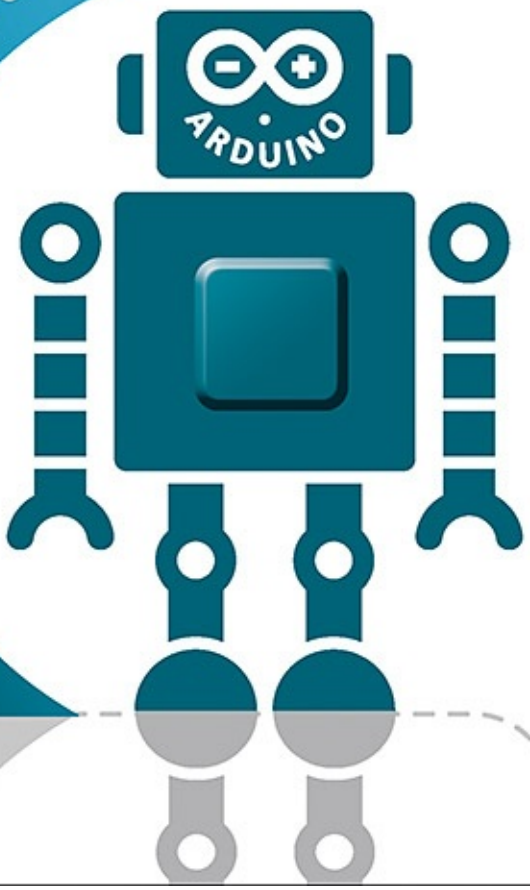
Саймон Монк

Программируем Arduino

Профессиональная работа со скетчами

БЕСТСЕЛЛЕР AMAZON



ПИТЕР

С. Монк

Программируем Arduino. Профессиональная работа со скетчами

2017

Переводчик *А. Макарова*
Технический редактор *Н. Суслова*
Литературный редактор *Н. Рощина*
Художники *Л. Егорова, С. Маликова*
Корректоры *С. Беляева, Н. Витько*
Верстка *Л. Егорова*

С. Монк
Программируем Arduino. Профессиональная работа со скетчами . — СПб.: Питер, 2017.

ISBN 978-5-496-02385-6

Об авторе

Доктор Саймон Монк (Dr. Simon Monk; Престон, Соединенное Королевство) *имеет степень бакалавра в области кибернетики и информатики, а также доктора наук в области программной инженерии.* Доктор Монк несколько лет занимался академической наукой, прежде чем уйти в промышленность. Является одним из основателей компании Momote Ltd, специализирующейся на разработке программного обеспечения для мобильных устройств. *Со школьных лет активно увлекается электроникой и много пишет для радиолюбительских журналов об электронике и открытом аппаратном обеспечении. Автор многочисленных книг по электронике, посвященных в основном открытым аппаратным платформам, особенно Arduino и Raspberry Pi. В соавторстве с Полом Шерцем написал третье издание книги «Practical Electronics for Inventors».*

Вы можете последовать за Саймоном в Twitter, где он зарегистрирован как **@simonmonk2**.

Благодарности

Хочу выразить большую признательность издательству *McGraw-Hill Education*, сотрудники которого приложили массу усилий, чтобы выпустить эту книгу. Отдельное спасибо моему редактору Роджеру Стюарту (Roger Stewart), а также Ваставикте Шарма (Vastavikta Sharma), Джоди Маккензи (Jody McKenzie), Ли-Энн Пикрелл (LeeAnn Pickrell) и Клер Сплан (Claire Splan).

Хочу также поблагодарить компании Adafruit, SparkFun и CPC за предоставленные модули и компоненты, использовавшиеся при подготовке этой книги.

И напоследок, но не в последнюю очередь, спасибо Линде за ее терпение и великодушие, благодаря которым я смог написать эту книгу.

Введение

Arduino — стандартный микроконтроллер, получивший широкое признание у инженеров, мастеров и преподавателей благодаря своей простоте, невысокой стоимости и большому разнообразию плат расширения. Платы расширения, подключаемые к основной плате Arduino, позволяют выходить в Интернет, управлять роботами и домашней автоматикой.

Простые проекты на основе Arduino не вызывают сложностей в реализации. Но, вступив на территорию, не охваченную вводными руководствами, и увеличивая сложность проектов, вы быстро столкнетесь с проблемой нехватки знаний — врагом всех программистов.

Эта книга задумана как продолжение бестселлера «Programming Arduino: Getting Started with Sketches»[1]. Несмотря на то что эта книга включает краткое повторение основ из книги «Programming Arduino», она познакомит читателя с более продвинутыми аспектами программирования плат Arduino. В частности, эта книга расскажет, как:

- обеспечить эффективную работу при минимальном объеме доступной памяти;
- решать сразу несколько задач без помощи механизмов многопоточного выполнения;
- упаковывать код в библиотеки, чтобы им могли пользоваться другие;
- использовать аппаратные прерывания и прерывания от таймера;
- добиться максимальной производительности;
- уменьшить потребление электроэнергии;
- взаимодействовать с последовательными шинами разных типов (I2C, 1-Wire, SPI и последовательный порт);
- взаимодействовать с портом USB;
- взаимодействовать с сетью;
- выполнять цифровую обработку сигналов (Digital Signal Processing, DSP).

Загружаемые примеры

Книга включает 75 примеров скетчей, которые распространяются в открытом виде и доступны на веб-сайте автора **www.simonmonk.org**. Перейдя по ссылке на страницу этой

книги, вы сможете загрузить исходный код примеров, а также самый актуальный список ошибок и опечаток, найденных в книге.

Что необходимо для чтения книги

Данная книга в первую очередь посвящена вопросам программирования. Поэтому для опробования большинства примеров будет достаточно платы Arduino, светодиода и мультиметра. Если у вас имеются дополнительные платы расширения Arduino, они тоже пригодятся. Для рассмотрения примеров из главы 12 вам понадобится плата Ethernet или Wi-Fi. На протяжении всей книги мы будем использовать разные модули для демонстрации разных интерфейсов.

В центре внимания находится Arduino Uno — наиболее широко используемая плата Arduino, но в главах, посвященных программированию порта USB и цифровой обработке сигналов, рассматриваются некоторые особенности других плат Arduino, таких как Leonardo и Arduino Due.

В приложении в конце книги вы найдете список поставщиков, у которых сможете приобрести все эти компоненты.

Как работать с этой книгой

Каждая глава посвящена отдельной теме, связанной с программированием Arduino. Главы книги, кроме главы 1, где приводится краткий обзор основ Arduino, можно читать в любом порядке. Если вы опытный разработчик, начните с главы 14, чтобы вникнуть в некоторые особенности программирования Arduino.

Далее следует краткое описание глав.

Глава 1 «Программирование Arduino». Эта глава содержит сводную информацию о программировании Arduino. Это учебник для тех, кому требуется быстро ознакомиться с основами Arduino.

Глава 2 «Под капотом». В этой главе мы заглянем под капот и посмотрим, как работают программы для Arduino и откуда они берутся.

Глава 3 «Прерывания и таймеры». Новички обычно стараются не использовать прерывания, и совершенно напрасно, так как часто они оказываются удобным инструментом и их программирование не представляет никаких сложностей. Однако прерывания имеют свои ловушки, и эта глава расскажет вам все, что вы должны знать, чтобы не попасть в них.

Глава 4 «Ускорение Arduino». Платы Arduino оснащены маломощными процессорами с невысоким быстродействием, поэтому иногда требуется выжать из них все, что только можно. Например, встроенная функция `digitalWrite` надежна и проста в использовании, но неэффективна, что особенно заметно, когда требуется одновременно включить несколько выходов. В этой главе вы узнаете, как увеличить ее

производительность, а также познакомитесь с другими приемами создания быстродействующих скетчей.

Глава 5 «Снижение потребления электроэнергии». Когда для питания платы Arduino используются аккумуляторы или солнечные батареи, желательно уменьшить потребление электроэнергии. Этого можно добиться не только оптимизацией конструкции устройства, но и применением особых приемов программирования.

Глава 6 «Память». В этой главе мы посмотрим, как уменьшить потребление памяти, а также познакомимся с достоинствами и недостатками, связанными с динамическим распределением памяти в скетчах.

Глава 7 «Интерфейс I2C». Интерфейс I2C на плате Arduino может существенно упростить взаимодействие с модулями и компонентами и позволит обойтись меньшим числом контактов на плате. Эта глава описывает, как действует интерфейс I2C и как им пользоваться.

Глава 8 «Взаимодействие с устройствами 1-Wire». В этой главе рассказывается о шине 1-Wire для связи с устройствами, такими как датчики температуры компании Dallas Semiconductor, которые часто применяются с платами Arduino. Здесь вы узнаете, как действует эта шина и как ею пользоваться.

Глава 9 «Взаимодействие с устройствами SPI». SPI — еще один стандартный интерфейс, поддерживаемый платами Arduino. В этой главе описывается, как он действует и как им пользоваться.

Глава 10 «Программирование последовательного интерфейса». Поддержка передачи данных через последовательный порт, порт USB или контакты **Rx** и **Tx** на плате Arduino — отличный способ организовать обмен данными с периферийными устройствами и другими платами Arduino. В этой главе вы узнаете, как пользоваться последовательным портом.

Глава 11 «Программирование интерфейса USB». В этой главе рассматриваются разные аспекты использования порта USB на плате Arduino. Вы познакомитесь с возможностью эмуляции клавиатуры и мыши, поддерживаемой платой Arduino Leonardo, а также узнаете, как подключить клавиатуру или мышь к соответственно оборудованной плате Arduino.

Глава 12 «Программирование сетевых взаимодействий». Плата Arduino давно стала обычным компонентом «Интернета вещей». В этой главе вы узнаете, как программировать Arduino для работы в Интернете. В число обсуждаемых здесь тем входит описание плат расширения Wi-Fi и Ethernet, использования веб-служб и применения Arduino в качестве маленького веб-сервера.

Глава 13 «Цифровая обработка сигналов». Плата Arduino способна выполнять простую обработку сигналов. В этой главе обсуждаются разные способы такой обработки, от фильтрации сигнала, поступающего на аналоговый вход, с применением программного обеспечения вместо внешних электронных устройств до вычисления относительной величины различных частотных сигналов с применением быстрого

преобразования Фурье.

Глава 14 «Многозадачность с единственным процессом». Программисты, пришедшие в мир Arduino из мира больших систем, часто отмечают отсутствие поддержки многозадачности в Arduino как существенное упущение. В этой главе я попробую исправить его и покажу, как преодолеть ограничения однопоточной модели встроенных систем.

Глава 15 «Создание библиотек». Рано или поздно вы создадите нечто замечательное, что, по вашему мнению, могли бы использовать другие. Это будет самый подходящий момент оформить свой код в виде библиотеки и выпустить ее в свет. Эта глава покажет вам, как это сделать.

Ресурсы

В поддержку этой книги на веб-сайте автора (**www.simonmonk.org**) создана страница. Перейдя по ссылке на страницу этой книги, вы найдете исходный код примеров, а также другие ресурсы.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу **comp@piter.com** (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства **www.piter.com** вы найдете подробную информацию о наших книгах.

1 Монк С. Программируем Arduino. Основы работы со скетчами. — СПб.: Питер, 2015. — *Примеч. пер.*

1. Программирование Arduino

Эта глава содержит сводную информацию о плате Arduino. Если вы ничего не знаете об Arduino, возможно, вам стоит также прочитать книгу «Programming Arduino: Getting Started with Sketches» (McGraw-Hill Professional, 2012)[2].

Что такое Arduino?

Термин «Arduino» используется для обозначения физических плат Arduino, наибольшей популярностью из которых пользуется модель Arduino Uno, и системы Arduino в целом. Система включает также программное обеспечение для компьютера (применяется для программирования платы) и периферийные платы, которые можно подключать к основной плате Arduino.

Для работы с платой Arduino вам понадобится подходящий компьютер. Это может быть персональный компьютер с операционной системой Mac, Windows, Linux или нечто более скромное, такое как Raspberry Pi. Компьютер нужен в основном для того, чтобы выгружать программы в плату Arduino. После установки на Arduino эти программы действуют совершенно автономно. На рис. 1.1 показано, как выглядит плата Arduino Uno.

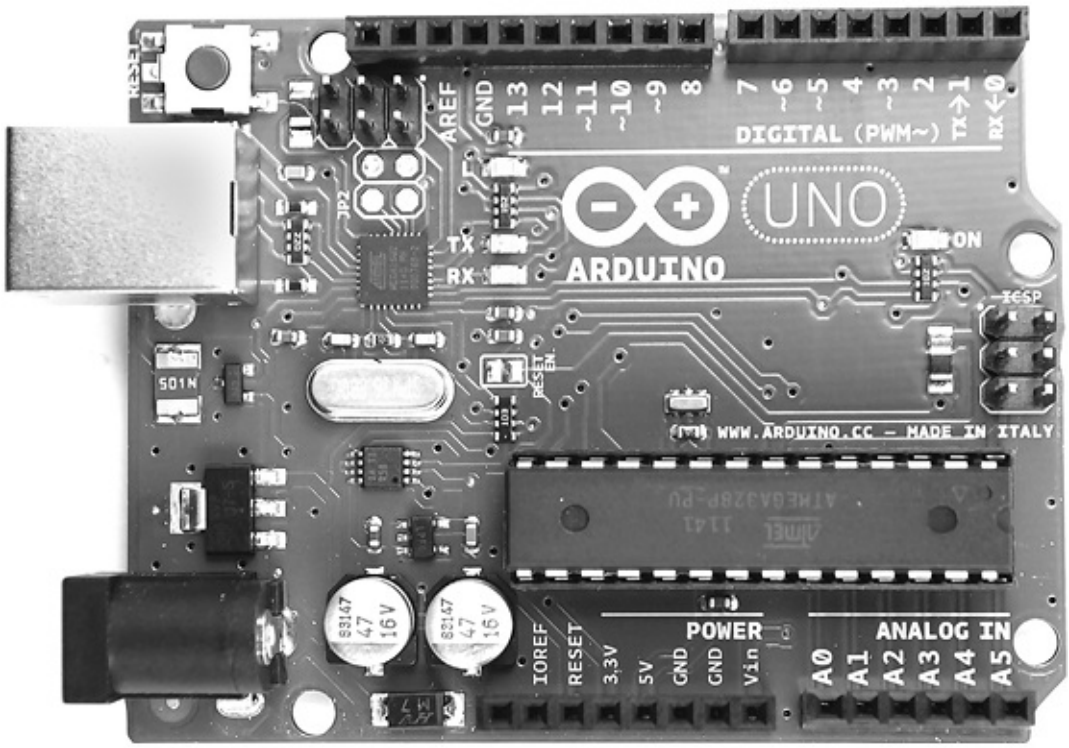


Рис. 1.1. Плата Arduino Uno

Плата Arduino может подключаться к порту USB компьютера. Когда она подключена, вы можете посылать сообщения в обоих направлениях. На рис. 1.2 показано, как соединяются плата Arduino и компьютер.

