(file_t file) {
    std::fclose(file);
}

size_t fileWrite(file_t file, const void* data, size_t size) {
    int res = std::fwrite((void*) data, size, size, file);
    return res;
}

size_t fileRead(file_t file, void* data, size_t size) {
    int res = std::fread(data, size, size, file);
    return res;
}

int fileSeek(file_t file, int offset, SeekOriginFlags origin) {
    return std::fseek(file, offset, origin);
}

bool fileIsEOF(file_t file) {
    return true;
}

int32_t fileTell(file_t file) {
    return 0;
}

int fileFlush(file_t file) {
    return 0;
}

void fileDelete(const String& name) {
    fs::remove(name.c_str());
}

void fileDelete(file_t file) {
    //
}

bool fileExist(const String& name) {
    std::error_code ec;
    bool ret = fs::is_regular_file(name.c_str(), ec);
    return ret;
}

int fileLastError(file_t fd) {
    return 0;
}

void fileClearLastError(file_t fd) {
```

```
    //
}

void fileSetContent(const String& fileName, const String& content) {
    fileSetContent(fileName, content.c_str());
}

void fileSetContent(const String& fileName, const char* content) {
    file_t file = fileOpen(fileName.c_str(), eFO_CreateNewAlways | eFO_WriteOnly);
    fileWrite(file, content, strlen(content));
    fileClose(file);
}

uint32_t fileGetSize(const String& fileName) {
    int size = 0;
    try {
         size = fs::file_size(fileName.c_str());
    }
    catch (fs::filesystem_error& e) {
         std::cout << e.what() << std::endl;
     }

    return size;
}

void fileRename(const String& oldName, const String& newName) {
    try {
          fs::rename(oldName.c_str(), newName.c_str());
    }
    catch (fs::filesystem_error& e) {
          std::cout << e.what() << std::endl;
    }
}

Vector<String> fileList() {
    Vector<String> result;
    return result;
}

String fileGetContent(const String& fileName) {
    std::ifstream ifs(fileName.c_str(), std::ios::in | std::ios::binary | std::ios::ate);
     std::ifstream::pos_type fileSize = ifs.tellg();
     ifs.seekg(0, std::ios::beg);
     std::vector<char> bytes(fileSize);
     ifs.read(bytes.data(), fileSize);

     return String(bytes.data(), fileSize);
}

int fileGetContent(const String& fileName, char* buffer, int bufSize) {
    if (buffer == NULL || bufSize == 0) { return 0; }
    *buffer = 0;
    std::ifstream ifs(fileName.c_str(), std::ios::in | std::ios::binary | std::ios::ate);
    std::ifstream::pos_type fileSize = ifs.tellg();
    if (fileSize <= 0 || bufSize <= fileSize) {
          return 0;
```

```
    }

    buffer[fileSize] = 0;
    ifs.seekg(0, std::ios::beg);
    ifs.read(buffer, fileSize);
    ifs.close();

    return (int) fileSize;
}
```

Выше приведены реализации всех стандартных операций с файлами, поэтому они не требуют каких-либо особых разъяснений. Основная причина объединения стилей доступа к файлам C и C++17 заключается в том, что методы исходного API предполагают обработку в стиле C, а кроме того, функциональность комплекта разработки SDK основана на языке C.

Можно было бы сформировать отображение всех методов API в чистые функции работы с файловой системой в стиле C++17, но это привело бы к дополнительным затратам времени, которые совершенно не оправдывают себя (это не дает каких-либо преимуществ).

Реализация функций таймера использует POCO-класс `Timer` в классе Sming `SimpleTimer` для обеспечения равноценной функциональности.

```
#include "Poco/Timer.h"
#include <iostream>

typedef void (*os_timer_func_t)(void* timer_arg);

class SimpleTimer {
 public:
    SimpleTimer() : timer(0) {
          cb = new Poco::TimerCallback<SimpleTimer>(*this, &SimpleTimer::onTimer);
    }

    ~SimpleTimer() {
          stop();
          delete cb;
          if (timer) {
                delete timer;
          }
    }

    __forceinline void startMs(uint32_t milliseconds, bool repeating = false) {
          stop();
          if (repeating) {
                timer = new Poco::Timer(milliseconds, 0);
          }
          else {
                timer = new Poco::Timer(milliseconds, milliseconds);
          }

          timer->start(*cb);
    }

    __forceinline void startUs(uint32_t microseconds, bool repeating = false) {
          stop();
          uint32_t milliseconds = microseconds / 1000;
```

```
        if (repeating) {
            timer = new Poco::Timer(milliseconds, 0);
        }
        else {
            timer = new Poco::Timer(milliseconds, milliseconds);
        }

        timer->start(*cb);
    }

    __forceinline void stop() {
        timer->stop();
        delete timer;
        timer = 0;
    }

    void setCallback(os_timer_func_t callback, void* arg = nullptr)   {
        stop();
        userCb = callback;
        userCbArg = arg;
    }

 private:
    void onTimer(Poco::Timer &timer) {
        userCb(userCbArg);
    }

    Poco::Timer* timer;
    Poco::TimerCallback<SimpleTimer>* cb;
    os_timer_func_t userCb;
    void* userCbArg;
};
```

Наконец, для измененной реализации класса `Clock` используется функциональность `chrono` из стандартной библиотеки STL.

```
#include "Clock.h"
#include <chrono>

unsigned long millis() {
    unsigned long now = std::chrono::duration_cast<std::chrono::milliseconds>
                        (std::chrono::system_clock::now().time_since_epoch()).count();
    return now;
}

unsigned long micros() {
    unsigned long now = std::chrono::duration_cast<std::chrono::microseconds>
                        (std::chrono::system_clock::now().time_since_epoch()).count();
    return now;
}

void delay(uint32_t milliseconds) {
    //
}

void delayMicroseconds(uint32_t time) {   //
}
```

Функции delay здесь не реализованы, так как в рассматриваемом примере проекта они не нужны.

Makefile

Содержимое Makefile для этой части проекта приведено ниже:

```
GPP = g++
GCC = gcc
MAKEDIR = mkdir -p
RM = rm
AR = ar
ROOT = test/node
OUTPUT = bmac_esp8266
OUTLIB = lib$(OUTPUT).a
INCLUDE = -I $(ROOT)/ \
          -I $(ROOT)/SmingCore/ \
          -I $(ROOT)/SmingCore/network \
          -I $(ROOT)/SmingCore/network/Http \
          -I $(ROOT)/SmingCore/network/Http/Websocket \
          -I $(ROOT)/SmingCore/network/libmosquitto \
          -I $(ROOT)/SmingCore/network/libmosquitto/cpp \
          -I $(ROOT)/SmingCore/wiring \
          -I $(ROOT)/Libraries/BME280 \
          -I $(ROOT)/esp8266/app
FLAGS := $(INCLUDE) -g3 -U__STRICT_ANSI__
LIB := -L$(ROOT)/lib -l$(OUTPUT) -lmosquittopp -lmosquitto  -lnymphrpc \
       -lPocoNet -lPocoUtil -lPocoFoundation -lPocoJSON -lstdc++fs \
       -lssl -lcrypto
LIB_WIN :=  -lws2_32
ifeq ($(OS),Windows_NT)
    LIB := $(LIB) $(LIB_WIN)
endif
include ./esp8266/version
include ./Makefile-user.mk
CPPFLAGS := $(FLAGS) -DVERSION="\"$(VERSION)\"" $(USER_CFLAGS) -std=c++17 -Wl,--gc-sections
CFLAGS := -g3
CPP_SOURCES := $(wildcard $(ROOT)/SmingCore/*.cpp) \
               $(wildcard $(ROOT)/SmingCore/network/*.cpp) \
               $(wildcard $(ROOT)/SmingCore/network/Http/*.cpp) \
               $(wildcard $(ROOT)/SmingCore/wiring/*.cpp) \
               $(wildcard $(ROOT)/Libraries/BME280/*.cpp)
FW_SOURCES := $(wildcard esp8266/app/*.cpp)
CPP_OBJECTS := $(addprefix $(ROOT)/obj/,$(notdir) $(CPP_SOURCES:.cpp=.o))
FW_OBJECTS := $(addprefix $(ROOT)/obj/,$(notdir) $(FW_SOURCES:.cpp=.o))

all: makedir $(FW_OBJECTS) $(CPP_OBJECTS) $(ROOT)/lib/$(OUTLIB) $(ROOT)/bin/$(OUTPUT)
    $(ROOT)/obj/%.o: %.cpp
    $(GPP) -c -o $@ $< $(CPPFLAGS) $(ROOT)/obj/%.o: %.c
    $(GCC) -c -o $@ $< $(CFLAGS) $(ROOT)/lib/$(OUTLIB): $(CPP_OBJECTS)
    -rm -f $@
    $(AR) rcs $@ $^ $(ROOT)/bin/$(OUTPUT):
    -rm -f $@
```

```
    $(GPP) -o $@ $(CPPFLAGS) $(FW_SOURCES) $(LIB)

makedir:
    $(MAKEDIR) $(ROOT)/bin
    $(MAKEDIR) $(ROOT)/lib
    $(MAKEDIR) $(ROOT)/obj
    $(MAKEDIR) $(ROOT)/obj/$(ROOT)/SmingCore/network
    $(MAKEDIR) $(ROOT)/obj/$(ROOT)/SmingCore/wiring
    $(MAKEDIR) $(ROOT)/obj/$(ROOT)/Libraries/BME280
    $(MAKEDIR) $(ROOT)/obj/esp8266/app

clean:
    $(RM) $(CPP_OBJECTS) $(FW_OBJECTS)
```

В этом Makefile следует отметить, что исходные файлы берутся из двух различных подкаталогов дерева исходного кода: для тестовых API и для специализированного ПО. В первом случае файлы исходного кода сначала компилируются в объектные файлы, которые затем объединяются в архивный файл. Файлы исходного кода специализированного ПО используются напрямую вместе с библиотекой тестовой рабочей среды, хотя скомпилированные объектные файлы также остаются доступными.

Причина создания архивного файла для тестового API перед редактированием связей – способ, которым линкер ищет символы. При использовании инструмента `ar` (создание архивных файлов) создается индекс всех символов в объектных файлах, содержащихся в архиве. Это предотвращает любые ошибки линкера. Особенно важен такой подход в крупных проектах, где часто требуется, чтобы большой набор объектных файлов без проблем связывался с преобразованием в бинарный файл.

Предварительная компиляция в объектные файлы также полезна в более крупномасштабных проектах, так как инструмент `make` обеспечивает перекомпиляцию только тех файлов, которые действительно были изменены. Это существенно ускоряет процесс компиляции, следовательно, и процесс разработки. Поскольку объем исходного кода для целевого специализированного ПО относительно невелик, в этом случае можно выполнять прямую компиляцию из файлов исходного кода.

Кроме того, в этот файл включены два дополнительных файла Makefile. Первый содержит номер версии компилируемого исходного кода специализированного ПО. Это удобно, поскольку эксплуатируемый бинарный файл узла будет сообщать в точности тот номер версии, который должен быть установлен в реальном модуле ESP8266. Подобный подход существенно упрощает проверку конкретной текущей версии специализированного ПО.

Второй включаемый Makefile содержит параметры настройки, определенные пользователем, в абсолютной точности скопированные из Makefile проекта специализированного ПО, но только с теми переменными, которые необходимы для компиляции исходных кодов и работы бинарного образа специализированного ПО, как показано в следующем фрагменте кода:

```
WIFI_SSID = MyWi-FiNetwork
WIFI_PWD = MyWi-FiPassword

MQTT_HOST = localhost
```

```
# Для поддержки протокола SSL необходимо раскомментировать следующую строку или компилировать
# с указанием этого параметра.
#ENABLE_SSL=1
# Порт MQTT SSL (пример):
ifdef ENABLE_SSL
 MQTT_PORT = 8883
else
 MQTT_PORT = 1883
endif

# Раскомментировать следующую строку, если используется аутентификация с паролем.
# USE_MQTT_PASSWORD=1
# MQTT имя пользователя и пароль (если необходимо):
# MQTT_USERNAME = esp8266
# MQTT_PWD = ESPassword

# Префикс темы MQTT: добавляется ко всем подпискам и публикациям MQTT.
# Его можно оставить пустым, но он обязательно должен быть определен.
# Если префикс не остается пустым, то он должен завершаться символом '/', чтобы избежать
# объединения с именами тем.
MQTT_PREFIX =

# URL для обновления по воздуху (OTA). Изменяйте только имя хоста (и порт).
OTA_URL = http://ota.host.net/ota.php?uid=

USER_CFLAGS := $(USER_CFLAGS) -DWIFI_SSID="\"$(WIFI_SSID)"\"
USER_CFLAGS := $(USER_CFLAGS) -DWIFI_PWD="\"$(WIFI_PWD)"\"
USER_CFLAGS := $(USER_CFLAGS) -DMQTT_HOST="\"$(MQTT_HOST)"\"
USER_CFLAGS := $(USER_CFLAGS) -DMQTT_PORT="$(MQTT_PORT)"
USER_CFLAGS := $(USER_CFLAGS) -DMQTT_USERNAME="\"$(MQTT_USERNAME)"\"
USER_CFLAGS := $(USER_CFLAGS) -DOTA_URL="\"$(OTA_URL)"\"
USER_CFLAGS := $(USER_CFLAGS) -DMQTT_PWD="\"$(MQTT_PWD)"\"
ifdef USE_MQTT_PASSWORD
 USER_CFLAGS := $(USER_CFLAGS) -DUSE_MQTT_PASSWORD="\"$(USE_MQTT_PASSWORD)"\"
endif
SER_CFLAGS := $(USER_CFLAGS) -DMQTT_PREFIX="\"$(MQTT_PREFIX)"\"
```

Включение этого Makefile позволяет сформировать все необходимые определения, которые должны быть переданы компилятору. Здесь приведены все директивы препроцессора, используемые для настройки строк или для определения тех частей кода, которые должны компилироваться, как, например, код SSL в данном случае.

Но в рассматриваемом примере проекта функциональность SSL не реализована, чтобы не усложнять проект.

Сборка проекта

На стороне сервера имеются следующие зависимости – библиотеки:

- NymphRPC;
- POCO.

Для узла определены следующие зависимости:

- NymphRPC;
- POCO;
- Mosquitto.

Библиотека NymphRPC (описанная в начале текущего раздела) компилируется в соответствии с проектными инструкциями и установлена в той локации, где линкер может ее найти. Библиотеки POCO установлены с использованием системного менеджера пакетов (Linux, BSD или MSYS2) или вручную.

Для обеспечения зависимости от библиотеки Mosquitto можно скомпилировать файлы *libmosquitto* и *libmosquittopp* в проектной версии библиотеки, используя для этого Makefile в подкаталоге *test/SmingCore/network/libmosquitto*. И в этом случае необходимо разместить полученные библиотечные файлы в том месте, где линкер сможет их найти.

Если не используется MinGW, то можно также воспользоваться общедоступной версией, устанавливаемой с помощью менеджера пакетов ОС или подобного механизма.

После выполнения всех описанных выше действий можно скомпилировать программы сервера и клиента, переместившись в корневой каталог проекта и выполнив следующую команду:

```
make
```

Будет выполнена компиляция проектов стороны сервера и клиента с использованием Makefile самого верхнего уровня. Итоговые выполняемые файлы для каждого проекта будут размещены в соответствующих каталогах *bin*. Необходимо проверить и убедиться в том, что путевое имя выполняемого файла в классе сервера `Node` полностью соответствует локации, в которой находится выполняемый файл узла.

Теперь можно приступить к работе с проектом и начать сбор результатов тестов. В проект включена сокращенная специализированная версия ПО для мониторинга и управления микроклиматом здания на основе микроконтроллера ESP8266, которая будет подробно рассматриваться в главе 9 «Пример: мониторинг и управление внутренним микроклиматом в здании». Обращайтесь к содержимому этой главы, чтобы понять, как выполняется обмен информацией с имитируемыми узлами по протоколу MQTT, как активизировать модули внутри специализированного ПО и как интерпретировать данные, полученные от этих модулей, с помощью механизма MQTT.

После настройки и установки всех компонентов, описанных в данной главе, – требуется, как минимум, брокер и соответствующий ему клиент MQTT – и активизации модуля BME280 в режиме имитации можно ожидать передачи через сервис MQTT значений температуры, влажности и атмосферного давления, определяемых для конкретного помещения в имитируемом узле.

Резюме

В этой главе подробно рассматривалась эффективная методика разработки для целевых систем на основе микроконтроллеров, позволяющая проводить тестирование без дорогостоящих и долговременных циклов разработки. Была описана реализация и интеграция среды, позволяющей отлаживать приложения на основе микроконтроллеров с использованием всех преимуществ и инструментальных средств, предоставляемых настольной ОС.

Предполагается, что после изучения этой главы читатель сможет самостоятельно разрабатывать комплексные тесты для проектов на основе микроконтроллеров и эффективно использовать инструментальные средства ОС для профилирования и отладки, прежде чем перейти к заключительному этапу переноса проекта на реальную аппаратуру. Кроме того, читатель сможет выполнять отладку непосредственно на микросхеме и получить полное представление об относительной стоимости специализированных реализаций (версий) ПО.

В следующей главе будет рассматриваться процесс разработки простой информационно-развлекательной системы на основе платформы одноплатного компьютера.

Глава 8

Пример: информационно-развлекательная система на основе ОС Linux

В этой главе представлен пример реализации информационно-развлекательной системы (infotainment system), использующей одноплатный компьютер (single-board computer – SBC) под управлением ОС Linux. Здесь также описаны способы установления соединения с удаленными устройствами по протоколу Bluetooth и использования онлайновых потоковых сервисов. Полученное в итоге устройство способно воспроизводить аудиоконтент, получаемый из различных источников, без применения сложного пользовательского интерфейса. В главе рассматриваются следующие темы:

- процесс разработки для одноплатного компьютера под управлением ОС Linux;
- использование Bluetooth в ОС Linux;
- воспроизведение аудиоконтента из различных источников и запись звука (аудио);
- использование интерфейса GPIO для простого голосового ввода и распознавания голоса;
- соединение с онлайновыми сервисами потокового аудио.

Одно устройство выполняет все задачи

В наши дни информационно-развлекательные системы (infotainment systems) уже стали широко распространенной функциональной возможностью. Все началось с автомобильных развлекательных систем (in-car entertainment – ICE) (также известных под названием In-Vehicle Infotainment – IVI), происходящих от простых радиоприемников и кассетных плееров, к которым были добавлены такие функции, как навигация и соединение со смартфонами по протоколу Bluetooth для доступа к музыкальным коллекциям (библиотекам) и многое другое. Еще одна значительная функциональная характеристика – обеспечение свободы рук водителя: возможность позвонить по телефону и управлять настройкой радио, не отрывая глаз от дороги и не убирая рук с руля.

Быстрый рост широкого распространения смартфонов, обеспечивающих пользователям непрерывный доступ к новостям, прогнозу погоды и разнобразным развлечениям, а также появление встроенных помощников с голосовым интерфейсом, как в смартфонах, так и в автомобильных системах, в конечном итоге привели к возникновению управляемых голосом информационно-развлекательных систем, предназначенных для повседневного («домашнего») использования. Такие системы обычно состоят из динамика и микрофона в комплекте с требуемой аппаратурой для управляемого голосом интерфейса и доступа к необходимым сервисам интернета.

В этой главе основное внимание сосредоточено именно на этом типе управляемых голосом информационно-развлекательных систем. В главе 10 «Разработка встроенных систем с использованием Qt» будет подробно рассматриваться дополнение такой системы графическим пользовательским интерфейсом.

Цели рассматриваемой в этой главе информационно-развлекательной системы:

- воспроизведение музыки из источника, соединяемого по протоколу Bluetooth, например смартфона;
- воспроизведение музыки из онлайнового потокового сервиса;
- воспроизведение музыки из локальной файловой системы, включая USB-устройства (флеш-карты);
- запись аудиоклипа и его воспроизведение по требованию;
- управление всеми действиями голосом, но для некоторых действий предусмотрены кнопки.

В следующих разделах рассматриваются эти цели и способы их достижения.

Необходимая аппаратура

Для этого проекта вполне пригоден любой одноплатный компьютер, способный работать под управлением ОС Linux. Кроме того, для полной реализации необходима поддержка следующих функциональных возможностей:

- соединение с интернетом (проводное или беспроводное) для доступа к онлайновому контенту;
- поддержка функциональности Bluetooth (встроенная или в виде дополнительного модуля) для обеспечения работы в системе Bluetooth-динамика;
- свободные входные контакты интерфейса GPIO для подключаемых кнопок;
- функции входа для микрофона и вывода звука для обеспечения голосового ввода и воспроизведения аудио соответственно;
- возможность подключения по интерфейсу SATA или по аналогичному интерфейсу для поддержки использования устройств хранения данных, таких как жесткие диски;
- шина периферийных устройств I2C для подключения I2C-дисплея.

Для примеров исходного кода в этой главе требуется только вход для микрофона и вывод звука, а также некоторое устройство хранения данных для локальных медиафайлов.

К контактам интерфейса GPIO можно подключить несколько кнопок, используемых для управления информационно-развлекательной системой без необхо-

димости обращения к подсистеме управления голосом. Это удобно в случаях, когда использование голосового управления может оказаться затруднительным или невозможным, например для временной приостановки или снижения громкости воспроизведения музыки при входящем телефонном звонке.

В рассматриваемом здесь примере подключение кнопок не демонстрируется, но соответствующую реализацию можно найти в ранее рассматриваемом проекте из главы 3, где используется библиотека WiringPi для соединения переключателей с контактами GPIO и конфигурирования подпрограмм прерываний для обработки изменения состояния переключателей.

К системе также можно подключить небольшой дисплей, если необходимо отображать текущую информацию, например название воспроизводимого музыкального призведения или другую информацию о состоянии. В настоящее время доступны дешевые дисплеи с размером 16×2 символов, управляемые через интерфейс I2C. Такие устройства вместе с широким диапазоном OLED и прочими компактными дисплеями вполне подходят для наших целей благодаря своим минимальным аппаратным требованиям.

В главе 3 кратко рассматривались типы аппаратуры, которые могли бы использоваться в информационно-развлекательной системе, подобной описываемой здесь, а также несколько возможных вариантов пользовательских интерфейсов и устройств хранения данных. Разумеется, правильный выбор конфигурации аппаратуры зависит от конкретных проектных требований. Если необходимо локально хранить огромный объем музыкальных файлов для воспроизведения, то непременным требованием становится наличие в системе жесткого диска SATA большой емкости.

Но для примера в этой главе мы не делаем подобных предположений, сосредоточив внимание на легко расширяемом начальном варианте системы. Таким образом, требования к аппаратуре минимальны – необходим только микрофон и устройство вывода звука.

Требования к программному обеспечению

В рассматриваемом проекте предполагается, что на целевом одноплатном компьютере уже установлена ОС Linux, и все драйверы, обеспечивающие функциональность аппаратуры, то есть микрофона и устройства вывода звука, установлены и сконфигурированы.

Поскольку в этом проекте используется рабочая среда Qt, все соответствующие зависимости также должны быть удовлетворены. Это означает наличие всех требуемых совместно используемых библиотек в системе, в которой планируется работа итогового бинарного файла проекта. Рабочую среду Qt можно установить с помощью менеджера пакетов операционной системы или непосредственно с веб-сайта Qt **http://qt.io/**.

В главе 10 «Разработка встроенных систем с использованием Qt» будет более подробно рассматриваться разработка на встраиваемых платформах с применением рабочей среды Qt. В этой главе лишь кратко описывается практическое использование API Qt.

В зависимости от того, нужно ли компилировать приложение непосредственно на одноплатном компьютере или на настольном компьютере, где производилась разработка, можно установить инструментальный комплект компиляторов и все требуемые зависимости на одноплатном компьютере, или инструментальный комплект кросс-компиляции для целевой системы Linux одноплатного компьютера (для ARM, x86 или для другой архитектуры). В главе 6 рассматривалась кросс-компиляция для систем на одноплатных компьютерах, а также локальное тестирование такой системы.

Поскольку пример проекта в этой главе не требует какой-либо особенной аппаратуры, компиляцию можно выполнять на любой системе, которая поддерживает рабочую среду Qt. Такой подход рекомендуется для полного тестирования кода перед развертыванием его на целевом одноплатном компьютере.

Bluetooth-источники и приемники аудио

Несмотря на то что Bluetooth представляет собой широко распространенную технологию, ее главным недостатком является проприетарная сущность. В результате ощущается недостаточность полноценной поддержки функциональности Bluetooth (в форме профилей). Для рассматриваемого здесь проекта интересен профиль Advanced Audio Distribution Profile (A2DP). Этот профиль используется разнообразными устройствами: от наушников до динамиков Bluetooth для воспроизведения потокового аудио.

Любое устройство с реализацией A2DP способно создавать поток аудио, направленный на A2DP-приемник, или может само выступить в роли приемника (в зависимости от реализации стека Bluetooth). Теоретически это должно позволять установить соединение со смартфоном или подобным устройством из разрабатываемой информационно-развлекательной системы и воспроизводить в ней музыку, как если бы она была независимым Bluetooth-динамиком.

Приемник в профиле A2DP обозначается как A2DP sink (собственно приемник), а другая сторона является источником (source) A2DP. Наушники или динамик Bluetooth всегда должны быть приемником, так как могут только потреблять поток аудио. Настольный компьтер, одноплатный компьютер или любое другое многофункциональное устройство можно сконфигурировать так, чтобы оно работало и как приемник, и как источник.

Как было отмечено выше, сложности реализации полноценного стека Bluetooth в основных операционных системах привели к весьма слабой поддержке функциональности Bluetooth, которая обычно ограничивается простой функцией последовательного соединения.

В операционных системах FreeBSD, macOS, Windows, Android имеются стеки Bluetooth, но количество поддерживаемых адаптеров ограничено (только один в Windows и только через интерфейс USB). Кроме того, ограничена поддержка профилей (в FreeBSD только профиль передачи данных data-transfer-only) и возможности конфигурирования (в Android ориентация исключительно на смартфоны).

В настоящее время поддержка профилей в Windows 10 сокращена по сравнению с функциональностью для Windows 7 из-за изменений в стеке Bluetooth. В macOS

с версии 10.5 (Leopard, 2007 год) добавлена поддержка A2DP, поэтому все должно работать.

Стек BlueZ Bluetooth, который стал официальным стеком для Linux, изначально был разработан компанией Qualcomm, а сейчас включен в официальные дистрибутивы ядра Linux. Это одна из наиболее полноценных реализаций стека Bluetooth.

При переходе с версии 4 на 5 BlueZ была отменена поддержка API звуковой системы ALSA, вместо которой стала применяться аудиосистема PulseAudio с переименованием старых API. Это означает, что приложения и код, использующий старые API (версии 4), больше не работает. К сожалению, огромное количество примеров кода и руководств, доступных онлайн, продолжают ориентироваться на версию 4. При изучении этих материалов следует быть особенно внимательными и осторожными, так как они работают совершенно по-другому.

BlueZ конфигурируется с помощью системы D-Bus Linux IPC (interprocess communication) или прямым редактированием вручную конфигурационных файлов. В действительности реализация поддержки BlueZ в приложении, подобном рассматриваемому в этой главе, для программируемой конфигурации может оказаться достаточно сложной из-за ограниченного набора API, а также из-за ограничений в настройке параметров конфигурации, которые относятся не только к стеку Bluetooth и требуют доступа к текстовым конфигурационным файлам. Таким образом, приложение должно запускаться с правильно установленными правами доступа к определенным свойствам и файлам для редактирования их напрямую, или эти действия придется выполнять вручную.

Еще одна сложность реализации рассматриваемого здесь проекта заключается в настройке автоматического пирингового режима (pairing mode), так как в противном случае удаленное устройство (смартфон) не сможет установить соединение с информационно-развлекательной системой. Это также потребует постоянного взаимодействия со стеком Bluetooth для опроса с целью определения новых подключаемых устройств.

Каждое новое устройство должно проверяться на обеспечение поддержки режима источника A2DP, и только в этом случае оно может быть добавлено как аудиовход для системы. Затем выполняется подключение в общую аудиосистему, чтобы использовать новое входное устройство.

Из-за сложности и масштабности реализации этой функции ее исходный код не включен в пример, рассматриваемый в текущей главе. Тем не менее ее можно добавить в код проекта. На одноплатных компьютерах Raspberry Pi 3 имеется встроенный адаптер Bluetooth. В другие одноплатные компьютеры адаптер Bluetooth может добавляться как устройство USB.

Организация потока в режиме онлайн

Существует множество онлайновых потоковых сервисов, которые можно интегрировать в информационно-развлекательную систему, подобную рассматриваемой в этой главе. Все они используют похожие потоковые API (обычно REST API на основе протокола HTTP), требующие создания учетной записи в конкретном сервисе, через которую можно получить специализированный для приложения

жетон (token), предоставляющий право доступа к этому API. Это позволяет формировать запросы по определенным исполнителям, музыкальным трекам, альбомам и т. д.

Используя HTTP-клиента, например предоставляемого рабочей средой Qt, можно без особого труда реализовать необходимое управление потоком. Поскольку требуется наличие зарегистрированного идентификатора приложения для вышеупомянутых потоковых сервисов, эта часть также остается за пределами рассматриваемого здесь примера кода.

Простая последовательность для организации потока из REST API обычно выглядит так, как показано ниже, с использованием простой обертки для HTTP-вызовов.

```
#include "soundFoo"
// Создание объекта клиента с регистрационными характеристиками конкретного приложения.
client = soundFoo.new('YOUR_CLIENT_ID');
// Передача трека в поток.
track = client.get("/tracks/293")
// Получение URL треков, включаемых в поток.
stream_url = client.get(track.stream_url, true);
// URL потока, разрешены перенаправления
// Вывод URL потока треков
std::cout << stream_url.location;
```

Управляемый голосом пользовательский интерфейс

В рассматриваемом здесь проекте применяется пользовательский интерфейс, полностью управляемый голосовыми командами. Для этого выполняется реализация интерфейса преобразования голоса в текст (voice-to-text), поддерживаемого библиотекой PocketSphinx (**https://cmusphinx.github.io/**), в которой используется выделение ключевых слов и грамматический поиск для распознавания и интерпретации передаваемых команд.

По умолчанию используется модель английского американского языка (USEnglish), включенная в дистрибутив PocketSphinx. Это означает, что для точного распознавания команд их необходимо произносить с англо-американским акцентом. Модель можно заменить, загрузив другую языковую модель с требуемым языком и акцентами. На веб-сайте PocketSphinx доступны разнообразные готовые модели, а кроме того, можно создать собственную языковую модель с небольшими трудозатратами.

Использование сценариев

Информационно-развлекательная система не должна активизироваться в каждом случае, когда интерфейс распознавания голоса пользователя определяет слова команд, призносимые с другими целями (не для управления системой). Для предотвращения таких ситуаций чаще всего применяется метод ключевых слов, активизирующих командный интерфейс. Если после ключевого слова не произ-

носится распознаваемая команда в течение определенного интервала времени, то система возвращается в режим определения ключевых слов.

В рассматриваемом здесь примере проекта используется ключевое слово `computer`. После того как система обнаружила (распознала) это ключевое слово, можно произнести одну из команд, приведенных в табл. 8.1.

Таблица 8.1

Команда	**Результат выполнения**
Play Bluetooth	Начинает воспроизведение из любого подключенного устройства-источника A2DP (здесь не реализовано)
Stop Bluetooth	Останавливает воспроизведение с любого устройства Bluetooth
Play local	Воспроизводит (жестко закодированный) локальный музыкальный файл
Stop local	Останавливает воспроизведение локального музыкального файла, если в текущий момент он воспроизводится
Play remote	Воспроизведение с онлайнового потокового сервиса или сервера (здесь не реализовано)
Stop remote	Останавливает воспроизведение, если активен онлайновый сервис
Record message	Записывает сообщение. Запись останавливается после нескольких секунд молчания
Play message	Воспроизводит записанное сообщение, если таковое имеется

Исходный код

Рассматриваемое здесь приложение реализовано с использованием рабочей среды (библиотеки) Qt как приложение с графическим пользовательским интерфейсом (GUI), поэтому мы также применяем графический интерфейс для упрощения отладки. Этот отладочный пользовательский интерфейс спроектирован с помощью инструмента Qt Designer из интегрированной среды разработки Qt Creator и сохранен как отдельный UI-файл.

Начинаем с создания экземпляра приложения с графическим интерфейсом пользователя (GUI).

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Здесь создается экземпляр класса `MainWindow`, в котором реализовано приложение, а также экземпляр `QApplication`, класса-обертки, используемого рабочей средой Qt.

Содержимое заголовочного файла *MainWindow*:

```
#include <QMainWindow>

#include <QAudioRecorder>
```

```
#include <QAudioProbe>
#include <QMediaPlayer>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow {
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
public slots:
    void playBluetooth();
    void stopBluetooth();
    void playOnlineStream();
    void stopOnlineStream();
    void playLocalFile();
    void stopLocalFile();
    void recordMessage();
    void playMessage();
    void errorString(QString err);
    void quit();
private:
    Ui::MainWindow *ui;
    QMediaPlayer* player;
    QAudioRecorder* audioRecorder;
    QAudioProbe* audioProbe;
    qint64 silence; // Продолжительность тишины в микросекундах, после чего запись
                    // сообщения останавливается.
private slots:
    void processBuffer(QAudioBuffer);
};
```

Реализация содержит почти все функциональное ядро с объявлением экземпляров записывающего и воспроизводящего устройств. Обработка голосовых команд реализована в отдельном классе.

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include "voiceinput.h"

#include <QThread>
#include <QMessageBox>

#include <cmath>

#define MSG_RECORD_MAX_SILENCE_US 5000000

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    // Настройка связей меню.
    connect(ui->actionQuit, SIGNAL(triggered()), this, SLOT(quit()));
    // Настройка связей UI.
```

```
    connect(ui->playBluetoothButton, SIGNAL(pressed), this, SLOT(playBluetooth));
    connect(ui->stopBluetoothButton, SIGNAL(pressed), this, SLOT(stopBluetooth));
    connect(ui->playLocalAudioButton, SIGNAL(pressed), this, SLOT(playLocalFile));
    connect(ui->stopLocalAudioButton, SIGNAL(pressed), this, SLOT(stopLocalFile));
    connect(ui->playOnlineStreamButton, SIGNAL(pressed), this, SLOT(playOnlineStream));
    connect(ui->stopOnlineStreamButton, SIGNAL(pressed), this, SLOT(stopOnlineStream));
    connect(ui->recordMessageButton, SIGNAL(pressed), this, SLOT(recordMessage));
    connect(ui->playBackMessage, SIGNAL(pressed), this, SLOT(playMessage));

    // Настройки значений по умолчанию.
    silence = 0;
    // Создание экземпляров аудиоинтерфейса.
    player = new QMediaPlayer(this);
    audioRecorder = new QAudioRecorder(this);
    audioProbe = new QAudioProbe(this);
    // Конфигурирование устройства записи звука.
    QAudioEncoderSettings audioSettings;
    audioSettings.setCodec("audio/amr");
    audioSettings.setQuality(QMultimedia::HighQuality);
    audioRecorder->setEncodingSettings(audioSettings);
    audioRecorder->setOutputLocation(QUrl::fromLocalFile("message/last_message.amr"));
    // Конфигурирование пробы звука.
    connect(audioProbe, SIGNAL(audioBufferProbed(QAudioBuffer)), this,
                        SLOT(processBuffer(QAudioBuffer)));
    audioProbe->setSource(audioRecorder);
    // Активизация голосового интерфейса в отдельном потоке и настройка соединений.
    QThread* thread = new QThread;
    VoiceInput* vi = new VoiceInput();
    vi->moveToThread(thread);
    connect(thread, SIGNAL(started()), vi, SLOT(run()));
    connect(vi, SIGNAL(finished()), thread, SLOT(quit()));
    connect(vi, SIGNAL(finished()), vi, SLOT(deleteLater()));
    connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));
    connect(vi, SIGNAL(error(QString)), this, SLOT(errorString(QString)));
    connect(vi, SIGNAL(playBluetooth), this, SLOT(playBluetooth));
    connect(vi, SIGNAL(stopBluetooth), this, SLOT(stopBluetooth));
    connect(vi, SIGNAL(playLocal), this, SLOT(playLocalFile));
    connect(vi, SIGNAL(stopLocal), this, SLOT(stopLocalFile));
    connect(vi, SIGNAL(playRemote), this, SLOT(playOnlineStream));
    connect(vi, SIGNAL(stopRemote), this, SLOT(stopOnlineStream));
    connect(vi, SIGNAL(recordMessage), this, SLOT(recordMessage));
    connect(vi, SIGNAL(playMessage), this, SLOT(playMessage));
    thread->start();
}
```

В приведенном выше конструкторе выполняется настройка всех связей пользовательского интерфейса с кнопками в окне GUI, которые позволяют работать с функциями приложения без использования голосового интерфейса. Это очень удобно при тестировании.

Кроме того, создается экземпляр устройства записи звука и экземпляр устройства воспроизведения медиа вместе со звуковой пробой, связанной с устройством записи звука. Таким образом, можно проверить образцы звукозаписи и поработать с ними.

Наконец, создается экземпляр класса голосового интерфейса, который перед запуском помещается в собственный отдельный поток. Его сигналы связываются с определенными командами и другими событиями в соответствующих слотах.

```
MainWindow::~MainWindow() {
    delete ui;
}

void MainWindow::playBluetooth() {
    // Используется связь со стеком BlueZ Bluetooth в ядре Linux для конфигурирования его
    // работы в качестве приемника A2DP и для установления соединения со смартфонами.
}

// --- STOP BLUETOOTH ---
void MainWindow::stopBluetooth() {
    //
}
```

Как уже было отмечено в разделе, описывающем технологию Bluetooth, в рассматриваемом здесь примере функциональность Bluetooth остается нереализованной по причинам, приведенным в том же разделе.

```
void MainWindow::playOnlineStream() {
    // Установление соединения с удаленным API потокового сервиса и инициализация потока.
}

void MainWindow::stopOnlineStream() {
    // Остановка потока, получаемого от удаленного сервиса.
}
```

То же самое относится и к функциональности онлайнового потокового аудио. В одном из предыдущих разделов текущей главы подробно описана реализация этой функциональности.

```
void MainWindow::playLocalFile() {
    player->setMedia(QUrl::fromLocalFile("music/coolsong.mp3"));
    player->setVolume(50);
    player->play();
}

void MainWindow::stopLocalFile() {
    player->stop();
}
```

Для воспроизведения локального файла предполагается наличие файла в формате MP3 с жестко закодированным путевым именем. Тем не менее возможно также воспроизведение всех музыкальных файлов в заданном каталоге при минимальных изменениях в исходном коде: считывание имен файлов и последовательное их воспроизведение.

```
void MainWindow::recordMessage() {
    audioRecorder->record();
}

void MainWindow::playMessage() {
    player->setMedia(QUrl::fromLocalFile("message/last_message.arm"));
```

```
    player->setVolume(50);
    player->play();
}
```

В конструкторе сконфигурировано звукозаписывающее устройство для записи в файл, сохраняемый в подкаталоге с именем *message*. Этот файл будет перезаписываться при каждой новой записи звукового сообщения, которое можно будет воспроизвести позже. Необязательный дисплей или другой аналогичный аксессуар можно использовать для отображения информации о новой сделанной записи сообщения, которое пока еще не прослушано.

```
void MainWindow::processBuffer(QAudioBuffer buffer) {
    const quint16 *data = buffer.constData<quint16>();
    // Получить RMS буфера, если отсутствует голосовой ввод (наступила тишина),
    // добавить продолжительность тишины к счетчику.
    int samples = buffer.sampleCount();
    double sumsquared = 0;
    for (int i = 0; i < samples; i++) {
        sumsquared += data[i] * data[i];
    }
    double rms = sqrt((double(1) / samples)*(sumsquared));
    if (rms <= 100) {
        silence += buffer.duration();
    }
    if (silence >= MSG_RECORD_MAX_SILENCE_US) {
        silence = 0;
        audioRecorder->stop();
    }
}
```

Этот метод вызывается предусмотренной звуковой пробой, когда устройство записи звука активно. В этой функции вычисляется среднее квадратическое значение (root mean square – RMS) содержимого аудиобуфера, чтобы определить отсутствие в нем голосовой информации (сообщения). В данном случае понятие «тишина» является относительным и может иметь регулируемую зависимость от среды, в которой производится запись.

После того как зафиксирован интервал отсутствия голоса (тишины) длиной в пять секунд, запись сообщения автоматически останавливается.

```
void MainWindow::errorString(QString err) {
    QMessageBox::critical(this, tr("Error"), err);
}

void MainWindow::quit() {
    exit(0);
}
```

Остальные методы выполняют обработку сообщений об ошибках, которые могут возникать в приложении, а также обеспечивают завершение приложения.

В заголовке класса `VoiceInput` определяется функциональность интерфейса голосового (речевого) ввода.

```
#include <QObject>
#include <QAudioInput>
```

```
extern "C" {
#include "pocketsphinx.h"
}

class VoiceInput : public QObject {
    Q_OBJECT
    QAudioInput* audioInput;
    QIODevice* audioDevice;
    bool state;
public:
    explicit VoiceInput(QObject *parent = nullptr);
    bool checkState() { return state; }
signals:
    void playBluetooth();
    void stopBluetooth();
    void playLocal();
    void stopLocal();
    void playRemote();
    void stopRemote();
    void recordMessage();
    void playMessage();
    void error(QString err);
public slots:
    void run();
};
```

Так как PocketSphinx является библиотекой, написанной на языке C, необходимо обеспечить для нее процедуру связывания в стиле C. После этого объявляются члены класса для аудиоввода и соответствующее устройство ввода/вывода, которое будет использовать этот голосовой ввод.

Далее следует определение класса.

```
#include <QDebug>
#include <QThread>

#include "voiceinput.h"

extern "C" {
#include <sphinxbase/err.h>
#include <sphinxbase/ad.h>
}

VoiceInput::VoiceInput(QObject *parent) : QObject(parent) {
    //
}
```

В конструкторе не выполняются какие-либо специальные действия, поскольку следующий метод обеспечивает все необходимое для инициализации и настройки основного цикла.

```
void VoiceInput::run() {
    const int32 buffsize = 2048;
    int16 adbuf[buffsize];
    uint8 utt_started, in_speech;
    uint32 k = 0;
```

```
char const* hyp;
static ps_decoder_t *ps;
state = true;
QAudioFormat format;
format.setSampleRate(16000);
format.setChannelCount(1);
format.setSampleSize(16);
format.setCodec("audio/pcm");
format.setByteOrder(QAudioFormat::LittleEndian);
format.setSampleType(QAudioFormat::UnSignedInt);
// Проверка поддержки этого формата используемым аудиоустройством.
QAudioDeviceInfo info = QAudioDeviceInfo::defaultInputDevice();
if (!info.isFormatSupported(format)) {
   qWarning() << "Default format not supported, aborting.";
   state = false;
   return;
}
audioInput = new QAudioInput(format, this);
audioInput->setBufferSize(buffsize * 2);
audioDevice = audioInput->start();

if (ps_start_utt(ps) < 0) {
    E_FATAL("Failed to start utterance\n");
}
utt_started = FALSE;
E_INFO("Ready....\n");
```

В первой части этого метода выполняется настройка аудиоинтерфейса и его конфигурирование для записи с использованием параметров аудиоформата, которых требует PocketSphinx: режима монофонии (mono), порядка битов от младшего к старшему (little endian), 16-битовой импульсно-кодовой модуляции (PCM) с учетом знака с частотой 16 000 Гц. После проверки поддержки такого формата устройством аудиоввода создается новый экземпляр этого устройства.

```
const char* keyfile = "COMPUTER/3.16227766016838e-13/\n";
if (ps_set_kws(ps, "keyword_search", keyfile) != 0) {
    return;
}
if (ps_set_search(ps, "keyword_search") != 0) {
    return;
}
const char* gramfile = "grammar asr;\
        \
        public <rule> = <action> [<preposition>] [<objects>] [<preposition>] [<objects>];\
        \
        <action> = STOP | PLAY | RECORD;\
        \
        <objects> = BLUETOOTH | LOCAL | REMOTE | MESSAGE;\
        \
        <preposition> = FROM | TO;";
ps_set_jsgf_string(ps, "jsgf", gramfile);
```

Далее выполняется настройка механизма обнаружения ключевых слов и файла грамматики JSGF (Java Speech Grammar Format), который будет использоваться

при обработке звукового фрагмента. При первом вызове функции `ps_set_search()` начинается процедура обнаружения ключевых слов. Приведенный ниже цикл будет продолжать процесс обработки звуковых фрагментов, пока не будет произнесено и обнаружено ключевое слово `computer`.

```
bool kws = true;
for (;;) {
    if ((k = audioDevice->read((char*) &adbuf, 4096))) {
        E_FATAL("Failed to read audio.\n");
    }
    ps_process_raw(ps, adbuf, k, FALSE, FALSE);
    in_speech = ps_get_in_speech(ps);
    if (in_speech && !utt_started) {
        utt_started = TRUE;
        E_INFO("Listening...\n");
    }
```

На каждой итерации цикла считывается очередное содержимое буфера, то есть звуковой фрагмент, который передается процессу PocketSphinx. Также выполняется обнаружение интервала тишины, чтобы определить начало голосового ввода через микрофон. Если пользователь говорит, но интерпретация пока еще не началась, то инициализируется новое высказывание (новая итерация цикла).

```
    if (!in_speech && utt_started) {
        ps_end_utt(ps);
        hyp = ps_get_hyp(ps, nullptr);
        if (hyp != nullptr) {
            // Получена предполагаемая команда.
            if (kws && strstr(hyp, "computer") != nullptr) {
                if (ps_set_search(ps, "jsgf") != 0) {
                    E_FATAL("ERROR: Cannot switch to jsgf mode.\n");
                }
                kws = false;
                E_INFO("Switched to jsgf mode \n");
                E_INFO("Mode: %s\n", ps_get_search(ps));
            }
            else if (!kws) {
                if (hyp != nullptr) {
                    // Проверка соответствия полученной команды одному из допустимых
                       действий.
                    if (strncmp(hyp, "play bluetooth", 14) == 0) {
                        emit playBluetooth();
                    }
                    else if (strncmp(hyp, "stop bluetooth", 14) == 0) {
                        emit stopBluetooth();
                    }
                    else if (strncmp(hyp, "play local", 10) == 0) {
                        emit playLocal();
                    }
                    else if (strncmp(hyp, "stop local", 10) == 0) {
                        emit stopLocal();
                    }
                    else if (strncmp(hyp, "play remote", 11) == 0) {
```

```
                        emit stopBluetooth();
                    }
                    else if (strncmp(hyp, "stop remote", 11) == 0) {
                        emit stopBluetooth();
                    }
                    else if (strncmp(hyp, "record message", 14) == 0) {
                        emit stopBluetooth();
                    }
                    else if (strncmp(hyp, "play message", 12) == 0) {
                        emit stopBluetooth();
                    }
                }
                else {
                    if (ps_set_search(ps, "keyword_search") != 0){
                        E_FATAL("ERROR: Cannot switch to kws mode.\n");
                    }
                    kws = true;
                    E_INFO("Switched to kws mode.\n");
                }
            }
        }

        if (ps_start_utt(ps) < 0) {
            E_FATAL("Failed to start utterance\n");
        }
        utt_started = FALSE;
        E_INFO("Ready....\n");
    }
    QThread::msleep(100);
  }
}
```

В завершающей части метода проверяется, действительно ли получена предполагаемая правильная команда, которую можно проанализировать. В зависимости от текущего состояния – в режиме обнаружения ключевых слов или в режиме грамматического анализа – проверяется факт обнаружения ключевого слова в первом случае и переключение в режим грамматического анализа. Если приложение уже находится в режиме грамматического анализа, то выполняется попытка преобразования произнесенной голосом фразы в конкретную команду. Если допустимая команда обнаружена, то генерируется соответствующий сигнал, активизирующий связанную с ним функцию.

Новая голосовая фраза (высказывание) начинается, если PocketSphinx обнаруживает, как минимум, одну секунду тишины. После выполнения команды система переключается обратно в режим обнаружения ключевых слов.

Сборка проекта

Для сборки проекта приложения сначала необходимо выполнить сборку проекта PocketSphinx. В примере, рассматриваемом в данной главе, в каталоге *sphinx* существуют два файла Makefile: один в подкаталоге *pocketsphinx*, другой в подка-

талоге *sphinxbase*. С помощью этих файлов будет выполнена сборка двух библиотечных файлов, которые формируют библиотеку PocketSphinx.

После этого можно выполнить сборку проекта Qt либо из среды Qt Creator, либо из командной строки с помощью следующих команд:

```
mkdir build
cd build
qmake ..
make
```

Расширение системы

В дополнение к аудиоформатам можно также добавить возможность воспроизведения видеофрагментов и включить возможность выполнения и ответа на телефонные звонки (используя Bluetooth API). Может потребоваться изменение приложения, чтобы сделать его более гибким и модульным, то есть ввести возможность расширения приложения собственными модулями. Например, кто-то пожелает дополнить приложение модулем, позволяющим добавлять собственные голосовые команды и соответствующие им действия.

Кроме того, очень удобной была бы функция голосового вывода, приближающая приложение к коммерческим системам, предлагаемым в настоящее время. Для этого можно воспользоваться API преобразования текста в речь, также доступным в рабочей среде Qt.

Не менее полезным стало бы добавление возможности получения информации такой системой с помощью запросов к удаленным API: прогноз погоды, новости и, возможно, даже регулярно обновляемая текстовая трансляция какого-либо спортивного события, например фубольного матча. Голосовой пользовательский интерфейс может применяться для установки таймеров и напоминаний о запланированных задачах в совокупности с календарным планированием и для многого другого.

В примере исходного кода в текущей главе не предусмотрена возможность ввода названия трека, альбома или имени исполнителя, который пользователь хотел бы воспроизвести. Обеспечение ввода произвольных названий и имен чрезвычайно удобно и полезно, но с реализацией этой функции связан ряд специфических проблем.

Основная задача – скорость (темп) распознавания системы преобразования голоса в текст, особенно для слов, не содержащихся в словаре системы. Возможно, кто-то из нас уже оценил преимущества речевого управления при использовании интерфейса, управляемого голосом, в телефоне, в автомобиле или в смартфонах, распознающих определенные слова.

В любом случае, это крупная область научных и практических исследований, в которой пока не найдено простое и быстрое решение. Возможен вариант с применением методики грубой силы (простого перебора) при распознавании и достижения большей точности при использовании индекса локальных имен файлов и исполнителей в совокупности с другими метаданными, являющимися частью словаря. То же самое можно сделать и для удаленных потоковых сервисов через запросы к их API. Но при выполнении операции распознавания возможна существенная задержка.

Резюме

В этой главе рассматривался способ создания относительно простой информационно-развлекательной системы на основе одноплатного компьютера с применением механизма преобразования речи в текст для обеспечения пользовательского интерфейса, управляемого голосом. Также разбирались некоторые возможные варианты расширения такой системы, позволяющие наращивать ее функциональность.

Предполагается, что после изучения этой главы пользователь сможет самостоятельно реализовать аналогичную систему и расширить ее с помощью подключения к онлайновым и сетевым сервисам. Также рекомендуется внимательно изучить реализации более продвинутых пользовательских интерфейсов, управляемых голосом, освоить добавление механизма преобразования текста в речь и практическое использование Bluetooth-устройств на основе A2DP.

В следующей главе будет рассматриваться реализация системы мониторинга и управления микроклиматом в здании с использованием микроконтроллеров и локальной сети.

Глава 9

Пример: мониторинг и управление внутренним микроклиматом в здании

Постоянное наблюдение за условиями окружающей среды в здании, включая температуру, влажность и уровень содержания углекислого газа CO_2, становится все более распространенной функцией, целью которой является регулирование систем отопления, охлаждения и вентиляции, чтобы обеспечить максимальный комфорт всем присутствующим в этом здании. В этой главе будет подробно описана и реализована такая система. Рассматриваются следующие темы:

- создание комплексного специализированного ПО для микроконтроллера ESP8266;
- интеграция микроконтроллера в сеть на основе протокола IP;
- добавление сенсорных датчиков определения уровня углекислого газа, работающих с шиной I2C;
- использование интерфейса GPIO и устройств широтно-импульсной модуляции (PWM) для управления реле и вентиляторов постоянного тока;
- установление соединений с сетевыми узлами с использованием центрального контроллера.

Растения, помещения и прочее

В главе 5 рассматривалась разработка специализированного ПО для микроконтроллера ESP8266, образующего систему в совокупности с сенсорным датчиком влажности почвы и насосом. Эта система обеспечивала снабжение наблюдаемого растения достаточным количеством воды из резервуара.

В главе 5 также отмечалось, что созданное специализированное ПО обеспечивает высокую степень модульности и предоставляет весьма гибкий интерфейс на основе протокола MQTT, поэтому может использоваться для широкого спектра разнообразных модулей. В этой главе подробно рассматривается система, для которой начиналась разработка специализированного ПО: система контроля и управления микроклиматом в здании (Building Management and Control – BMaC), изначально созданная только для наблюдения (мониторинга) за температурой,

влажностью и уровнем углекислого газа в помещениях. В дальнейшем система была усовершенствована для отслеживания режима работы автоматов приготовления кофе и присутствия людей в комнатах для совещаний. Потом была добавлена функция управления кондиционированием воздуха во всем здании.

Текущее состояние проекта BMaC можно узнать в соответствующем репозитории автора на GitHub **https://github.com/MayaPosch/BMaC**. Версия, рассматриваемая в этой главе, является последней на момент написания книги, и мы рассмотрим, как появилась эта система, как она выглядит сейчас и почему реализована именно так.

История разработки

Разработка проекта BMaC началась, когда в офисном здании планировалось добавление сенсорных датчиков для измерения температуры и других параметров, таких как относительная влажность воздуха. После принятия решения об использовании микроконтроллеров ESP8266 вместе с сенсорными датчиками температуры и влажности DHT22 был собран простой прототип с использованием базовой версии специализированного ПО, написанного с применением рабочей среды Sming.

Выяснилось, что сенсоры DHT22 относительно громоздки и не слишком точны. Кроме того, на используемых макетных платах были смонтированы не соответствующие сопротивления, из-за чего показания температуры были неверными. К тому же крупным недостатком этого типа сенсоров являлось использование собственного однопроводного протокола (единственная линия связи) вместо стандартного интерфейса.

Сенсоры DHT22 были заменены на сенсорные датчики BME280 MEMS, измеряющие температуру, влажность и атмосферное давление. Также был добавлен сенсорный датчик определения концентрации углекислого газа MH-Z19. Разумеется, для поддержки этих изменений и дополнений потребовалась новая версия специализированного ПО. Получаемые показания сенсоров должны были передаваться как сообщения по протоколу MQTT с внутренней организацией службы подписки на соответствующие темы и с записью этих сообщений (показаний) в базу данных временны́х последовательностей (InfluxDB) для просмотра и анализа.

Далее рассматривались предполагаемые решения, обеспечивающие возможность считывания контрольных счетчиков продукта из полностью автоматических устройств для приготовления кофе Jura, в частности – необходима ли для этого разработка отдельного специализированного ПО.

Вместо варианта разработки отдельной программы было принято решение об использовании единого комплекта специализированного ПО для всех узлов ESP8266. Это означало обеспечение функциональности, предоставляющей специфические возможности и свойства, а также обеспечение поддержки специализированных сенсорных датчиков и прочих функциональных объектов. Результатом стала разработка нового специализированного ПО, позволяющего выполнять удаленные команды, передаваемые по протоколу MQTT, для включения и отключения различных функциональных модулей в совокупности с другими управляющими функциями.

В дополнение к новому специализированному ПО был добавлен сервер контроля и управления (command and control – C&C), используемый отдельными узлами для извлечения их конфигурации. Кроме того, на сервере использовалось административное приложение, обеспечивающее добавление новых узлов и добавление или редактирование конфигурации узла.

После ввода созданной рабочей среды в эксплуатацию появилась возможность быстрого добавления новых функциональных возможностей, например добавление датчиков движения для обнаружения присутствия людей в помещении. В конечном итоге была добавлена функция управления устройствами кондиционирования воздуха, так как существующая централизованная система управления кондиционированием для всего здания была признана не соответствующей требованиям.

Общая схема всей созданной системы показана на рис. 9.1.

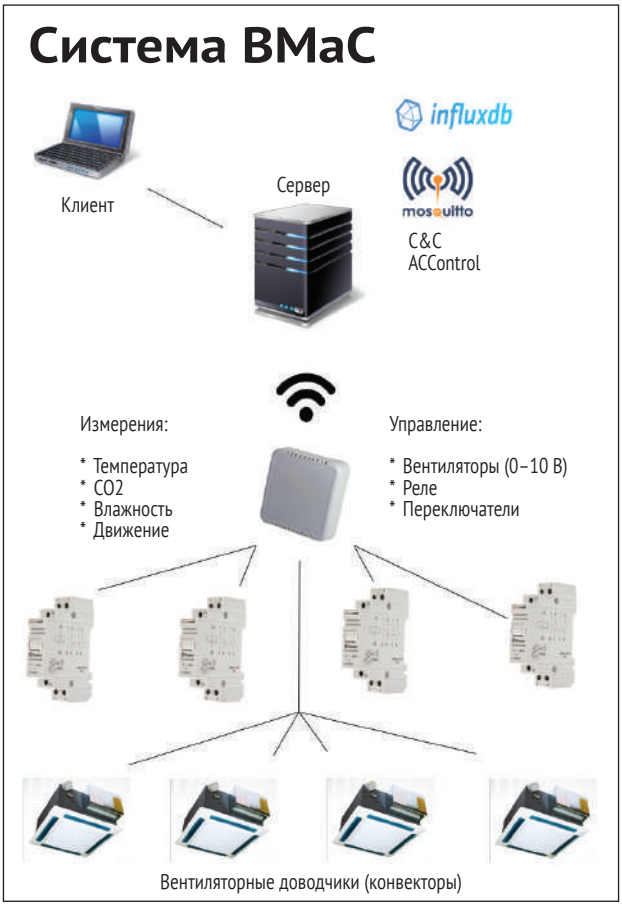

Рис. 9.1

В следующих разделах будут подробно рассматриваться все компоненты и особенности этой системы.

Функциональные модули

В табл. 9.1 приведен список модулей специализированного ПО для системы BMaC.

Таблица 9.1

Имя	Функции	Описание
THP	Температура, влажность, давление	Центральный класс для сенсорных датчиков THP. По умолчанию обеспечивает функциональность сенсора BME280
CO2	Концентрация CO_2	Измеряет концентрацию углекислого газа: значения <indexentry content="functional modules, Building Management and Control (BMaC):CO_2"> принимаются с MH-Z19 или с совместимого сенсорного датчика
Jura	Счетчики TopTronics EEPROM	Считывает счетчики расхода различных продуктов из EEPROM
JuraTerm	Устройства удаленного управления TopTronics	Обеспечивает удаленное обслуживание для передачи команд TopTronics (классический стиль v5) для поддерживаемых автоматов по приготовлению кофе
Motion	Обнаружение движения	Использует сенсорный датчик движения HC-SR501 PIR или совместимый сенсор для обнаружения движения
PWM	Вывод PWM	Настройка вывода широтно-импульсной модуляции (PWM) на одном или нескольких контактах
I/O	Расширение ввода/вывода	Поддержка восьмиканального модуля расширения ввода/вывода MCP23008 через шину I2C
Switch	Постоянно действующий переключатель	Управление переключателем, который использует реле с защелкой (с запоминанием) или равнозначное устройство для переключения (коммутации)
Plant	Полив растений	Считывает показания аналогового сенсорного датчика состояния почвы для определения ее влажности и при необходимости активизирует водяной насос

Исходный код специализированного ПО

В этом разделе подробно рассматривается исходный код специализированного ПО для микроконтроллера ESP8266, используемого в системе BMaC.

Ядро

Ядро этого специализированного ПО уже рассматривалось в главе 5 в примере мониторинга влажности почвы с применением протокола Wi-Fi, включая точку входа – класс OtaCore и класс BaseModule, предоставляющий всю функциональность, необходимую для инициализации отдельных модулей, и позволяющий подключать и отключать их с использованием интерфейса MQTT.

Модули

Один из модулей специализированного ПО – модуль водоснабжения растений – уже был подробно описан в главе 5. Здесь мы рассматриваем остальные модули, начиная с THP.

```
#include "base_module.h"

class THPModule {
    public:
    static bool initialize();
    static bool start();
    static bool shutdown();
};

#include "thp_module.h"
#include "dht_module.h"
#include "bme280_module.h"

bool THPModule::initialize() {
    BaseModule::registerModule(MOD_IDX_TEMPERATURE_HUMIDITY, THPModule::start,
                               THPModule::shutdown);
    return true;
}

bool THPModule::start() {
    BME280Module::init();
    return true;
}

bool THPModule::shutdown() {
    BME280Module::shutdown();
    return true;
}
```

Этот модуль предоставляет обобщенный интерфейс для широкого разнообразия сенсорных датчиков температуры, влажности и атмосферного давления. Во время первоначального варианта разработки модуль не требовался, поэтому функционировал как промежуточное звено для модуля BME280. Модуль регистрируется в базе модулей при вызове и обращается к соответствующим функциям модуля BME280, когда они вызываются.

Для обеспечения большей гибкости класс должен быть расширен так, чтобы позволять принимать команды, возможно, через интерфейс MQTT в собственной теме подписки. Затем эти команды должны подключать модуль конкретного заданного сенсорного датчика или даже группу таких сенсоров, например при использовании отдельных датчиков температуры и атмосферного давления.

Вне зависимости от использования в данной версии специализированного ПО рассмотрим модуль DHT, чтобы можно было в дальнейшем сравнить его с модулем BME280.

```
#include "ota_core.h"

#include <Libraries/DHTesp/DHTesp.h>

#define DHT_PIN 5 // Сенсор DHT: GPIO5 ('D1' on NodeMCU)

class DHTModule {
    static DHTesp* dht;
    static int dhtPin;
    static Timer dhtTimer;

 public:
    static bool init();
    static bool shutdown();
    static void config(String cmd);
    static void readDHT();
};
```

Несмотря на то что этот класс является статическим, все переменные, потребляющие значительный объем памяти, такие как экземпляры библиотечного класса, определены как указатели. Это создает компромисс между доступностью модуля для упрощения его использования и переходом к более сложному, полностью динамическому решению. Поскольку большинство микроконтроллеров способно хранить объем программного кода, который вмещается в используемое ПЗУ, необходимо обеспечить минимальное использование ПЗУ и статического ОЗУ.

```
#include "dht_module.h"

DHTesp* DHTModule::dht = 0;
int DHTModule::dhtPin = DHT_PIN;
Timer DHTModule::dhtTimer;

bool DHTModule::init() {
    if (!OtaCore::claimPin(dhtPin)) { return false; }
    if (!dht) { dht = new DHTesp(); dht->setup(dhtPin, DHTesp::DHT22); }
    dhtTimer.initializeMs(2000, DHTModule::readDHT).start();
    return true;
}
```

Для инициализации этого модуля обеспечивается возможность безопасного использования контактов интерфейса ввода/вывода общего назначения (general-purpose input/output – GPIO), создание нового экземпляра класса сенсорного датчика из библиотеки и его настройка перед созданием 2-секундного таймера, который будет регулировать запланированное считывание этого сенсорного датчика.