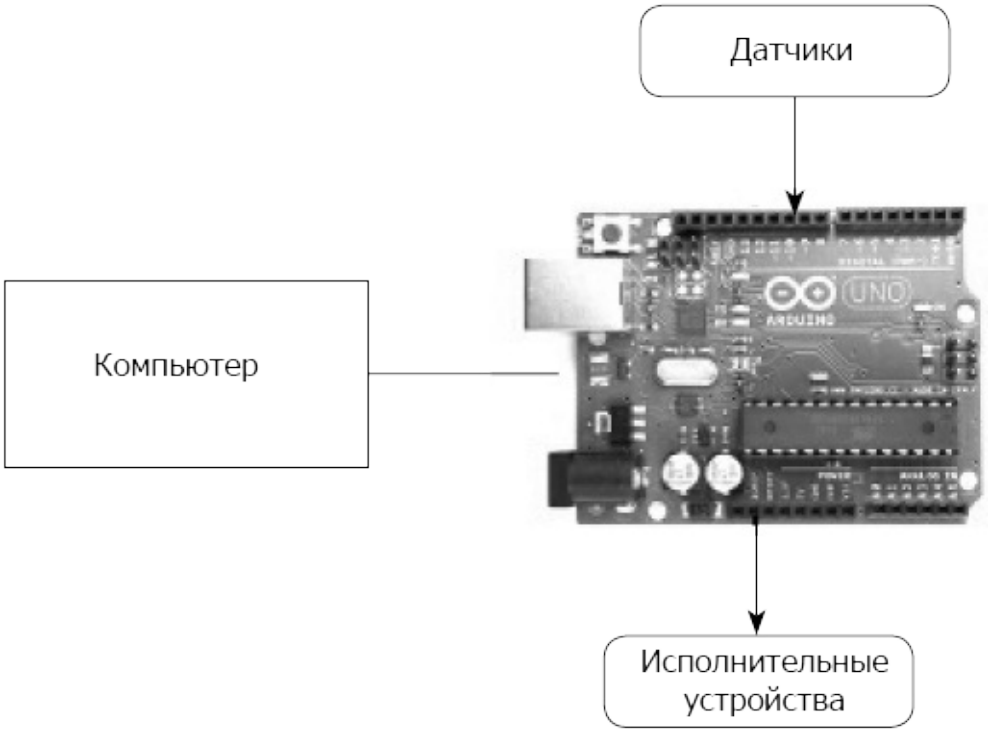


Рис. 1.2. Плата Arduino и компьютер

В отличие от компьютера, Arduino почти не имеет памяти, а также не имеет операционной системы, клавиатуры с мышью и экрана. Главная ее задача — чтение данных с датчиков и управление исполнительными устройствами. То есть можно, к примеру, подключить к плате датчик измерения температуры и управлять мощностью обогревателя.

На рис. 1.3 перечислены некоторые устройства, которые можно подсоединять к плате Arduino. Это далеко не все виды устройств, которые можно подключить к Arduino.

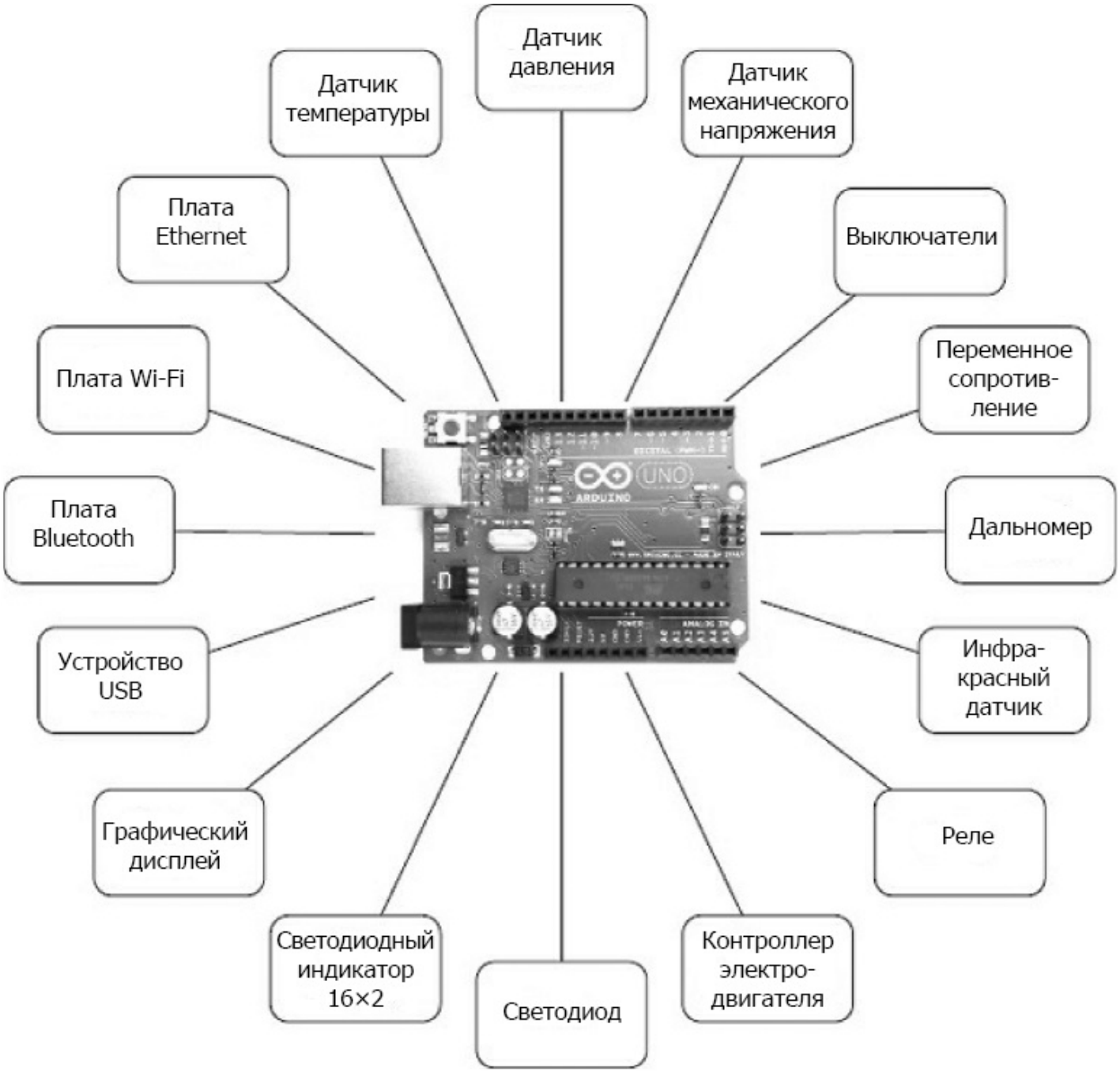


Рис. 1.3. Устройства, подключаемые к Arduino

Далее перечислены некоторые очень интересные проекты, реализованные на основе Arduino:

- Bubblino — подключенная к Arduino машина для мыльных пузырей, которая выпускает пузыри, когда вы посылаете ей сообщение в Twitter;
- светодиодные кубы;
- счетчики Гейгера;
- музыкальные инструменты;
- дистанционные датчики;

• роботы.

Установка и среда разработки

Для программирования Arduino используется *интегрированная среда разработки Arduino* (Arduino Integrated Development Environment, IDE). Если вы программист и имеете опыт работы со сложными средами разработки, такими как Eclipse или Visual Studio, то Arduino IDE покажется вам очень простой и, возможно, вы отметите отсутствие интеграции с репозиториями, функции автодополнения команд и прочих удобств. Если вы новичок в программировании, то вам понравятся простота и легкость использования Arduino.

Установка среды разработки

Прежде всего загрузите программное обеспечение для своей операционной системы с официального веб-сайта Arduino: **http://arduino.cc/en/Main/Software**.

После загрузки вы сможете найти подробные инструкции по установке для каждой платформы по адресу **http://arduino.cc/en/Guide/HomePage**[3].

Самое замечательное в Arduino то, что для работы с этой платой вам понадобятся лишь сама плата Arduino, компьютер и кабель USB для их соединения между собой. Плата Arduino может даже питаться от порта USB компьютера.

Blink

Чтобы убедиться в работоспособности платы, запишем в нее программу, которая будет включать и выключать светодиод, отмеченный на плате Arduino меткой **L**, из-за чего его часто называют светодиодом **L**.

Запустите Arduino IDE на своем компьютере. Затем в меню **File** (Файл) (рис. 1.4) выберите пункт **Examples**—>**01 Basics**—>**Blink** (Примеры—>01 Basics—>Blink).

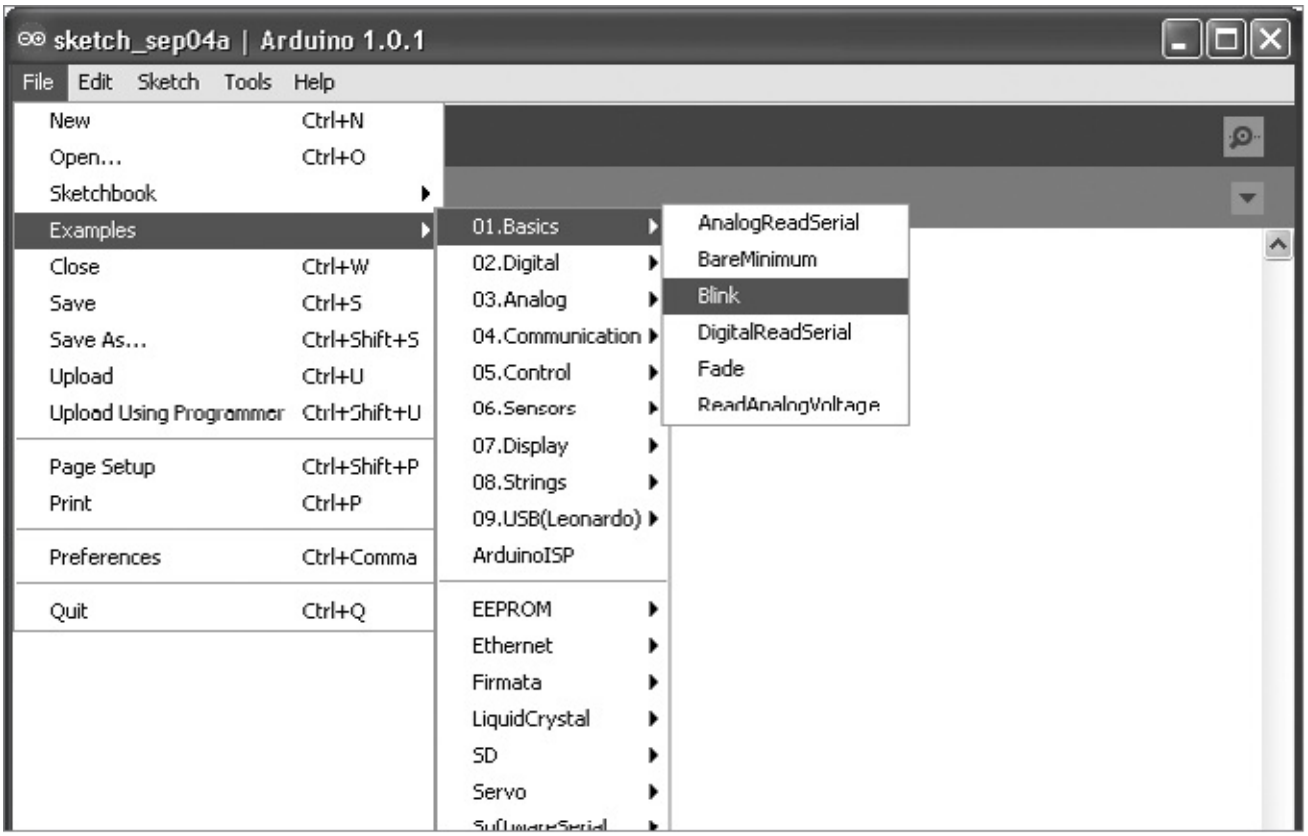


Рис. 1.4. Загрузка скетча Blink в Arduino IDE

Чтобы не отпугивать непрограммистов, программы в мире Arduino называют *скетчами*. Прежде чем выгрузить скетч Blink в свою плату Arduino, вам нужно настроить в Arduino IDE тип платы. Набольшее распространение получила плата Arduino Uno, и в этой главе я буду полагать, что вы используете именно ее. Поэтому в меню **Tools—>Board** (Инструменты—>Плата) выберите пункт **Arduino Uno**[4] (рис. 1.5).

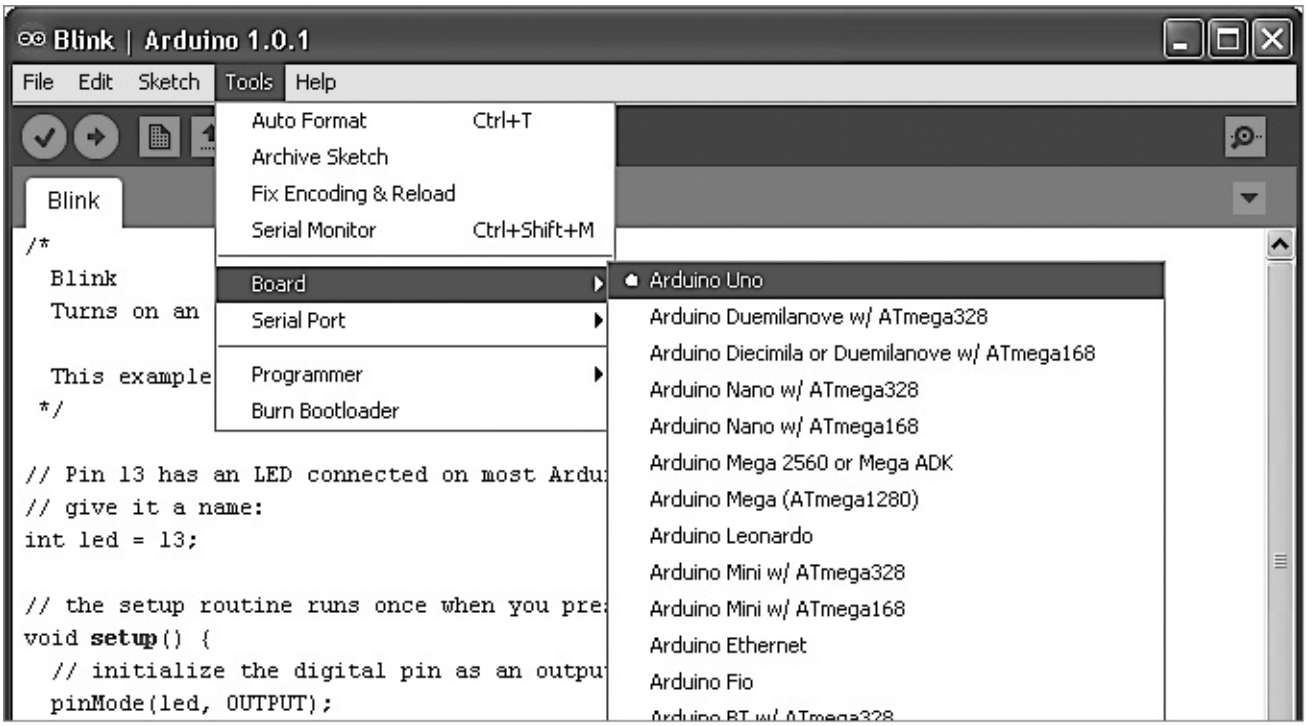


Рис. 1.5. Выбор типа платы

После настройки типа платы выберите порт, к которому она подключена. Сделать такой выбор в Windows очень просто, поскольку в списке, скорее всего, будет единственный порт COM4 (рис. 1.6). Однако в Mac или Linux список обычно содержит большее количество последовательных устройств. Среда разработки Arduino IDE помещает последние подключенные устройства в начало списка, поэтому плата Arduino должна находиться вверху.