Поскольку новый экземпляр класса сенсорного датчика создается при инициализации, он никогда не должен быть уже существующим, тем не менее этот случай проверяется при повторном вызове функции `init()` по каким-либо причинам. Обращение к функции инициализации в таймере во второй раз также можно было бы включить в этот блок, но это не строгое требование, так как повторная инициализация таймера не приводит к какому-либо отрицательному эффекту.

```
bool DHTModule::shutdown() {
    dhtTimer.stop();
    if (!OtaCore::releasePin((ESP8266_pins) dhtPin)) { delete dht; return false; }
```

```
    delete dht;
    dht = 0;
    return true;
}
```

Для завершения работы модуля таймер останавливается, и освобождаются все используемые контакты GPIO. После этого освобождаются все используемые ресурсы. Так как используемый контакт был объявлен ранее при инициализации модуля, не должно возникать никаких проблем при его освобождении, но мы все же выполняем проверку.

```
void DHTModule::config(String cmd) {
    Vector<String> output;
    int numToken = splitString(cmd, '=', output);
    if (output[0] == "set_pin" && numToken > 1) {
          dhtPin = output[1].toInt();
    }
}
```

Это пример возможного последующего изменения контакта GPIO, используемого модулем. Здесь применяется старый текстовый формат команд из ранних версий специализированного ПО для системы BMaC. Эту информацию также можно принимать через тему подписки MQTT или при помощи активного запроса команды и управляющего сервера.

Отметим, что для изменения контакта, используемого сенсорным датчиком, необходимо перезапустить сенсор, удалив экземпляр этого класса и создав новый экземпляр.

```
void DHTModule::readDHT() {
    TempAndHumidity th;
    th = dht->getTempAndHumidity();

    OtaCore::publish("nsa/temperature", OtaCore::getLocation() + ";" + th.temperature);
    OtaCore::publish("nsa/humidity", OtaCore::getLocation() + ";" + th.humidity);
}
```

Далее приведен исходный код для модуля сенсорного датчика BME280.

```
#include "ota_core.h"

#include <Libraries/BME280/BME280.h>

class BME280Module {
    static BME280* bme280;
    static Timer timer;

 public:
    static bool init();
    static bool shutdown();
    static void config(String cmd);
    static void readSensor();
};
```

Реализация класса модуля, которая выглядит уже знакомой.

```
#include "bme280_module.h"

BME280* BME280Module::bme280 = 0;
```

```
Timer BME280Module::timer;

bool BME280Module::init() {
    if (!OtaCore::starti2c()) { return false; }
    if (!bme280) { bme280 = new BME280(); }

    if (bme280->EnsureConnected()) {
          OtaCore::log(LOG_INFO, "Connected to BME280 sensor.");
          bme280->SoftReset();
          bme280->Initialize();
    }
    else {
          OtaCore::log(LOG_ERROR, "Not connected to BME280 sensor.");
          return false;
    }

    timer.initializeMs(2000, BME280Module::readSensor).start();

    return true;
}

bool BME280Module::shutdown() {
    timer.stop();
    delete bme280;
    bme280 = 0;
    return true;
}

void BME280Module::config(String cmd) {
    Vector<String> output;
    int numToken = splitString(cmd, '=', output);
    if (output[0] == "set_pin" && numToken > 1) {
          //
    }
}

void BME280Module::readSensor() {
    float t, h, p;
    if (bme280->IsConnected) {
          t = bme280->GetTemperature();
          h = bme280->GetHumidity();
          p = bme280->GetPressure
          OtaCore::publish("nsa/temperature", OtaCore::getLocation() + ";" + t);
          OtaCore::publish("nsa/humidity", OtaCore::getLocation() + ";" + h);
          OtaCore::publish("nsa/pressure", OtaCore::getLocation() + ";" + p);
    }
    else {
          OtaCore::log(LOG_ERROR, "Disconnected from BME280 sensor.");
    }
}
```

Здесь можно видеть, что код этого модуля в основном скопирован из модуля DHT, а затем отредактирован для соответствия сенсорному датчику BME280. Схожесть этих двух модулей стала одной из причин принятия решения о разработке модуля THP с целью рационального использования одинаковых свойств и характеристик.

Как и в модуле DHT, здесь используется внешняя библиотека, выполняющая всю самую трудную работу, а нам остается лишь вызывать функции из библиотечного класса для настройки сенсорного датчика и получения от него данных.

Модуль CO2

Для модуля CO2 пока еще не было выполнено попыток сделать его универсальным, то есть работающим с несколькими типами сенсорных датчиков определения концентрации углекислого газа. Сначала использовался сенсорный датчик MH-Z14, затем он был заменен на более компактный сенсор MH-Z19. Но обе эти модели использовали один и тот же протокол в своем интерфейсе универсального асинхронного приемопередатчика (universal asynchronous receiver/transmitter – UART).

В микроконтроллере ESP8266 имеется два интерфейса UART, но только один из них полнофункциональный с линиями приема (RX) и передачи (TX). Второй интерфейс UART содержит только линию передачи TX. По существу, возможности этого микроконтроллера по использованию UART ограничены лишь одним интерфейсом, следовательно, можно применить только один сенсорный датчик на основе UART.

Эти сенсоры также имеют однопроводной интерфейс (одна линия) в дополнение к интерфейсу UART. По этой линии сенсор выводит текущие считываемые данные в специальной кодировке, которая должна быть принята и декодирована с использованием специфического интервала между импульсами в этой единственной проводной линии. Это похоже на применение однопроводного протокола сенсора DHT22.

Разумеется, использование интерфейса UART значительно проще, поэтому именно он применяется в этом модуле.

```
#include "base_module.h"

class CO2Module {
    static Timer timer;
    static uint8_t readCmd[9];
    static uint8 eventLevel;
    static uint8 eventCountDown;
    static uint8 eventCountUp;
    static void onSerialReceived(Stream &stream, char arrivedChar,
                                 unsigned short availableCharsCount);
 public:
    static bool initialize();
    static bool start();
    static bool shutdown();
    static void readCO2();
    static void config(String cmd);
};
```

Здесь можно видеть функцию обратного вызова, которая будет использоваться для работы с интерфейсом UART во время приема данных. Также объявлено несколько переменных, смысл которых будет объяснен немного позже.

```
#include "CO2_module.h"

Timer CO2Module::timer;
```

```
uint8_t CO2Module::readCmd[9] = {0xFF,0x01,0x86,0x00,0x00,0x00,0x00,0x00,0x79};
uint8 CO2Module::eventLevel = 0;
uint8 CO2Module::eventCountDown = 10;
uint8 CO2Module::eventCountUp = 0;
```

В статических инициализациях определена команда, которая будет передаваться в сенсорный датчик концентрации углекислого газа. По этой команде сенсор передает текущее измеренное значение. Здесь же определяется группа счетчиков и связанный с ними экземпляр таймера, который будет использоваться для анализа полученных данных об уровнях концентрации углекислого газа.

```
bool CO2Module::initialize() {
    BaseModule::registerModule(MOD_IDX_CO2, CO2Module::start, CO2Module::shutdown);
    return true;
}

bool CO2Module::start() {
    if (!OtaCore::claimPin(ESP8266_gpio03)) { return false; }
    if (!OtaCore::claimPin(ESP8266_gpio01)) { return false; }

    Serial.end();
    delay(10);
    Serial.begin(9600);
    Serial.setCallback(&CO2Module::onSerialReceived);

    timer.initializeMs(30000, CO2Module::readCO2).start();
    return true;
}
```

При запуске этого модуля инициализируется регистрация контактов, необходимых для работы с UART. Интерфейс UART начинает работу со скоростью 9600 бод. Также регистрируется функция обратного вызова для приема данных. Подпрограмма регистрации контактов в классе ядра предназначена для служебного использования, следовательно, в действительности не может привести к критическому сбою. В случае конфликта, связанного с распределением контакта для другого модуля, может потребоваться освобождение от регистрации первого контакта, если для второго процедура регистрации закончилась неудачно.

Контакты интерфейса GPIO, используемые последовательным интерфейсом, настраиваются в том же классе ядра и должны изменяться там же. Главная причина такого подхода – недостаточная конфигурабельность, поскольку контакты GPIO на микроконтроллере ESP8266 ограничены исключительно теми функциями, которые они действительно поддерживают. Именно поэтому аппаратный интерфейс UART всегда связан с этими двумя конкретными контактами, игнорируя все прочие контакты с другой функциональностью.

Запускаемый таймер регулирует считывание данных с сенсорного датчика через каждые 30 секунд с учетом того, что в течение первых 3 минут данные с сенсора бесполезны, так как этот интервал необходим сенсору для разогрева.

```
bool CO2Module::shutdown() {
    if (!OtaCore::releasePin(ESP8266_gpio03)) { return false; }
    if (!OtaCore::releasePin(ESP8266_gpio01)) { return false; }

    timer.stop();
```

```
    Serial.end();
    return true;
}

void CO2Module::readCO2() {
    Serial.write(readCmd, 9);
}
```

Считывание данных с сенсорного датчика выполняется так же просто, как запись последовательности байтов, которая была определена на этапе статической инициализации сенсора, и последующее ожидание ответа сенсора в виде данных, возвращаемых в буфер RX и активизирующих функцию обратного вызова.

```
void CO2Module::config(String cmd) {
    Vector<String> output;
    int numToken = splitString(cmd, '=', output);
    if (output[0] == "event" && numToken > 1) {
          //
    }
}
```

Здесь также не реализован метод конфигурирования, но его можно было бы использовать для запрещения событий (описываемых в следующем подразделе) и динамического редактирования и внесения изменений и усовершенствований.

```
void CO2Module::onSerialReceived(Stream &stream, char arrivedChar,
                                 unsigned short availableCharsCount)
{
    if (availableCharsCount >= 9) {
          char buff[9];
          Serial.readBytes(buff, 9);

          int responseHigh = (int) buff[2];
          int responseLow = (int) buff[3];
          int ppm = (responseHigh * 0xFF) + responseLow;
          String response = OtaCore::getLocation() + ";" + ppm;
          OtaCore::publish("nsa/CO2", response);

          if (ppm > 1000) { // T3
                if (eventLevel < 2 && eventCountUp < 10) {
                      if (++eventCountUp == 10) {
                            eventLevel = 2;
                            eventCountDown = 0;
                            eventCountUp = 0;
                            response = OtaCore::getLocation() + ";" + eventLevel + ";1;" + ppm;
                            OtaCore::publish("nsa/events/CO2", response);
                      }
                }
          }
          else if (ppm > 850) { // T2
                if (eventLevel == 0 && eventCountUp < 10) {
                      if (++eventCountUp == 10) {
                            eventLevel = 1;
                            eventCountDown = 0;
                            eventCountUp = 0;
```

```
                        response = OtaCore::getLocation() + ";" + eventLevel + ";1;" + ppm;
                        OtaCore::publish("nsa/events/CO2", response);
                    }
                }
                else if (eventLevel == 2 && eventCountDown < 10) {
                    if (++eventCountDown == 10) {
                        eventLevel = 1;
                        eventCountUp = 0;
                        eventCountDown = 0;
                        response = OtaCore::getLocation() + ";" + eventLevel + ";0;" + ppm;
                        OtaCore::publish("nsa/events/CO2", response);
                    }
                }
            }
            else if (ppm < 750) { // T1
                if (eventLevel == 1 && eventCountDown < 10) {
                    if (++eventCountDown == 10) {
                        eventLevel = 0;
                        eventCountDown = 0;
                        eventCountUp = 0;
                        response = OtaCore::getLocation() + ";" + eventLevel + ";0;" + ppm;
                        OtaCore::publish("nsa/events/CO2", response);
                    }
                }
            }
    }
}
```

В этой функции обратного вызова мы получаем символы по мере их поступления по линии приема RX. Организуется интервал ожидания до тех пор, пока в буфере RX не накопится девять символов, то есть количество байтов, которое предполагается принять от сенсорного датчика, следящего за концентрацией углекислого газа. Также можно выполнить проверку контрольной суммы для принятых данных, для которых спецификация MH-Z19 рекомендует использовать следующий код на языке C:

```
char getCheckSum(char* packet) {
    char i, checksum;
    for ( i = 1; i < 8; i++) {
        checksum += packet[i];
    }

    checksum = 0xff - checksum;
    checksum += 1;
    return checksum;
}
```

Эта подпрограмма вычисляет контрольную сумму для принятых данных в форме одного байта, который в дальнейшем можно сравнить со значением, переданным в девятом байте, и убедиться, что эти значения совпадают.

Вернемся к нашему коду: далее выполняется обработка принятых байтов для вычисления количества частиц на миллион (parts per million – PPM), то есть числа

молекул CO_2, обнаруженных сенсорным датчиком. Это значение сразу же публикуется в соответствующей теме MQTT.

После этого сравнивается новое значение PPM, чтобы узнать, не перешло ли оно через одну из трех границ предварительно установленных уровней срабатывания. Первая граница обозначает безопасный уровень содержания углекислого газа, вторая граница – повышенный уровень содержания углекислого газа, третья граница – чрезвычайно высокий уровень содержания углекислого газа, который требует особого внимания. Если была пересечена граница уровней в том или ином направлении, то это событие публикуется в соответствующей теме MQTT.

Модуль Jura

Это еще один модуль, использующий UART. Он применяется для управления несколькими автоматами приготовления кофе Jura на основе электронных устройств общего назначения TopTronics, которые широко используются и другими производителями кофе-автоматов. Для обеспечения считывания данных с этих кофе-автоматов модуль микроконтроллера ESP8266 был помещен в небольшой пластиковый корпус, имеющий на одной из своих сторон только один разъем для последовательного соединения. Подключение выполняется с помощью стандартного последовательного кабеля с девятью контактами, соединяемого с так называемым сервисным портом, расположенным на задней панели кофе-автомата.

Последовательный порт кофе-автомата после включения электропитания дает напряжение 5 В, которое позволяет также включить и присоединенный микроконтроллер ESP8266. Пластиковый корпус с микроконтроллером можно скрыть за кофе-автоматом.

Модуль для обеспечения этой функции реализован следующим образом:

```
#include "base_module.h"

class JuraModule {
    static String mqttTxBuffer;
    static Timer timer;

    static bool toCoffeemaker(String cmd);
    static void readStatistics();
    static void onSerialReceived(Stream &stream, char arrivedChar,
                                 unsigned short availableCharsCount);
 public:
    static bool initialize();
    static bool start();
    static bool shutdown();
};
```

Здесь следует особо отметить, что при объявлении этого класса применено включение наименования кофе-автомата в имя класса. Далее рассмотрим его реализацию.

```
#include "jura_module.h"
#include <stdlib.h>

Timer JuraModule::timer;
String JuraModule::mqttTxBuffer;
```

```
bool JuraModule::initialize() {
    BaseModule::registerModule(MOD_IDX_JURA, JuraModule::start, JuraModule::shutdown);
}

bool JuraModule::start() {
    if (!OtaCore::claimPin(ESP8266_gpio03)) { return false; }
    if (!OtaCore::claimPin(ESP8266_gpio01)) { return false; }
    Serial.end();
    delay(10);
    Serial.begin(9600);
    Serial.setCallback(&JuraModule::onSerialReceived);
    timer.initializeMs(60000, JuraModule::readStatistics).start();
    return true;
}
```

По умолчанию интерфейс UART кофе-автомата работает со скоростью 9600 бод. Выполняется настройка метода последовательного обратного вызова, запускается таймер, управляющий чтением устройств памяти EEPROM в счетчиках расхода продуктов. Поскольку речь идет о кофе-автоматах, процедура считывания этих счетчиков более одного раза в минуту реализована достаточно примитивно.

```
bool JuraModule::shutdown() {
    if (!OtaCore::releasePin(ESP8266_gpio03)) { return false; } // RX 0
    if (!OtaCore::releasePin(ESP8266_gpio01)) { return false; } // TX 0
    timer.stop();
    Serial.end();
    return true;
}

void JuraModule::readStatistics() {
    String message = "RT:0000";
    JuraModule::toCoffeemaker(message);
}
```

Для считывания счетчиков EEPROM необходимо отправить соответствующую команду в интерфейс UART кофе-автомата. Эта команда предписывает передать содержимое первой «строки» EEPROM. К сожалению, протокол кофе-автомата не использует обычный текст, а требует специального кодирования, поэтому необходим приведенный ниже метод.

```
bool JuraModule::toCoffeemaker(String cmd) {
    OtaCore::log(LOG_DEBUG, "Sending command: " + cmd);
    cmd += "\r\n";
    for (int i = 0; i < cmd.length(); ++i) {
          uint8_t ch = static_cast<uint8_t>(cmd[i]);
          uint8_t d0 = 0xFF;
          uint8_t d1 = 0xFF;
          uint8_t d2 = 0xFF;
          uint8_t d3 = 0xFF;
          bitWrite(d0, 2, bitRead(ch, 0));
          bitWrite(d0, 5, bitRead(ch, 1));
          bitWrite(d1, 2, bitRead(ch, 2));
          bitWrite(d1, 5, bitRead(ch, 3));
          bitWrite(d2, 2, bitRead(ch, 4));
```

```
            bitWrite(d2, 5, bitRead(ch, 5));
            bitWrite(d3, 2, bitRead(ch, 6));
            bitWrite(d3, 5, bitRead(ch, 7));
            delay(1);
            Serial.write(d0);
            delay(1);
            Serial.write(d1);
            delay(1);
            Serial.write(d2);
            delay(1);
            Serial.write(d3);
            delay(7);
    }
    return true;
}
```

Метод принимает строку, добавляет требуемые символы конца строки и кодирует каждый байт в четыре байта, помещая отдельные биты данных в каждый второй и пятый бит нового байта, а все остальные биты устанавливаются в 1. Затем эти четыре сформированных байта передаются в интерфейс UART кофе-автомата с небольшой задержкой между передачей каждого байта, чтобы обеспечить корректный прием команды.

```
void JuraModule::onSerialReceived(Stream &stream, char arrivedChar,
                                  unsigned short availableCharsCount) {
    OtaCore::log(LOG_DEBUG, "Receiving UART 0.");

    while(stream.available()){
        delay(1);
        uint8_t d0 = stream.read();
        delay(1);
        uint8_t d1 = stream.read();
        delay(1);
        uint8_t d2 = stream.read();
        delay(1);
        uint8_t d3 = stream.read();
        delay(7);

        uint8_t d4;
        bitWrite(d4, 0, bitRead(d0, 2));
        bitWrite(d4, 1, bitRead(d0, 5));
        bitWrite(d4, 2, bitRead(d1, 2));
        bitWrite(d4, 3, bitRead(d1, 5));
        bitWrite(d4, 4, bitRead(d2, 2));
        bitWrite(d4, 5, bitRead(d2, 5));
        bitWrite(d4, 6, bitRead(d3, 2));
        bitWrite(d4, 7, bitRead(d3, 5));
        OtaCore::log(LOG_TRACE, String(d4));
        mqttTxBuffer += (char) d4;

        if ('\n' == (char) d4) {
            long int espressoCount = strtol(mqttTxBuffer.substring(3, 7).c_str(), 0, 16);
            long int espresso2Count = strtol(mqttTxBuffer.substring(7, 11).c_str(), 0, 16);
            long int coffeeCount = strtol(mqttTxBuffer.substring(11, 15).c_str(), 0, 16);
```

```
            long int coffee2Count = strtol(mqttTxBuffer.substring(15, 19).c_str(), 0, 16);
            OtaCore::publish("nsa/espresso", OtaCore::getLocation() + ";" + espressoCount);
            OtaCore::publish("nsa/espresso2", OtaCore::getLocation() + ";" + espresso2Count);
            OtaCore::publish("nsa/coffee", OtaCore::getLocation() + ";" + coffeeCount);
            OtaCore::publish("nsa/coffee2", OtaCore::getLocation() + ";" + coffee2Count);
            mqttTxBuffer = "";
          }
    }
}
```

В последовательном обратном вызове приема декодируется каждый принятый байт с использованием той же процедуры, которая применялась для кодирования данных, отправляемых в кофе-автомат, с буферизацией декодированных байтов до тех пор, пока не будет обнаружен признак конца ответа – символ перехода на новую строку (LF – linefeed). После этого считываются сформированные 16-битовые счетчики и публикуются в соответствующей теме подписки MQTT.

Модуль JuraTerm

Модуль JuraTerm похож на модуль Jura, но принимает удаленные команды, кодирует их тем же способом, что и модуль Jura, и возвращает декодированный ответ. В рассматриваемом проекте этот модуль использовал класс `Jura`, пока тот не был заменен на новый класс `Jura`. После замены JuraTerm делегирован исключительно как завершающий класс в иерархии. Таким образом, в следующей версии планируется объединить функциональность этого модуля с основным классом `Jura`.

```
#include "base_module.h"

class JuraTermModule
{
    static String mqttTxBuffer;
    static bool toCoffeemaker(String cmd);
    static void onSerialReceived(Stream &stream, char arrivedChar,
                                 unsigned short availableCharsCount);
 public:
    static bool initialize();
    static bool start();
    static bool shutdown();
    static void commandCallback(String message);
 };

#include "juraterm_module.h"

String JuraTermModule::mqttTxBuffer;

bool JuraTermModule::initialize() {
    BaseModule::registerModule(MOD_IDX_JURATERM, JuraTermModule::start,
                               JuraTermModule::shutdown);
}

bool JuraTermModule::start() {
    if (!OtaCore::claimPin(ESP8266_gpio03)) { return false; } // RX 0
    if (!OtaCore::claimPin(ESP8266_gpio01)) { return false; } // TX 0
    OtaCore::registerTopic("coffee/command/" + OtaCore::getLocation(),
                           JuraTermModule::commandCallback);
```

```
    Serial.end();
    delay(10);
    Serial.begin(9600);
    Serial.setCallback(&JuraTermModule::onSerialReceived);

    return true;
}

bool JuraTermModule::shutdown() {
    if (!OtaCore::releasePin(ESP8266_gpio03)) { return false; } // RX 0
    if (!OtaCore::releasePin(ESP8266_gpio01)) { return false; } // TX 0

    Serial.end();
    OtaCore::deregisterTopic("coffee/command/" + OtaCore::getLocation());
    return true;
}

void JuraTermModule::commandCallback(String message) {
    if (message == "AN:0A") { return; }
    JuraTermModule::toCoffeemaker(message);
}
```

При инициализации этого модуля регистрируется соответствующая тема MQTT для приема команд. Это позволяет получать команды кофе-автомата. Модуль в основном действует как «пропускной пункт» для этих команд, за исключением одной особой команды. Эта выделяемая команда предназначена для стирания устройства памяти EEPROM кофе-автомата, то есть ее выполнение чрезвычайно нежелательно.

Здесь также используется описанный выше метод кодирования команд.

```
bool JuraTermModule::toCoffeemaker(String cmd) {
    OtaCore::log(LOG_DEBUG, "Sending command: " + cmd);

    cmd += "\r\n";

    for (int i = 0; i < cmd.length(); ++i) {
          uint8_t ch = static_cast<uint8_t>(cmd[i]);
          uint8_t d0 = 0xFF;
          uint8_t d1 = 0xFF;
          uint8_t d2 = 0xFF;
          uint8_t d3 = 0xFF;
          bitWrite(d0, 2, bitRead(ch, 0));
          bitWrite(d0, 5, bitRead(ch, 1));
          bitWrite(d1, 2, bitRead(ch, 2));
          bitWrite(d1, 5, bitRead(ch, 3));
          bitWrite(d2, 2, bitRead(ch, 4));
          bitWrite(d2, 5, bitRead(ch, 5));
          bitWrite(d3, 2, bitRead(ch, 6));
          bitWrite(d3, 5, bitRead(ch, 7));

          delay(1);
          Serial.write(d0);
          delay(1);
          Serial.write(d1);
          delay(1);
          Serial.write(d2);
```

```
            delay(1);
            Serial.write(d3);
            delay(7);
        }
        return true;
}

void JuraTermModule::onSerialReceived(Stream &stream, char arrivedChar,
                                      unsigned short availableCharsCount) {
        OtaCore::log(LOG_DEBUG, "Receiving UART 0.");

        while(stream.available()){
            delay(1);
            uint8_t d0 = stream.read();
            delay(1);
            uint8_t d1 = stream.read();
            delay(1);
            uint8_t d2 = stream.read();
            delay(1);
            uint8_t d3 = stream.read();
            delay(7);
            uint8_t d4;
            bitWrite(d4, 0, bitRead(d0, 2));
            bitWrite(d4, 1, bitRead(d0, 5));
            bitWrite(d4, 2, bitRead(d1, 2));
            bitWrite(d4, 3, bitRead(d1, 5));
            bitWrite(d4, 4, bitRead(d2, 2));
            bitWrite(d4, 5, bitRead(d2, 5));
            bitWrite(d4, 6, bitRead(d3, 2));
            bitWrite(d4, 7, bitRead(d3, 5));

            OtaCore::log(LOG_TRACE, String(d4));

            mqttTxBuffer += (char) d4;
            if ('\n' == (char) d4) {
                OtaCore::publish("coffee/response", OtaCore::getLocation() + ";" +
                                 mqttTxBuffer);
                mqttTxBuffer = "";
            }
        }
}
```

Вместо какой-либо интерпретации данных ответ просто возвращается в соответствующую тему подписки MQTT.

Модуль Motion

Модуль определения движения предназначен для работы с пассивными инфракрасными датчиками (passive infrared – PIR – sensors). В эти датчики встроена логика, позволяющая определять достижение порогового значения срабатывания. В этот момент изменяется состояние контакта прерывания, то есть генерируется мощный сигнал. Эту функциональность можно использовать для определения присутствия людей в помещении или перемещения людей по коридору.

Ниже приведен исходный код этого модуля.

```
#include "base_module.h"

#define GPIO_PIN 0

class MotionModule
{
    static int pin;
    static Timer timer;
    static Timer warmup;
    static bool motion;
    static bool firstLow;

 public:
    static bool initialize();
    static bool start();
    static bool shutdown();
    static void config(String cmd);
    static void warmupSensor();
    static void readSensor();
    static void IRAM_ATTR interruptHandler();
};
```

Здесь следует отметить явное перемещение метода обработки прерываний в статическое ОЗУ микроконтроллера с использованием ключевого слова `IRAM_ATTR` для предотвращения каких-либо задержек после генерации прерывания и вызова обработчика.

Реализация класса:

```
#include "motion_module.h"

int MotionModule::pin = GPIO_PIN;
Timer MotionModule::timer;
Timer MotionModule::warmup;
bool MotionModule::motion = false;
bool MotionModule::firstLow = true;

bool MotionModule::initialize() {
      BaseModule::registerModule(MOD_IDX_MOTION, MotionModule::start, MotionModule::shutdown);
}

bool MotionModule::start() {
    if (!OtaCore::claimPin(ESP8266_gpio00)) { return false; }
    pinMode(pin, INPUT);
    warmup.initializeMs(60000, MotionModule::warmupSensor).start();
   return true;
}
```

Пассивный инфракрасный датчик требует времени для разогрева и стабилизации считываемых данных. На это ему предоставляется одна минута, отсчитываемая по таймеру разогрева. Также устанавливается необходимый режим для используемого контакта интерфейса GPIO.

```
bool MotionModule::shutdown() {
    if (!OtaCore::releasePin(ESP8266_gpio00)) { return false; } // RX 0
```

```
    timer.stop();
    detachInterrupt(pin);

    return true;
}

void MotionModule::config(String cmd) {
    Vector<String> output;
    int numToken = splitString(cmd, '=', output);
    if (output[0] == "set_pin" && numToken > 1) {
          //
    }
}

void MotionModule::warmupSensor() {
    warmup.stop();
    attachInterrupt(pin, &MotionModule::interruptHandler, CHANGE);

    timer.initializeMs(5000, MotionModule::readSensor).start();
}
```

После завершения разогрева датчика таймер останавливается, и выполняется привязка прерывания для обработки любых сигналов от датчика. Проверяется переменная, совместно используемая с подпрограммой прерывания, чтобы обнаружить изменение ее значения. Текущее считанное значение публикуется каждые 5 секунд.

```
void MotionModule::readSensor() {
    if (!motion) {
          if (firstLow) { firstLow = false; }
          else {
                OtaCore::publish("nsa/motion", OtaCore::getLocation() + ";0");
                firstLow = true;
          }
    }
    else if (motion) {
          OtaCore::publish("nsa/motion", OtaCore::getLocation() + ";1");
          firstLow = true;
    }
}
```

При проверке текущего значения датчика создается ветвь игнорирования его первоначального состояния, которая сообщает об отсутствии сигналов (`LOW`). Это сделано для того, чтобы не фиксировать ситуацию, когда в помещении нет существенных перемещений людей. После этого получаемое значение публикуется в соответствующей теме MQTT.

```
void IRAM_ATTR MotionModule::interruptHandler() {
    int val = digitalRead(pin);
    if (val == HIGH) { motion = true; }
    else { motion = false; }
}
```

Обработчик прерываний явно обновляет локальное логическое значение. Из-за относительно длительного времени перехода между состояниями в большинстве

схем обработки пассивного инфракрасного датчика требуется некоторое время (несколько секунд), прежде чем датчик снова обнаружит движение, то есть создаются так называемые «мертвые зоны». Поэтому мы продолжаем отслеживать последнее зарегистрированное значение.

Модуль PWM

Причина разработки модуля поддержки импульсно-широтной модуляции (PWM) заключается в необходимости способа генерации аналогового вывода как величины напряжения с использованием внешней схемы фильтра RC. Это нужно для управления вентиляторами устройств кондиционирования воздуха, монтируемых на потолке, поскольку для контроллеров таких вентиляторов допускается напряжение от 0 до 10 В.

Интересной особенностью данного модуля является использование собственного бинарного протокола, поддерживающего удаленное управление. Таким образом, служба управления кондиционированием воздуха может напрямую управлять скоростью вентиляторов через узлы, также смонтированные на потолке.

```
#include "base_module.h"

#include <HardwarePWM.h>

class PwmModule
{
    static HardwarePWM* hw_pwm;
    static Vector<int> duty;
    static uint8 pinNum;
    static Timer timer;
    static uint8* pins;

 public:
    static bool initialize();
    static bool start();
    static bool shutdown();
    static void commandCallback(String message);
};
```

Реализация этого класса приведена ниже.

```
#include "pwm_module.h"

HardwarePWM* PwmModule::hw_pwm = 0;
uint8 PwmModule::pinNum = 0;
Timer PwmModule::timer;
uint8* PwmModule::pins = 0;

enum {
    PWM_START = 0x01,
    PWM_STOP = 0x02,
    PWM_SET_DUTY = 0x04,
    PWM_DUTY = 0x08,
    PWM_ACTIVE = 0x10
};
```

Здесь в форме перечисления определяются команды, доступные для модуля PWM.

```
bool PwmModule::initialize() {
    BaseModule::registerModule(MOD_IDX_PWM, PwmModule::start, PwmModule::shutdown);
}

bool PwmModule::start() {
    OtaCore::registerTopic(MQTT_PREFIX + String("pwm/") + OtaCore::getLocation(),
                           PwmModule::commandCallback);
    return true;
}

bool PwmModule::shutdown() {
    OtaCore::deregisterTopic(MQTT_PREFIX + String("pwm/") + OtaCore::getLocation());
    if (hw_pwm) {
          delete hw_pwm;
          hw_pwm = 0;
    }
    return true;
}
```

При инициализации этого модуля регистрируется соответствующая тема MQTT, в которой модуль сможет принимать команды. При завершении работы модуля выполняется отмена регистрации этой темы. Для обеспечения работы модуля PWM с отдельными контактами используется класс `HardwarePWM` из рабочей среды Sming.

Остальная часть модуля – это просто командный процессор (обработчик команд).

```
void PwmModule::commandCallback(String message) {
    OtaCore::log(LOG_DEBUG, "PWM command: " + message);
    if (message.length() < 1) { return; }
    int index = 0;
    uint8 cmd = *((uint8*) &message[index++]);

    if (cmd == PWM_START) {
          if (message.length() < 2) { return; }
          uint8 num = *((uint8*) &message[index++]);

          OtaCore::log(LOG_DEBUG, "Pins to add: " + String(num));

          if (message.length() != (2 + num)) { return; }

          pins = new uint8[num];
          for (int i = 0; i < num; ++i) {
                pins[i] = *((uint8*) &message[index++]);
                if (!OtaCore::claimPin(pins[i])) {
                      OtaCore::log(LOG_ERROR, "Pin is already in use: " + String(pins[i]));
                      OtaCore::publish("pwm/response", OtaCore::getLocation() + ";0", 1);
                      return;
                }

                OtaCore::log(LOG_INFO, "Adding GPIO pin " + String(pins[i]));
          }
          hw_pwm = new HardwarePWM(pins, num);
          pinNum = num;

          OtaCore::log(LOG_INFO, "Added pins to PWM: " + String(pinNum));

          OtaCore::publish("pwm/response", OtaCore::getLocation() + ";1", 1);
```

```
    }
    else if (cmd == PWM_STOP) {
          delete hw_pwm;
          hw_pwm = 0;

          for (int i = 0; i < pinNum; ++i) {
                if (!OtaCore::releasePin(pins[i])) {
                      OtaCore::log(LOG_ERROR, "Pin cannot be released: " + String(pins[i]));
                      OtaCore::publish("pwm/response", OtaCore::getLocation() + ";0", 1);
                      return;
                }

                OtaCore::log(LOG_INFO, "Removing GPIO pin " + String(pins[i]));
          }

          delete[] pins;
          pins = 0;

          OtaCore::publish("pwm/response", OtaCore::getLocation() + ";1");
    }
    else if (cmd == PWM_SET_DUTY) {
          if (message.length() < 3) { return; }

          uint8 pin = *((uint8*) &message[index++]);
          uint8 duty = *((uint8*) &message[index++]);
          bool ret = hw_pwm->setDuty(pin, ((uint32) 222.22 * duty));

          if (!ret) {
                OtaCore::publish("pwm/response", OtaCore::getLocation() + ";0");
                return;
          }
          OtaCore::publish("pwm/response", OtaCore::getLocation() + ";1");
    }
    else if (cmd == PWM_DUTY) {
          if (message.length() < 2) { return; }

          uint8 pin = *((uint8*) &message[index++]);
          uint32 duty = hw_pwm->getDuty(pin);

          uint8 dutyp = (duty / 222.22) + 1;
          String res = "";
          res += (char) pin;
          res += (char) dutyp;
          OtaCore::publish("pwm/response", OtaCore::getLocation() + ";" + res);
    }
    else if (cmd == PWM_ACTIVE) {
          String res;
          if (pins && pinNum > 0) {
                res = String((char*) pins, pinNum);
          }

          OtaCore::publish("pwm/response", OtaCore::getLocation() + ";" + res);
    }
}
```

Протокол, реализованный в приведенном выше методе, описан в табл. 9.2.

Таблица 9.2

Команда	Смысл	Полезная нагрузка	Возвращаемое значение
0x01	Запуск модуля	uint8 (количество контактов) uint8* (по одному байту на каждый номер контакта)	0x00/0x01
0x02	Останов модуля	–	0x00/0x01
0x04	Настройка уровня функциональных привилегий PWM	uint8 (номер контакта) uint8 (цикл функциональных обязанностей/привилегий, 0–100)	0x00/0x01
0x08	Считывание уровня функциональных привилегий PWM	uint8 (номер контакта)	uint8 (уровень функциональных привилегий)
0x10	Возврат активных контактов	–	uint8* (в каждом байте номер отдельного контакта)

Для каждой команды выполняется синтаксический разбор (парсинг) полученной строки байтов, проверяется количество байтов (должно соответствовать ожидаемому количеству), затем содержимое байтов интерпретируется как команды и полезная нагрузка (аргументы команд и ответы). Возвращается либо 0 (неудачное завершение), либо 1 (успешное завершение), либо полезная нагрузка с требуемой информацией.

Здесь можно было бы внести очевидное дополнение – проверку некоторого вида контрольной суммы принятой команды в совокупности с проверками корректности принимаемых данных. Приведенный выше код будет отлично работать в безопасной среде с зашифрованными каналами MQTT и надежными сетевыми соединениями, но другие реальные среды могут оказаться не столь лояльными, в них возможны повреждения, искажения и потери данных.

Модуль ввода/вывода

Иногда требуется настолько большое количество контактов интерфейса GPIO, соединенных с различными устройствами, такими как, например, реле, что приходится постоянно включать и отключать «дымящиеся клапаны». Именно это стало причиной появления рассматриваемого здесь модуля. Узлы, которые должны устанавливаться на потолке, содержат не только шину I2C, используемую для датчиков наблюдения за средой, но и интерфейс UART для датчиков концентрации углекислого газа, а также четыре контакта для вывода устройства импульсно-широтной модуляции (PWM).

Поскольку потребовалось большее число контактов интерфейса GPIO для подключения реле, управляющих клапанами системы подачи воды в устройства кондиционирования воздуха, на шину I2C была добавлена необходимая микросхема расширения GPIO с восемью дополнительными контактами.

Этот модуль обеспечивает поддержку дополнительного сервиса, например той же службы кондиционирования воздуха, с возможностью прямой установки для этих новых контактов GPIO высокого или низкого уровня сигнала.

```
#include "base_module.h"

#include <Libraries/MCP23008/MCP23008.h>
```

```
class IOModule
{
    static MCP23008* mcp;
    static uint8 iodir;
    static uint8 gppu;
    static uint8 gpio;
    static String publishTopic;

 public:
    static bool initialize();
    static bool start();
    static bool shutdown();
    static void commandCallback(String message);
};
```

Этот класс представляет собой обертку для устройства расширения ввода/вывода MCP23008 с сохранением локальной копии его регистров направления, нагрузки и состояния интерфейса GPIO для упрощения процедур обновления и управления.

```
#include "io_module.h"

#include <Wire.h>

MCP23008* IOModule::mcp = 0;
uint8 IOModule::iodir;
uint8 IOModule::gppu;
uint8 IOModule::gpio;
String IOModule::publishTopic;
```

Здесь сохраняются локальные копии трех регистров устройства расширения GPIO на шине I2C: регистр направления ввода/вывода (`iodir`), регистр нагрузки (`gppu`) и регистр контакта уровня ввода/вывода (`gpio`).

```
enum {
    IO_START = 0x01,
    IO_STOP = 0x02,
    IO_STATE = 0x04,
    IO_SET_MODE = 0x08,
    IO_SET_PULLUP = 0x10,
    IO_WRITE = 0x20,
    IO_READ = 0x40,
    IO_ACTIVE = 0x80
};

enum {
    MCP_OUTPUT = 0,
    MCP_INPUT = 1
};
```

Также определяется группа команд в форме перечисления и еще одно перечисление, определяющее направление для контактов устройства расширения GPIO.

```
bool IOModule::initialize() {
    BaseModule::registerModule(MOD_IDX_IO, IOModule::start, IOModule::shutdown);
}
```

```
bool IOModule::start() {
    publishTopic = "io/response/" + OtaCore::getLocation();
    OtaCore::registerTopic("io/" + OtaCore::getLocation(), IOModule::commandCallback);
    OtaCore::starti2c();
}

bool IOModule::shutdown() {
    OtaCore::deregisterTopic("io/" + OtaCore::getLocation());
    if (mcp) {
          delete mcp;
          mcp = 0;
    }
}
```

Процедуры инициализации и запуска этого модуля аналогичны процедурам модуля PWM, как и регистрация темы MQTT для обеспечения приема команд. Различие лишь в том, что поскольку используется I2C-устройство, необходимо убедиться в полной готовности шины I2C к работе, то есть в том, что шина I2C уже была активизирована.

Ниже приведена реализация метода обработки команд.

```
void IOModule::commandCallback(String message) {
    OtaCore::log(LOG_DEBUG, "I/O command: " + message);
    uint32 mlen = message.length();
    if (mlen < 1) { return; }
    int index = 0;
    uint8 cmd = *((uint8*) &message[index++]);
    if (cmd == IO_START) {
        if (mlen > 2) {
            OtaCore::log(LOG_INFO, "Enabling I/O Module failed: too many parameters.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() +  ";" + (char) 0x01 +
                            (char) 0x00);
            return;
        }
        // Считывание требуемого адреса или использование адреса по умолчанию.
        uint8 addr = 0;
        if (mlen == 2) {
            addr = *((uint8*) &message[index++]);
            if (addr > 7) {
            // Сообщение о критическом сбое. QoS 1.
            OtaCore::log(LOG_INFO, "Enabling I/O Module failed: invalid i2c address.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x01 +
                             (char) 0x00);
            return;
        }
    }
    if (!mcp) {
        mcp = new MCP23008(0x40);
    }
    // Установка для всех контактов режима вывода (0) и низкого уровня сигнала (0)
    mcp->writeIODIR(0x00);
    mcp->writeGPIO(0x00);
    // Считывание текущих значений микросхемы.
```

```
iodir = mcp->readIODIR();
gppu = mcp->readGPPU();
gpio = mcp->readGPIO();
// Проверка регистров IODIR и GPIO.
if (iodir != 0 || gpio != 0) {
    delete mcp;
    mcp = 0;
    OtaCore::log(LOG_INFO, "Enabling I/O Module failed: not connected.");
    OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" +  (char) 0x01 +
                    (char) 0x00);
    return;
}
OtaCore::log(LOG_INFO, "Enabled I/O Module.");
OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x01 + (char) 0x01);
}
else if (cmd == IO_STOP) {
    if (mlen > 1) {
        OtaCore::log(LOG_INFO, "Disabling I/O Module failed: too many parameters.");
        OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x02 +
                        (char) 0x00);
        return;
    }
    if (mcp) {
        delete mcp;
        mcp = 0;
    }
    OtaCore::log(LOG_INFO, "Disabled I/O Module.");
    OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" +  (char) 0x02 +
                    (char) 0x01);
}
else if (cmd == IO_STATE) {
      if (mlen > 1) {
            OtaCore::log(LOG_INFO, "Reading state failed: too many parameters.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x04 +
                            (char) 0x00);
            return;
      }
      OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x04 +
                      (char) 0x01 +
                      ((char) iodir) + ((char) gppu) + ((char) gpio));
}
else if (cmd == IO_SET_MODE) {
      if (mlen != 3) {
            OtaCore::log(LOG_INFO, "Reading state failed: incorrect number of parameters.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x08 +
                            (char) 0x00);
            return;
      }

      uint8 pnum = *((uint8*) &message[index++]);
      uint8 pstate = *((uint8*) &message[index]);

      if (pnum > 7) {
```

```
            OtaCore::log(LOG_INFO, "Setting pin mode failed: unknown pin.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x08 +
                        (char) 0x00);
            return;
      }
      if (pstate > 1) {
            // Сообщение о критическом сбое. QoS 1.
            OtaCore::log(LOG_INFO, "Setting pin mode failed: invalid pin mode.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x08 +
                        (char) 0x00);
            return;
      }
      // Установка нового состояния регистра IODIR.
      if (pstate == MCP_INPUT) { iodir |= 1 << pnum; }
      else { iodir &= ~(1 << pnum); }
      if (mcp) {
            OtaCore::log(LOG_DEBUG, "Setting pinmode in library...");
            mcp->writeIODIR(iodir);
      }
      OtaCore::log(LOG_INFO, "Set pin mode for I/O Module.");
      OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x08 +
                  (char) 0x01);
}
else if (cmd == IO_SET_PULLUP) {
      if (mlen != 3) {
            OtaCore::log(LOG_INFO, "Reading state failed: incorrect number of parameters.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x10 +
                        (char) 0x00);
            return;
      }

      uint8 pnum = *((uint8*) &message[index++]);
      uint8 pstate = *((uint8*) &message[index]);

      if (pnum > 7) {
            OtaCore::log(LOG_INFO, "Setting pull-up failed: unknown pin.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x10 +
                        (char) 0x00);
            return;
      }
      if (pstate > 1) {
            OtaCore::log(LOG_INFO, "Setting pull-up failed: invalid state.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x10 +
                        (char) 0x00);
            return;
      }
      if (pstate == HIGH) { gppu |= 1 << pnum; }
      else { gppu &= ~(1 << pnum); }
      if (mcp) {
            OtaCore::log(LOG_DEBUG, "Setting pull-up in library...");
            mcp->writeGPPU(gppu);
      }

      OtaCore::log(LOG_INFO, "Changed pull-up for I/O Module.");
```

```
      OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x10 +
                  (char) 0x01);
}
else if (cmd == IO_WRITE) {
      if (mlen != 3) {
            OtaCore::log(LOG_INFO, "Writing pin failed: incorrect number of parameters.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x20 +
                        (char) 0x00);
            return;
      }

      // Настройка нового уровня контактов GPIO.
      uint8 pnum = *((uint8*) &message[index++]);
      uint8 pstate = *((uint8*) &message[index]);
      if (pnum > 7) {
            OtaCore::log(LOG_INFO, "Writing pin failed: unknown pin.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x20 +
                        (char) 0x00);
            return;
      }
      if (pstate > 1) {
            OtaCore::log(LOG_INFO, "Writing pin failed: invalid state.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x20 +
                        (char) 0x00);
            return;
      }
      String state = "low";
      if (pstate == HIGH) { gpio |= 1 << pnum; state = "high"; }
      else { gpio &= ~(1 << pnum); }
      OtaCore::log(LOG_DEBUG, "Changed GPIO to: " + ((char) gpio));
      if (mcp) {
            OtaCore::log(LOG_DEBUG, "Setting state to " + state +
                              " in library for pin " + ((char) pnum));
            mcp->writeGPIO(gpio);
      }
      OtaCore::log(LOG_INFO, "Wrote pin state for I/O Module.");
      OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x20 +
                  (char) 0x01);
}
else if (cmd == IO_READ) {
      if (mlen > 2) {
            OtaCore::log(LOG_INFO, "Reading pin failed: too many parameters.");
            OtaCore::publish(publishTopic, OtaCore::getLocation()(char)0x40 + (char) 0x00);
            return;
      }
      // Считывание состояния контактов GPIO и возврат состояния.
      uint8 pnum = *((uint8*) &message[index]);
    if (pnum > 7) {
        OtaCore::log(LOG_INFO, "Reading pin failed: unknown pin.");
        OtaCore::publish(publishTopic, OtaCore::getLocation() + ";"  + (char) 0x40 +
                    (char) 0x00);
    }
      uint8 pstate;
```

```
        if (mcp) {
            OtaCore::log(LOG_DEBUG, "Reading pin in library...");
            pstate = (mcp->readGPIO() >> pnum) & 0x1;
        }
        OtaCore::log(LOG_INFO, "Read pin state for I/O Module.");
        OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x40 +
                         (char) 0x01 + (char) pnum + (char) pstate);
    }
    else if (cmd == IO_ACTIVE) {
        if (mlen > 1) {
            OtaCore::log(LOG_INFO, "Reading active status failed: too many parameters.");
            OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x80 +
                             (char) 0x00);
            return;
        }
        uint8 active = 0;
        if (mcp) { active = 1; }
        char output[] = { 0x80, 0x01, active };
        OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + String(output, 3));
    }
}
```

Используемый протокол описан в табл. 9.3.

Таблица 9.3

Команда	Смысл	Полезная нагрузка	Возвращаемое значение
0x01	Запуск модуля	uint8 (смещение адреса шины I2C; 0–7, необязательный параметр)	0x01 0x00/0x01
0x02	Останов модуля	–	0x02 0x00/0x01
0x04	Возврат режима ввода/вывода, нагрузки и состояния (уровня)	–	0x04 0x00/0x01 (результат) uint8 (регистр iodir) uint8(регистр gppu) uint8(регистр gpio)
0x08	Установка контакта в заданный режим (ввод/вывод)	uint8 (номер контакта, 0–7) uint8 (0: вывод, 1: ввод)	0x08 0x00/0x01
0x10	Установка контакта нагрузочного резистора (низкий/высокий)	uint8 (номер контакта, 0–7) uint8 (состояние нагрузки контакта, 0/1)	0x10 0x00/0x01
0x20	Установка для контакта уровня сигнала (низкий/высокий)	uint8 (номер контакта, 0–7) uint8 (состояние контакта, 0/1)	0x20 0x00/0x01
0x40	Считывание текущего значения контакта (низкий, высокий)	uint8 (номер контакта)	0x40 0x00/0x01 uint8 (номер контакта) uint8 (значение с контакта)
0x80	Возврат состояния: был ли модуль инициализирован	–	0x80 0x00/0x01 uint8 (состояние модуля, 0/1)

Как и по протоколу модуля PWM, возвращается логическое значение для обозначения успешного завершения действия или запрошенная полезная нагрузка (ответ). Кроме того, в ответе возвращается команда, к которой было выполнено обращение.

Команда представляет собой один байт, то есть доступно восемь команд, поскольку здесь используются битовые флаги. При необходимости можно расширить набор до 256 команд.

Возможные улучшения для этого модуля включают объединение дублирующегося кода в (встроенные) вызовы функций и использование подкласса, который должен управлять установкой и переключением отдельных битов с помощью API более высокого уровня.

Модуль Switch

Так как в каждой части (помещении) офиса имеется собственный центральный переключатель, регулирующий подачу воды по трубам к вентиляторным доводчикам (конвекторам), управление должно быть организовано с внутреннего сервера. Используя определенную конфигурацию двустабильных реле (с запоминанием), можно регулировать переключение между режимами обогрева и охлаждения, а также ввести запоминающий элемент, который может считываться узлом.

Такая система была собрана на одной плате, которая заменила ранее используемый ручной переключатель. Для управления системой применялся модуль, исходный код которого приведен ниже.

```
#include "base_module.h"

class SwitchModule {
    static String publishTopic;

 public:
    static bool initialize();
    static bool start();
    static bool shutdown();
    static void commandCallback(String message);
};
```

Реализация класса:

```
#include "switch_module.h"
#include <Wire.h>
#define SW1_SET_PIN 5
#define SW2_SET_PIN 4
#define SW1_READ_PIN 14
#define SW2_READ_PIN 12

String SwitchModule::publishTopic;

enum {
    SWITCH_ONE = 0x01,//Включение первой подключенной нагрузки, отключение второй.
    SWITCH_TWO = 0x02,//Включение второй подключенной нагрузки, отключение первой.
    SWITCH_STATE = 0x04,//Возврат положения переключателя (0x01/0x02).
};

bool SwitchModule::initialize() {
```

```
    BaseModule::registerModule(MOD_IDX_SWITCH, SwitchModule::start, SwitchModule::shutdown);
}

bool SwitchModule::start() {
    // Регистрация контактов.
    if (!OtaCore::claimPin(ESP8266_gpio05)) { return false; }
    if (!OtaCore::claimPin(ESP8266_gpio04)) { return false; }
    if (!OtaCore::claimPin(ESP8266_gpio14)) { return false; }
    if (!OtaCore::claimPin(ESP8266_gpio12)) { return false; }
    publishTopic = "switch/response/" + OtaCore::getLocation();
    OtaCore::registerTopic("switch/" + OtaCore::getLocation(),
    SwitchModule::commandCallback);
    // Установка нагрузок на входных контактах и конфигурирование выходных контактов.
    pinMode(SW1_SET_PIN, OUTPUT);
    pinMode(SW2_SET_PIN, OUTPUT);
    pinMode(SW1_READ_PIN, INPUT_PULLUP);
    pinMode(SW2_READ_PIN, INPUT_PULLUP);
    digitalWrite(SW1_SET_PIN, LOW);
    digitalWrite(SW2_SET_PIN, LOW);
}

bool SwitchModule::shutdown() {
    OtaCore::deregisterTopic("switch/" + OtaCore::getLocation());
    // Отмена регистрации контактов.
    if (!OtaCore::releasePin(ESP8266_gpio05)) { return false; }
    if (!OtaCore::releasePin(ESP8266_gpio04)) { return false; }
    if (!OtaCore::releasePin(ESP8266_gpio14)) { return false; }
    if (!OtaCore::releasePin(ESP8266_gpio12)) { return false; }
}

void SwitchModule::commandCallback(String message) {
    // Сообщение – это команда.
    OtaCore::log(LOG_DEBUG, "Switch command: " + message);
    uint32 mlen = message.length();
    if (mlen < 1) { return; }
    int index = 0;
    uint8 cmd = *((uint8*) &message[index++]);
    if (cmd == SWITCH_ONE) {
          if (mlen > 1) {
                // Сообщение о критическом сбое. QoS 1.
                OtaCore::log(LOG_INFO, "Switching to position 1 failed: too many
                            parameters.");
                OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x01 +
                                  (char) 0x00);
                return;
          }
          // Настройка реле в первоначальной позиции (условие сброса).
          // Принудительный перевод контактов 3 и 10 реле с запоминанием в активное состояние.
          digitalWrite(SW1_SET_PIN, HIGH);
          delay(1000); // Ожидание 1 секунда для переключения позиции (состояния) реле.
          digitalWrite(SW1_SET_PIN, LOW);
          OtaCore::log(LOG_INFO, "Switched to position 1.");
          OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x01 +
```

```
                        (char) 0x01);
    }
    else if (cmd == SWITCH_TWO) {
          if (mlen > 1) {
                OtaCore::log(LOG_INFO, "Switching to position 2 failed: too many
                            parameters.");
                OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x02 +
                                  (char) 0x00);
                return;
          }
          // Настройка реле в первоначальной позиции (условие сброса).
          // Принудительный перевод контактов 3 и 10 реле с запоминанием в активное состояние.
          digitalWrite(SW2_SET_PIN, HIGH);
          delay(1000); // Ожидание 1 секунда для переключения позиции (состояния) реле.
          digitalWrite(SW2_SET_PIN, LOW);
          OtaCore::log(LOG_INFO, "Switched to position 1.");
          OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x02 +
                           (char) 0x01);
    }
    else if (cmd == SWITCH_STATE) {
          if (mlen > 1) {
                OtaCore::log(LOG_INFO, "Reading state failed: too many parameters.");
                OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x04 +
                                  (char) 0x00);
                return;
          }
          // Проверка значения двух входных контактов. Если один из них имеет значение низкий
          // (low), то это активная позиция.
          uint8 active = 2;
          if (digitalRead(SW1_READ_PIN) == LOW) { active = 0; }
          else if (digitalRead(SW2_READ_PIN) == LOW) { active = 1; }
          if (active > 1) {
                OtaCore::log(LOG_INFO, "Reading state failed: no active state found.");
                OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x04 +
                                  (char) 0x00);
                return;
          }
          OtaCore::publish(publishTopic, OtaCore::getLocation() + ";" + (char) 0x04 +
                           (char) 0x01 + (char) active);
    }
}
```

Код этого модуля очень похож на код модулей PWM и ввода/вывода с регистрацией темы MQTT для обеспечения обмена информацией с использованием собственного бинарного протокола. Здесь устройство управляется очень просто. Имеется двустабильное реле с запоминанием, то есть с двумя состояниями, одна сторона которого включается в соединения, между которыми должны происходить переключения, а вторая сторона используется как однобитовая ячейка памяти.

Так как обе стороны этого типа реле будут переключаться одновременно, можно создать счетчик на стороне, подключенной к микроконтроллеру, для соответ-

ствия позиции на стороне, подключенной к остальной системе. Даже после сбоя электропитания или перезагрузки микроконтроллера можно будет просто прочитать значения контактов, соединенных с реле, чтобы определить состояние системы.

Используемый протокол описан в табл. 9.4.

Таблица 9.4

Команда	Смысл	Полезная нагрузка	Возвращаемое значение
0x01	Переключение в позицию 1	–	0x01 0x00/0x01
0x02	Переключение в позицию 2	–	0x02 0x00/0x01
0x04	Возврат текущего состояния	–	0x04 0x00/0x01 (результат) uint8 (активный контакт 0x00, 0x01)

Сервер управления и контроля

Как уже отмечалось ранее в этой главе, так называемый сервер управления и контроля (command and control server – C&C), по существу, является базой данных, содержащей информацию об отдельных узлах и их конфигурации. Эта информация используется как самими узлами, так и инструментальными средствами администрирования, подобными описанному в следующем разделе.

C&C-сервер также включает HTTP-сервер для обеспечения обновлений по воздуху (on-the-air – OTA) с использованием протокола HTTP. Поскольку система BMaC работает на основе протокола MQTT, C&C-сервер также написан в виде клиента MQTT.

```
#include "listener.h"
#include <iostream>
#include <string>

using namespace std;

#include <Poco/Util/IniFileConfiguration.h>
#include <Poco/AutoPtr.h>
#include <Poco/Net/HTTPServer.h>

using namespace Poco::Util;
using namespace Poco;
using namespace Poco::Net;

#include "httprequestfactory.h"

int main(int argc, char* argv[]) {
   cout << "Starting MQTT BMaC Command & Control server...\n";
   int rc;
   mosqpp::lib_init();
   cout << "Initialised C++ Mosquitto library.\n";
   string configFile;
   if (argc > 1) { configFile = argv[1]; }
```

```
   else { configFile = "config.ini"; }
   AutoPtr<IniFileConfiguration> config(new IniFileConfiguration(configFile));
   string mqtt_host = config->getString("MQTT.host", "localhost");
   int mqtt_port = config->getInt("MQTT.port", 1883);
   string defaultFirmware = config->getString("Firmware.default", "ota_unified.bin");
   Listener listener("Command_and_Control", mqtt_host, mqtt_port, defaultFirmware);
   UInt16 port = config->getInt("HTTP.port", 8080);
   HTTPServerParams* params = new HTTPServerParams;
   params->setMaxQueued(100);
   params->setMaxThreads(10);
   HTTPServer httpd(new RequestHandlerFactory, port, params);
   httpd.start();
   cout << "Created listener, entering loop...\n";
   while(1) {
         rc = listener.loop();
         if (rc){
               cout << "Disconnected. Trying to reconnect...\n";
               listener.reconnect();
         }
   }
   cout << "Cleanup...\n";
   mosqpp::lib_cleanup();
   return 0;
}
```

Здесь используется клиент MQTT из библиотеки Mosquitto C++ в совокупности с рабочей средой POCO для обеспечения требуемой функциональности.

Реализация класса `Listener`:

```
#include <mosquittopp.h>
#include <string>

using namespace std;

#include <Poco/Data/Session.h>
#include <Poco/Data/SQLite/Connector.h>

using namespace Poco;

class Listener : public mosqpp::mosquittopp
{
   Data::Session* session;
   string defaultFirmware;
public:
   Listener(string clientId, string host, int port, string defaultFirmware);
   ~Listener();
   void on_connect(int rc);
   void on_message(const struct mosquitto_message* message);
   void on_subscribe(int mid, int qos_count, const int* granted_qos);
};
```

Заголовочные файлы из рабочей среды POCO включены для обеспечения функциональности базы данных SQLite, которая используется как внутренняя база данных для этого приложения. Сам класс является производным от класса библиотеки Mosquitto C++, предоставляя все основные функциональные возможно-

сти в сочетании с некоторыми функциями-заглушками, которые будут реализованы немного позже.

```
#include "listener.h"

#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;

#include <Poco/StringTokenizer.h>
#include <Poco/String.h>
#include <Poco/Net/HTTPSClientSession.h>
#include <Poco/Net/HTTPRequest.h>
#include <Poco/Net/HTTPResponse.h>
#include <Poco/File.h>

using namespace Poco::Data::Keywords;

struct Node {
   string uid;
   string location;
   UInt32 modules;
   float posx;
   float posy;
};
```

Здесь определена структура для отдельного узла.

```
Listener::Listener(string clientId, string host, int port, string defaultFirmware) :
                   mosquittopp(clientId.c_str())
{
   int keepalive = 60;
   connect(host.c_str(), port, keepalive);
   Data::SQLite::Connector::registerConnector();
   session = new Poco::Data::Session("SQLite", "nodes.db");
   (*session) << "CREATE TABLE IF NOT EXISTS nodes (uid TEXT UNIQUE, \
                  location TEXT, \
                  modules INT, \
                  posx FLOAT, \
                  posy FLOAT)", now;
   (*session) << "CREATE TABLE IF NOT EXISTS firmware (uid TEXT UNIQUE, file TEXT)", now;
   this->defaultFirmware = defaultFirmware;
}
```

В конструкторе выполняется попытка установить соединение с брокером MQTT. Для этого используется заданный хост и номер порта. Также устанавливается соединение с базой данных SQLite и обеспечивается создание таблицы нормально работающих узлов и специализированного ПО.

```
Listener::~Listener() {
   //
}

void Listener::on_connect(int rc) {
```

```
    cout << "Connected. Subscribing to topics...\n";
    if (rc == 0) {
          string topic = "cc/config";   // Оповещение от узлов, подключенных в режим онлайн.
          subscribe(0, topic.c_str());
          topic = "cc/ui/config";       // C&C-клиент: запрос конфигурации.
          subscribe(0, topic.c_str());
          topic = "cc/nodes/new";       // C&C-клиент: добавление нового узла.
          subscribe(0, topic.c_str());
          topic = "cc/nodes/update";    // C&C-клиент: обновление узла.
          subscribe(0, topic.c_str());
          topic = "nsa/events/CO2";     // События, связанные с наблюдением за концентрацией CO2.
          subscribe(0, topic.c_str());
          topic = "cc/firmware";        // C&C-клиент: команды специализированного ПО.
          subscribe(0, topic.c_str());
    }
    else {
          cerr << "Connection failed. Aborting subscribing.\n";
    }
}
```

Изменена реализация обратного вызова при установлении соединения с брокером MQTT. В этом методе выполняется подписка на все темы MQTT, в которых мы заинтересованы.

Следующий метод вызывается при получении сообщения MQTT в одной из тем, на которые оформлена подписка.

```
void Listener::on_message(const struct mosquitto_message* message) {
    string topic = message->topic;
    string payload = string((const char*) message->payload, message->payloadlen);
    if (topic == "cc/config") {
          if (payload.length() < 1) {
                cerr << "Invalid payload: " << payload << ". Reject.\n";
                return;
          }
```

Проверяется полезная нагрузка, принимаемая в каждой теме. В первой теме ожидается полезная нагрузка, содержащая MAC-адрес узла, сделавшего запрос на получение конфигурации. После проверки правильности полезной нагрузки процедура продолжается.

```
          Data::Statement select(*session);
          Node node;
          node.uid = payload;
          select << "SELECT location, modules FROM nodes WHERE uid=?",
                    into (node.location),
                    into (node.modules),
                    use (payload);
          size_t rows = select.execute();
          if (rows == 1) {
                string topic = "cc/" + payload;
                string response = "mod;" + string((const char*)&node.modules, 4);
                publish(0, topic.c_str(), response.length(), response.c_str());
                response = "loc;" + node.location;
```

```
            publish(0, topic.c_str(), response.length(), response.c_str());
        }
        else if (rows < 1) {
            // Не найден узел с заданным UID.
            cerr << "Error: No data set found for uid " << payload << endl;
        }
        else {
            // Обнаружено несколько наборов данных, не являющихся корректными
            // (допустимыми)...
            cerr << "Error: Multiple data sets found for uid " << payload << "\n";
        }
    }
```

Выполняется попытка найти заданный MAC-адрес в базе данных. Если адрес найден, то считывается соответствующая конфигурация и формируется полезная нагрузка для ответного сообщения.