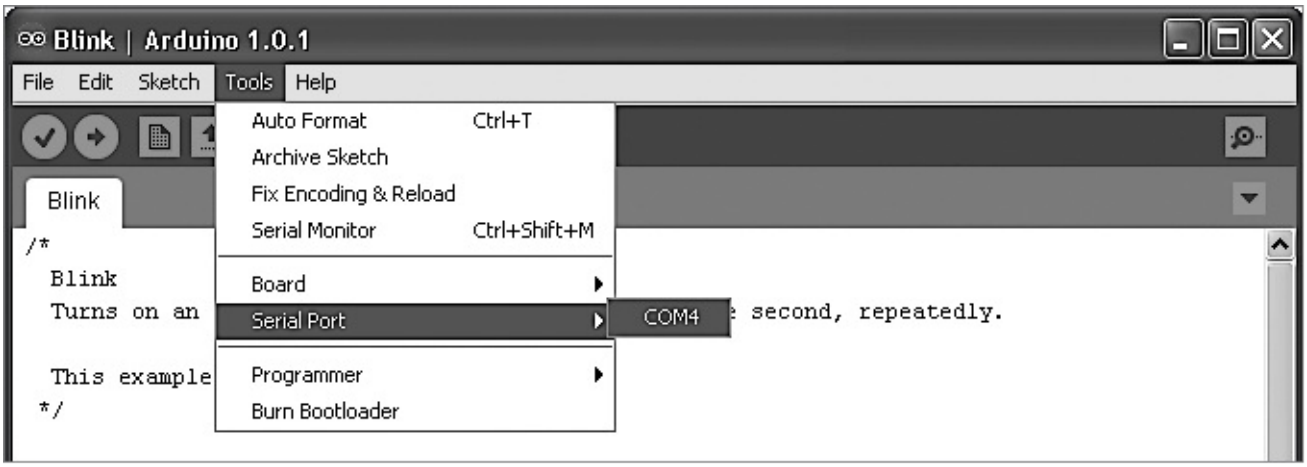


Рис. 1.6. Выбор последовательного порта

Чтобы выгрузить скетч в плату Arduino, щелкните на кнопке **Upload** (Загрузка) на панели инструментов — второй слева, которая подсвечена на рис. 1.7.

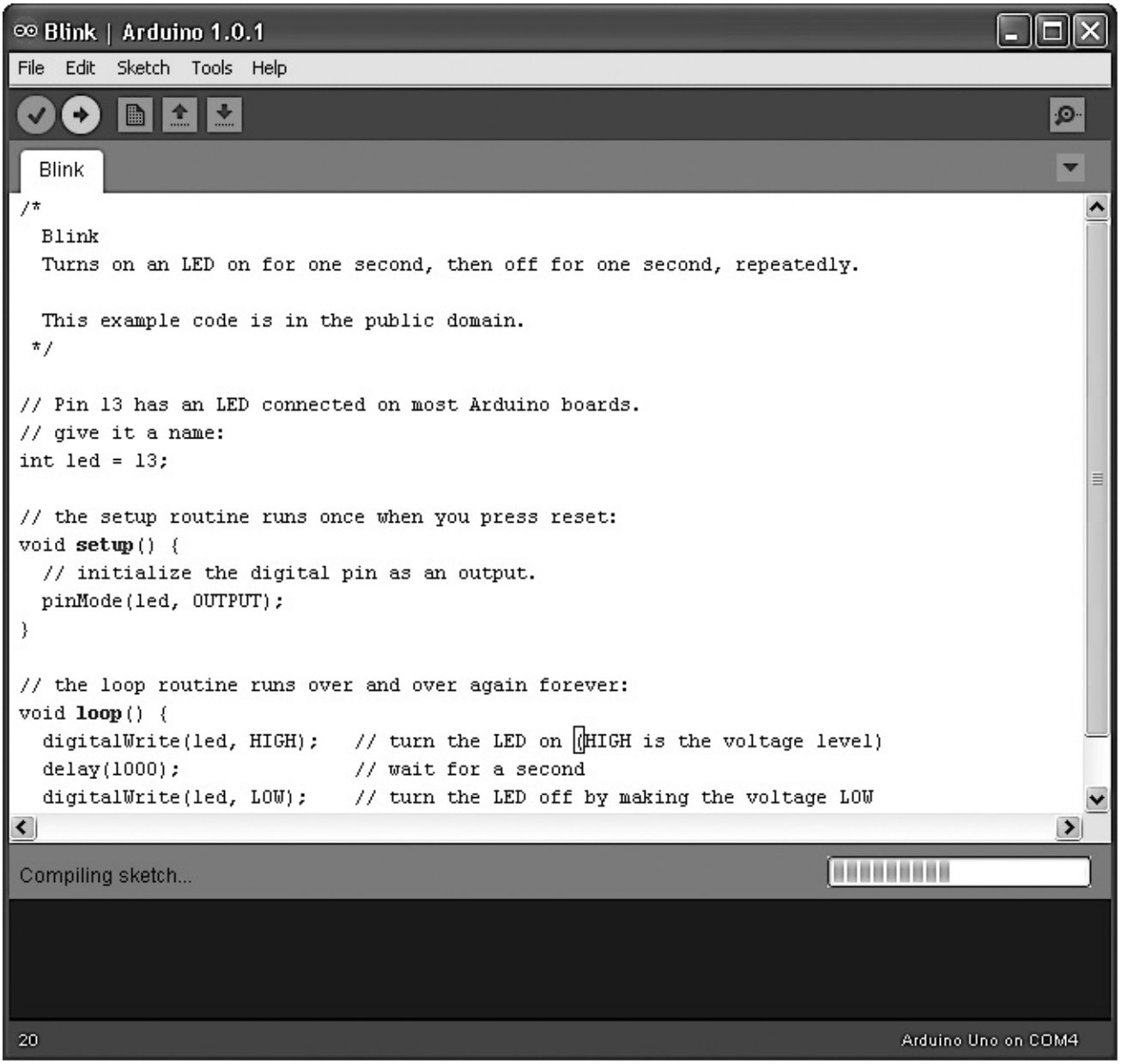


Рис. 1.7. Выгрузка скетча Blink

После щелчка на кнопке **Upload** (Загрузка) справа внизу в окне Arduino IDE появится индикатор выполнения, отражающий ход компиляции скетча (то есть его преобразования в формат, подходящий для выгрузки). Затем какое-то время должны мигать светодиоды с метками **Rx** и **Tx** на плате Arduino. И наконец, должен начать мигать светодиод с меткой **L**. В нижней части окна Arduino IDE должно также появиться сообщение «Binary sketch size: 1,084 bytes (of a 32,256 byte maximum)» («Скетч использует 1084 байт (3%) памяти устройства. Всего доступно 32 256 байт»)[5]. Оно означает, что скетч занимает около 1 Кбайт флеш-памяти из 32 Кбайт, доступных в Arduino для программ.

Прежде чем приступить к программированию, давайте познакомимся с аппаратным окружением, в котором вашим программам, или скетчам, предстоит работать и которое они смогут использовать.

Обзор платы Arduino

На рис. 1.8 показано устройство платы Arduino. В левом верхнем углу рядом с разъемом USB находится кнопка сброса. Нажатие на нее посылает логический импульс на вывод **Reset** микроконтроллера, который в ответ

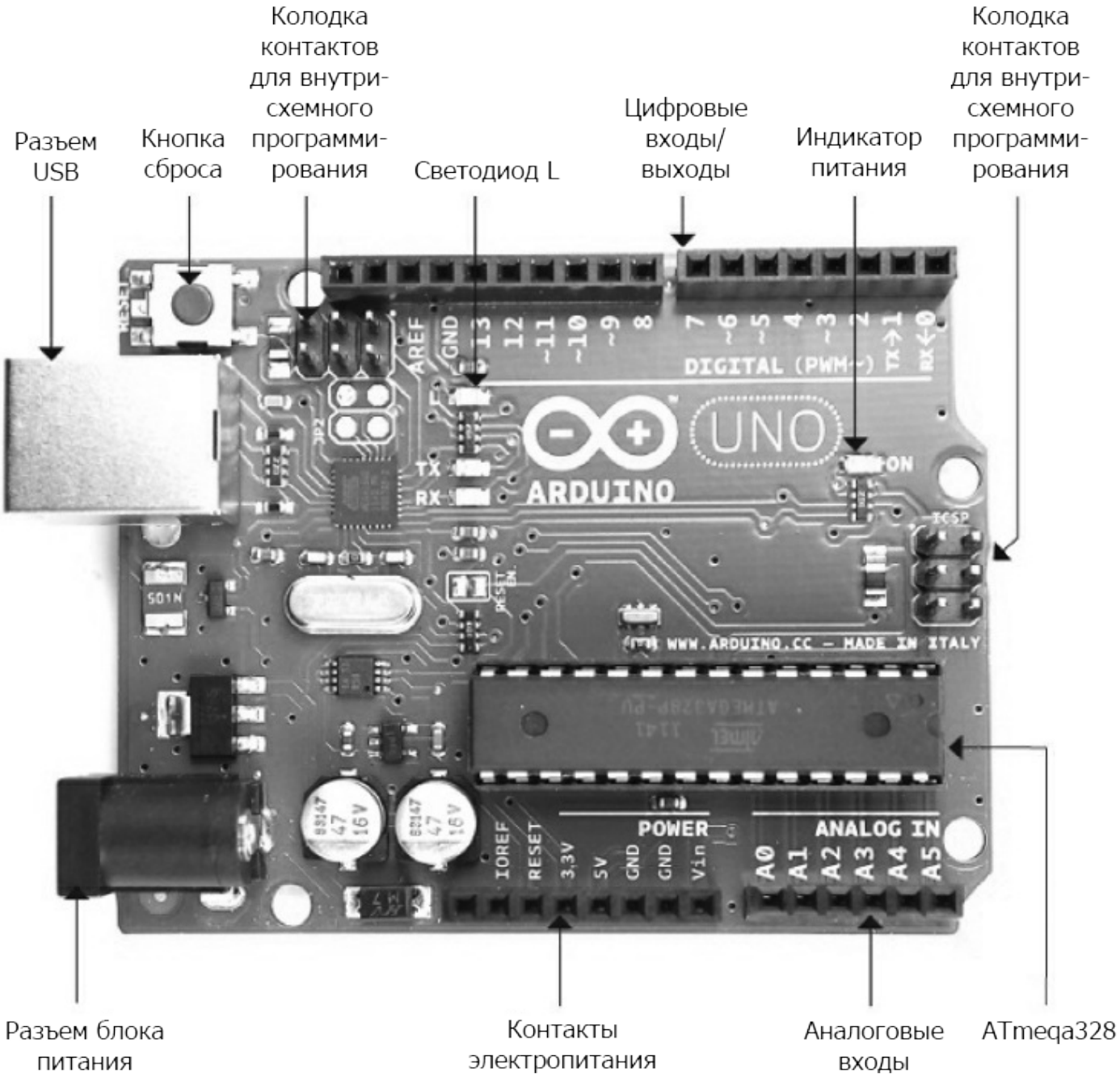


Рис. 1.8. Устройство платы Arduino

очищает свою память и запускает программу с самого начала. Обратите внимание на то, что программа, хранящаяся в устройстве, сохраняется, потому что находится в энергонезависимой флеш-памяти, то есть в памяти, не утрачивающей свое содержимое даже при выключении электропитания.

Электропитание

Электропитание платы Arduino возможно через разъем USB или через разъем внешнего блока питания, находящийся ниже. На этот разъем допускается подавать постоянное напряжение от 7,5 до 12 В. Сама плата Arduino потребляет около 50 мА. Поэтому небольшой 9-вольтовой батареи типа «Крона» (200 мА·ч) достаточно, чтобы питать плату в течение примерно четырех часов.

При подаче питания на плату загорается индикатор питания (справа на плате Uno, слева на плате Leonardo).

Контакты электропитания

Рассмотрим теперь контакты в нижнем ряду на рис. 1.8. Все контакты, кроме первого, подписаны.

Первый контакт, без метки, зарезервирован для использования в будущем. Следующий контакт, **IOREF**, служит для определения опорного напряжения, на котором работает плата. Обе модификации, Uno и Leonardo, используют напряжение 5 В, поэтому на данном контакте всегда будет присутствовать напряжение 5 В, но в этой книге он не будет использоваться. Его назначение — позволить платам расширения, подключаемым к 3-вольтовым модификациям Arduino, таким как Arduino Due, определять напряжение, на котором работает плата, и адаптироваться к нему.

Следующий контакт — **Reset**. Он служит той же цели, что и кнопка сброса. По аналогии с перезагрузкой компьютера контакт **Reset** позволяет сбросить микроконтроллер в исходное состояние и заставить его выполнять программу с самого начала. Чтобы сбросить микроконтроллер с помощью этого контакта, необходимо кратковременно подать на него низкое напряжение (замкнуть на «землю»). Маловероятно, что вам когда-нибудь потребуется этот контакт, но знать о его существовании полезно.

Остальные контакты в этой группе служат для вывода электропитания с разными уровнями напряжения (**3.3V**, **5V**, **GND** и **9V**) в соответствии с обозначениями. **GND**, или *ground* («земля»), означает 0 В. Контакты **GND** служат опорными точками, относительно которых измеряется напряжение во всех других точках на плате.

Два контакта **GND** совершенно идентичны, наличие двух контактов **GND** иногда оказывается полезно. В действительности в верхнем ряду на плате есть еще один контакт **GND**.