Следующие темы используются инструментальным средством администрирования.

```
else if (topic == "cc/ui/config") {
    if (payload == "map") {
        ifstream mapFile("map.png", ios::binary);
        if (!mapFile.is_open()) {
            cerr << "Failed to open map file.\n";
            return;
        }

        stringstream ss;
        ss << mapFile.rdbuf();
        string mapData = ss.str();
        publish(0, "cc/ui/config/map", mapData.length(),

        mapData.c_str());
}
```

Для этого варианта строки полезной нагрузки возвращаются бинарные данные для отображения образа, который должен существовать в локальном каталоге. Этот образ содержит общую схему администрируемого здания для представления ее в инструментальном средстве.

```
        else if (payload == "nodes") {
            Data::Statement countQuery(*session);
            int rowCount;
            countQuery << "SELECT COUNT(*) FROM nodes", into(rowCount), now;
            if (rowCount == 0) {
                cout << "No nodes found in database, returning...\n";
                return;
            }
            Data::Statement select(*session);
            Node node;
            select << "SELECT uid, location, modules, posx, posy FROM nodes",
                        into (node.uid),
                        into (node.location),
                        into (node.modules),
```

```
                        into (node.posx),
                        into (node.posy),
                        range(0, 1);
            string header;
            string nodes;
            string nodeStr;
            UInt32 nodeCount = 0;
            while (!select.done()) {
                  select.execute();
                  nodeStr = "NODE";
                  UInt8 length = (UInt8) node.uid.length();
                  nodeStr += string((char*) &length, 1);
                  nodeStr += node.uid;
                  length = (UInt8) node.location.length();
                  nodeStr += string((char*) &length, 1);
                  nodeStr += node.location;
                  nodeStr += string((char*) &node.posx, 4);
                  nodeStr += string((char*) &node.posy, 4);
                  nodeStr += string((char*) &node.modules, 4);
                  UInt32 segSize = nodeStr.length();
                  nodes += string((char*) &segSize, 4);
                  nodes += nodeStr;
                  ++nodeCount;
            }
            UInt64 messageSize = nodes.length() + 9;
            header = string((char*) &messageSize, 8);
            header += "NODES";
            header += string((char*) &nodeCount, 4);
            header += nodes;
            publish(0, "cc/nodes/all", header.length(), header.c_str());
      }
}
```

В приведенном выше фрагменте кода считывается каждый отдельный узел в базе данных и возвращается в сериализованном бинарном формате.

Далее создается новый узел и добавляется в базу данных.

```
else if (topic == "cc/nodes/new") {
      UInt32 index = 0;
      UInt32 msgLength = *((UInt32*) payload.substr(index, 4).data());
      index += 4;
      string signature = payload.substr(index, 4);
      index += 4;
      if (signature != "NODE") {
            cerr << "Invalid node signature.\n";
            return;
      }
      UInt8 uidLength = (UInt8) payload[index++];
      Node node;
      node.uid = payload.substr(index, uidLength);
      index += uidLength;
      UInt8 locationLength = (UInt8) payload[index++];
      node.location = payload.substr(index, locationLength);
```

```
        index += locationLength;
        node.posx = *((float*) payload.substr(index, 4).data());
        index += 4;
        node.posy = *((float*) payload.substr(index, 4).data());
        index += 4;
        node.modules = *((UInt32*) payload.substr(index, 4).data());
        cout << "Storing new node for UID: " << node.uid << "\n";
        Data::Statement insert(*session);
        insert << "INSERT INTO nodes VALUES(?, ?, ?, ?, ?)",
                  use(node.uid),
                  use(node.location),
                  use(node.modules),
                  use(node.posx),
                  use(node.posy),
                  now;
        (*session) << "INSERT INTO firmware VALUES(?, ?)",
                  use(node.uid),
                  use(defaultFirmware),
                  now;
}
```

Также поддерживается операция обновления конфигурации узла.

```
else if (topic == "cc/nodes/update") {
        UInt32 index = 0;
        UInt32 msgLength = *((UInt32*) payload.substr(index, 4).data());
        index += 4;
        string signature = payload.substr(index, 4);
        index += 4;
        if (signature != "NODE") {
              cerr << "Invalid node signature.\n";
              return;
        }
        UInt8 uidLength = (UInt8) payload[index++];
        Node node;
        node.uid = payload.substr(index, uidLength);
        index += uidLength;
        UInt8 locationLength = (UInt8) payload[index++];
        node.location = payload.substr(index, locationLength);
        index += locationLength;
        node.posx = *((float*) payload.substr(index, 4).data());
        index += 4;
        node.posy = *((float*) payload.substr(index, 4).data());
        index += 4;
        node.modules = *((UInt32*) payload.substr(index, 4).data());
        cout << "Updating node for UID: " << node.uid << "\n";
        Data::Statement update(*session);
        update << "UPDATE nodes SET location = ?, posx = ?, posy = ?, modules = ?
               WHERE uid = ?",
                    use(node.location),
                    use(node.posx),
                    use(node.posy),
                    use(node.modules),
```

```
                use(node.uid),
                now;
}
```

Далее следует обработка темы для удаления конфигурации узла.

```
else if (topic == "cc/nodes/delete") {
      cout << "Deleting node with UID: " << payload << "\n";
      Data::Statement del(*session);
      del << "DELETE FROM nodes WHERE uid = ?", use(payload), now;
      (*session) << "DELETE FROM firmware WHERE uid = ?", use(payload), now;
}
```

При изучении модуля CO2 в специализированном ПО было отмечено, что в модуле генерируются события, связанные с наблюдением за концентрацией углекислого газа. То же самое можно отметить и в рассматриваемом здесь примере: события генерируются в формате JSON и передаются в некоторый API на основе протокола HTTP. Затем используется HTTP-клиент из рабочей среды POCO для отправки JSON-сообщения на удаленный сервер (установленный на localhost).

```
else if (topic == "nsa/events/CO2") {
      StringTokenizer st(payload, ";", StringTokenizer::TOK_TRIM |
                         StringTokenizer::TOK_IGNORE_EMPTY);
      if (st.count() < 4) {
            cerr << "CO2 event: Wrong number of arguments. Payload: " << payload << "\n";
            return;
      }
      string state = "ok";
      if (st[1] == "1") { state = "warn"; }
      else if (st[1] == "2") { state = "crit"; }
      string increase = (st[2] == "1") ? "true" : "false";
      string json = "{ \"state\": \"" + state + "\", \
                              \"location\": \"" + st[0] + "\", \
                              \"increase\": " + increase + ", \
                              \"ppm\": " + st[3] + " }";
      Net::HTTPSClientSession httpsClient("localhost");
      try {
            Net::HTTPRequest request(Net::HTTPRequest::HTTP_POST, "/",
                                     Net::HTTPMessage::HTTP_1_1);
            request.setContentLength(json.length());
            request.setContentType("application/json");
            httpsClient.sendRequest(request) << json;
            Net::HTTPResponse response;
            httpsClient.receiveResponse(response);
      }
      catch (Exception& exc) {
            cout << "Exception caught while attempting to connect." << std::endl;
            cerr << exc.displayText() << std::endl;
            return;
      }
}
```

Для управления хранимыми образами специализированного ПО можно воспользоваться следующей темой. Конкретную версию специализированного ПО,

используемую конкретным узлом, можно установить в каждой конфигурации для отдельного узла, хотя, как мы видели ранее, по умолчанию используется самая последняя версия.

В этой теме можно получить список доступных образов специализированного ПО или выгрузить новую версию.

```
   else if (topic == "cc/firmware") {
         if (payload == "list") {
               std::vector<File> files;
               File file("firmware");
               if (!file.isDirectory()) { return; }
               file.list(files);
               string out;
               for (int i = 0; i < files.size(); ++i) {
                     if (files[i].isFile()) {
                           out += files[i].path();
                           out += ";";
                     }
               }
               out.pop_back();
               publish(0, "cc/firmware/list", out.length(), out.c_str());
         }
         else {
               StringTokenizer st(payload, ";", StringTokenizer::TOK_TRIM |
                                  StringTokenizer::TOK_IGNORE_EMPTY);
               if (st[0] == "change") {
                     if (st.count() != 3) { return; }
                     (*session) << "UPDATE firmware SET file = ? WHERE uid = ?",
                                                use (st[1]),
                                                use (st[2]),
                                                now;
               }
               else if (st[0] == "upload") {
                     if (st.count() != 3) { return; }
                     // Write file & truncate if exists.
                     string filepath = "firmware/" + st[1];
                     ofstream outfile("firmware/" + st[1], ofstream::binary| ofstream::trunc);
                     outfile.write(st[2].data(), st[2].size());
                     outfile.close();
               }
         }
   }
}

void Listener::on_subscribe(int mid, int qos_count, const int* granted_qos)
{
   //
}
```

Указанный последним метод вызывается при каждой успешной операции подписки на тему MQTT, позволяя при необходимости выполнить какие-либо действия.

Далее рассмотрим реализацию HTTP-сервера. Начнем с класса-фабрики (factory) обработчика запросов HTTP.

```
#include <Poco/Net/HTTPRequestHandlerFactory.h>
#include <Poco/Net/HTTPServerRequest.h>

using namespace Poco::Net;

#include "datahandler.h"

class RequestHandlerFactory: public HTTPRequestHandlerFactory {
public:
   RequestHandlerFactory() {}
   HTTPRequestHandler* createRequestHandler(const HTTPServerRequest& request) {
         return new DataHandler();
   }
};
```

Этот обработчик всегда будет возвращать экземпляр следующего класса.

```
#include <iostream>
#include <vector>

using namespace std;

#include <Poco/Net/HTTPRequestHandler.h>
#include <Poco/Net/HTTPServerResponse.h>
#include <Poco/Net/HTTPServerRequest.h>
#include <Poco/URI.h>
#include <Poco/File.h>

#include <Poco/Data/Session.h>
#include <Poco/Data/SQLite/Connector.h>

using namespace Poco::Data::Keywords;
using namespace Poco::Net;
using namespace Poco;

class DataHandler: public HTTPRequestHandler
{
public:
   void handleRequest(HTTPServerRequest& request, HTTPServerResponse& response) {
         cout << "DataHandler: Request from " + request.clientAddress().toString() << endl;
         URI uri(request.getURI());
         string path = uri.getPath();
         if (path != "/") {
               response.setStatus(HTTPResponse::HTTP_NOT_FOUND);
               ostream& ostr = response.send();
               ostr << "File Not Found: " << path;
               return;
         }
         URI::QueryParameters parts;
         parts = uri.getQueryParameters();
         if (parts.size() > 0 && parts[0].first == "uid") {
               Data::SQLite::Connector::registerConnector();
               Data::Session* session = new Poco::Data::Session("SQLite", "nodes.db");
               Data::Statement select(*session);
```

```
            string filename;
            select << "SELECT file FROM firmware WHERE uid=?",
                        into (filename),
                        use (parts[0].second);
            size_t rows = select.execute();
            if (rows != 1) {
                  response.setStatus(HTTPResponse::HTTP_NOT_FOUND);
                  ostream& ostr = response.send();
                  ostr << "File Not Found: " << parts[0].second;
                  return;
            }
            string fileroot = "firmware/";
            File file(fileroot + filename);
            if (!file.exists() || file.isDirectory()) {
                  response.setStatus(HTTPResponse::HTTP_NOT_FOUND);
                  ostream& ostr = response.send();
                  ostr << "File Not Found.";
                  return;
            }
            string mime = "application/octet-stream";
            try {
                  response.sendFile(file.path(), mime);
            }
            catch (FileNotFoundException &e) {
                  cout << "File not found exception triggered..." << endl;
                  cerr << e.displayText() << endl;
                  response.setStatus(HTTPResponse::HTTP_NOT_FOUND);
                  ostream& ostr = response.send();
                  ostr << "File Not Found.";
                  return;
            }
            catch (OpenFileException &e) {
                  cout << "Open file exception triggered..." << endl;
                  cerr << e.displayText() << endl;
                  response.setStatus(HTTPResponse::HTTP_INTERNAL_SERVER_ERROR);
                  ostream& ostr = response.send();
                  ostr << "Internal Server Error. Couldn't open file.";
                  return;
            }
      }
      else {
            response.setStatus(HTTPResponse::HTTP_BAD_REQUEST);
            response.send();
            return;
      }
   }
};
```

Реализация этого класса выглядит достаточно сложной, но содержит всего лишь операции поиска в базе данных SQLite идентификатора узла (MAC-адреса) и обеспечивает возврат соответствующего образа специализированного ПО по найденному идентификатору.

Инструментальное средство администрирования

С помощью API, реализованных в коде сервера управления и контроля, было создано инструментальное средство администрирования с графическим пользовательским интерфейсом (GUI) с использованием рабочей среды Qt5. Также была разработана библиотека клиента Mosquitto MQTT, обеспечивающая поддержку основных средств управления узлами. Основой для этих инструментов послужил графический план администрируемых зданий и помещений.

Несмотря на общие преимущества и удобства, было обнаружено, что разработка графического инструментального средства слишком сложна. Кроме того, такой инструмент ограничен только одним этажом здания, если не создавать действительно огромную карту, содержащую все этажи с размещенными на них узлами. Стало очевидно, что это было бы весьма громоздким решением.

В исходном коде, приведенном в этой главе, можно найти и элементы такого инструментального средства администрирования, представленного в качестве примера одного из вариантов его реализации. Но из соображений экономии места и краткости изложения полный код такого инструмента здесь не приводится.

Система кондиционирования воздуха

Для управления устройствами кондиционирования воздуха, то есть системой, во многом похожей на ранее разработанный сервер управления и контроля, использовался тот же основной шаблон. Самые интересные фрагменты исходного кода приведены ниже в этом разделе.

```
#include <string>
#include <vector>

using namespace std;

#include <Poco/Data/Session.h>
#include <Poco/Data/SQLite/Connector.h>

#include <Poco/Net/HTTPClientSession.h>
#include <Poco/Net/HTTPSClientSession.h>

#include <Poco/Timer.h>

using namespace Poco;
using namespace Poco::Net;

class Listener;

struct NodeInfo {
    string uid;
    float posx;
    float posy;
    float current;
    float target;
    bool ch0_state;
    UInt8 ch0_duty;
    bool ch0_valid;
```

```
    bool ch1_state;
    UInt8 ch1_duty;
    bool ch1_valid;
    bool ch2_state;
    UInt8 ch2_duty;
    bool ch2_valid;
    bool ch3_state;
    UInt8 ch3_duty;
    bool ch3_valid;
    UInt8 validate;
};

struct ValveInfo {
    string uid;
    UInt8 ch0_valve;
    UInt8 ch1_valve;
    UInt8 ch2_valve;
    UInt8 ch3_valve;
};

struct SwitchInfo {
    string uid;
    bool state;
};

#include "listener.h"

class Nodes {
    static Data::Session* session;
    static bool initialized;
    static HTTPClientSession* influxClient;
    static string influxDb;
    static bool secure;
    static Listener* listener;
    static Timer* tempTimer;
    static Timer* nodesTimer;
    static Timer* switchTimer;
    static Nodes* selfRef;

 public:
    static void init(string influxHost, int influxPort, string influxDb, string influx_sec,
                     Listener* listener);
    static void stop();
    static bool getNodeInfo(string uid, NodeInfo &info);
    static bool getValveInfo(string uid, ValveInfo &info);
    static bool getSwitchInfo(string uid, SwitchInfo &info);
    static bool setTargetTemperature(string uid, float temp);
    static bool setCurrentTemperature(string uid, float temp);
    static bool setDuty(string uid, UInt8 ch0, UInt8 ch1, UInt8 ch2, UInt8 ch3);
    static bool setValves(string uid, bool ch0, bool ch1, bool ch2, bool ch3);
    static bool setSwitch(string uid, bool state);
    void updateCurrentTemperatures(Timer& timer);
    void checkNodes(Timer& timer);
    void checkSwitch(Timer& timer);
```

```
    static bool getUIDs(vector<string> &uids);
    static bool getSwitchUIDs(vector<string> &uids);
};
```

Определение этого класса в сервисе AC (переменный ток) предоставляет подробный обзор его функциональности. По существу, это обертка базы данных SQLite, содержащей информацию об узлах, клапанах и переключателях охлаждения/обогрева. В этот класс также включен таймер, отслеживающий и управляющий активизацией приложения для проверки состояния системы, для сравнения его с целевым состоянием и для внесения изменений при необходимости.

Этот класс интенсивно используется классом `Listener` того же приложения для наблюдения за состоянием узлов и подключенных устройств переменного тока в совокупности с переключателями и клапанами, управляющими подачей воды.

```
#include <mosquittopp.h>

#include <string>
#include <map>

using namespace std;

#include <Poco/Mutex.h>

using namespace Poco;

struct NodeInfo;
struct ValveInfo;
struct SwitchInfo;

#include "nodes.h"

class Listener : public mosqpp::mosquittopp {
    map<string, NodeInfo> nodes;
    map<string, ValveInfo> valves;
    map<string, SwitchInfo> switches;
    Mutex nodesLock;
    Mutex valvesLock;
    Mutex switchesLock;
    bool heating;
    Mutex heatingLock;

 public:
    Listener(string clientId, string host, int port);
    ~Listener();
    void on_connect(int rc);
    void on_message(const struct mosquitto_message* message);
    void on_subscribe(int mid, int qos_count, const int* granted_qos);
    bool checkNodes();
    bool checkSwitch();
};
```

Основной принцип работы этого приложения заключается в том, что таймеры класса `Nodes` активизируют класс `Listener` для публикации сообщений в темах для модулей PWM, IO и Switch, запрашивая о состоянии тех устройств, которые предполагаются активно работающими.

Такой тип системы с активным циклом является общепринятым в промышленных приложениях, поскольку обеспечивает постоянную проверку системы, позволяя быстро определять те компоненты, которые не работают так, как ожидается.

База данных InfluxDB для записи показаний датчиков

Запись показаний датчиков и статистические характеристики от кофе-автоматов были определены как приоритетные данные с самого начала разработки проекта. В идеальном варианте для этого типа данных наиболее подходящей является база данных временны́х последовательностей, среди которых самой широко применяемой является Influx. Самая большая проблема при использовании этой базы данных – отсутствие поддержки протокола MQTT, она предлагает только протокол HTTP и собственный встроенный интерфейс.

Для устранения этой проблемы был написан простой линейный преобразователь протоколов MQTT-to-Influx HTTP. И в этом случае использовалась клиентская библиотека Mosquitto и функциональность HTTP из рабочей среды POCO.

```
#include "mth.h"

#include <iostream>

using namespace std;

#include <Poco/Net/HTTPRequest.h>
#include <Poco/Net/HTTPResponse.h>
#include <Poco/StringTokenizer.h>
#include <Poco/String.h>

using namespace Poco;

MtH::MtH(string clientId, string host, int port, string topics, string influxHost,
         int influxPort, string influxDb, string influx_sec) : mosquittopp(clientId.c_str())
{
    this->topics  = topics;
    this->influxDb = influxDb;
    if (influx_sec == "true") {
          cout << "Connecting with HTTPS..." << std::endl;
          influxClient = new Net::HTTPSClientSession(influxHost, influxPort);
          secure = true;
    }
    else {
          cout << "Connecting with HTTP..." << std::endl;
          influxClient = new Net::HTTPClientSession(influxHost, influxPort);
          secure = false;
    }

    int keepalive = 60;
    connect(host.c_str(), port, keepalive);
}
```

В приведенном выше конструкторе устанавливается соединение с брокером MQTT и создается клиент HTTP или HTTPS в зависимости от настройки конкретного протокола в файле конфигурации.

```
MtH::~MtH() {
    delete influxClient;
}

void MtH::on_connect(int rc) {
    cout << "Connected. Subscribing to topics...\n";

    if (rc == 0) {
          StringTokenizer st(topics, ",", StringTokenizer::TOK_TRIM |
                                          StringTokenizer::TOK_IGNORE_EMPTY);
          for (StringTokenizer::Iterator it = st.begin(); it != st.end(); ++it) {
                string topic = string(*it);
                cout << "Subscribing to: " << topic << "\n";
                subscribe(0, topic.c_str());
                // Добавляется имя последовательности в таблицу отображений 'series'.
                StringTokenizer st1(topic, "/", StringTokenizer::TOK_TRIM |
                                                StringTokenizer::TOK_IGNORE_EMPTY);
                string s = st1[st1.count() - 1]; // Взять самый последний элемент.
                series.insert(std::pair<string, string>(topic, s));
          }
    }
    else {
          cerr << "Connection failed. Aborting subscribing.\n";
    }
}
```

Вместо жестко заданных тем подписки MQTT здесь используются темы, определенные в файле конфигурации, представленные в виде отдельных строк каждой темы, разделенных запятыми.

Также создается отображение (map) стандартной библиотеки STL, содержащее имя временно́й последовательности для записи конкретной темы. При этом берется конечная часть темы MQTT после последнего символа слеша. Можно было бы обеспечить более гибкую конфигурацию, но для тем, используемых в системе BMaC, даже такой подход не является ограничением, так как нет необходимости в более сложной структуре тем подписки.

```
void MtH::on_message(const struct mosquitto_message* message) {
    string topic = message->topic;
    map<string, string>::iterator it = series.find(topic);
    if (it == series.end()) {
          cerr << "Topic not found: " << topic << "\n";
          return;
    }

    if (message->payloadlen < 1) {
          cerr << "No payload found. Returning...\n";
          return;
    }

    string payload = string((const char*) message->payload, message->payloadlen);
    size_t pos = payload.find(";");
    if (pos == string::npos || pos == 0) {
        cerr << "Invalid payload: " << payload << ". Reject.\n";
        return;
```

```
    }

    string uid = payload.substr(0, pos);
    string value = payload.substr(pos + 1);
    string influxMsg;
    influxMsg = series[topic];
    influxMsg += ",location=" + uid;
    influxMsg += " value=" + value;
    try {
        Net::HTTPRequest request(Net::HTTPRequest::HTTP_POST, "/write?db=" + influxDb,
                                 Net::HTTPMessage::HTTP_1_1);
        request.setContentLength(influxMsg.length());
        request.setContentType("application/x-www-form-urlencoded");
        influxClient->sendRequest(request) << influxMsg;
        Net::HTTPResponse response;
        influxClient->receiveResponse(response);
    }
    catch (Exception& exc) {
        cout << "Exception caught while attempting to connect." << std::endl;
        cerr << exc.displayText() << std::endl;
        return;
    }
}
```

При получении нового сообщения MQTT выполняется поиск соответствующего имени временно́й последовательности в базе данных Influx, затем создается строка для передачи на сервер InfluxDB. Здесь предполагается, что полезная нагрузка состоит из MAC-адреса узла, отправившего это сообщение, после которого следует точка с запятой.

Часть после точки с запятой просто считывается и устанавливается как значение. MAC-адрес используется как локация. Затем эти данные отправляются на сервер базы данных.

Вопросы обеспечения безопасности

В процессе разработки этой системы быстро стала очевидной чрезвычайная важность обеспечения ее безопасности. Поэтому было принято решение о вводе механизма шифрования с использованием TLS. Использовалась библиотека шифрования axTLS, встроенная в рабочую среду Sming, в сочетании с сертификатами AES (для хоста и клиента) для поддержки процедур проверки подлинности хоста (сервера) и клиентов (узлов). Кроме того, обеспечивалось создание защищенного шифрованного соединения.

В главе 5 уже рассматривалась обработка подобных сертификатов клиента и настройка зашифрованного MQTT-соединения. Остались без внимания лишь подробности о затруднениях, которые возникли при настройке этой системы сертификации. Как было отмечено в главе 5, микроконтроллер ESP8266 не имеет достаточного объема памяти для размещения принятых по умолчанию буферов для процедуры рукопожатия TLS, поэтому требуется использование расширения размера SSL-фрагмента на стороне сервера (на хосте).

К сожалению, выяснилось, что используемый в проекте брокер MQTT (широко применяемая библиотека Mosquitto) не поддерживает это расширение SSL, следовательно, потребуется используемый клиентом по умолчанию двойной буфер размером 16 Кб. Первый вариант решения предусматривал рекомпиляцию брокера Mosquitto после внесения нескольких изменений в его исходный код для устранения возникшей проблемы.

Более эффективным решением, реализованным в конечном итоге, стала установка программного обеспечения прокси HAProxy, которое функционировало как конечный пункт TLS, обрабатывая сертификаты и перенаправляя расшифрованный трафик в брокер MQTT через локальный интерфейс обратной петли (localhost).

При установленном параметре размера фрагмента SSL в 1–2 Кб все работало, как предполагалось. Таким образом, мы получили беспроводную систему мониторинга и управления для всего здания, которая обеспечивала защищенный обмен важной информацией и передачу управляющих команд, требующих осторожного обращения с ними.

Дальнейшие разработки

К рассмотренной здесь системе можно разработать множество дополнений и усовершенствований. Увеличение количества поддерживаемых датчиков, дополнительные микросхемы расширения интерфейса GPIO, различные конфигурации системы кондиционирования воздуха, определение присутствия людей в помещениях с учетом внутреннего календаря, отмена запланированных встреч в офисе при очевидном отсутствии людей и т. д.

Также возможна замена ESP8266 на другой микроконтроллер, например на модели на основе ARM, поддерживающие проводные соединения Ethernet, в совокупности с более эффективными инструментами разработки и отладки. Полезно и наличие микроконтроллера с Wi-Fi, который можно установить где угодно и теоретически обеспечить его бесперебойную работу. Инструментальные средства разработки для ESP8266 не столь хороши, а отсутствие вариантов проводных соединений (без использования внешних дополнительных микросхем) означает, что либо все будет работать, либо не зависеть от качества сети Wi-Fi.

Поскольку система BMaC подразумевает автоматизацию здания, рекомендуется обеспечить определенный уровень надежности, которого трудно достичь при использовании сети Wi-Fi, хотя для менее важных компонентов (статистические данные кофе-автоматов, показания датчиков и т. п.) это вряд ли будет проблемой. Предполагается, что в будущем могут применяться варианты гибридной сети с проводными и беспроводными соединениями.

Резюме

В этой главе рассматривалась разработка системы мониторинга и управления микроклиматом здания, его компонентов, а также уроки, извлеченные в процессе разработки.

Предполагается, что после изучения материала этой главы читатель сможет понять, как создается и функционирует подобная крупномасштабная встроенная система, а также будет способен воспользоваться системой BMaC как основой или самостоятельно реализовать аналогичную систему.

В следующей главе рассматривается разработка проектов встроенных систем с использованием рабочей среды Qt.

Часть III

ИНТЕГРАЦИЯ С ДРУГИМИ ИНСТРУМЕНТАЛЬНЫМИ СРЕДСТВАМИ И РАБОЧИМИ СРЕДАМИ

После изучения процесса разработки и тестирования встроенных систем можно перейти к рассмотрению процесса разработки более усовершенствованных графических пользовательских интерфейсов и процесса разработки для гибридных платформ FPGA/SoC.

В эту часть включены следующие главы:

- глава 10 «Разработка встроенных систем с использованием Qt»;
- глава 11 «Разработка для гибридных систем FPGA/SoC».

Глава 10

Разработка встроенных систем с использованием Qt

Qt (произносится cute [кьют]) – это весьма развитая рабочая среда на основе языка C++, которая охватывает широчайший спектр API, позволяя реализовывать сетевые системы, графические пользовательские интерфейсы, механизмы синтаксического анализа (парсинга) различных форматов данных, системы воспроизведения и записи аудио и многое другое. В этой главе главное внимание сосредоточено на графическом аспекте Qt, в основном на способах создания продвинутых графических пользовательских интерфейсов (GUI) для встроенных систем, чтобы обеспечить удобный, привлекательный и функциональный интерфейс для пользователей.

В этой главе рассматриваются следующие темы:

- создание продвинутых графических пользовательских интерфейсов (GUI) для встроенных систем с использованием Qt;
- использование трехмерного инструмента проектирования (3D designer) Qt для создания пользовательского интерфейса информационно-развлекательной системы;
- дополнение существующей встроенной системы графическим пользовательским интерфейсом.

Главное преимущество правильно выбранной рабочей среды

Рабочая среда (framework, часто просто фреймворк) – это, в сущности, крупная совокупность кода, предназначенная для упрощения разработки программного обеспечения для конкретных приложений. Рабочая среда предоставляет разработчику обширный диапазон классов или равнозначных сущностей другого языка, позволяющий реализовать логику приложения, не заботясь об организации взаимодействия с аппаратурой на более низком уровне и с API более низких уровней, предоставляемых операционными системами.

В предыдущих главах уже использовались некоторые рабочие среды для упрощения процесса разработки, например рабочая среда Nodate (в главе 4), CMSIS

и Arduino for microcontrollers, а также рабочая среда низкого уровня POCO для кроссплатформенной разработки и рабочая среда более высокого уровня Qt.

Каждая из этих рабочих сред предназначена для определенного типа систем. Для Nodate, CMSIS и Arduino целевыми являются семейства микроконтроллеров от 8-битовых AVR до 32-битовых ARM. Это чисто аппаратные системы без какой-либо промежуточной ОС или аналогичного ПО. С точки зрения сложности уровнем выше располагаются рабочие среды операционных систем реального времени (ОСРВ – RTOS), в которые включены полнофункциональные операционные системы.

Такие рабочие среды, как POCO и Qt, ориентированы на обобщенное применение ОС – от настольных и серверных платформ до систем на кристаллах (SoC). Поэтому их функциональность главным образом обеспечивает некоторый уровень абстракции между API конкретных ОС, но в то же время предоставляет функциональность, связанную с этим уровнем абстракции. Это позволяет быстро создавать полнофункциональные приложения без больших затрат времени и сил на реализацию каждой функции.

Это особенно важно для обеспечения сетевой функциональности, где нежелательно раз за разом писать с нуля сервер на основе TCP-сокетов, а вместо этого в идеальном случае просто создать экземпляр готового к работе класса и воспользоваться им. Кроме того, Qt предоставляет API, связанные с графическим интерфейсом пользователя, позволяющие упростить разработку GUI, независимого от платформы. Другие рабочие среды, например GTK+ и WxWidgets, также предоставляют этот тип функциональности. Но в этой главе мы будем рассматривать только разработку с использованием Qt.

В главе 8 уже рассматривались основы разработки с использованием рабочей среды Qt. Но в том случае часть, отвечающая за графический пользовательский интерфейс (GUI), практически осталась без внимания, хотя, возможно, это наиболее интересная часть рабочей среды Qt по сравнению с другими рабочими средами на основе ОС. Возможность использовать один и тот же GUI в различных операционных системах может оказаться невероятно полезной и удобной.

В основном рабочая среда Qt используется для приложений, работающих в настольных системах, где графический пользовательский интерфейс является важнейшей частью приложения. Благодаря кроссплатформенности Qt нет необходимости тратить время и силы на перенос приложений между различными ОС. Это справедливо и для встраиваемых платформ, хотя в этом случае предоставляется возможность еще более глубокой интеграции в систему, чем в настольной системе, как мы вскоре убедимся сами.

Здесь также будут рассматриваться различные типы приложений Qt, которые можно разрабатывать. Начнем с простого приложения с интерфейсом командной строки (command-line interface – CLI).

Использование Qt для приложений с интерфейсом командной строки

Несмотря на то что графический пользовательский интерфейс является компонентом, привлекающим наибольшее внимание к рабочей среде Qt, кроме него, существует возможность разработки приложений, использующих только команд-

ную строку. Необходимо воспользоваться классом QCoreApplication для создания потока ввода и цикла обработки событий, как показано в следующем примере.

```
#include <QCoreApplication>
#include <core.h>

int main(int argc, char *argv[]) {
   QCoreApplication app(argc, argv);
   Core core;

   connect(&core, &Core::done, &app, &app::quit, Qt::QueuedConnection);
   core.start();

   return app.exec();
}
```

Здесь весь код приложения реализован в классе Core. В главной функции main создается экземпляр класса QCoreApplication, который принимает параметры командной строки. Затем создается экземпляр нашего класса.

Устанавливается соединение между сигналом от экземпляра нашего класса с экземпляром QCoreApplication, так что при подаче сигнала о завершении работы будет активизирован слот в экземпляре QCoreApplication для освобождения ресурсов и завершения приложения.

После этого вызывается метод нашего класса для активизации его функциональности, наконец, начинает работу цикл обработки событий с помощью вызова exec() из экземпляра QCoreApplication. Начиная с этого момента можно использовать сигналы.

Следует отметить, что можно также использовать синтаксис связывания сигналов и слотов в стиле Qt4 вместо более раннего синтаксиса.

```
connect(core, SIGNAL(done()), &app, SLOT(quit()), Qt::QueuedConnection);
```

С функциональной точки зрения нет никакой разницы, но такой синтаксис может оказаться более удобным в большинстве ситуаций.

Ниже приведена реализация нашего класса Core.

```
#include <QObject>

class Core : public QObject {
   Q_OBJECT
public:
   explicit Core(QObject *parent = 0);
signals:
   void done();
public slots:
   void start();
};
```

Каждый класс в приложении на основе Qt, где предполагается применение архитектуры сигнал–слот, непременно должен быть производным от класса QObject и включать определение макро Q_OBJECT в объявление класса. Это необходимо для того, чтобы инструмент qmake preprocessor точно знал, какие классы подлежат обработке перед компиляцией кода приложения основным рабочим комплектом.

Реализация класса Core:

```
#include "core.h"
#include <iostream>

Core::Core(QObject *parent) : QObject(parent) {
   //
}

void hang::start() {
   std::cout << "Start emitting done()" << std::endl;
   emit done();
}
```

Следует отметить, что можно позволить конструктору любого класса, производного от QObject, узнать о существовании объемлющего родительского класса. Это обеспечивает передачу родительскому классу информации о принадлежащих ему классах-потомках и вызов их деструкторов при удалении.

Приложения с использованием графического пользовательского интерфейса Qt

Возвращаясь к примеру проекта на основе Qt из главы 8, можно сравнить его главную функцию main с версией приложения командной строки из предыдущего раздела, чтобы наблюдать изменения, внесенные для добавления GUI в проект.

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Самое явное изменение – использование класса QApplication вместо класса QCoreApplication. Другое значительное изменение состоит в использовании полностью специализированного пользовательского класса, хотя он и является производным от класса QMainWindow.

```
#include <QMainWindow>

#include <QAudioRecorder>
#include <QAudioProbe>
#include <QMediaPlayer>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT
```

```
public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
public slots:
    void playBluetooth();
    void stopBluetooth();
    void playOnlineStream();
    void stopOnlineStream();
    void playLocalFile();
    void stopLocalFile();
    void recordMessage();
    void playMessage();
    void errorString(QString err);
    void quit();
private:
    Ui::MainWindow *ui;
    QMediaPlayer* player;
    QAudioRecorder* audioRecorder;
    QAudioProbe* audioProbe;
    qint64 silence;
private slots:
    void processBuffer(QAudioBuffer);
};
```

Здесь можно видеть, что класс `MainWindow` действительно является производным от класса `QMainWindow`, что позволяет воспользоваться методом `show()`. Отметим, что класс `MainWindow` объявлен в пространстве имен `UI`. Это позволяет установить связь с автоматически генерируемым кодом, который создается при обработке инструментом `qmake` файла UI, как мы вскоре увидим. Далее следует конструктор.

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow) {
    ui->setupUi(this);
```

Здесь прежде всего следует отметить способ формирования графического пользовательского интерфейса (GUI) из файла описания интерфейса UI. Этот файл обычно создается методом визуальной компоновки графического интерфейса с помощью инструментального средства Qt Designer, входящего в состав интегрированной среды разработки (IDE) Qt Creator. Такой файл UI содержит описание свойств каждого виджета (widget), общую схему размещения всех виджетов и прочих элементов и т. д.

Кроме того, можно программно создавать те же виджеты и добавлять их в схему компоновки. Но при больших схемах компоновки это становится слишком утомительной и скучной работой. В общем случае создается один файл UI для основного окна и по одному дополнительному файлу UI для каждого дочернего и диалогового подокна. Затем эти окна также формируются в графическом представлении.

```
    connect(ui->actionQuit, SIGNAL(triggered()), this, SLOT(quit()));
```

Пункты меню в графическом пользовательском интерфейсе связываются с внутренними слотами с определением конкретного сигнала для каждого пункта меню (выполняемого действия) (экземпляр `QAction`). Здесь можно видеть, что

пункты меню находятся в объекте ui, созданном в автоматически сгенерированном исходном коде для файла UI, как было описано ранее.

```
connect(ui->playBluetoothButton, SIGNAL(pressed), this, SLOT(playBluetooth));
connect(ui->stopBluetoothButton, SIGNAL(pressed), this, SLOT(stopBluetooth));
connect(ui->playLocalAudioButton, SIGNAL(pressed), this, SLOT(playLocalFile));
connect(ui->stopLocalAudioButton, SIGNAL(pressed), this, SLOT(stopLocalFile));
connect(ui->playOnlineStreamButton, SIGNAL(pressed), this, SLOT(playOnlineStream));
connect(ui->stopOnlineStreamButton, SIGNAL(pressed), this, SLOT(stopOnlineStream));
connect(ui->recordMessageButton, SIGNAL(pressed), this, SLOT(recordMessage));
connect(ui->playBackMessage, SIGNAL(pressed), this, SLOT(playMessage));
```

Виджеты кнопок в графическом интерфейсе связываются аналогичным способом, хотя они, разумеется, генерируют совсем другие сигналы, поскольку являются другим типом виджетов.

```
silence = 0;
// Создание экземпляров интерфейса для аудио.
player = new QMediaPlayer(this);
audioRecorder = new QAudioRecorder(this);
audioProbe = new QAudioProbe(this);
// Конфигурирование устройства записи аудио.
QAudioEncoderSettings audioSettings;
audioSettings.setCodec("audio/amr");
audioSettings.setQuality(QMultimedia::HighQuality);
audioRecorder->setEncodingSettings(audioSettings);
audioRecorder->setOutputLocation(QUrl::fromLocalFile("message/last_message.amr"));
// Конфигурирование аудиопробы.
connect(audioProbe, SIGNAL(audioBufferProbed(QAudioBuffer)), this,
                    SLOT(processBuffer(QAudioBuffer)));
audioProbe→setSource(audioRecorder);
```

Как и в любом другом конструкторе, здесь можно выполнять всевозможные действия, включая установки параметров по умолчанию и создание экземпляров классов, которые потребуются в дальнейшем.

```
QThread* thread = new QThread;
VoiceInput* vi = new VoiceInput();
vi->moveToThread(thread);
connect(thread, SIGNAL(started()), vi, SLOT(run()));
connect(vi, SIGNAL(finished()), thread, SLOT(quit()));
connect(vi, SIGNAL(finished()), vi, SLOT(deleteLater()));
connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));
connect(vi, SIGNAL(error(QString)), this, SLOT(errorString(QString)));
connect(vi, SIGNAL(playBluetooth), this, SLOT(playBluetooth));
connect(vi, SIGNAL(stopBluetooth), this, SLOT(stopBluetooth));
connect(vi, SIGNAL(playLocal), this, SLOT(playLocalFile));
connect(vi, SIGNAL(stopLocal), this, SLOT(stopLocalFile));
connect(vi, SIGNAL(playRemote), this, SLOT(playOnlineStream));
connect(vi, SIGNAL(stopRemote), this, SLOT(stopOnlineStream));
connect(vi, SIGNAL(recordMessage), this, SLOT(recordMessage));
connect(vi, SIGNAL(playMessage), this, SLOT(playMessage));
thread->start();
}
```

Чрезвычайно важно помнить о том, что этот класс активизируется в потоке UI, таким образом, в нем не должна выполняться какая-либо интенсивная ресурсоемкая работа. Именно поэтому экземпляры этого класса выносятся в собственный отдельный поток, как показано выше.

```
MainWindow::~MainWindow() {
    delete ui;
}
```

В деструкторе удаляется объект UI и все связанные с ним элементы.

Qt для встроенных систем

Помимо настольных систем, одной из главных целей применения рабочей среды Qt являются встроенные системы, в частности Embedded Linux, для которых существует несколько различных способов использования Qt. Основной пункт применения Qt для встроенных систем – оптимизация основного комплекта ПО, позволяющая загружаться непосредственно в Qt-оптимизированную среду и предоставляющая разнообразные методы формирования и отрисовки графического экрана.

Qt для Embedded Linux поддерживает подключаемые модули (plugins) формирования и отрисовки графики, перечисленные в табл. 10.1.

Таблица 10.1

Подключаемый модуль	Описание
EGLFS	Предоставляет интерфейс для OpenGL ES или аналогичных API 3D-отрисовки. Обычно это конфигурация по умолчанию для Embedded Linux. Более подробную информацию о EGLFS можно найти на сайте **https://www.khronos.org/egl**
LinuxFB	Выполняет запись напрямую во фреймбуфер через подисистему Linux fbdev. Поддерживается только программно формируемый контент. В результате некоторые настройки производительности вывода графики на дисплей, вероятнее всего, будут ограничены
DirectFB	Выполняет запись напрямую во фреймбуфер графической карты, используя для этого библиотеку DirectFB
Wayland	Использует оконную систему Wayland. Это позволяет создавать несколько параллельно работающих окон, но, разумеется, создает бóльшую зависимость от аппаратуры

В дополнение к этим подключаемым модулям в сам комплект Qt для Embedded Linux включены разнообразные API для обработки сенсорного и перьевого ввода и т. д. В целях оптимизации системы для приложения на основе Qt обычно удаляются все ненужные сервисы, процессы и библиотеки. В результате получается система, загружаемая во встроенное приложение буквально за считанные секунды.

Графические пользовательские интерфейсы с использованием таблиц стилей

Стандартные графические пользовательские интерфейсы на основе виджетов, которые все чаще используются в настольных системах, как правило, не предоставляют особых возможностей для специализации и детальной настройки под конкретного пользователя. Поэтому в большинстве случаев приходится либо переписывать функцию отрисовки в экземпляре `QWidget` и обрабатывать каждый отображаемый пиксел виджета, либо применять специализацию на основе таблиц стилей.

Таблицы стилей (stylesheets) Qt позволяют тонко настраивать внешний вид отдельных виджетов и нюансы взаимодействия с ними даже в динамическом режиме. По существу, для их создания используется синтаксис каскадных таблиц стилей (cascading style sheet – CSS), применяемый в HTML-страницах. Таблицы стилей позволяют изменять элементы виджета, такие как границы, закругленные углы, а также толщину и цвет элементов.

QML

Qt Modeling Language (QML) – это язык разметки пользовательского интерфейса. Его основой является язык JavaScript, более того – используется даже встроенный (inline) JavaScript. QML можно применять для создания динамических и полностью настраиваемых специализированных пользовательских интерфейсов. Обычно QML используется в сочетании с модулем Qt Quick.

Немного позже в этой главе будет более подробно рассматриваться создание динамического графического пользовательского интерфейса.

3D Designer

В версии Qt5 появился модуль Qt 3D, предоставляющий прямой и удобный доступ к API рендеринга OpenGL. Новый модуль использовался как основа для графического редактора Qt 3D Designer и для создания соответствующей среды времени выполнения. Его также можно применять для создания высокодинамических графических пользовательских интерфейсов, позволяющих сочетать двумерные и трехмерные элементы.

Это в определенной степени похоже на создаваемые вручную графические пользовательские интерфейсы на основе языка QML, но такой подход предоставляет более удобно организованный рабочий поток, легкость добавления анимационных элементов и предварительный просмотр проекта. Это также напоминает Qt Designer Studio, в большей степени ориентированный на создание двумерных графических пользовательских интерфейсов, но этот инструмент не бесплатный, поэтому требуется приобретение лицензии на его использование.

Пример добавления графического пользовательского интерфейса в информационно-развлекательную систему

В этом примере будет использоваться язык C++, Qt и QML для создания графического пользовательского интерфейса (GUI), способного отображать текущий воспроизводимый аудиотрек, формировать визуализацию аудио, показывать время воспроизведения и обеспечивать возможность для пользователя переключаться в различные режимы ввода с помощью экранных кнопок.

Рассматриваемый здесь пример основан на примере *Audio Visualizer* из документации Qt, который можно найти в соответствующем подкаталоге установленного дистрибутива Qt (если были установлены примеры) или на сайте Qt: **https://doc.qt.io/qt-5/qt3d-audio-visualizer-qml-example.html**.

Основное различие между представленным здесь кодом и примером из документации состоит в том, что медиаплеер `QMediaPlayer` был перемещен в код C++ вместе с рядом других функций. Поэтому в рассматриваемом примере используется набор сигналов и слотов для связи между QML UI и внутренним кодом C++ в новом классе `QmlInterface` для обработки нажатий кнопок, обновления интерфейса и взаимодействия с медиаплеером.

Графический пользовательский интерфейс, представленный здесь, можно подключить к коду существующего проекта информационно-развлекательной системы для управления ее функциональностью и использования GUI в дополнение к интерфейсу, управляемому голосом.

GUI, созданный в рассматриваемом здесь примере, показан на рис. 10.1.

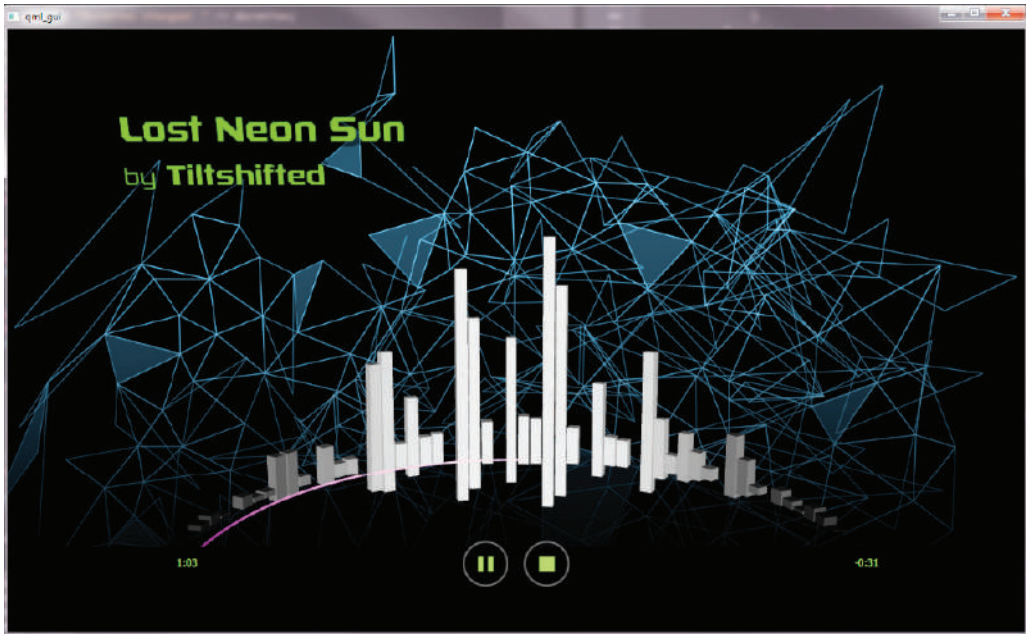

Рис. 10.1

Основной файл исходного кода (main)

Ниже приведен исходный код, содержащийся в основном файле.

```
#include "interface.h"
#include <QtGui/QGuiApplication>
#include <QtGui/QOpenGLContext>
#include <QtQuick/QQuickView>
#include <QtQuick/QQuickItem>
#include <QtQml/QQmlContext>
#include <QObject>

int main(int argc, char* argv[]) {
    QGuiApplication app(argc, argv);
    QSurfaceFormat format;
    if (QOpenGLContext::openGLModuleType() == QOpenGLContext::LibGL) {
        format.setVersion(3, 2);
        format.setProfile(QSurfaceFormat::CoreProfile);
    }
    format.setDepthBufferSize(24);
    format.setStencilBufferSize(8);
    QQuickView view;
    view.setFormat(format);
    view.create();
    QmlInterface qmlinterface;
    view.rootContext()->setContextProperty("qmlinterface", &qmlinterface);
    view.setSource(QUrl("qrc:/main.qml"));
    qmlinterface.setPlaying();

    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.setMaximumSize(QSize(1820, 1080));
    view.setMinimumSize(QSize(300, 150));
    view.show();

    return app.exec();
}
```

Специализированный класс добавляется в инструмент просмотра QML `QQuickView` как контекстный класс. Он работает в качестве прокси между QML UI и нашим кодом C++, как мы вскоре увидим. Сам просмотрщик использует поверхность OpenGL для отрисовки на ней графического интерфейса UI.

QmlInterface

В заголовочном файле специализированного класса описывается ряд дополнений для создания свойств и методов, которые должны быть видимыми в коде QML.

```
#include <QtCore/QObject>
#include <QMediaPlayer>
#include <QByteArray>

class QmlInterface : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString durationTotal READ getDurationTotal NOTIFY durationTotalChanged)
    Q_PROPERTY(QString durationLeft READ getDurationLeft NOTIFY durationLeftChanged)
```

Тег Q_PROPERTY сообщает синтаксическому анализатору (парсеру) qmake о том, что объявляемый здесь класс содержит свойство – переменную, которая должна быть видимой в коде QML, с параметрами, определяющими имя этой переменной, с методами, используемыми для чтения и записи значений этой переменной (если требуется), наконец, с определением сигнала, генерируемого при изменении данного свойства.

Это позволяет автоматически обновлять функцию настройки для поддержания синхронизации данного свойства в коде C++ и на стороне QML.

```
    QString formatDuration(qint64 milliseconds);
    QMediaPlayer mediaPlayer;
    QByteArray magnitudeArray;
    const int millisecondsPerBar = 68;
    QString durationTotal;
    QString durationLeft;
    qint64 trackDuration;

public:
    explicit QmlInterface(QObject *parent = nullptr);
    Q_INVOKABLE bool isHoverEnabled() const;
    Q_INVOKABLE void setPlaying();
    Q_INVOKABLE void setStopped();
    Q_INVOKABLE void setPaused();
    Q_INVOKABLE qint64 duration();
    Q_INVOKABLE qint64 position();
    Q_INVOKABLE double getNextAudioLevel(int offsetMs);
    QString getDurationTotal() { return durationTotal; }
    QString getDurationLeft() { return durationLeft; }

public slots:
    void mediaStatusChanged(QMediaPlayer::MediaStatus status);
    void durationChanged(qint64 duration);
    void positionChanged(qint64 position);

signals:
    void start();
    void stopped();
    void paused();
    void playing();
    void durationTotalChanged();
    void durationLeftChanged();
};
```

Точно так же тег Q_INVOKABLE гарантирует, что эти методы будут видимыми на стороне QML, следовательно, могут быть вызваны оттуда.

Ниже приведен код реализации специализированного класса.

```
#include "interface.h"
#include <QtGui/QTouchDevice>
#include <QDebug>
#include <QFile>
#include <QtMath>

QmlInterface::QmlInterface(QObject *parent) : QObject(parent) {
```

```
    // Установка трека для медиаплеера.
    mediaPlayer.setMedia(QUrl("qrc:/music/tiltshifted_lost_neon_sun.mp3"));
    // Загрузка файла визуализации для аудиотрека.
    QFile magFile(":/music/visualization.raw", this);
    magFile.open(QFile::ReadOnly);
    magnitudeArray = magFile.readAll();
    // Установка связей для медиаплеера.
    connect(&mediaPlayer, SIGNAL(mediaStatusChanged(QMediaPlayer::MediaStatus)), this,
                          SLOT(mediaStatusChanged(QMediaPlayer::MediaStatus)));
    connect(&mediaPlayer, SIGNAL(durationChanged(qint64)), this,
            SLOT(durationChanged(qint64)));
    connect(&mediaPlayer, SIGNAL(positionChanged(qint64)), this,
            SLOT(positionChanged(qint64)));
}
```

Конструктор существенно изменился по сравнению с исходным примером проекта: здесь создается экземпляр медиаплеера и устанавливаются его связи.

Загружается тот же самый музыкальный файл, который использовался в исходном проекте. При интеграции кода в реальный проект информационно-развлекательной или аналогичной системы рекомендуется сделать процедуру загрузки динамически настраиваемой. Кроме того, операция загрузки конкретного файла, предназначенного для визуализации амплитудных характеристик музыкального файла, также должна быть заменена на процедуру динамической настройки визуализации.

```
bool QmlInterface::isHoverEnabled() const {
#if defined(Q_OS_IOS) || defined(Q_OS_ANDROID) || defined(Q_OS_QNX) || defined(Q_OS_WINRT)
    return false;
#else
    bool isTouch = false;
    foreach (const QTouchDevice *dev, QTouchDevice::devices()) {
        if (dev->type() == QTouchDevice::TouchScreen) {
            isTouch = true;
            break;
        }
    }
    bool isMobile = false;
    if (qEnvironmentVariableIsSet("QT_QUICK_CONTROLS_MOBILE")) {
        isMobile = true;
    }
    return !isTouch && !isMobile;
#endif
}
```

Это был единственный метод, ранее существовавший в контекстном классе QML. Он использовался для того, чтобы определить, выполняется ли код на мобильном устройстве с сенсорным экраном.

```
void QmlInterface::setPlaying() {
   mediaPlayer.play();
}

void QmlInterface::setStopped() {
```

```
    mediaPlayer.stop();
}

void QmlInterface::setPaused() {
    mediaPlayer.pause();
}
```

Определяется группа методов управления, которые связываются с соответствующими кнопками в UI и обеспечивают управление экземпляром медиаплеера.

```
void QmlInterface::mediaStatusChanged(QMediaPlayer::MediaStatus status) {
    if (status == QMediaPlayer::EndOfMedia) {
        emit stopped();
    }
}
```

Этот слот-метод используется для определения достижения конца активного трека в медиаплеере, следовательно, необходимо оповестить UI и передать ему сигнал о том, что он должен обновить отображение трека.

```
void QmlInterface::durationChanged(qint64 duration) {
    qDebug() << "Duration changed: " << duration;
    durationTotal = formatDuration(duration);
    durationLeft = "-" + durationTotal;
    trackDuration = duration;
    emit start();
    emit durationTotalChanged();
    emit durationLeftChanged();
}

void QmlInterface::positionChanged(qint64 position) {
    qDebug() << "Position changed: " << position;
    durationLeft = "-" + formatDuration((trackDuration - position));
    emit durationLeftChanged();
}
```

Следующие два слот-метода связаны с экземпляром медиаплеера. Слот продолжительности требуется потому, что длина (продолжительность) нового загружаемого трека не становится доступной сразу после загрузки. Это асинхронно обновляемое свойство.

В результате необходимо ждать, пока медиаплеер не завершит операцию и не сгенерирует сигнал об успешном завершении этого процесса.

Далее для обновления времени, оставшегося до конца текущего трека, также постоянно считываются обновления текущей позиции из медиаплеера, поэтому имеется возможность обновления грфического интерфейса UI с использованием новых получаемых значений.

Оба свойства продолжительности и позиции обновляются в интерфейсе UI с использованием метода связывания (linkage), который мы видели выше в заголовочном файле для этого класса.

В конце генерируется сигнал `start()`, связанный со слотом в коде QML, который инициализирует процесс визуализации, как мы увидим несколько позже при рассмотрении кода QML.

```
qint64 QmlInterface::duration() {
    qDebug() << "Returning duration value: " << mediaPlayer.duration();
    return mediaPlayer.duration();
}

qint64 QmlInterface::position() {
    qDebug() << "Returning position value: " << mediaPlayer.position();
    return mediaPlayer.position();
}
```

Свойство продолжительности также используется кодом визуализации. Здесь обеспечивается прямое получение этого свойства. Точно так же доступно напрямую и свойство текущей позиции.

```
double QmlInterface::getNextAudioLevel(int offsetMs) {
    // Вычисление целочисленного индекса позиции в массиве величин амплитуды.
    qint64 index = ((mediaPlayer.position() + offsetMs) / millisecondsPerBar) | 0;
    if (index < 0 || index >= (magnitudeArray.length() / 2)) {
        return 0.0;
    }
    return (((quint16*) magnitudeArray.data())[index] / 63274.0);
}
```

Этот метод был перенесен из кода JavaScript в исходный проект. Он выполняет задачу определения уровня аудио на основе данных об амплитуде, ранее считанных из файла.

```
QString QmlInterface::formatDuration(qint64 milliseconds) {
    qint64 minutes = floor(milliseconds / 60000);
    milliseconds -= minutes * 60000;
    qint64 seconds = milliseconds / 1000;
    seconds = round(seconds);
    if (seconds < 10) {
        return QString::number(minutes) + ":0" + QString::number(seconds);
    }
    else {
        return QString::number(minutes) + ":" + QString::number(seconds);
    }
}
```

И этот метод также был перенесен из кода JavaScript исходного проекта, поскольку код, основанный на нем, перемещен в код C++. Он получает счетчик миллисекунд для продолжительности трека или счетчик позиции и выполняет преобразование в строку, содержащую минуты и секунды, соответствующие исходному значению.

QML

Продолжая изучать все, что было добавлено с использованием языка C++, рассмотрим теперь пользовательский интерфейс UI, сформированный на языке QML.

Сначала приводится содержимое основного файла QML.

```
import QtQuick 2.0
import QtQuick.Scene3D 2.0
```

```
import QtQuick.Layouts 1.2
import QtMultimedia 5.0

Item {
    id: mainview
    width: 1215
    height: 720
    visible: true
    property bool isHoverEnabled: false
    property int mediaLatencyOffset: 68
```

Файл QML состоит из иерархии элементов. Здесь определен элемент самого верхнего уровня, его размеры и имя (идентификатор).

```
    state: "stopped"
    states: [
        State {
            name: "playing"
            PropertyChanges {
                target: playButtonImage
                source: {
                    if (playButtonMouseArea.containsMouse)
                        "qrc:/images/pausehoverpressed.png"
                    else
                        "qrc:/images/pausenormal.png"
                }
            }
            PropertyChanges {
                target: stopButtonImage
                source: "qrc:/images/stopnormal.png"
            }
        },
        State {
            name: "paused"
            PropertyChanges {
                target: playButtonImage
                source: {
                    if (playButtonMouseArea.containsMouse)
                        "qrc:/images/playhoverpressed.png"
                    else
                        "qrc:/images/playnormal.png"
                }
            }
            PropertyChanges {
                target: stopButtonImage
                source: "qrc:/images/stopnormal.png"
            }
        },
        State {
            name: "stopped"
            PropertyChanges {
                target: playButtonImage
                source: "qrc:/images/playnormal.png"
```

```
            }
            PropertyChanges {
                target: stopButtonImage
                source: "qrc:/images/stopdisabled.png"
            }
        }
    ]
```

Далее определяется ряд состояний пользовательского интерфейса и соответствующие изменения, которые должны быть сгенерированы, если состояние изменилось.

```
    Connections {
        target: qmlinterface
        onStopped: mainview.state = "stopped"
        onPaused: mainview.state = "paused"
        onPlaying: mainview.state = "started"
        onStart: visualizer.startVisualization()
    }
```

Определяются связи (соединения), которые устанавливаются между сигналами со стороны кода C++ и локальным обработчиком. Мы устанавливаем наш специализированный класс как источник этих сигналов, затем определяем обработчик для каждого типа сигнала, требующего обработки, с указанием соответствующего префикса и добавлением указателя на код, который должен быть выполнен.

Здесь можно видеть, что сигнал запуска связан с обработчиком, активизирующим функцию в модуле визуализации, которая запускает этот модуль.

```
    Component.onCompleted: isHoverEnabled = qmlinterface.isHoverEnabled()

    Image {
        id: coverImage
        anchors.fill: parent
        source: "qrc:/images/albumcover.png"
    }
```

Элемент `Image` определяет фоновое изображение, загружаемое из ресурсов, которые добавляются в выполняемый файл при сборке проекта.

```
    Scene3D {
        anchors.fill: parent
        Visualizer {
            id: visualizer
            animationState: mainview.state
            numberOfBars: 120
            barRotationTimeMs: 8160 // 68 миллисекунд на каждый элемент анимации
        }
    }
```

Трехмерная сцена, которая будет заполняться содержимым определяемого здесь визуализатора.

```
    Rectangle {
        id: blackBottomRect
        color: "black"
```

```
        width: parent.width
        height: 0.14 * mainview.height
        anchors.bottom: parent.bottom
    }

    Text {
        text: qmlinterface.durationTotal
        color: "#80C342"
        x: parent.width / 6
        y: mainview.height - mainview.height / 8
        font.pixelSize: 12
    }

    Text {
        text: qmlinterface.durationLeft
        color: "#80C342"
        x: parent.width - parent.width / 6
        y: mainview.height - mainview.height / 8
        font.pixelSize: 12
    }
```

Далее определяются два текстовых элемента, связанных со свойством нашего специализированного класса C++, как мы видели выше. Эти значения будут постоянно обновляться значением из экземпляра данного класса в соответствии с происходящими изменениями.

```
property int buttonHorizontalMargin: 10
Rectangle {
    id: playButton
    height: 54
    width: 54
    anchors.bottom: parent.bottom
    anchors.bottomMargin: width
    x: parent.width / 2 - width - buttonHorizontalMargin
    color: "transparent"

    Image {
        id: playButtonImage
        source: "qrc:/images/pausenormal.png"
    }

    MouseArea {
        id: playButtonMouseArea
        anchors.fill: parent
        hoverEnabled: isHoverEnabled
        onClicked: {
            if (mainview.state == 'paused' || mainview.state == 'stopped')
                mainview.state = 'playing'
            else
                mainview.state = 'paused'
        }
        onEntered: {
            if (mainview.state == 'playing')
                playButtonImage.source = "qrc:/images/pausehoverpressed.png"
```

```
                else
                    playButtonImage.source = "qrc:/images/playhoverpressed.png"
            }
            onExited: {
                if (mainview.state == 'playing')
                    playButtonImage.source = "qrc:/images/pausenormal.png"
                else
                    playButtonImage.source = "qrc:/images/playnormal.png"
            }
        }
    }

    Rectangle {
        id: stopButton
        height: 54
        width: 54
        anchors.bottom: parent.bottom
        anchors.bottomMargin: width
        x: parent.width / 2 + buttonHorizontalMargin
        color: "transparent"

        Image {
            id: stopButtonImage
            source: "qrc:/images/stopnormal.png"
        }

        MouseArea {
            anchors.fill: parent
            hoverEnabled: isHoverEnabled
            onClicked: mainview.state = 'stopped'
            onEntered: {
                if (mainview.state != 'stopped')
                    stopButtonImage.source = "qrc:/images/stophoverpressed.png"
            }
            onExited: {
                if (mainview.state != 'stopped')
                    stopButtonImage.source = "qrc:/images/stopnormal.png"
            }
        }
    }
}
```

Оставшаяся часть этого файла предназначена для настройки отдельных кнопок, управляющих воспроизведением аудио, то есть кнопок воспроизведения, останова и паузы, которые меняются местами при необходимости.

Теперь рассмотрим содержимое файла для определения гистограммы амплитуды.

```
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Extras 2.0
import QtQuick 2.4 as QQ2

Entity {
```

```
property int rotationTimeMs: 0
property int entityIndex: 0
property int entityCount: 0
property int startAngle: 0 + 360 / entityCount * entityIndex
property bool needsNewMagnitude: true
property real magnitude: 0
property real animWeight: 0

property color lowColor: "black"
property color highColor: "#b3b3b3"
property color barColor: lowColor

property string entityAnimationsState: "stopped"
property bool entityAnimationsPlaying: true

property var entityMesh: null
```

Определяется группа свойств до перехода к обработчику изменения состояния анимации амплитуды.