Аналоговые входы

Контакты в следующей группе подписаны **Analog In** (аналоговые входы) с номерами от 0 до 5. Эти шесть контактов можно использовать для измерения напряжения и его анализа в скетче. Несмотря на то что они обозначены как аналоговые входы, их можно использовать и как цифровые входы или выходы. Но по умолчанию они действуют как аналоговые входы.

Цифровые входы

Теперь перейдем к верхнему ряду контактов (см. рис. 1.8) и начнем движение справа налево. Здесь находятся контакты, обозначенные **Digital 0**...**13**. Их можно использовать как цифровые входы или выходы. Если включить такой контакт из скетча, на нем появится напряжение 5 В, а если выключить — напряжение упадет до 0 В. Подобно контактам электропитания, их следует использовать осторожно, чтобы не превысить максимально допустимый ток.

Цифровые выходы могут отдавать ток до 40 мА с напряжением 5 В — этого более чем достаточно для питания светодиода, но недостаточно для непосредственного управления электромотором.

Платы Arduino

Модель Arduino Uno (см. рис. 1.1) является последней версией оригинальной платы Arduino. Это самая распространенная модель Arduino, и обычно, когда кто-то говорит, что использует Arduino, подразумевается именно эта модель.

Все остальные модели плат Arduino сконструированы для удовлетворения особых потребностей, таких как большая величина тока на входных и выходных контактах, более высокая производительность, меньший размер, возможность вшивания в элементы одежды и подключения телефонов на Android, простота подключения к беспроводным сетям и т.д.

Независимо от конструктивных особенностей, все платы программируются из Arduino IDE, немного различаясь лишь некоторыми особенностями программного обеспечения, которое они могут использовать. Поэтому, узнав, как использовать одну плату Arduino, вы сможете применять полученные знания для работы с другими моделями.

Давайте рассмотрим спектр официальных версий платы Arduino. Существуют разные модели Arduino, отличные от обсуждаемых в этой книге, но они не так популярны. Полный их перечень можно найти на официальном веб-сайте Arduino (**www.arduino.cc**).

Uno и похожие модели

Модель Uno R3 является последней в серии стандартных плат, включающей также модели Uno, Duemilanove, Diecimila и NG. Все эти платы построены на основе микропроцессоров ATmega168 и ATmega328, которые различаются только объемом памяти.

Другой современной моделью Arduino того же размера и с тем же набором контактов, что и Uno R3, является Arduino Leonardo (рис. 1.9). Как видите, эта плата содержит меньше электронных компонентов, чем Uno. Это объясняется

использованием другого процессора. Плата Leonardo сконструирована на основе процессора ATmega32u4, схожего с ATmega328, но имеющего встроенный интерфейс USB, благодаря чему отпала необходимость в дополнительных компонентах, которые можно увидеть на плате Uno. Кроме того, модель Leonardo имеет немного больше памяти, больше аналоговых входов и обладает некоторыми другими преимуществами. Она также немного дешевле Uno. Во многих отношениях она имеет также более удачную конструкцию, чем Uno.

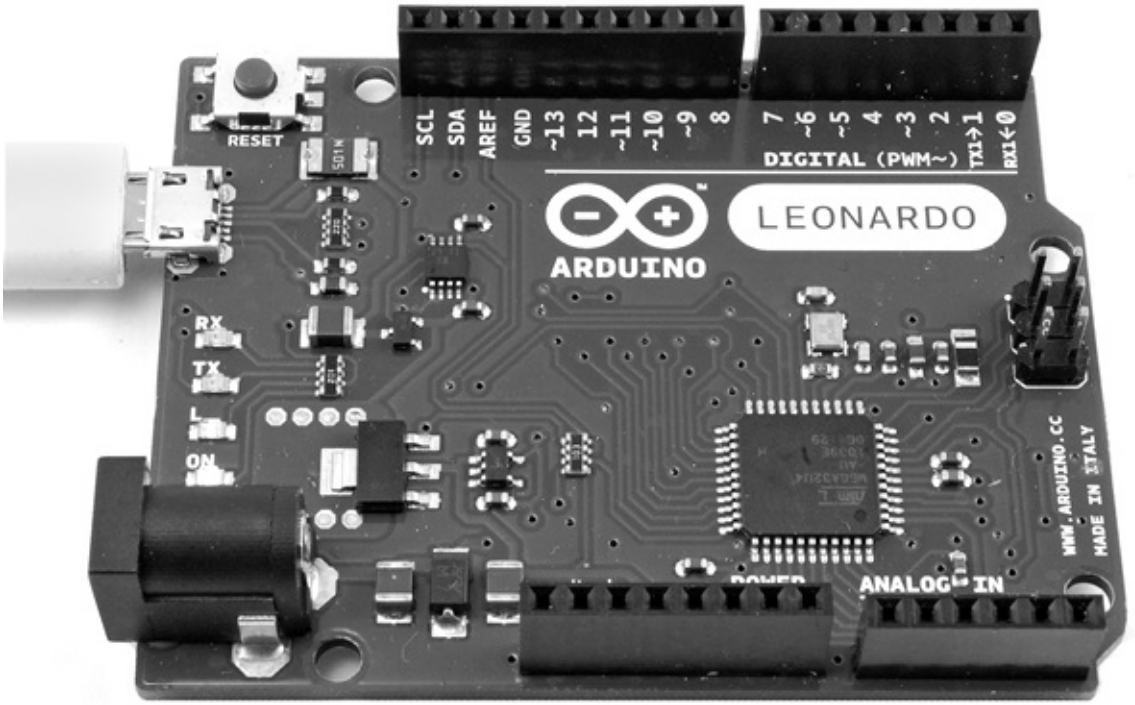


Рис. 1.9. Arduino Leonardo

Но если все перечисленное верно, возникает резонный вопрос: почему Leonardo не пользуется большей популярностью, чем Uno? Причина в том, что усовершенствования, внесенные в плату Leonardo, ухудшили обратную совместимость с Uno и другими предшествующими моделями. Некоторые платы расширения, особенно старой конструкции, не будут работать с Leonardo. Со временем эти отличия станут доставлять все меньше хлопот, и будет интересно посмотреть, смогут ли модель Leonardo и ее последующие версии завоевать наибольшую популярность.

Относительно недавно в арсенале Arduino появилась плата Arduino Ethernet. Она объединяет основные характеристики Uno с интерфейсом Ethernet, позволяющим подключаться к сети без использования дополнительной платы расширения Ethernet.

Большие платы Arduino

Иногда количества контактов ввода/вывода на платах Uno и Leonardo оказывается недостаточно для решения поставленных задач. В таких ситуациях вы оказываетесь перед выбором между приобретением дополнительных плат расширения для Uno или

переходом на использование плат большего размера.

СОВЕТ

Если вы только начинаете знакомиться с Arduino, воздержитесь от покупки большой платы. Они выглядят привлекательно, обладая большим числом контактов и большим быстродействием, но имеют проблемы совместимости с платами расширения. Пока вам лучше остановить свой выбор на стандартной модели Uno.

Модели Arduino большего размера имеют тот же набор контактов, что и Uno, а также двойной ряд дополнительных контактов ввода/вывода с торцевой стороны и более длинные ряды контактов по боковым сторонам (рис. 1.10).

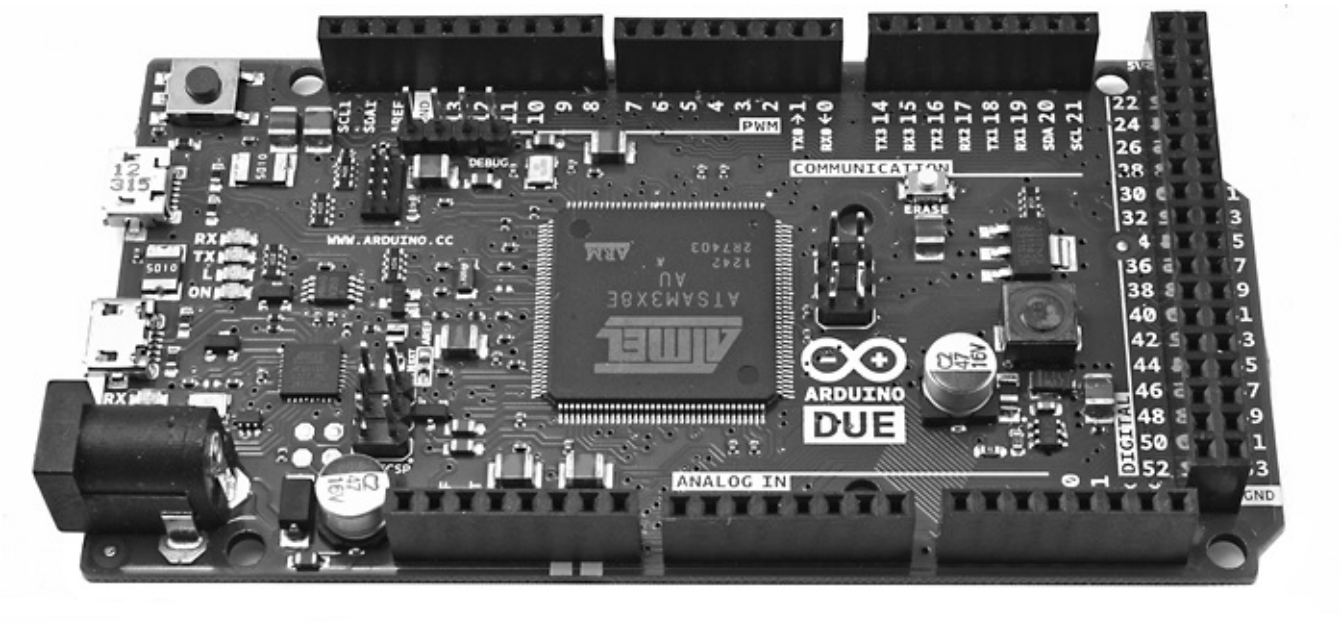


Рис. 1.10. Arduino Due

Традиционно самой большой считается Arduino Mega 2560. Эти платы, подобно всем другим большим платам Arduino, имеют больше памяти каждого вида. Платы Mega 2560 и Mega ADK комплектуются процессорами с производительностью, схожей с производительностью процессора в модели Arduino Uno. Но в целом Arduino Due — более «мощная машина». Эта плата комплектуется процессором с тактовой частотой 84 МГц (сравните с 16 МГц модели Uno), но имеет проблемы совместимости с другими моделями. Самая большая из них состоит в том, что для электропитания Due должно использоваться напряжение 3,3 В вместо 5 В, как для большинства предыдущих моделей Arduino. Неудивительно, что многие платы расширения несовместимы с ней.

Однако эта плата имеет множество преимуществ, значимых для большинства проектов с высокими требованиями:

- большой объем памяти для программ и данных;

- аппаратная поддержка вывода звуков (аппаратные цифроаналоговые преобразователи);
- четыре последовательных порта;
- два порта USB;
- интерфейсы USB-хоста и USB OTG;
- имитация USB-клавиатуры и USB-мыши.

Маленькие платы Arduino

Для одних проектов модель Uno может оказаться слишком маленькой, но для других — слишком большой. Несмотря на невысокую стоимость плат Arduino, они становятся слишком дорогим удовольствием, если включать их в каждый проект. Существует целый спектр маленьких и специализированных плат Arduino, которые имеют меньший размер, чем обычная модель Uno, или более низкую цену за счет отсутствия каких-то особенностей, не требующихся в большинстве проектов.

На рис. 1.11 изображена плата Arduino Mini. Эта модель не имеет интерфейса USB, а ее программирование осуществляется с применением отдельного модуля расширения. Помимо Mini существуют также модели Nano и Micro. Обе они имеют встроенный интерфейс USB, но и стоят дороже.

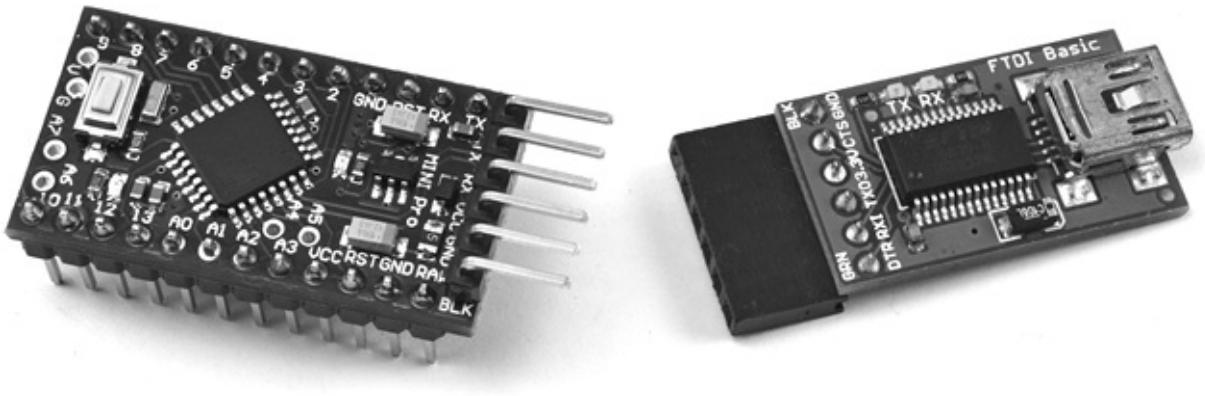


Рис. 1.11. Arduino Mini и Arduino Programmer

Платы LilyPad и LilyPad USB

Плата LilyPad и более новая ее версия LilyPad USB — одни из самых интересных моделей Arduino (рис. 1.12). Эти платы можно вшивать в элементы одежды и соединять их токопроводящими нитями со светодиодами, выключателями, акселерометрами и другими устройствами. Для программирования более старых плат LilyPad требуется использовать отдельный интерфейс USB, как в случае с Arduino Mini.

Однако эти платы постепенно вытесняются более новой модификацией Arduino LilyPad USB, имеющей встроенный разъем USB.

Рис. 1.12. Arduino LilyPad

Неофициальные платы Arduino

Благодаря статусу открытого аппаратного обеспечения помимо «официальных» плат, описанных ранее, появилось множество неофициальных копий и модификаций Arduino. Прямые клоны Arduino, которые без труда можно найти на eBay и других недорогих торговых площадках, являются простыми копиями плат Arduino. Единственное их преимущество — невысокая цена. Но существует также ряд интересных Arduino-совместимых разработок, предлагающих дополнительные возможности.

В числе примеров такого рода плат, которым стоит уделить внимание, можно назвать:

- EtherTen — аналог платы Arduino Ethernet (www.freetronics.com/products/etherten);
- Leostick A — малогабаритный аналог платы Leonardo со встроенным разъемом USB (www.freetronics.com/collections/arduino/products/leostick).

Теперь, после знакомства с аппаратной стороной Arduino, можно перейти к знакомству с возможностями их программирования.