```
onEntityAnimationsStateChanged: {
    if (animationState == "paused") {
        if (angleAnimation.running)
            angleAnimation.pause()
        if (barColorAnimations.running)
            barColorAnimations.pause()
    } else if (animationState == "playing"){
        needsNewMagnitude = true;
        if (heightDecreaseAnimation.running)
            heightDecreaseAnimation.stop()
        if (angleAnimation.paused) {
            angleAnimation.resume()
        } else if (!entityAnimationsPlaying) {
            magnitude = 0
            angleAnimation.start()
            entityAnimationsPlaying = true
        }
        if (barColorAnimations.paused)
            barColorAnimations.resume()
    } else {
        if (animWeight != 0)
            heightDecreaseAnimation.start()
        needsNewMagnitude = true
        angleAnimation.stop()
        barColorAnimations.stop()
        entityAnimationsPlaying = false
    }
}
```

Каждый раз при останове, паузе или запуске воспроизведения аудио анимация должна быть обновлена, чтобы соответствовать текущему измененному состоянию.

```
property Material barMaterial: PhongMaterial {
    diffuse: barColor
```

```
        ambient: Qt.darker(barColor)
        specular: "black"
        shininess: 1
    }
```

Определяется внешний вид полос гистограммы амплитуды с использованием типа закраски Phong.

```
    property Transform angleTransform: Transform {
        property real heightIncrease: magnitude * animWeight
        property real barAngle: startAngle

        matrix: {
            var m = Qt.matrix4x4()
            m.rotate(barAngle, Qt.vector3d(0, 1, 0))
            m.translate(Qt.vector3d(1.1, heightIncrease / 2 - heightIncrease * 0.05, 0))
            m.scale(Qt.vector3d(0.5, heightIncrease * 15, 0.5))
            return m;
        }

        property real compareAngle: barAngle
        onBarAngleChanged: {
            compareAngle = barAngle
            if (compareAngle > 360)
                compareAngle = barAngle - 360
            if (compareAngle > 180) {
                parent.enabled = false
                animWeight = 0
                if (needsNewMagnitude) {
                    // Вычисление смещения в мс, когда полоса будет в центральной точке
                    // визуализации и вовремя получит правильную величину для этой точки.
                    var offset = (90.0 + 360.0 - compareAngle) * (rotationTimeMs / 360.0)
                    magnitude = qmlinterface.getNextAudioLevel(offset)
                    needsNewMagnitude = false
                }
            } else {
                parent.enabled = true
                // Вычисление кривой 2 порядка для анимации гистограммы с максимумом
                // на 90 градусах
                animWeight = Math.min((compareAngle / 90), (180 - compareAngle) / 90)
                animWeight = animWeight * animWeight
                if (!needsNewMagnitude) {
                    needsNewMagnitude = true
                    barColorAnimations.start()
                }
            }
        }
    }
```

Поскольку полосы гистограммы амплитуды перемещаются по экрану, они изменяют свое положение относительно камеры, поэтому необходимо постоянно вычислять новый угол и высоту выводимых элементов.

В этом разделе также заменяется исходное обращение к методу регулирования уровня аудио на вызов нового метода из нашего специализированного класса C++.

```
    components: [entityMesh, barMaterial, angleTransform]

    QQ2.NumberAnimation {
        id: angleAnimation
        target: angleTransform
        property: "barAngle"
        duration: rotationTimeMs
        loops: QQ2.Animation.Infinite
        running: true
        from: startAngle
        to: 360 + startAngle
    }

    QQ2.NumberAnimation {
        id: heightDecreaseAnimation
        target: angleTransform
        property: "heightIncrease"
        duration: 400
        running: false
        from: angleTransform.heightIncrease
        to: 0
        onStopped: barColor = lowColor
    }

    property int animationDuration: angleAnimation.duration / 6

    QQ2.SequentialAnimation on barColor {
        id: barColorAnimations
        running: false

        QQ2.ColorAnimation {
            from: lowColor
            to: highColor
            duration: animationDuration
        }

        QQ2.PauseAnimation {
            duration: animationDuration
        }

        QQ2.ColorAnimation {
            from: highColor
            to: lowColor
            duration: animationDuration
        }
    }
}
```

Заключительная часть файла содержит описание еще нескольких анимационных преобразований.

Последним рассматривается содержимое модуля визуализации.

```
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Extras 2.0
import QtQuick 2.2 as QQ2
```

```
Entity {
    id: sceneRoot
    property int barRotationTimeMs: 1
    property int numberOfBars: 1
    property string animationState: "stopped"
    property real titleStartAngle: 95
    property real titleStopAngle: 5

    onAnimationStateChanged: {
        if (animationState == "playing") {
            qmlinterface.setPlaying();
            if (progressTransformAnimation.paused)
                progressTransformAnimation.resume()
            else
                progressTransformAnimation.start()
        } else if (animationState == "paused") {
            qmlinterface.setPaused();
            if (progressTransformAnimation.running)
                progressTransformAnimation.pause()
        } else {
            qmlinterface.setStopped();
            progressTransformAnimation.stop()
            progressTransform.progressAngle = progressTransform.defaultStartAngle
        }
    }
```

В этом разделе взаимодействие с локальным экземпляром медиаплеера заменено на взаимодействие с новым экземпляром из кода C++. Все прочее остается неизменным. Это основной обработчик всевозможных изменений в визуальной сцене, происходящих из-за взаимодействий с пользователем, при запуске либо останове аудиотрека.

```
    QQ2.Item {
        id: stateItem

        state: animationState
        states: [
            QQ2.State {
                name: "playing"
                QQ2.PropertyChanges {
                    target: titlePrism
                    titleAngle: titleStopAngle
                }
            },
            QQ2.State {
                name: "paused"
                QQ2.PropertyChanges {
                    target: titlePrism
                    titleAngle: titleStopAngle
                }
            },
            QQ2.State {
                name: "stopped"
```

```
            QQ2.PropertyChanges {
                target: titlePrism
                titleAngle: titleStartAngle
            }
        }
    ]

    transitions: QQ2.Transition {
        QQ2.NumberAnimation {
            property: "titleAngle"
            duration: 2000
            running: false
        }
    }
}
```

Изменяется группа свойств, и определяются преобразования для объекта названия трека.

```
function startVisualization() {
    progressTransformAnimation.duration = qmlinterface.duration()
    mainview.state = "playing"
    progressTransformAnimation.start()
}
```

Эта функция инициализирует всю последовательность визуализации. Она использует значение продолжительности трека, полученное от экземпляра класса C++, для определения размеров индикатора воспроизведения и его анимации до начала процесса визуализации анимации в целом.

```
Camera {
    id: camera
    projectionType: CameraLens.PerspectiveProjection
    fieldOfView: 45
    aspectRatio: 1820 / 1080
    nearPlane: 0.1
    farPlane: 1000.0
    position: Qt.vector3d(0.014, 0.956, 2.178)
    upVector: Qt.vector3d(0.0, 1.0, 0.0)
    viewCenter: Qt.vector3d(0.0, 0.7, 0.0)
}
```

Определяется камера для трехмерной сцены.

```
Entity {
    components: [
        DirectionalLight {
            intensity: 0.9
            worldDirection: Qt.vector3d(0, 0.6, -1)
        }
    ]
}

RenderSettings {
    id: external_forward_renderer
```

```
    activeFrameGraph: ForwardRenderer {
        camera: camera
        clearColor: "transparent"
    }
}
```

Создается механизм отрисовки (рендеринга) и освещения сцены.

```
components: [external_forward_renderer]

CuboidMesh {
    id: barMesh
    xExtent: 0.1
    yExtent: 0.1
    zExtent: 0.1
}
```

Формируется сетка для гистограммы амплитуды.

```
NodeInstantiator {
    id: collection
    property int maxCount: parent.numberOfBars
    model: maxCount

    delegate: BarEntity {
        id: cubicEntity
        entityMesh: barMesh
        rotationTimeMs: sceneRoot.barRotationTimeMs
        entityIndex: index
        entityCount: sceneRoot.numberOfBars
        entityAnimationsState: animationState
        magnitude: 0
    }
}
```

Определяется количество полос гистограммы и некоторые другие свойства.

```
Entity {
    id: titlePrism
    property real titleAngle: titleStartAngle

    Entity {
        id: titlePlane

        PlaneMesh {
            id: titlePlaneMesh
            width: 550
            height: 100
        }

        Transform {
            id: titlePlaneTransform
            scale: 0.003
            translation: Qt.vector3d(0, 0.11, 0)
        }

        NormalDiffuseMapAlphaMaterial {
```

```
            id: titlePlaneMaterial
            diffuse: TextureLoader { source: "qrc:/images/demotitle.png" }
            normal: TextureLoader { source: "qrc:/images/normalmap.png" }
            shininess: 1.0
        }

        components: [titlePlaneMesh, titlePlaneMaterial, titlePlaneTransform]
    }
```

Определяемая здесь плоскость содержит объект названия при отсутствии воспроизводимого трека.

```
    Entity {
        id: songTitlePlane

        PlaneMesh {
            id: songPlaneMesh
            width: 550
            height: 100
        }

        Transform {
            id: songPlaneTransform
            scale: 0.003
            rotationX: 90
            translation: Qt.vector3d(0, -0.03, 0.13)
        }

        property Material songPlaneMaterial: NormalDiffuseMapAlphaMaterial {
            diffuse: TextureLoader { source: "qrc:/images/songtitle.png" }
            normal: TextureLoader { source: "qrc:/images/normalmap.png"}
            shininess: 1.0
        }

        components: [songPlaneMesh, songPlaneMaterial, songPlaneTransform]
    }
```

Следующая плоскость содержит название композиции, когда трек активен.

```
    property Transform titlePrismPlaneTransform: Transform {
        matrix: {
            var m = Qt.matrix4x4()
            m.translate(Qt.vector3d(-0.5, 1.3, -0.4))
            m.rotate(titlePrism.titleAngle, Qt.vector3d(1, 0, 0))
            return m;
        }
    }

    components: [titlePlane, songTitlePlane, titlePrismPlaneTransform]
}
```

Эта функция используется для преобразования плоскости воспроизведения в плоскость отсутствия воспроизведения и обратно.

```
Mesh {
    id: circleMesh
    source: "qrc:/meshes/circle.obj"
```

```
    }

    Entity {
        id: circleEntity
        property Material circleMaterial: PhongAlphaMaterial {
            alpha: 0.4
            ambient: "black"
            diffuse: "black"
            specular: "black"
            shininess: 10000
        }

        components: [circleMesh, circleMaterial]
    }
```

Добавляется круговая сетка, обеспечивающая эффект отражения.

```
    Mesh {
        id: progressMesh
        source: "qrc:/meshes/progressbar.obj"
    }

    Transform {
        id: progressTransform
        property real defaultStartAngle: -90
        property real progressAngle: defaultStartAngle
        rotationY: progressAngle
    }

    Entity {
        property Material progressMaterial: PhongMaterial {
            ambient: "purple"
            diffuse: "white"
        }
        components: [progressMesh, progressMaterial, progressTransform]
    }

    QQ2.NumberAnimation {
        id: progressTransformAnimation
        target: progressTransform
        property: "progressAngle"
        duration: 0
        running: false
        from: progressTransform.defaultStartAngle
        to: -270
        onStopped: if (animationState != "stopped") animationState = "stopped"
    }
}
```

В завершающей части файла создается сетка, формирующая индикатор воспроизведения, который перемещается слева направо, отображая продолжительность уже воспроизведенной части аудиофайла.

Весь проект в целом компилируется при выполнении команды `qmake` и последующем выполнении команды `make`. Другой вариант – открыть среду разработки Qt Creator и произвести сборку проекта в ней. После запуска приложение авто-

матически начнет воспроизведение жестко закодированной композиции и демонстрацию визуализации с возможностью управления кнопками графического пользовательского интерфейса.

Резюме

В этой главе рассматривались различные способы использования рабочей среды Qt при разработке встроенных систем. Рабочая среда Qt сравнивалась с другими рабочими средами, описывались возможности оптимизации Qt для встраиваемых платформ. Далее подробно рассматривался пример разработки графического пользовательского интерфейса на основе языка разметки QML, которые можно добавить в ранее созданную информационно-развлекательную систему.

Предполагается, что после изучения материала этой главы читатель сможет самостоятельно создавать простые Qt-приложения как с интерфейсом командной строки, так и с графическим пользовательским интерфейсом. Кроме того, должен быть совершенно понятным основной принцип разработки GUI, предлагаемый рабочей средой Qt.

В следующей главе будет рассматриваться следующая ступень развития встраиваемых платформ с использованием программируемых пользователем вентильных матриц (field-programmable gate arrays – FPGA) для добавления специализированной функциональности на основе аппаратуры для ускорения работы встраиваемых платформ.

Глава 11

Разработка для гибридных систем SoC/FPGA

В дополнение к стандартным встроенным системам на основе центрального процессора (CPU) стал широко применяться метод комбинирования ЦП в форме объединения систем на кристалле (SoC) с программируемыми пользователями вентильными матрицами (field programmable gate arrays – FGPA). Это позволяет реализовать алгоритмы и методы обработки, весьма интенсивно использующие ЦП, в том числе методы числовой обработки сигналов (DSP) и обработки изображений, на FPGA-части системы. Для ЦП остаются менее интенсивные задачи, такие как взаимодействие с пользователем, организация хранения данных и сетевые функции.

В этой главе рассматриваются следующие темы:

- методы обмена информацией со стороны FPGA в гибридной системе FPGA/SoC;
- изучение методик реализации разнообразных алгоритмов в FPGA и использования их на стороне SoC;
- пример реализации простого осциллографа на основе гибридной системы FPGA/SoC.

Организация исключительно параллельного выполнения

При возникновении вопросов и проблем, связанных с производительностью, выполнение одной инструкции на одноядерном процессоре в единицу времени, по существу, представляет собой самый медленный способ реализации алгоритма или какой-либо другой функциональности. Существует возможность разделить такой одиночный поток выполнения на несколько потоков, использующих одновременное планирование выполнения на единственном ядре процессора, как отдельных функциональных элементов.

Следующий шаг к увеличению производительности – добавление ядер процессора, что, разумеется, усложняет планирование и создает проблему потенциальных задержек, когда выполнение наиболее важных задач откладывается из-за того, что менее важные задачи блокируют требуемые ресурсы. Использование процессоров общего назначения также существенно ограничено кругом определенных задач, особенно тех, для которых распараллеливание затруднительно.

Для задач, в которых необходимо обрабатывать единственный крупный набор данных с использованием того же алгоритма, который применяется к каждому элементу в этом наборе, стало весьма распространенным использование метода неспециализированных вычислений на графических процессорах (general-purpose graphical processor unit-based – GPGPU) в сочетании с применением процессоров обработки цифровых сигналов (Digital Signal Processors – DSP) для значительного ускорения больших блоков операций с помощью специализированного аппаратного оборудования.

С другой стороны, существуют задачи, которые по своей сущности являются параллельными, но включают множество разнородных операций, выполняемых над входящими данными, внутренними данными или обоими этими типами данных. Это уровень сложности, на котором становится весьма трудно обеспечить приемлемую производительность, если реализация выполняется исключительно на основе программного обеспечения, предназначенного для ограниченного диапазона ядер микропроцессоров.

Здесь может помочь использование дорогостоящего оборудования для цифровой обработки сигналов (DSP), но даже такой подход не является оптимальным для решения задачи производительности. Обычно в подобных ситуациях компания может расматривать вариант применения специализированной интегральной микросхемы (integrated circuit – IC), спроектированной и выполненной как интегральная схема специального назначения (application-specific integrated circuit – ASIC). Но затраты на это чрезвычайно высоки, и такой подход становится обоснованным только при крупномасштабном производстве, когда есть возможность конкуренции с другими вариантами.

Были разработаны различные решения, позволяющие практически реализовать подобную специализированную аппаратуру. Одно из таких решений – микросхема с программируемой логикой. Например, система, подобная Commodore 64, содержала программируемую логическую матрицу (programmable logic array – PLA), то есть микросхему, представляющую собой однократно программируемый массив элементов комбинационной логики. Это позволяло процессору переконфигурировать встроенные подпрограммы управления адресной шины, чтобы изменить место расположения в активном адресном пространстве блоков микросхем памяти DRAM, микросхем ПЗУ и прочих периферийных устройств.

После программирования такой логической матрицы PLA она работала точно так же, как большинство микросхем 74-logic (микросхемы дискретной логики), но для такого дискретного решения требовалась некоторая часть пространства. Применяемый подход, по существу, придавал системе Commodore собственную специализированную интегральную схему ASIC, но без дополнительных затрат на проектирование и производство. Фактически эта микросхема использовалась как компонент «из коробки» без каких-либо ограничений на внесение усовершенствований в логику, прошитую в микросхеме PLA, в течение всего жизненного цикла Commodore 64.

Со временем микросхемы PLA (также называемые PAL – Programmable Array Logic[1]) стали более продвинутыми, что привело к разработке программируемых

[1] Следует отметить, что в Википедии не рекомендуется путать PLA и PAL: «Not to be confused with Programmable logic array»; **https://en.wikipedia.org/wiki/Programmable_Array_Logic**. – *Прим. перев.*

логических интегральных схем (ПЛИС; Complex Programmable Logic Devices – CPLD), основанных на матрицах макроячеек, позволяющих реализовывать более развитые функциональные возможности по сравнению с простой комбинационной логикой. В конечном итоге ПЛИС эволюционировали в программируемые пользователем вентильные матрицы (ППВМ; field-programmable gate array – FPGA), которые добавили еще больше развитых функциональных возможностей и периферийных устройств.

В настоящее время ППВМ можно найти практически везде, где требуется какой-либо тип расширенной обработки данных и управления. В оборудовании для обработки видео и аудио контента используются ППВМ вместе с микросхемами обработки цифровых сигналов (DSP), микроконтроллерами и системами на кристаллах (SoC) для поддержки пользовательского интерфейса и прочей низкоприоритетной функциональности.

Устройства, подобные осциллографам, сейчас реализованы с помощью аналогового (и цифрового, если обеспечена поддержка) внешнего звена, микросхемы DSP выполняют предварительное преобразование исходных получаемых данных и начальную их обработку до передачи в одну или несколько ППВМ, выполняющих дальнейшую обработку и анализ данных. После обработки данные могут быть сохранены в буфере (компонент «цифровое хранилище» цифрового запоминающего осциллографа, ЦЗО; digital storage oscilloscope – DSO) или переданы во внешнее звено, где программное обеспечение системы на кристалле выполнит их визуальное отображение в пользовательском интерфейсе и позволит пользователю вводить команды, управляющие отображенными данными.

В этой главе рассматривается проект простого осциллографа, реализованного с использованием несложной аппаратуры и программируемой пользователем вентильной матрицы (ППВМ) с применением кода на языке VHDL.

Языки описания аппаратуры

Поскольку сложность сверхбольших интегральных схем (СБИС; very large scale integrated – VLSI) возросла за последние десятилетия, все большую важность приобретает поиск методов усовершенствования процесса разработки, включая возможность верификации проектного решения. Это привело к разработке языков описания аппаратуры (hardware description languages – HDL), среди которых в настоящее время наиболее широко используемыми являются VHDL и Verilog.

Главная цель языков описания аппаратуры – позволить разработчику с легкостью описывать аппаратные схемы такого типа, который можно было бы напрямую внедрять в интегральные схемы специального назначения (ASIC) или использовать для внутреннего программирования ППВМ. Кроме того, языки описания аппаратуры дают возможность имитировать проектное решение и проверять правильность его функционирования.

В этой главе будет рассматриваться пример с использованием языка VHDL для той части программирования, которая должна быть реализована внутри ППВМ. Язык VHSIC Hardware Description Language (VHDL) появился в 1983 году как разработка министерства обороны США. Он был предназначен для документирования поведения интегральных схем специального назначения, предоставляемых

поставщиками электронного оборудования.

Через некоторое время возникла идея возможного использования файлов документации для имитации поведения интегральных схем специального назначения. Вскоре после осуществления этой идеи последовала разработка инструментальных средств синтеза для функциональной аппаратной реализации, которую можно было бы использовать для создания самих интегральных схем специального назначения.

Язык VHDL основан главным образом на языке программирования Ada, корни которого также обнаруживаются в военных ведомствах США. Несмотря на то что VHDL в первую очередь используется как язык описания аппаратуры, его также можно применять как обычный язык программирования, во многом похожий на Ada и подобные языки.

Архитектура ППВМ

Несмотря на то что не все ППВМ имеют одинаковую структуру, общий принцип остается неизменным: это матрицы логических элементов, которые могут быть сконфигурированы для формирования специализированных схем. Таким образом, сложность этих логических элементов (ЛЭ; logic elements – LE) определяет тип формируемых логических схем. Именно этот аспект принимается во внимание при написании кода VHDL для специализированной архитектуры ППВМ.

Термины логические элементы (ЛЭ; logic elements – LE) и логические ячейки (ЛЯ; logic cells – LC) используются как взаимозаменяемые. Логические элементы содержат одну или несколько таблиц поиска (lookup table – LUT) или таблиц истинности. Обычно таблица истинности имеет от четырех до шести входов. Вне зависимости от точной конфигурации каждый логический элемент окружен логическими связями (контактами), позволяющими соединять логические элементы друг с другом. Для самих логических элементов программируется специализированная конфигурация. Таким образом, формируется требуемая общая логическая схема.

К потенциальным недостаткам разработок для ППВМ относится строгое предположение производителей ППВМ о том, что ППВМ будут использоваться в синхронизированных проектных решениях (с использованием центрального источника синхронизации (таймера) и доменов синхронизации), а не с применением комбинационной (несинхронизированной) логики. Вообще говоря, лучше всего заранее изучить целевую систему ППВМ до включения ее в новый проект, чтобы узнать, в какой степени она способна поддерживать требуемые функциональные возможности.

Гибридные микросхемы FPGA/SoC

Несмотря на то что системы, содержащие ППВМ (FPGA) и SoC, чрезвычайно широко используются уже в течение многих лет, в последнее время появились комбинированные микросхемы FPGA/SoC, содержащие интегральные пластины (заготовки) и для ППВМ, и для системы на кристалле в одном корпусе. Затем они

соединяются с помощью шины и могут эффективно обмениваться данными друг с другом, используя отображаемый в память ввод/вывод или аналогичную схему.

Широко известными примерами таких ППВМ в настоящее время являются Altera (теперь Intel), Cyclone V SoC и Xilinx Zynq. Показанная на рис. 11.1 блок-схема Cyclone V SoC, взятая из официальной спецификации, дает общее представление о работе подобной системы.

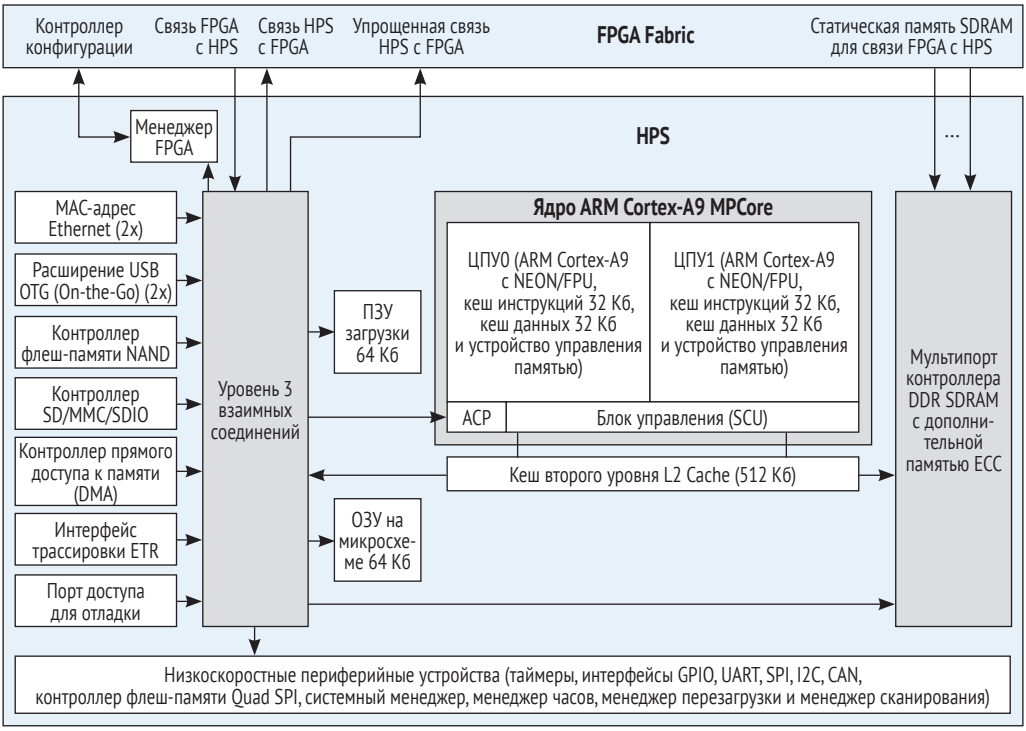

Рис. 11.1

Здесь можно видеть несколько возможных вариантов взаимодействия и обмена информацией между системой процессора Hard Processor System (HPS) и ППВМ, например через совместно используемый контроллер статической памяти SDRAM, через два канала типа «точка-точка» (point-to-point) и через ряд других интерфейсов. Для системы Cyclone V SoC либо сторона ППВМ, либо сторона SoC может быть исходным пунктом загрузки при запуске системы, позволяя использовать широкий набор параметров системной конфигурации.

Пример: простой осциллограф

Этот пример демонстрирует обобщенную методику практического использования ППВМ в проекте встроенной системы. Здесь ППВМ используется для сбора входных данных и измерения напряжения или аналогичной характеристики, которую можно измерять с помощью осциллографа. Затем преобразованные с по-

мощью аналого-цифрового преобразователя (ADC) данные передаются по последовательному каналу в приложение, написанное на C++/Qt, которое визуально отображает полученные данные.

Аппаратура

В рассматриваемом здесь проекте используется плата Fleasystems FleaFPGA Ohm (**http://fleasystems.com/fleaFPGA_Ohm.html**). Это компактная дешевая (цена менее 50 долл., менее 40 евро) макетная плата ППВМ с форм-фактором Raspberry Pi Zero (рис. 11.2).

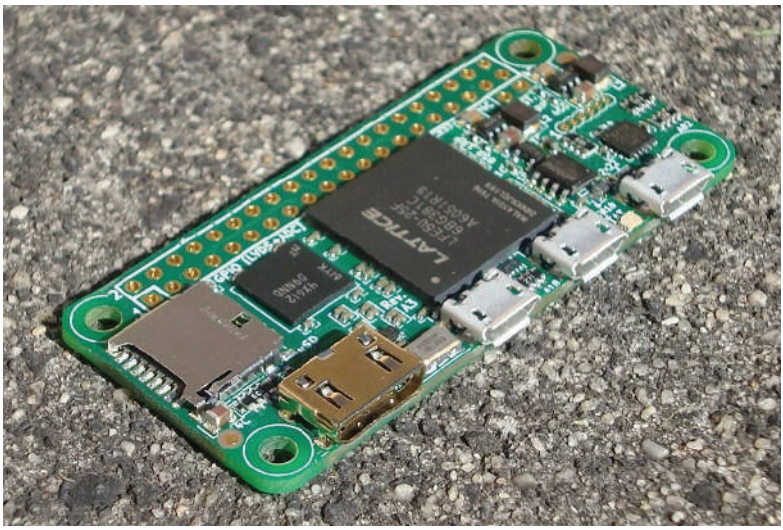

Рис. 11.2

Основные характеристики платы:

- ППВМ Lattice ECP5 FPGA с 24K элементами таблиц истинности (поиска) и блоком ОЗУ 112 Кб;
- статическая память SDRAM 256 Мбит с длиной слов 16 бит и таймером 167 МГц;
- флеш-ПЗУ SPI 8 Мбит для хранилища конфигурации ППВМ;
- кристалл осциллографа 25 МГц;
- выход видео HDMI (с возможными режимами 1080p30 или 720p60);
- слот для карты твердотельного диска µSD;
- два порта для хоста микро-USB с поддержкой альтернативной функциональности хост-порта PS/2;
- 29 контактов пользовательского интерфейса GPIO, включая 4× среднескоростных входа ADC и 12× сигнальных пар LVDS, доступных с 40-контактного расширения (совместимого с Raspberry Pi), и 2-контактные направляющие разъемы перезагрузки соответственно;
- один подчиненный порт микро-USB. Предоставляет +5 В электропитание для Ohm, последовательные каналы обмена данными консоль/UART, а так-

же доступ к встроенному программному интерфейсу JTAG (для конфигурирования ППВМ ECP5);

- поддержка внешней подключаемой программируемой панели JTAG для обеспечения отладки в реальном времени.

К этой плате подключается электрическая схема, показанная на рис. 11.3, которая позволяет воспользоваться осциллографом.

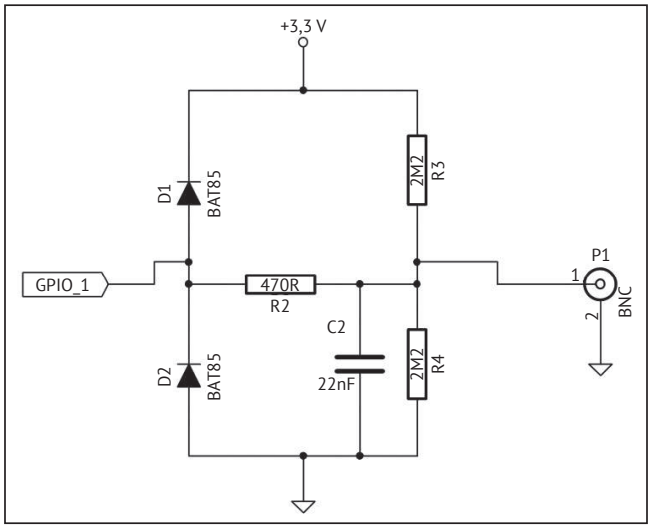

Рис. 11.3

Эта схема подключается к контакту с номером 29 в разъеме интерфейса GPIO платы Ohm, соответствующего контакту GPIO 5. Это позволяет измерять сигналы от 0 до 3 В постоянного тока и сигналы 1.5 В переменного тока (RMS) в режиме измерений 1× датчика осциллографа. Ширина измеряемого диапазона немного больше 10 Мгц.

Код VHDL

В этом разделе рассматривается объект верхнего уровня в проекте VHDL для получения представления о его функционировании. Сначала в проект включается стандартная библиотека, как показано ниже.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;
use IEEE.numeric_std.all;

entity FleaFPGA_Ohm_A5 is
   port(
   sys_clock       : in        std_logic;  -- вход таймера 25MHz clock с внешнего осциллографа
                                           xtal.
   sys_reset       : in        std_logic;  -- вход управления перезагрузкой с контакта
                                           перезагрузки.
```

Здесь устанавливается соответствие с системным таймером и линией перезагрузки ППВМ (это более низкий, аппаратный уровень). Также можно видеть, как осуществляется отображение на порт с определением направления объекта порта и его типа. В рассматриваемом примере тип порта `std_logic`, то есть стандартный логический сигнал в форме бинарной единицы или нуля.

```
n_led1          : buffer    std_logic;
LVDS_Red        : out       std_logic_vector(0 downto 0);
LVDS_Green      : out       std_logic_vector(0 downto 0);
LVDS_Blue       : out       std_logic_vector(0 downto 0);
LVDS_ck         : out       std_logic_vector(0 downto 0);
slave_tx_o      : out       std_logic;
slave_rx_i      : in        std_logic;
slave_cts_i     : in        std_logic;  -- Прием сигнала от контакта #RTS на FT230x
```

Также используется световой индикатор (LED) состояния на плате, отображение контактов видео HDMI (сигналы LVDS) и интерфейс UART, который пользуется микросхемой FDTI USB-UART на плате. Эта микросхема будет применяться для передачи данных из ППВМ в приложение на языке C++.