Язык программирования

Многие ошибочно полагают, что платы Arduino имеют собственный язык программирования. В действительности программы для них пишутся на языке с простым названием C. Этот язык существует с самых первых дней развития вычислительной техники. А вот что действительно привносит Arduino — это набор простых в использовании команд, написанных на C, которые вы можете использовать в своих программах.

Пуристы могут заметить, что в Arduino используется C++, объектно-ориентированное расширение языка C. Строго говоря, они правы, однако наличие всего 1–2 Кбайт памяти обычно означает, что использование объектно-ориентированных приемов при программировании для Arduino не самая лучшая идея, за исключением особых ситуаций, и фактически программы пишутся на C.

Начнем с изменения скетча Blink.

Изменение скетча Blink

Может так случиться, что при первом включении ваша плата Arduino уже мигает светодиодом. Это объясняется тем, что платы Arduino часто поставляются с установленным скетчем Blink.

Если у вас именно такая плата, вам может понравиться предложение изменить частоту мигания, чтобы убедиться, что вы можете сделать что-то своими руками. Давайте рассмотрим скетч Blink, чтобы понять, какие изменения следует внести, чтобы заставить светодиод мигать чаще.

Первая часть скетча — это комментарий, описывающий назначение скетча. Комментарий не является программным кодом. В процессе подготовки кода к выгрузке все такие комментарии удаляются. Все, что находится между парами символов `/*` и `*/`, игнорируется компьютером и адресовано людям.

```
/*
  Blink
   Включает светодиод на одну секунду, затем выключает на одну
секунду, и так много раз.


  Этот пример кода находится в свободном доступе.
*/
```

Далее идут два однострочных комментария. Они похожи на блочные комментарии, но в отличие от них начинаются с пары символов `//`. Эти комментарии описывают происходящее. В данном случае комментарий сообщает вам, что контакт с номером 13 — это тот самый контакт, которым мы собираемся управлять. Мы выбрали этот контакт, потому что на плате Arduino Uno он подключен к светодиоду **L**.

```
// На большинстве плат Arduino к контакту 13 подключен светодиод.
// Дадим ему имя:
int led = 13;
```

Следующая часть скетча — функция `setup`. Эта функция должна присутствовать в каждом скетче, и она выполняется всякий раз, когда происходит сброс платы Arduino, либо в результате (как сообщает комментарий) нажатия на кнопку сброса **Reset**, либо после подачи электропитания на плату.

```
// процедура setup выполняется один раз после нажатия на кнопку
сброса
void setup(){
  // инициализировать контакт как цифровой выход
  pinMode(led, OUTPUT);
```

```
}
```

Структура этого текста может показаться немного странной тем, кто только начинает изучать программирование. *Функция* — это фрагмент программного кода, имеющий собственное имя (в данном случае `setup`). Пока просто используйте предыдущий текст как шаблон и помните, что скетч должен начинаться строкой `void setup() {`, за которой следуют необходимые команды, каждая в отдельной строке, завершающиеся точкой с запятой (`;`). Конец функции отмечается символом `}`.

В данном случае Arduino выполнит единственную команду `pinMode(led, OUTPUT)`, которая настраивает контакт на работу в режиме выхода.

Далее следует основная часть скетча, функция `loop`.

По аналогии с функцией `setup` каждый скетч должен иметь функцию `loop`. Но в отличие от функции `setup`, которая выполняется только один раз после сброса, `loop` выполняется снова и снова. То есть как только будут выполнены все ее инструкции, она тут же запускается снова.

Функция `loop` включает светодиод, выполняя инструкцию `digitalWrite(led, HIGH)`. Затем выполняется команда `delay(1000)`, приостанавливающая скетч на 1 с. Значение 1000 здесь обозначает 1000 мс, или 1 с. После этого светодиод выключается, скетч приостанавливается еще на 1 с, и процесс повторяется.

```
// Процедура loop выполняется снова и снова, до бесконечности
void loop() {
  digitalWrite(led, HIGH);  // включить светодиод (HIGH — уровень
напряжения)
  delay(1000);              // ждать 1 с
   digitalWrite(led, LOW);    // выключить светодиод, установив
уровень напряжения LOW
  delay(1000);              // ждать 1 с
}
```

Чтобы увеличить частоту мигания светодиода, заменим оба числа 1000 числом 200. Обе замены должны быть произведены в функции `loop`, поэтому теперь она должна выглядеть так:

```
void loop() {
  digitalWrite(led, HIGH);  // включить светодиод (HIGH — уровень
напряжения)
  delay(200);               // ждать 1 с
   digitalWrite(led, LOW);    // выключить светодиод, установив
уровень напряжения LOW
  delay(200);               // ждать 1 с
```

```
}
```

Если попытаться сохранить скетч перед выгрузкой, Arduino IDE напомнит, что этот скетч является примером и доступен только для чтения, но предложит сохранить копию, которую вы затем сможете изменять по своему усмотрению.

Однако сейчас этого делать не нужно — просто выгрузите скетч в плату, не сохраняя его. Если сохранить этот или другой скетч, вы увидите, что он появится в меню **File—>Sketchbook** (Файл—>Папка со скетчами) Arduino IDE.

Итак, щелкните на кнопке **Upload** (Загрузка) еще раз, и, когда выгрузка завершится, плата Arduino сама сбросится и светодиод должен начать мигать чаще.

Переменные

Переменные помогают дать имена числам. В действительности их возможности намного шире, но пока мы будем использовать их только для этой цели.

При объявлении переменной в языке C необходимо указать ее тип. Например, если нужна переменная, хранящая целое число, ее следует объявить с типом **int** (сокращенно от *integer* — целое со знаком). Чтобы определить переменную с именем `delayPeriod` и значением 200, нужно записать такое объявление:

```
int delayPeriod = 200;
```

Так как `delayPeriod` — это имя, в нем не может быть пробелов. По общепринятым соглашениям имена переменных должны начинаться с буквы в нижнем регистре, а каждое новое слово в имени — с буквы в верхнем регистре. Такую «горбатую» форму записи имен программисты часто называют *верблюжьей нотацией* (camel case).

Давайте добавим эту переменную в скетч Blink, чтобы вместо жестко зашитого значения 200, определяющего продолжительность паузы, можно было использовать имя переменной:

```
int led = 13;
int delayPeriod = 200;

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
  delay(delayPeriod);
  digitalWrite(led, LOW);
```

```
  delay(delayPeriod);
}
```

Везде в скетче, где прежде использовалось число `200`, сейчас стоит ссылка на переменную `delayPeriod`.

Если теперь вы пожелаете заставить светодиод мигать еще чаще, достаточно будет изменить значение `delayPeriod` в одном месте.

If

Обычно строки программы выполняются по порядку, одна за другой, без исключений. Но как быть, если потребуется изменить порядок выполнения? Что если какая-то часть скетча должна выполняться только при определенном условии?

Хорошим примером может служить выполнение фрагмента, только когда нажата кнопка, подключенная к плате Arduino. Такой код мог бы выглядеть примерно так:

```
void setup()
{
  pinMode(5, INPUT_PULLUP);
  pinMode(9, OUTPUT);
}
void loop()
{
  if (digitalRead(5) == LOW)
  {
    digitalWrite(9, HIGH);
  }
}
```

В этом случае условие (после оператора `if`) выполняется, если с контакта **5** прочитано значение `LOW`. Два знака «равно» `==` обозначают операцию определения равенства двух значений. Ее легко спутать с единственным знаком «равно», обозначающим операцию присваивания значения переменной. Оператор `if` говорит: если условие истинно (то есть выполняется), то должны быть выполнены команды в фигурных скобках. В данном случае команда, устанавливающая уровень напряжения `HIGH` на цифровом выходе **9**.

Если условие ложно (не выполняется), то Arduino просто перешагнет через фигурные скобки и продолжит выполнение программы. В данном случае будет достигнут конец функции `loop`, и она запустится снова.

Циклы

По аналогии с выполнением некоторых операций по условию иногда в скетчах возникает необходимость повторять операции снова и снова. Конечно, вы и так получаете это поведение, помещая команды в функцию `loop`. То есть именно так действует Blink.

Но иногда требуется выполнить команды определенное число раз. Добиться этого можно с помощью команды `for`, позволяющей использовать переменную-счетчик. Например, напишем скетч, который мигает светодиодом 10 раз. Далее вы узнаете, почему это решение нельзя признать идеальным при определенных обстоятельствах, но сейчас оно вполне отвечает нашим потребностям.

```
// sketch 01_01_blink_10
int ledPin = 13;
int delayPeriod = 200;
void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
  for (int i = 0; i < 10; i++)
  {
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
  }
}
```

ПРИМЕЧАНИЕ

Это наш первый законченный скетч, поэтому его имя указано в строке комментария в начале файла. Все скетчи с подобными именами можно загрузить на веб-сайте автора www.simonmonk.org. Чтобы установить все скетчи в окружение Arduino, распакуйте файл со скетчами в каталог Arduino, который находится в папке Documents (Документы). Среда разработки Arduino IDE автоматически создает эту папку при первом запуске.

Команда `for` определяет переменную с именем `i` и присваивает ей начальное значение `0`. За точкой с запятой (`;`) следует текст `i < 10`. Это условие продолжения

цикла. Иными словами, пока значение `i` остается меньше 10, продолжают выполняться команды, заключенные в фигурные скобки.

Последний элемент в команде `for` — `i++`. Это сокращенная форма записи выражения `i = i + 1`, которое прибавляет 1 к значению `i`. Увеличение значения `i` на единицу происходит в конце каждого цикла. Это выражение гарантирует прекращение цикла, потому что, увеличивая `i` на 1, вы в конечном счете получите значение больше 10.

Функции

Функции — это способ группировки программных команд в удобный для использования блок. Функции помогают поделить скетч на управляемые и простые в использовании фрагменты.

Например, напишем скетч, который сначала быстро мигает светодиодом 10 раз, а затем начинает мигать с частотой один раз в секунду.

Прочитайте следующий листинг, а потом я объясню, как он работает.

```
// sketch 01_02_blink_fast_slow
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
  flash(10, 100);
}

void loop()
{
  flash(1, 500);
}

void flash(int n, int delayPeriod)
{
  for (int i = 0; i < n; i++)
  {
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
```

```
  }
}
```

Функция `setup` теперь включает строку `flash(10, 100);`. Она означает: «мигнуть 10 раз с периодом задержки `delayPeriod` 100 мс». Команда `flash` не является встроенной командой Arduino — вы создадите эту очень полезную функцию сами.

Определение функции находится в конце скетча. Первая строка в определении функции

```
void flash(int n, int delayPeriod)
```

сообщает Arduino, что определяется новая функция с именем `flash`, которая принимает два параметра, оба типа `int`. Первый параметр, с именем `n`, определяет, сколько раз светодиод должен мигнуть, а второй, с именем `delayPeriod`, определяет величину паузы после включения или выключения светодиода.

Эти два параметра можно использовать только внутри функции. Так, `n` используется в команде `for`, где определяет количество повторений цикла, а `delayPeriod` — внутри команд `delay`.

Функция `loop` скетча также использует функцию `flash`, но с более длинным периодом задержки `delayPeriod` и количеством повторений, равным 1. Так как эта команда находится внутри функции `loop`, она будет выполняться снова и снова, заставляя светодиод мигать непрерывно.

Цифровые входы

Чтобы извлечь максимум пользы из данного раздела, найдите короткий кусок провода или просто металлическую скрепку.

Загрузите следующий скетч и запустите его:

```
// sketch 01_03_paperclip
int ledPin = 13;
int switchPin = 7;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(switchPin, INPUT_PULLUP);
}

void loop()
```

```
{
  if (digitalRead(switchPin) == LOW)
  {
    flash(100);
  }
  else
  {
    flash(500);
  }
}

void flash(int delayPeriod)
{
  digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
  digitalWrite(ledPin, LOW);
  delay(delayPeriod);
}
```

Используя кусок провода или выпрямленную скрепку, замкните контакты **GND** и **7**, как показано на рис. 1.13. Это можно делать на включенной плате Arduino, но только после выгрузки в нее скетча. Это связано с тем, что некий предыдущий скетч мог настроить контакт **7** на работу в режиме выхода — в этом случае замыкание на **GND** может повредить контакт. Так как скетч настраивает контакт **7** на работу в режиме входа, его безопасно соединять с контактом **GND**.

Вот что происходит в результате: когда контакты замкнуты, светодиод мигает чаще, а когда не замкнуты — реже.

Давайте исследуем скетч и посмотрим, как он работает.

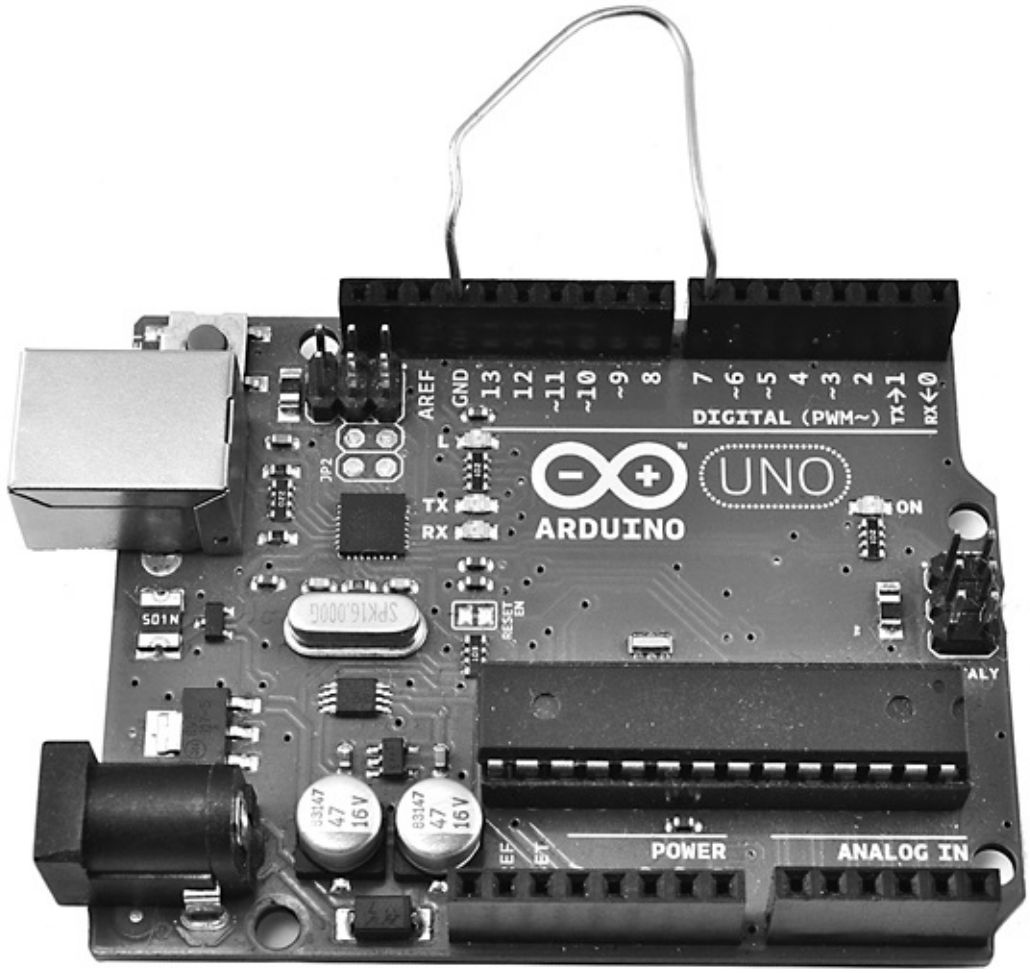


Рис. 1.13. Опыт с цифровым входом

Во-первых, в скетче появилась новая переменная с именем `switchPin`. Этой переменной присвоен номер контакта **7**. Скрепка в данном эксперименте играет роль выключателя. В функции `setup` контакт **7** настраивается на работу в режиме входа командой `pinMode`. Во втором аргументе команде `pinMode` передается не просто значение `INPUT`, а `INPUT_PULLUP`. Оно сообщает плате Arduino, что по умолчанию вход должен иметь уровень напряжения `HIGH`, если на него не подан уровень `LOW` соединением этого контакта с контактом **GND** (скрепкой).

В функции `loop` мы используем команду `digitalRead` для проверки уровня напряжения на входном контакте. Если он равен `LOW` (скрепка замыкает контакты), вызывается функция с именем `flash` и значением 100 (в параметре `delayPeriod`). Это заставляет светодиод мигать чаще.

Если входной контакт имеет уровень напряжения `HIGH`, выполняются команды в разделе `else` инструкции `if`. Здесь вызывается та же функция `flash`, но с более продолжительной задержкой, заставляющей светодиод мигать реже. Функция `flash` является упрощенной версией функции `flash` из предыдущего скетча, она просто включает и выключает светодиод один раз с указанной задержкой.

Цифровые входы могут соединяться с цифровыми выходами других модулей, которые действуют не так, как выключатель, но устанавливают те же уровни

напряжения HIGH и LOW. В таких случаях функции pinMode следует передавать аргумент INPUT вместо INPUT_PULLUP.

Цифровые выходы

Немного нового можно сказать о цифровых выходах с точки зрения программирования после экспериментов с контактом **13**, к которому подключен встроенный светодиод.

Настройка контактов на работу в режиме цифровых выходов осуществляется в функции setup с помощью следующей команды:

```
pinMode(outputPin, OUTPUT);
```

Чтобы на цифровом выходе установить уровень напряжения HIGH или LOW, нужно вызывать команду digitalWrite:

```
digitalWrite(outputPin, HIGH);
```

Монитор последовательного порта

Так как плата Arduino подключается к компьютеру через порт USB, есть возможность пересылать сообщения между ними, используя компонент Arduino IDE, который называется *монитором последовательного порта* (Serial Monitor). Для иллюстрации изменим скетч 01_03 так, чтобы вместо изменения частоты мигания светодиода после установки уровня напряжения LOW на цифровом входе **7** он посылал сообщение.

Загрузите следующий скетч:

```
// sketch 01_04_serial
int switchPin = 7;

void setup()
{
  pinMode(switchPin, INPUT_PULLUP);
  Serial.begin(9600);
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {
    Serial.println("Paperclip connected");
  }
  else
```

```
  {
    Serial.println("Paperclip NOT connected");
  }
  delay(1000);
}
```

Теперь откройте монитор последовательного порта в Arduino IDE, щелкнув на кнопке с изображением, напоминающим лупу. Вы сразу же должны увидеть несколько сообщений, появляющихся одно за другим (рис. 1.14).

Рис. 1.14. Монитор последовательного порта

Разъедините контакты, убрав скрепку, и вы должны увидеть, что текст сообщения изменился.

Так как встроенный светодиод в этом скетче не используется, отпала и необходимость в переменной `ledPin`. Зато появилась новая команда `Serial.begin`, запускающая обмен сообщениями через последовательный порт. Ее параметр определяет скорость передачи. Подробнее о взаимодействиях через последовательный порт рассказывается в главе 13.

Чтобы записать сообщение в монитор порта, достаточно выполнить команду `Serial.println`.

В данном примере Arduino посылает сообщения в монитор последовательного порта.

Массивы и строки

Массивы предназначены для хранения списков значений. Переменные, которые нам встречались до сих пор, могли хранить только одно значение, обычно типа `int`. Массив, напротив, может хранить список значений и позволяет обращаться к отдельным значениям по их позициям в списке.

В C, как и в большинстве других языков программирования, нумерация позиций в массиве начинается с 0, а не с 1. Это означает, что первый элемент фактически является нулевым элементом.

Мы уже сталкивались с одной из разновидностей массивов в предыдущем разделе, где знакомились с монитором последовательного порта. Сообщения, такие как «Paperclip NOT connected» (скрепка не замыкает контакты), называют *массивами символов*, потому что фактически они являются коллекциями символов.

Например, научим Arduino посылать в монитор порта всякую чепуху.

Следующий скетч имеет массив массивов символов. Он выбирает их по одному в случайном порядке и посылает в монитор последовательного порта через случайные интервалы времени. Попутно этот скетч показывает, как в Arduino получать случайные числа.

```
// sketch 01_05_gibberish

char* messages[] = {
                "My name is Arduino",
                "Buy books by Simon Monk",
                "Make something cool with me",
                "Raspberry Pis are fruity"};

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  int delayPeriod = random(2000, 8000);
  delay(delayPeriod);
  int messageIndex = random(4);
  Serial.println(messages[messageIndex]);
}
```

Все сообщения, или *строки*, как часто называют коллекции символов, имеют тип `char*`. Символ звездочки (*) говорит о том, что это указатель на что-то. Подробнее об указателях будет рассказываться в главе 6. Квадратные скобки ([]) в конце объявления переменной указывают, что данная переменная хранит массив данных типа `char*`, а не единственное значение `char*`.

Внутри функции `loop` переменной `delayPeriod` присваивается случайное значение из диапазона от `2000` до `7999` (второй аргумент `random` не входит в диапазон). Затем вызовом функции `delay` выполняется пауза, продолжительность которой равна полученному промежутку.

Переменной `messageIndex` также присваивается случайное значение с помощью команды `random`, но на этот раз ей передается единственный параметр, в результате чего она возвращает случайное число в диапазоне от 0 до 3, которое затем используется как индекс сообщения для отправки в монитор порта.

Наконец, сообщение, находящееся в выбранной позиции, посылается в монитор порта. Опробуйте этот скетч, не забыв открыть окно монитора последовательного порта.

Аналоговые входы

Контакты с метками от **A0** до **A5** на плате Arduino можно использовать для измерения приложенного к ним напряжения. Уровень напряжения должен находиться в диапазоне от 0 до 5 В. Измерение выполняется с помощью встроенной функции `analogRead`, которая возвращает значение в диапазоне от 0 до 1023: значение 0 соответствует напряжению 0 В, а значение 1023 — напряжению 5 В. То есть, чтобы преобразовать число в значение, находящееся в диапазоне от 0 до 5, нужно разделить полученное число на 5: 1023/5 = 204,6.

Тип данных `int` не очень хорошо подходит для измерения напряжения, так как представляет только целые числа, а нам было бы желательно видеть также дробную часть. Для этого следует использовать тип данных `float`.

Загрузите следующий скетч в плату Arduino и затем замкните скрепкой контакты **A0** и **3.3V** (рис. 1.15).

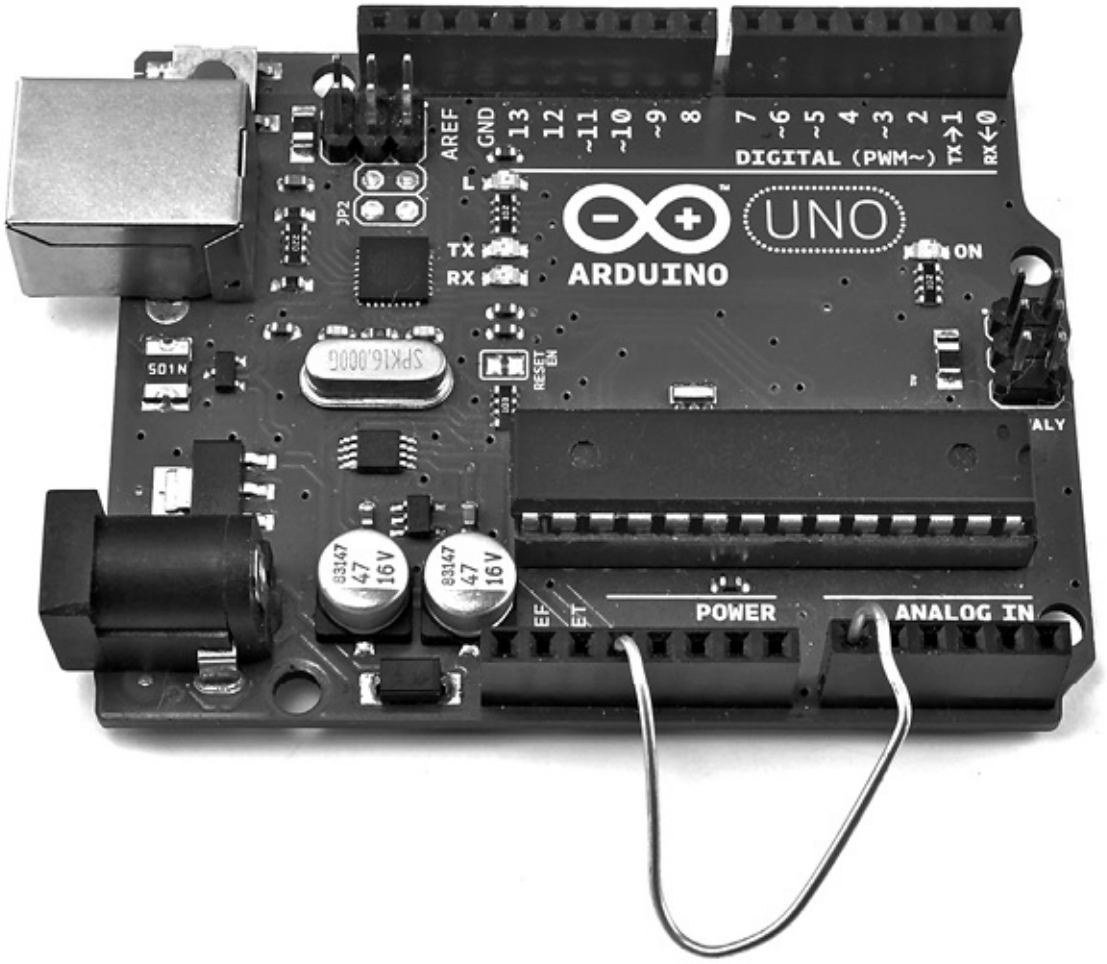


Рис. 1.15. Соединение контактов A0 и 3.3V

```
// sketch 01_06_analog
int analogPin = A0;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  int rawReading = analogRead(analogPin);
  float volts = rawReading / 204.6;
  Serial.println(volts);
  delay(1000);
}
```

Откройте монитор последовательного порта, и вы должны увидеть поток чисел (рис. 1.16). Числа должны быть близки к значению 3,3.

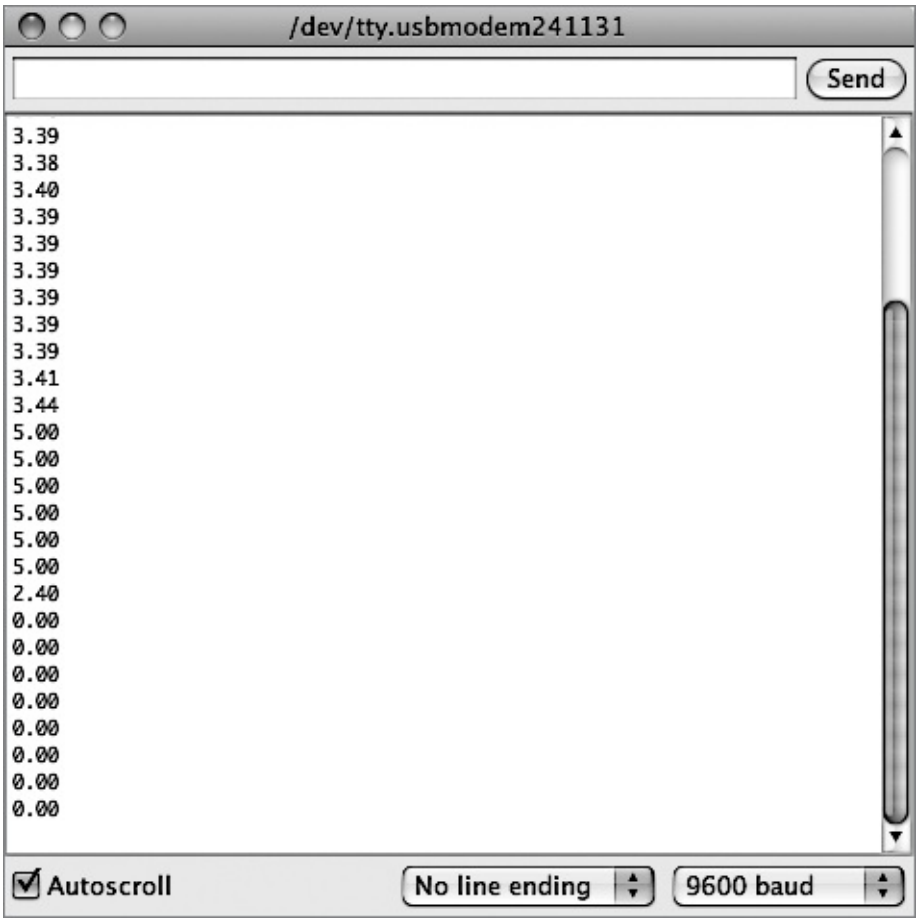


Рис. 1.16. Вывод значений напряжения

ВНИМАНИЕ

Не замыкайте между собой контакты электропитания (**5V**, **3.3V** и **GND**). Это может привести к выходу из строя платы Arduino, а может быть, и компьютера.

Если теперь один конец скрепки оставить соединенным с контактом **A0**, а другой подключить к контакту **5V**, числа в мониторе изменятся и будут близки к 5 В. Теперь соедините контакт **A0** с контактом **GND**, и вы увидите числа 0 В.

Аналоговые выходы

Плата Arduino Uno не имеет настоящих аналоговых выходов (такие выходы вы найдете на плате Arduino Due), но она имеет несколько выходов с широтно-импульсной модуляцией (Pulse-Width Modulation, PWM). Они имитируют аналоговые выходы, управляя длительностью импульсов (рис. 1.17).