Далее определяется отображение соответствий контактов разъема, совместимого с Raspberry Pi, как показано ниже.

```
GPIO_2              : inout         std_logic;
GPIO_3              : inout         std_logic;
GPIO_4              : inout         std_logic;
-- GPIO_5           : inout         std_logic;
GPIO_6              : inout         std_logic;
GPIO_7              : inout         std_logic;
GPIO_8              : inout         std_logic;
GPIO_9              : inout         std_logic;
GPIO_10             : inout         std_logic;
GPIO_11             : inout         std_logic;
GPIO_12             : inout         std_logic;
GPIO_13             : inout         std_logic;
GPIO_14             : inout         std_logic;
GPIO_15             : inout         std_logic;
GPIO_16             : inout         std_logic;
GPIO_17             : inout         std_logic;
GPIO_18             : inout         std_logic;
GPIO_19             : inout         std_logic;
GPIO_20             : in            std_logic;
GPIO_21             : in            std_logic;
GPIO_22             : inout         std_logic;
GPIO_23             : inout         std_logic;
GPIO_24             : inout         std_logic;
GPIO_25             : inout         std_logic;
GPIO_26             : inout         std_logic;
GPIO_27             : inout         std_logic;
GPIO_IDSD           : inout         std_logic;
GPIO_IDSC           : inout         std_logic;
```

Отображение контакта GPIO 5 закомментировано, потому что предполагается его использование для обеспечения функциональности ADC, а не в качестве контакта ввода/вывода общего назначения.

Таким образом, контакт GPIO 5 назначается для обеспечения работы периферийного устройства, совместимого с сигма-дельта модулятором АЦП ADC3, как показано ниже.

```
   --ADC0_input      : in        std_logic;
   --ADC0_error      : buffer    std_logic;
   --ADC1_input      : in        std_logic;
   --ADC1_error      : buffer    std_logic;
   --ADC2_input      : in        std_logic;
   --ADC2_error      : buffer    std_logic;
   ADC3_input  : in        std_logic;
   ADC3_error  : buffer    std_logic;
```

Здесь можно видеть, что имеются еще три периферийных устройства АЦП, которые при необходимости можно было бы использовать для добавления дополнительных каналов к осциллографу.

```
   mmc_dat1          : in        std_logic;
   mmc_dat2          : in        std_logic;
   mmc_n_cs          : out       std_logic;
   mmc_clk           : out       std_logic;
   mmc_mosi          : out       std_logic;
   mmc_miso          : in        std_logic;

   PS2_enable        : out       std_logic;
   PS2_clk1          : inout           std_logic;
   PS2_data1         : inout           std_logic;
   PS2_clk2          : inout           std_logic;
   PS2_data2         : inout           std_logic
   );
end FleaFPGA_Ohm_A5;
```

Определение объекта верхнего уровня завершается описанием интерфейсов MMC (для карты твердотельного диска SD) и PS/2.

Далее следует определение архитектуры модуля. Эта часть похожа на файл исходного кода на C++ приложения с определением функциональности объекта, как в заголовочном файле.

```
architecture arch of FleaFPGA_Ohm_A5 is
   signal clk_dvi  : std_logic := '0';
   signal clk_dvin : std_logic := '0';
   signal clk_vga  : std_logic := '0';
   signal clk_50  : std_logic := '0';
   signal clk_pcs   : std_logic := '0';

   signal vga_red     : std_logic_vector(3 downto 0) := (others => '0');
   signal vga_green   : std_logic_vector(3 downto 0) := (others => '0');
   signal vga_blue    : std_logic_vector(3 downto 0) := (others => '0');
   signal ADC_lowspeed_raw     : std_logic_vector(7 downto 0) := (others => '0');
   signal red     : std_logic_vector(7 downto 0) := (others => '0');
```

```
signal green   : std_logic_vector(7 downto 0) := (others => '0');
signal blue    : std_logic_vector(7 downto 0) := (others => '0');
signal hsync   : std_logic := '0';
signal vsync   : std_logic := '0';
signal blank   : std_logic := '0';
```

Здесь определяется группа сигналов. Эти сигналы позволяют связывать друг с другом порты, объекты, процессы и прочие элементы модуля VHDL.

Можно видеть, что некоторые сигналы определены для поддержки VGA. Это обеспечивает совместимость с платами ППВМ, поддерживающими VGA, но часть из них также совместима с периферийными устройствами HDMI (или DVI), как мы увидим немного ниже.

```
begin
 Dram_CKE <= '0';    -- Таймер DRAM отключен (запрещен).
 Dram_n_cs <= '1';   -- Микросхема DRAM отключена (запрещена).
 PS2_enable <= '1';  -- Конфигурирование обоих хост-портов USB для поддержки режима PS/2.
 mmc_n_cs <= '1';    -- Микросхема карты микро-SD отключена (запрещена).
```

Ключевое слово `begin` обозначает пункт, в котором необходимо начать выполнение команд в определении архитектуры. Все инструкции, следующие за этим ключевым словом до завершающего ключевого слова `end architecture`, будут выполняться одновременно, за исключением блока инструкций, обозначенного ключевым словом `process` (в рассматриваемом здесь коде такого блока нет).

Некоторые аппаратные функции отключаются (запрещаются) с помощью явно записанных инструкций для соответствующих контактов. В приведенном выше фрагменте определения пропущен раздел DRAM (внешняя память) для краткости. Запрещается использование функциональности DRAM и SD-карты, тогда как функциональность PS/2 (клавиатура, мышь) разрешена. Это позволяет при необходимости подключать устройства ввода с интерфейсом PS/2.

```
user_module1 : entity work.FleaFPGA_DSO
    port map(
         rst => not sys_reset,
         clk => clk_50,
         ADC_1 => n_led1,
         ADC_lowspeed_raw => ADC_lowspeed_raw,
         Sampler_Q => ADC3_error,
         Sampler_D => ADC3_input,
         Green_out => vga_green,
         Red_out => vga_red,
         Blue_out => vga_blue,
         VGA_HS => hsync,
         VGA_VS => vsync,
         blank => blank,
         samplerate_adj => GPIO_20,
         trigger_adj => GPIO_21
    );
```

Здесь определяется возможность использования экземпляра модуля FleaFPGA Digital Storage Oscilloscope. Определено отображение только для первого канала, хотя модуль способен поддерживать четыре канала. Такое упрощение помогает продемонстрировать общий принцип действий.

Модуль DSO отвечает за чтение данных из аналого-цифрового преобразователя (ADC) по мере накопления характеристик сигналов, измеряемых с помощью датчика осциллографа. Данные передаются в локальную кеш-память для вывода на локальный монитор (HDMI или VGA) и отправляются через последовательный интерфейс в модуль UART (показанный в конце текущего раздела).

```
red <= vga_red & "0000";
green <= vga_green & "0000";
blue <= vga_blue & "0000";
```

Далее определяются цвета для вывода на дисплей с помощью выходных сигналов HDMI.

```
u0 : entity work.DVI_clkgen
port map(
      CLKI              =>    sys_clock,
      CLKOP             =>    clk_dvi,
      CLKOS             =>  clk_dvin,
      CLKOS2            =>  clk_vga,
      CLKOS3            =>  clk_50
      );

u100 : entity work.dvid PORT MAP(
   clk       => clk_dvi,
   clk_n     => clk_dvin,
   clk_pixel => clk_vga,
   red_p     => red,
   green_p   => green,
   blue_p    => blue,
   blank     => blank,
   hsync     => hsync,
   vsync     => vsync,
   -- выходные сигналы для драйверов TMDS
   red_s     => LVDS_Red,
   green_s   => LVDS_Green,
   blue_s    => LVDS_Blue,
   clock_s   => LVDS_ck
);
```

Приведенный выше раздел предназначен для вывода видеосигнала, генерируемого модулем DSO. Это позволяет использовать плату ППВМ как независимое устройство осциллографа.

```
myuart : entity work.simple_uart

      port map(
            clk => clk_50,
            reset => sys_reset, -- active low
            txdata => ADC_lowspeed_raw,
            --txready => ser_txready,
            txgo => open,
            --rxdata => ser_rxdata,
            --rxint => ser_rxint,
            txint => open,
```

```
            rxd => slave_rx_i,
            txd => slave_tx_o
        );
end architecture;
```

В заключительной секции определена простая реализация UART, позволяющая модулю DSO обмениваться информацией с приложением, написанным на C++.

Интерфейс UART конфигурируется для работы на скорости 19 200 бод, 8 бит, 1 стоп-бит, без контроля четности. После сборки этого проекта VHDL и программирования с его помощью платы ППВМ можно подключать плату к последовательной линии связи.

Код C++

Код VHDL предназначен для реализации вывода на дисплей с простыми параметрами и опциями ввода, но если необходимо воспользоваться большим дисплеем (с высоким разрешением), выполнять анализ сигналов, вести записи в течение нескольких минут или даже часов и т. п., то более удобно было бы выполнять все эти операции на одноплатном компьютере.

Приведенный ниже исходный код представляет собой графическое приложение, написанное на C++/Qt, которое принимает необработанные данные из аналого-цифрового преобразователя (ADC) на плате ППВМ (FPGA) и выводит их в графическом виде. Это приложение является основой рабочей среды для полнофункциональной системы на кристалле (SoC).

Сначала рассмотрим содержимое заголовочного файла.

```
#include <QMainWindow>

#include <QSerialPort>
#include <QChartView>
#include <QLineSeries>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow {
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
public slots:
    void connectUart();
    void disconnectUart();
    void about();
    void quit();
private:
    Ui::MainWindow *ui;
    QSerialPort serialPort;
    QtCharts::QLineSeries* series;
    quint64 counter = 0;
```

```
private slots:
    void uartReady();
};
```

Здесь используется реализация поддержки последовательного порта на Qt и модуль QChart для визуализации данных.

В следующем фрагменте кода представлена реализация класса, объявленного в заголовочном файле.

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <QSerialPortInfo>
#include <QInputDialog>
#include <QMessageBox>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    // Определение связей меню.
    connect(ui->actionQuit, SIGNAL(triggered()), this, SLOT(quit()));
    connect(ui->actionConnect, SIGNAL(triggered()), this, SLOT(connectUart()));
    connect(ui->actionDisconnect, SIGNAL(triggered()), this, SLOT(disconnectUart()));
    connect(ui->actionInfo, SIGNAL(triggered()), this, SLOT(about()));
    // Другие связи
    connect(&serialPort, SIGNAL(readyRead()), this, SLOT(uartReady()));
    // Конфигурирование графического представления.
    QChart* chart = ui->chartView->chart();
    chart->setTheme(QChart::ChartThemeBlueIcy);
    chart->createDefaultAxes();
    series = new QtCharts::QLineSeries(chart);
    chart->setAnimationOptions(QChart::NoAnimation);
    chart->addSeries(series);
}
```

В приведенном выше конструкторе создаются связи с пунктами меню в графическом пользовательском интерфейсе, обеспечивающими выход из приложения, соединение с последовательным портом, разрыв соединения с последовательным портом, если оно было установлено ранее, и получение информации о приложении.

Устанавливается соединение экземпляра последовательного порта со слотом, который будет вызываться при готовности к чтению новых данных.

Далее конфигурируется графическое представление в GUI с получением ссылки на экземпляр QChart в виджете QChartView. С помощью этой ссылки устанавливается тема графика, добавляются оси по умолчанию и пустая последовательность, которая будет заполняться данными, передаваемыми с ППВМ.

```
MainWindow::~MainWindow() {
    delete ui;
}

void MainWindow::connectUart() {
    QList<QSerialPortInfo> comInfo = QSerialPortInfo::availablePorts();
    QStringList comNames;
```

```
    for (QSerialPortInfo com: comInfo) {
        comNames.append(com.portName());
    }
    if (comNames.size() < 1) {
        QMessageBox::warning(this, tr("No serial port found"),
                                   tr("No serial port was found on the system. \
                                       Please check all connections and try again."));
        return;
    }
    QString comPort = QInputDialog::getItem(this, tr("Select serial port"),
                                                  tr("Available ports:"), comNames, 0, false);
    if (comPort.isEmpty()) { return; }
    serialPort.setPortName(comPort);
    if (!serialPort.open(QSerialPort::ReadOnly)) {
        QMessageBox::critical(this, tr("Error"), tr("Failed to open the serial port."));
        return;
    }
    serialPort.setBaudRate(19200);
    serialPort.setParity(QSerialPort::NoParity);
    serialPort.setStopBits(QSerialPort::OneStop);
    serialPort.setDataBits(QSerialPort::Data8);
}
```

Когда пользователь хочет установить соединение с ППВМ через интерфейс UART, должна быть выбрана линия последовательного соединения с ППВМ, после чего устанавливается соединение со скоростью обмена 19 200 бод и параметрами 8N1, ранее установленными в секции VHDL проекта.

Для жестко заданной конфигурации с одним и тем же последовательным портом можно рассмотреть вариант автоматизации установки порта при загрузке системы.

```
void MainWindow::disconnectUart() {
    serialPort.close();
}
```

Разрыв соединения с последовательным портом выполняется чрезвычайно просто.

```
void MainWindow::uartReady() {
    QByteArray data = serialPort.readAll();
    for (qint8 value: data) {
        series->append(counter++, value);
    }
}
```

Когда UART принимает новые данные с платы ППВМ, вызывается этот слот. Здесь считываются все данные из буфера UART, затем они добавляются в последовательность, добавленную в виджет формирования графика, который обновляет выводимые данные. Переменная `counter` используется для наращивания временно́й базы графика. Здесь это работает как упрощенная метка времени.

Здесь же необходимо начать удаление данных из последовательности, чтобы их объем не становился слишком большим. Также обеспечивается возможность поиска в данных и их сохранение. Метка времени на основе счетчика может со-

общать реальное время, в которое получен сигнал, хотя в идеальном случае это должно быть частью данных, получаемых от ППВМ.

```
void MainWindow::about() {
    QMessageBox::aboutQt(this, tr("About"));
}

void MainWindow::quit() {
    exit(0);
}
```

Завершают исходный код два простых слота. Для вывода информации о приложении выводится стандартное диалоговое окно Qt. Его можно заменить на подробную справку или на более интерактивное информационное диалоговое окно.

Сборка проекта

Сборку проекта VHDL и программирование платы ППВМ (FPGA) Ohm можно выполнить с помощью свободно распространяемого программного обеспечения Lattice Semiconductor Diamond IDE (**http://www.latticesemi.com/latticediamond**). Для программирования платы потребуется установка утилиты FleaFPGA JTAG (**https://github.com/Basman74/FleaFPGA-Ohm**), чтобы среда разработки Diamond IDE могла воспользоваться ею.

Следуя инструкциям из оперативного руководства по использованию платы FleaFPGA Ohm, относительно просто сформировать соответствующую часть проекта и выполнить ее. Для части C++ необходимо убедиться в том, что плата ППВМ (FPGA) и одноплатный компьютер (или равноценное устройство) соединены правильно, чтобы можно было получить доступ к интерфейсу UART на ППВМ.

После проверки правильности сборки системы нужно просто скомпилировать проект C++ вместе с рабочей средой Qt (непосредственно на одноплатном компьютере или на настольной системе с применением кросс-компиляции, что более предпочтительно). После этого можно запустить приложение при активной (снабженной электропитанием) плате ППВМ, установить соединение с интерфейсом UART и наблюдать трассировку сигнала, изображаемую в окне приложения.

Резюме

В этой главе рассматривалось применение программируемых пользователем вентильных матриц (ППВМ; FPGA) в разработке встроенных систем, рост значимости ППВМ в течение последнего десятилетия и варианты их возможного использования. Был подробно разобран пример реализации простого осциллографа на основе ППВМ и одноплатного компьютера. Предполагается, что после изучения материала данной главы читатель сможет самостоятельно выбрать ППВМ для нового проекта встроенной системы с глубоким пониманием того, как можно использовать подобное устройство и как организовать обмен данными с ним.

Приложение A

Эффективные практические методики

Как и в любом программном проекте, при разработке встроенных систем возникает ряд общих проблем и затруднений, к которым добавляется фактор аппаратуры, создающий специфический набор проблем. Учитывая весь диапазон проблем от вопросов управления ресурсами до некорректных прерываний и непредсказуемого поведения аппаратной части, это приложение предлагает методы предотвращения и устранения многих из подобных проблем. Кроме того, здесь описаны разнообразные способы оптимизации и варианты их практического применения. В приложении рассматриваются следующие темы:

- безопасные способы оптимизации кода для встроенных систем;
- способы предотвращения и устранения общих проблем, связанных с программной и аппаратной частями;
- выявление несовершенных аппаратных компонентов и возможностей их интеграции в проектное решение.

Тщательно продуманные планы

Как и в любом проекте, неизбежно возникает различие между запланированным проектным решением и тем, как работает система в действительности. Даже при самом скрупулезном планировании и солидном практическом опыте всегда будут возникать непредвиденные и неожиданные проблемы. Самое лучшее, что можно сделать, – как можно тщательнее подготовиться к неожиданностям.

В первую очередь необходимо получить доступ ко всей доступной информации о целевой платформе, изучить все доступные инструментальные средства и составить подробный план разработки и тестирования. Многие из этих аспектов достаточно подробно рассматривались в этой книге.

В данном приложении обобщаются наиболее эффективные практические методики, которые должны помочь вам избежать или устранить большинство общих проблем.

Работа с аппаратурой

Для каждой целевой платформы существуют собственные тонкости и характеристики. Многие из них связаны с историей разработок для конкретной платформы. Например, для такой платформы, как AVR, эта связь очевидна, поскольку на протяжении многих лет разработку вела одна компания (Atmel), поэтому установилась тесная связь между различными моделями микросхем и инструментальными средствами, используемыми для этой платформы.

Такая платформа, как ESP8266 (и в определенной степени ее последователь ESP32), никогда не предназначалась для применения в качестве системы микроконтроллеров общего назначения, что отражено в ее достаточно схематичной и фрагментированной экосистеме программного обеспечения. Хотя за последние несколько лет ситуация немного улучшилась с появлением разнообразных рабочих сред и инструментальных средств с открытым исходным кодом, исправляющих самые крупные недостатки, на этой платформе все еще высока вероятность совершения ошибок из-за недостаточно подробной документации, проблем с инструментальными средствами и слабой поддержки отладки непосредственно на микросхеме.

Микроконтроллеры ARM (Cortex-M) выпускаются множеством производителей в невероятном количестве конфигураций. Хотя программирование для этих микроконтроллеров становится более или менее согласованным при использовании таких инструментальных средств, как OpenOCD, периферийные устройства, добавляемые к каждому микроконтроллеру, совершенно различны у разных производителей, но об этом – в следующем разделе.

Наконец, системы на кристалле ARM и их аналоги занимают позицию, схожую с микроконтроллерами ARM, но имеют значительно более сложную архитектуру и большее количество периферийных устройств, чем их собратья. Поэтому в системы на кристалле ARM добавляются сложные подпрограммы инициализации, требующие полноценных загрузчиков. Именно по этой причине большинство разработчиков предпочитают использовать готовый к работе образ Linux или нечто подобное для системы на кристалле и вести разработку в такой рабочей среде.

В действительности однозначного ответа не существует. Большинство аппаратных устройств имеет комплектацию, подходящую для проектов, но чрезвычайно важно иметь правильное представление обо всех аппаратных платформах, с которыми вы работаете.

Огромный мир периферийных устройств

Для микроконтроллеров ARM характерно наличие разнообразных и часто несовместимых периферийных устройств, отображаемых в абсолютно различные области пространства памяти. Хуже всего дело обстоит с периферийными устройствами таймеров, отличающихся разными уровнями сложности и индивидуальными особенностями, благодаря которым они, вообще говоря, способны генерировать любой требуемый выходной сигнал на контакте интерфейса GPIO, в том числе и широтно-импульсную модуляцию (PWM), а также работать в качест-

ве таймеров на основе прерываний для управления выполнением специализированного ПО.

Конфигурирование таймеров и подобных сложных периферийных устройств – задача не для слабонервных. Кроме того, использование MAC-адреса и внешнего физического интерфейса Ethernet (PHY) требует обширных и глубоких знаний в этой области и практического опыта конфигурирования. Весьма важно тщательно изучать спецификации и прочую документацию на периферийные устройства.

Применение кода, автоматически сгенерированного такими инструментальными средствами, как ST CubeMX, для линейки микроконтроллеров ARM STM32 может привести к ситуации, в которой придется иметь дело с нефункциональным кодом, потому что вы забыли активизировать несколько чек-боксов в редакторе CubeMX, так как не знали, для чего предназначены соответствующие опции и параметры.

Нет ничего плохого в использовании подобных инструментов автоматической генерации кода или библиотек высокого уровня, предоставляемых производителем аппаратуры, поскольку они существенно облегчают работу. Но при этом чрезвычайно важно принимать во внимание потенциальные риски, сопутствующие этому решению, поскольку приходится полагаться на то, что предоставленный код корректен, или тратить время на проверку корректности этого кода.

Чтобы сделать использование периферийных устройств с разнообразными микроконтроллерами и системами на кристалле менее затруднительным, необходимо добавить некоторый уровень абстракции для обеспечения переносимости кода. Самое главное здесь – убедиться в том, что это действительно упрощает работу, а не только добавляет еще один потенциальный источник проблем, который может отрицательно повлиять на текущий проект или на будущие проекты.

Изучайте свои инструментальные средства

При работе над проектом встроенной системы необходимо знать, какие инструментальные средства существуют для целевой платформы и как они работают: от программирования микроконтроллера через JTAG или другой интерфейс и организации сеанса отладки непосредственно на микросхеме до ограничений процесса отладки на микросхеме. Необходимо тщательно изучать руководства и документацию на инструментальные средства, прежде чем использовать их, и перенимать опыт других разработчиков, которые применяли эти инструменты.

В предыдущих главах рассматривались некоторые из инструментальных средств как для микроконтроллеров, так и для систем на кристалле, вместе с методами проверки проектного решения с применением микроконтроллера до переноса на реальную целевую аппаратуру.

Выбор асинхронных методов

Многие аппаратные устройства и соответствующие операции требуют некоторого времени для выполнения. Следовательно, имеет смысл выбирать асинхронные действия с применением прерываний и таймеров вместо блокировки операций.

При программировании непосредственно на аппаратуре имеется тенденция к использованию единственного цикла с подпрограммами обработки прерываний и сигналов таймеров, которые позволяют выполнять ответные действия и опрашивать устройства, чтобы определить возникновение событий. При программировании в полностью асинхронном стиле такой главный цикл будет эффективно выполнять все задачи, пока обработчики прерываний обновляют данные, требующие обработки.

Использование асинхронных методов даже на платформах систем на кристалле является хорошей идеей, поскольку сетевые операции, операции ввода/вывода и т. п. могут занимать больше времени, чем предполагалось. Отдельной проблемой может стать обработка незавершенных операций.

Изучение спецификаций

Спецификации, особенно для микроконтроллеров, предоставляют солидный объем полезной информации о том, как работает аппаратура, как конфигурируется внутренний системный таймер, как работают отдельные периферийные устройства, а также характеристики и предназначение доступных регистров.

Даже при использовании существующей готовой платы вместо специализированной аппаратной системы важно понимать, как работает конкретная аппаратура, хотя бы бегло ознакомившись со спецификацией микроконтроллера или системы на кристалле.

Обеспечение краткости обработчиков прерываний

Сама сущность прерывания определяет тот факт, что оно прерывает обычное выполнение инструкций процессора, переключаясь на обработку прерывания. Каждая микросекунда, потраченная в коде обработчика прерывания, – это время, в течение которого невозможно выполнять другие подпрограммы или обрабатывать иные прерывания.

Для устранения возникающих при этом проблем обработчики прерываний (ISR) необходимо делать максимально короткими. В идеальном случае должно выполняться обновление единственного значения самым быстрым и безопасным способом перед завершением работы обработчика прерываний и возобновлением выполнения обычных операций.

8 бит означает 8 бит

Никого не удивляет, что обработка 16-битовых и 32-битовых целых чисел на 8-битовых микроконтроллерах выполняется достаточно медленно. Система вынуждена выполнять несколько операций с одним и тем же целочисленным значением, поскольку в ее регистры одновременно может помещаться только 8 бит.

Кроме того, использование переменных с плавающей точкой в системе без сопроцессора FPU приводит к значительному замедлению работы всей системы

в целом, так как обычный процессор вынужден формировать дополнительный поток инструкций, имитирующий выполнение операций с плавающей точкой.

Не следует заново изобретать колесо

Если существует библиотека или рабочая среда высокого качества, они доступны для целевой платформы и могут быть лицензированы в проекте, то следует воспользоваться ими, а не увлекаться созданием собственной реализации.

Рекомендуется также хранить библиотеку часто используемых фрагментов кода и примеров как справочник не только для себя, но и для других членов группы разработки. Так проще вспомнить, где найти пример применения некоторого функционального свойства или метода, чем воспроизводить в памяти весь процесс реализации с многочисленными подробностями.

Подумайте, прежде чем начать оптимизацию

Никогда не следует пытаться оптимизировать код без полного понимания того, как повлияют на работу системы предполагаемые изменения. Недостаточно просто чувствовать или иметь неопределенное представление о том, что будет происходить в системе после внесения изменений.

Платформы на основе систем на кристаллах с полноценной ОС предоставляют бóльшую свободу действий, но для платформ микроконтроллеров весьма важно понимать, что именно будет означать добавление единственного ключевого слова инструкции или использование другой структуры данных для хранения некоторой информации.

Самый плохой случай – предположение о том, что методы оптимизации, примененные на одноплатном компьютере и на настольных системах, дадут тот же эффект на платформе микроконтроллера. Из-за модифицированной гарвардской архитектуры и различных особенностей платформ, подобных AVR, вероятнее всего, эффект окажется обратным (отрицательным) или, если повезет, вы вообще не увидите никакого эффекта.

Практические рекомендации и указания по применению для платформы микроконтроллера весьма полезны для понимания того, как следует выполнять оптимизацию аппаратуры. Заключительное замечание: проводите исследования и эксперименты до начала реальной оптимизации и не приступайте к написанию кода без принятия проектного решения.

Требования – это основа, а не дополнение

Создание программного обеспечения для встроенной системы без четко определенного набора требований к проекту напоминает начало строительства нового здания без ясно сформулированного представления о том, сколько помещений должно быть в этом здании, где будут располагаться окна и двери, где необходимо проложить электропроводку и прочие коммуникации.

Конечно, можно сразу приступить к написанию работающего кода и сформировать функционирующий прототип, но в действительности подобные прототипы обычно вводятся в эксплуатацию без тщательного рассмотрения полного жизненного цикла конечного продукта, поэтому приходится постоянно вносить изменения и исправления (патчи) на протяжении многих лет, чтобы добавлять функциональные возможности в изначально созданный код специализированного ПО, которое никогда не проектировалось надлежащим образом.

После завершения составления полного списка требований к конечному продукту выполняется преобразование этих требований в архитектуру (то есть в общую структуру приложения), которая, в свою очередь, трансформируется в проектное решение (которое, собственно, и будет реализовано). После этого проектное решение переводится в исходный код.

Преимущество такого подхода заключается не только в том, что вы должны ответить на многочисленные вопросы о том, почему то или иное действие выполняется конкретным выбранным способом, но также в том, что при этом создается достаточный объем документации, которая в дальнейшем будет практически использоваться вплоть до завершения проекта.

Кроме того, проект встроенной системы с полным набором требований может помочь сэкономить немало времени и средств, поскольку позволяет выбрать правильный микроконтроллер или систему на кристалле без излишних затрат на более мощную микросхему, чем требуется в действительности. При этом также предотвращается ситуация в ходе разработки проекта, когда вдруг вспоминают о «забытой» функциональной возможности, добавление которой может привести к необходимости изменения всего проектного решения аппаратуры.

Документация жизненно важна

Ставшее широко известным шутливое высказывание утверждает, что программисты не любят писать документацию и называют создаваемый ими код «самодокументированным кодом». В действительности без подробной и ясной документации, описывающей требования к проектному решению, архитектуру, планирование проекта и API, ставится под угрозу будущее проекта, а также будущее разработчиков и конечных пользователей, которые надеются на корректную работу программного обеспечения.

Следование стандартным процедурам и выполнение всей скучной бумажной работы перед началом написания первых строк кода может отбить всякое желание к работе. К сожалению, действительность такова, что без этих усилий все знания и замыслы останутся лишь в головах разработчиков проекта. Это крайне затрудняет интеграцию специализированного ПО с остальной частью проекта встроенной системы и осложняет сопровождение системы в будущем, особенно если она передается другой группе.

Простой факт состоит в том, что «самодокументированного» кода не существует. Но даже если бы он существовал, ни один инженер по аппаратуре не станет тратить свое время на изучение тысяч строк исходного кода, чтобы узнать, какой тип сигнала определен для конкретного контакта GPIO при возникновении некоторого входного условия в микроконтроллере.

Тестирование кода означает попытку нарушить его выполнение

Самая распространеная ошибка при создании тестов – написание сценариев тестирования, в которых предполагается, что все будет работать правильно. Это неверный подход. Очень хорошо, что конкретная программа синтаксического анализа правильно выполняет свою работу при обработке абсолютно корректно подготовленных и отформатированных данных, но это не слишком полезно для ситуации при реальной эксплуатации.

Разумеется, вы можете получать идеально корректные данные, но с той же вероятностью возможно получение полностью искаженных данных или даже «мусора» вместо данных для обработки в вашем коде. Поэтому главная цель тестирования – убедиться в том, что вне зависимости от того, насколько невероятными и некорректными могут быть входные данные, они не смогут оказать какого-либо негативного воздействия на остальную часть системы.

Все входные данные должны быть проверены, и их корректность должна быть подтверждена. Если что-то выглядит некорректным, то такой фрагмент данных должен быть отброшен, чтобы не допустить возникновения проблем при дальнейшем выполнении кода.

Резюме

В этом приложении кратко рассматривались общие проблемы и затруднения, возникновение которых возможно в процессе работы над проектом встроенной системы.

Предполагается, что после изучения материала этого приложения читатель сможет определить этапы проекта и обосновать необходимость документирования каждого этапа проекта.

Предметный указатель

K

M

Н

О

П

Р

С

Т

У

Ф

Ц

Ш

Э

Я

A

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Майа Пош

Программирование встроенных систем на C++17

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Снастин А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 32,01. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

C++ — превосходный выбор для разработки встроенных систем, особенно с учетом того, что этот язык не добавляет каких-либо излишеств, улучшает удобство сопровождения и предлагает множество преимуществ над прочими языками программирования.

Из этой книги вы узнаете, как можно применять C++ для создания надежных, конкурентоспособных систем, рационально использующих все доступные аппаратные ресурсы.

Начиная с простого примера программирования встроенной системы и описания самых свежих функциональных возможностей, введенных стандартом C++ 17, книга демонстрирует все тонкости правильного программирования. Вы узнаете, как использовать параллельный режим выполнения, управление памятью и возможности функционального программирования C++ при создании встроенных систем, а также как объединять ваши системы с внешними периферийными устройствами и применять эффективные способы работы с драйверами. Автор приводит правила и рекомендации по тестированию и оптимизации кода для улучшения производительности и реализации полезных шаблонов проектирования. Также в книге представлено подробное описание работы с Qt, широко известной библиотекой графического интерфейса, используемой для создания встроенных систем.

Вы научитесь:

- выбирать правильный тип встроенной платформы для конкретного проекта;
- разрабатывать драйверы для встроенных систем на основе ОС;
- использовать параллельный режим выполнения и средства управления памятью для различных моделей микроконтроллеров;
- отлаживать и тестировать кросс-платформенный код в среде ОС Linux;
- практически реализовать информационно-развлекательную систему с использованием одноплатного компьютера на основе Linux;
- усовершенствовать существующую встроенную систему с помощью графического пользовательского интерфейса на основе Qt;
- организовать обмен информацией со стороной FPGA в гибридной системе FPGA/SoC.

Для понимания тем, рассматриваемых в книге, нужен опыт разработки на языке C++. Какие-либо знания о встроенных системах не требуются.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

Packt>

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-785-5