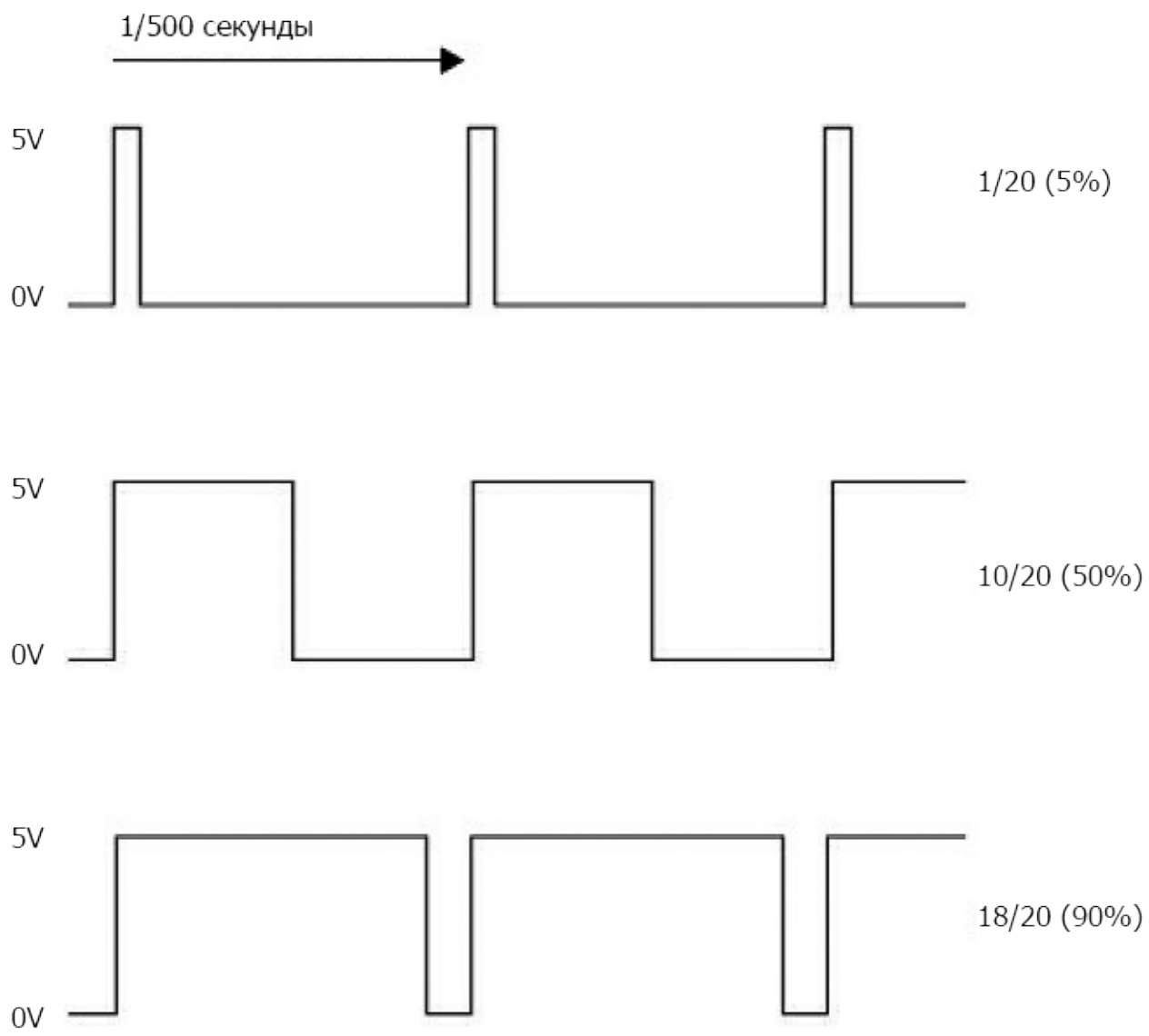


Рис. 1.17. Широтно-импульсная модуляция

Чем длиннее положительный импульс, тем выше среднее напряжение на выходе. Так как импульсы следуют с частотой 500 раз в секунду, а большинство устройств, которые вам доведется подключать к выходам **PWM**, не обладают мгновенной реакцией, возникает эффект изменения напряжения.

Контакты, отмеченные на плате Arduino Uno значком **~** (контакты **3**, **5**, **6**, **9**, **10** и **11**), можно использовать как аналоговые выходы.

Рис. 1.18. Измерение напряжения на выходе

Если у вас есть вольтметр, установите на нем диапазон измерения 0...20 В постоянного тока и подключите положительный щуп к контакту **6**, а отрицательный — к контакту **GND** (рис. 1.18). Затем загрузите следующий скетч.

```
// sketch 01_07_pwm
int pwmPin = 6;

void setup()
{
  pinMode(pwmPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  if (Serial.available())
  {
    int dutyCycle = Serial.parseInt();
```

```
    analogWrite(pwmPin, dutyCycle);
  }
}
```

Откройте монитор последовательного порта и введите число в диапазоне от 0 до 255 в текстовое поле в верхней части окна, слева от кнопки **Send** (Отправить). Затем щелкните на кнопке **Send** (Отправить) — вы должны увидеть, как на мультиметре изменится напряжение. Если послать число 0, напряжение должно упасть до 0. Отправка числа 127 должна дать середину между 0 и 5 В (2,5 В), а число 255 должно дать напряжение около 5 В.

В этом скетче функция `loop` начинается с оператора `if`. Условием для `if` является результат выполнения команды `Serial.available()`. То есть если монитор последовательного порта получил сообщение, выполняются команды внутри фигурных скобок. В этом случае команда `Serial.parseInt` преобразует текст сообщения, введенного в окне монитора порта, в значение типа `int`, которое затем передается как аргумент команде `analogWrite` для вывода импульсов на контакт.

Использование библиотек

Так как платы Arduino обладают весьма ограниченным объемом памяти, имеет смысл включать в программу, которая в конечном итоге окажется в плате, только тот код, который действительно потребуется. Один из способов добиться этого — использовать библиотеки. В Arduino и вообще в языке C под *библиотекой* понимается коллекция функций.

Например, Arduino IDE включает библиотеку для работы со светодиодным жидкокристаллическим дисплеем. Она занимает примерно 1,5 Кбайт памяти для программ. Нет никакого смысла подключать эту библиотеку, если она не используется, поэтому такие библиотеки подключаются только при необходимости.

Подключение выполняется добавлением директив `#include` в начало скетча. Добавить инструкции `include` для подключения любых библиотек, поставляемых в составе Arduino IDE, можно с помощью пунктов меню **Sketch—>Import Library...** (Скетч—>Подключить библиотеку).

В состав Arduino IDE входит большая коллекция официальных библиотек, в том числе:

- EEPROM — для сохранения данных в электрически стираемую программируемую постоянную память (ЭСППЗУ) (Electrically Erasable Programmable Read-Only Memory);
- Ethernet — для реализации сетевых взаимодействий;

- Firmata — стандартная библиотека для реализации взаимодействий через последовательный порт;
- LiquidCrystal — для работы с алфавитно-цифровыми жидкокристаллическими дисплеями;
- SD — для чтения и записи данных на карты флеш-памяти;
- Servo — для управления сервоприводами;
- SPI — для реализации взаимодействий по шине последовательного периферийного интерфейса;
- Software Serial — для реализации взаимодействий по последовательным линиям с использованием любых цифровых выходов;
- Stepper — для управления шаговыми электромоторами;
- WiFi — для доступа к беспроводной сети WiFi;
- Wire — для реализации взаимодействий с периферией по протоколу I2C.

Некоторые библиотеки предназначены для конкретных моделей плат Arduino:

- Keyboard — позволяет платам Arduino имитировать USB-клавиатуру (Leonardo, Due и Micro);
- Mouse — позволяет платам Arduino имитировать USB-мышь (Leonardo, Due и Micro);
- Audio — утилиты для проигрывания звука (только Due);
- Scheduler — для управления выполнением нескольких потоков (только Due);
- USBHost — для подключения USB-периферии (только Due).

Наконец, существует огромное число библиотек, написанных другими пользователями Arduino, которые можно загрузить из Интернета. Далее перечислены некоторые из них, пользующиеся особой популярностью:

- OneWire — для чтения данных из цифровых устройств с интерфейсом 1-wire, выпускаемых компанией Dallas Semiconductor;
- Xbee — для реализации беспроводных взаимодействий;

- GFX — графическая библиотека для работы с разными дисплеями, выпускаемыми компанией Adafruit;
- Capacitive Sensing — для работы с емкостными датчиками;
- FFT — библиотека частотного анализа.

Новые библиотеки появляются постоянно, и их можно найти на официальном сайте Arduino (**http://arduino.cc/en/Reference/Libraries**) или с помощью поисковых систем.

Если вам понадобится использовать одну из сторонних библиотек, ее нужно установить, загрузив и сохранив в папку **Libraries**, находящуюся в папке **Arduino** (в папке **Documents** (Документы)). Обратите внимание на то, что в случае отсутствия папки **Libraries** ее сначала нужно создать и только потом добавлять в нее библиотеки.

Чтобы среда разработки Arduino IDE обнаружила вновь установленную библиотеку, ее нужно перезапустить.

Типы данных в Arduino

Для переменных типа `int` в Arduino C отводится 2 байта памяти. Если только скетч не предъявляет особых требований к экономии памяти, значения `int` используются практически для любых видов информации, даже для логических значений и маленьких целых чисел, которые можно было бы хранить в однобайтовом значении.

Полный список доступных типов данных приводится в табл. 1.1.

Таблица 1.1. Типы данных в Arduino C

Тип	Занимаемая память, байт	Диапазон значений	Примечания
boolean	1	true или false (1 или 0)	Используется для представления логических значений
char	1	–128...+128	Используется для представления кодов символов ASCII, например, A имеет код 65. Отрицательные значения обычно не используются
byte	1	0...255	Часто используется как элементарная единица данных при обмене через последовательные интерфейсы. Подробнее об этом рассказывается в главе 9
int	2	–32 768...+32 767	Целые 16-битные значения со знаком
unsigned int	2	0...65 535	Используется для увеличения точности в расчетах, где не используются отрицательные числа. Применяйте с осторожностью, так как при использовании в выражениях совместно со значениями типа int могут получаться неожиданные результаты
long	4	–2 147 483 648...+ 2 147 483 647	Требуется только для представления очень больших чисел

unsigned long	4	0...4 294 967 295	См. описание типа unsigned int
float	4	–3,4028235E+38...+3,4028235E+38	Используется для представления вещественных чисел
double	4	Как для типа float	Этот тип должен был бы занимать 8 байт и иметь более широкий диапазон и более высокую точность по сравнению с типом float. Но в Arduino тип double является полным аналогом типа float

Команды Arduino

В библиотеке Arduino доступно большое число команд. В табл. 1.2 перечислены наиболее часто используемые из них вместе с примерами.

Таблица 1.2. Функции из библиотеки Arduino

Команда	Пример	Описание
Цифровой ввод/вывод		
pinMode	pinMode(8, OUTPUT);	Переводит контакт 8 в режим работы цифрового выхода. Поддерживаются также режимы INPUT и INPUT_PULLUP
digitalWrite	digitalWrite(8, HIGH);	Устанавливает высокий уровень напряжения на контакте 8. Чтобы установить низкий уровень напряжения, используйте константу LOW вместо HIGH
digitalRead	int i; i = digitalRead(8);	Присваивает переменной i значение HIGH или LOW в зависимости от уровня напряжения на указанном контакте (в данном случае — на контакте 8)
pulseIn	i = pulseIn(8, HIGH);	Возвращает продолжительность в микросекундах следующего импульса с напряжением HIGH на контакте 8
tone	tone(8, 440, 1000);	Генерирует на контакте 8 серию импульсов с частотой 440 Гц продолжительностью 1000 мс
noTone	noTone();	Прерывает любые серии импульсов, запущенные вызовом tone
Аналоговый ввод/вывод		
analogRead	int r; r = analogRead(0);	Присваивает переменной r значение в диапазоне от 0 до 1023. Значение 0 соответствует напряжению 0 В на контакте 0, а значение 1023 — напряжению 5 В (или 3,3 В, если для питания платы используется напряжение 3,3 В)
analogWrite	analogWrite(9, 127);	Выводит широтно-импульсный сигнал. Протяженность положительного импульса может изменяться в диапазоне от 0 до 255, где число 255 соответствует 100%. Этой функции можно передавать номера контактов, обозначенных на плате как PWM (контакты 3, 5, 6, 9, 10 и 11)
Команды для работы со временем		
millis	unsigned long l; l = millis();	Переменные типа long в Arduino хранят 32-битные значения. Значение, возвращаемое функцией millis(), — это число миллисекунд, прошедших с момента последнего сброса платы. Примерно через 50 дней значение обнуляется и счет начинается заново
micros	long l; l = micros();	Действует подобно millis, но возвращает число микросекунд, прошедших с момента последнего сброса платы. Значение обнуляется примерно через 70 минут
delay	delay(1000);	Приостанавливает работу скетча на 1000 мс, или на 1 с
		Приостанавливает работу скетча на 10 000 мкс. Обратите внимание:

delayMicroseconds	delayMicroseconds(10000)	минимальная задержка составляет 3 мкс, максимальная — около 16 мс
Прерывания (глава 3)		
attachInterrupt	attachInterrupt(1, myFunction, RISING);	Устанавливает функцию myFunction, как обработчик положительного фронта прерывания 1 (контакт D3 в UNO)
detachInterrupt	detachInterrupt(1);	Запрещает обработку сигналов от прерывания 1

Полный перечень всех команд Arduino вы найдете в официальной документации на сайте Arduino: **http://arduino.cc**[6].

В заключение

Из-за ограниченного объема книги в этой главе было дано очень краткое введение в мир Arduino. Более подробную информацию об основах вы найдете на многочисленных онлайн-ресурсах, включая бесплатные руководства по Arduino на сайте **http://www.learn.adafruit.com**.

В следующей главе мы заглянем под капот Arduino и посмотрим, как действует эта плата и что происходит у нее внутри, используя для этого удобную среду программирования Arduino.

2 *Монк С.* Программируем Arduino. Основы работы со скетчами. — СПб.: Питер, 2015. — *Примеч. пер.*

3 Существует замечательный сайт об Arduino на русском языке: **http://arduino.ru/**. — *Примеч. пер.*

4 В новых версиях Arduino IDE этот пункт называется **Arduino/Genuino Uno**. — *Примеч. пер.*

5 В русифицированной версии формат сообщений не всегда совпадает с форматом англоязычных сообщений. — *Примеч. пер.*

6 Аналогичный справочник с описанием команд на русском языке можно найти по адресу **http://arduino.ru/Reference**. — *Примеч. пер.*

2. Под капотом

Самое замечательное в Arduino — в большинстве случаев нет необходимости знать, что происходит за кулисами после того, как вы выгрузите скетч. Но так как вы собираетесь вникнуть в особенности работы Arduino и узнать больше о ее возможностях, вы должны знать чуть больше о происходящем в ее глубинах.

Краткая история Arduino

Первая плата Arduino была создана в 2005 году в итальянском Институте проектирования взаимодействий (Interaction Design Institute) в городе Ивреа, близ Турина. Целью было создание недорогого и простого в использовании инструмента для обучения студентов искусству проектирования интерактивных систем. Программное обеспечение для Arduino, которое обеспечило этой плате значительную долю успеха, является доработкой открытого фреймворка с названием Wiring, созданного студентом этого же института.

Доработанная версия для Arduino получилась очень близкой к оригиналу Wiring, а среда разработки Arduino IDE написана с использованием фреймворка Processing, старшего брата Wiring, способного работать на PC, Mac и других персональных компьютерах. Вам стоит обратить внимание на Processing, если вы работаете над проектом, в котором плата Arduino должна обмениваться информацией с PC через USB или Bluetooth.

Аппаратура Arduino все эти годы продолжала развиваться, но современные платы Arduino Uno и Leonardo сохранили форму и набор контактов, доставшиеся от оригинала.

Устройство Arduino

На рис. 2.1 изображена структурная схема платы Arduino Uno. Модель Leonardo имеет схожее устройство, но интерфейс USB в ней интегрирован в чип микроконтроллера. Похожее устройство имеет и модель Due, но в ней используется процессор с напряжением питания 3,3 В вместо 5 В.

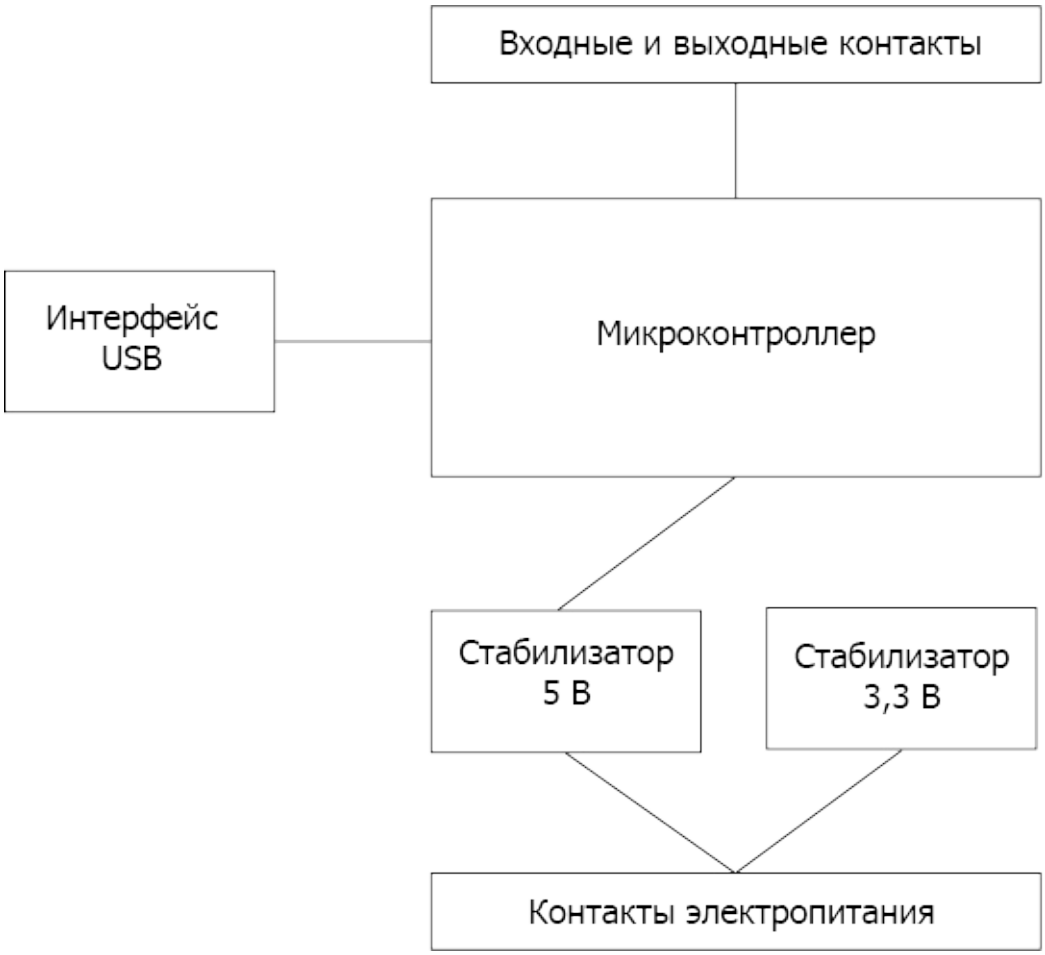


Рис. 2.1. Устройство платы Arduino Uno

Во многих отношениях Arduino — не более чем микроконтроллер с небольшим количеством поддерживающих компонентов. В действительности вполне можно создать Arduino на макетной плате, добавив микропроцессор и несколько дополнительных компонентов, или создать свою печатную плату, взяв за основу плату Arduino. Готовые платы Arduino избавляют от такой необходимости, но вообще можно повторить любую плату Arduino, используя микроконтроллер и несколько действительно необходимых компонентов. Например, если ваша разработка предназначена только для одной цели, есть смысл отказаться от интерфейса USB, так как программы можно выгружать в микросхемы памяти на плате Arduino, а затем переносить их на свои платы.

Далее в книге будет показано, как можно программировать платы Arduino непосредственно через последовательный интерфейс внутрисхемного программирования ICSP (In Circuit Serial Programming).

Процессоры AVR

Во всех платах семейства Arduino используются микроконтроллеры, производимые компанией Atmel. Все они имеют схожую аппаратную архитектуру и, за исключением микроконтроллера, используемого в модели Due (SAM3X8E ARM Cortex-M3 CPU), схожую конструкцию.

ATmega328

В Arduino Uno и ее предшественнице Duemilanove используется микроконтроллер ATmega328. Фактически первые версии Arduino комплектовались микроконтроллером ATmega168, который, по сути, является полным аналогом ATmega328, но имеет в два раза меньше памяти каждого вида.

На рис. 2.2 изображена функциональная схема ATmega328 из технического описания микроконтроллера. Полное описание можно найти по адресу www.atmel.com/Images/doc8161.pdf. Вам стоит просмотреть ее, чтобы лучше понимать, как действует это устройство.

Центральный процессор — то место, где выполняются все операции. Процессор читает инструкции (скомпилированный код скетча) из флеш-памяти по одной за раз. Этот процесс отличается от обычного для компьютеров, где программы хранятся на диске и перед выполнением загружаются в оперативное запоминающее устройство (ОЗУ). Переменные, используемые в программе, хранятся отдельно, в статическом ОЗУ. В отличие от флеш-памяти, где хранится код программы, ОЗУ является энергозависимым и теряет свое содержимое при выключении питания.

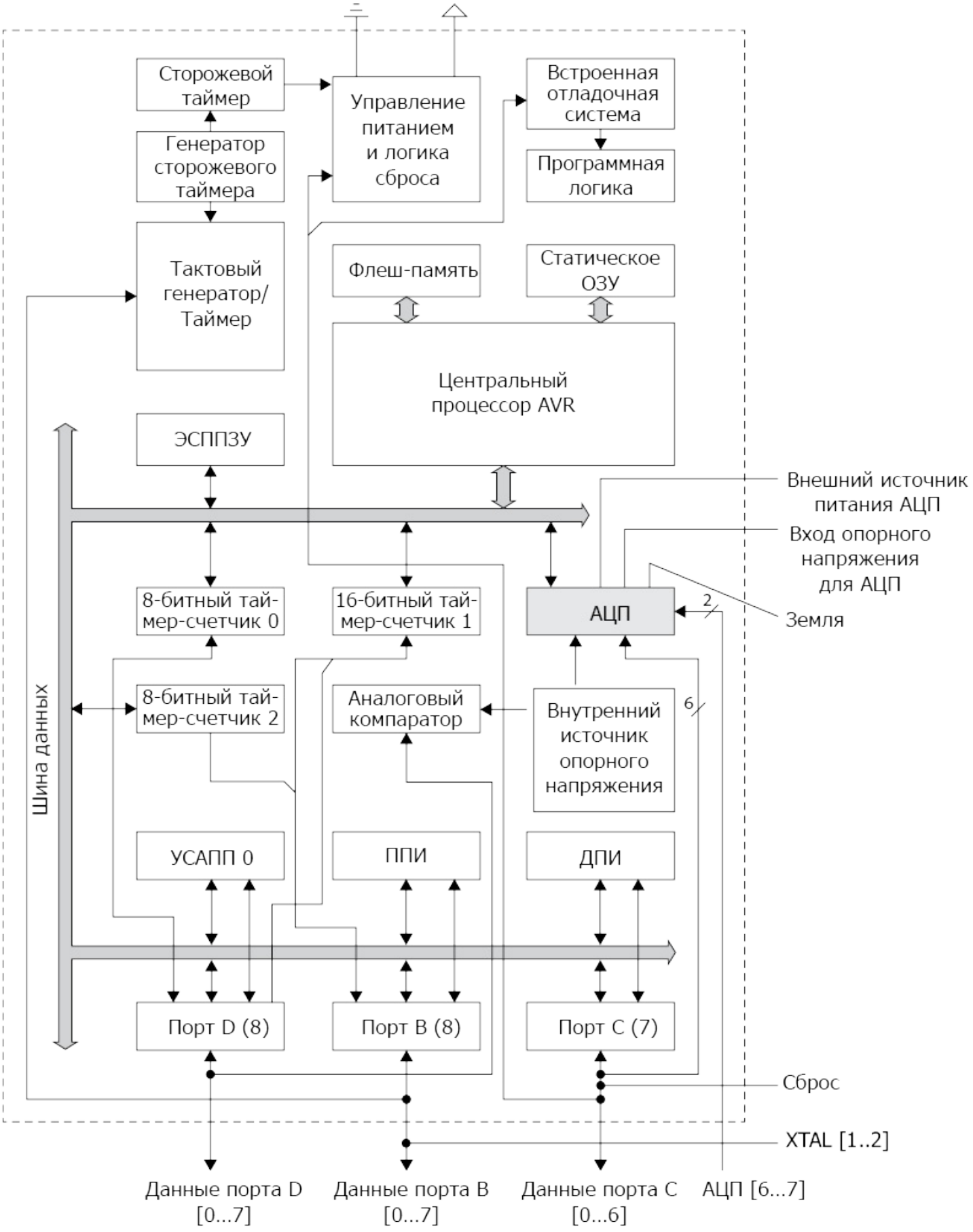


Рис. 2.2. ATmega328

Для данных, которые должны сохраняться даже после выключения питания, следует использовать память третьего типа — *электрически стираемую программируемую постоянную память (Electrically Erasable Programmable Read-Only Memory, EEPROM)*.

Еще одна интересная область — сторожевой таймер и устройство управления питанием. Они открывают для микроконтроллера множество дополнительных возможностей, обычно скрытых за уровнем упрощения Arduino, в том числе перевод микропроцессора в экономичный режим с последующей установкой таймера для периодического перевода в нормальный режим. Этот трюк может пригодиться для создания приложений с низким энергопотреблением. Подробнее об этом рассказывается в главе 5.

Все остальное на рис. 2.2 так или иначе связано с аналого-цифровым преобразователем, портами ввода/вывода и последовательными интерфейсами трех типов, поддерживаемыми микроконтроллером: УСАПП, ППИ и ДПИ.

ATmega32u4

Микроконтроллер ATmega32u4B используется в моделях Arduino Leonardo и LilyPad USB, а также Micro и Nano. Он похож на ATmega328, но имеет более современный чип с несколькими дополнениями, отсутствующими в ATmega328:

- со встроенным интерфейсом USB, благодаря чему отпала необходимость в дополнительных аппаратных компонентах поддержки USB;
- с большим числом контактов с возможностью ШИМ;
- с двумя последовательными портами;
- с отдельными контактами интерфейса I2C (в Arduino эти контакты также играют роль аналоговых входов/выходов);
- с объемом статического ОЗУ больше на 0,5 Кбайт.

В модели Leonardo используется микроконтроллер в корпусе для поверхностного монтажа, то есть он припаивается непосредственно к плате, тогда как ATmega328 выпускается в корпусе с двумя рядами контактов, вставляемом в панель для интегральных микросхем, как на плате Arduino Uno.

ATmega2560

Микроконтроллер ATmega2560 используется в моделях Arduino Mega 2560 и Arduino Mega ADK. Он не быстрее других микроконтроллеров ATmega, но имеет больше памяти каждого типа (256 Кбайт флеш-памяти, 8 Кбайт статического ОЗУ и 4 Кбайт ЭСППЗУ), а также намного больше контактов ввода/вывода.

AT91SAM3X8E

Этот микроконтроллер является сердцем Arduino Due. Он существенно быстрее других микроконтроллеров ATmega, упоминавшихся ранее, и работает на тактовой частоте 84 МГц против обычных для ATmega 16 МГц. Имеет 512 Кбайт флеш-памяти и 96 Кбайт статического ОЗУ. Данный микроконтроллер не имеет ЭСППЗУ. Поэтому для долговременно хранения данных требуется использовать дополнительные устройства, такие как держатели карт памяти или устройства с флеш-памятью или ЭСППЗУ. Сам микроконтроллер обладает множеством дополнительных особенностей, включая два аналоговых выхода, делающих его идеальным инструментом для генерации звуков.

Arduino и Wiring

Фреймворк Wiring включает простые в использовании функции управления контактами на плате Arduino, однако основная его часть написана на языке C.

До недавнего времени в каталоге установки Arduino IDE можно было найти файл **WProgram.h** (программа Wiring). Теперь его замещает похожий файл с именем **Arduino.h**, что свидетельствует о постепенном отдалении Arduino от первоначального проекта Wiring.

Заглянув в каталог установки Arduino IDE, можно увидеть папку **hardware**, внутри нее — папку **arduino**, а внутри этой папки — папку **cores**. Обратите внимание на то, что в Mac в эту папку можно попасть, только если щелкнуть правой кнопкой на ярлыке приложения Arduino, выбрать в контекстном меню пункт **View Package Contents** (Показать содержимое пакета) и затем перейти в папку **Resources/Java/**.

Внутри папки **cores** находится еще одна папка с именем **arduino**, в которой вы найдете множество заголовочных файлов на языке C с расширением **.h** и файлов реализации на языке C++ с расширением **.cpp** (рис. 2.3).

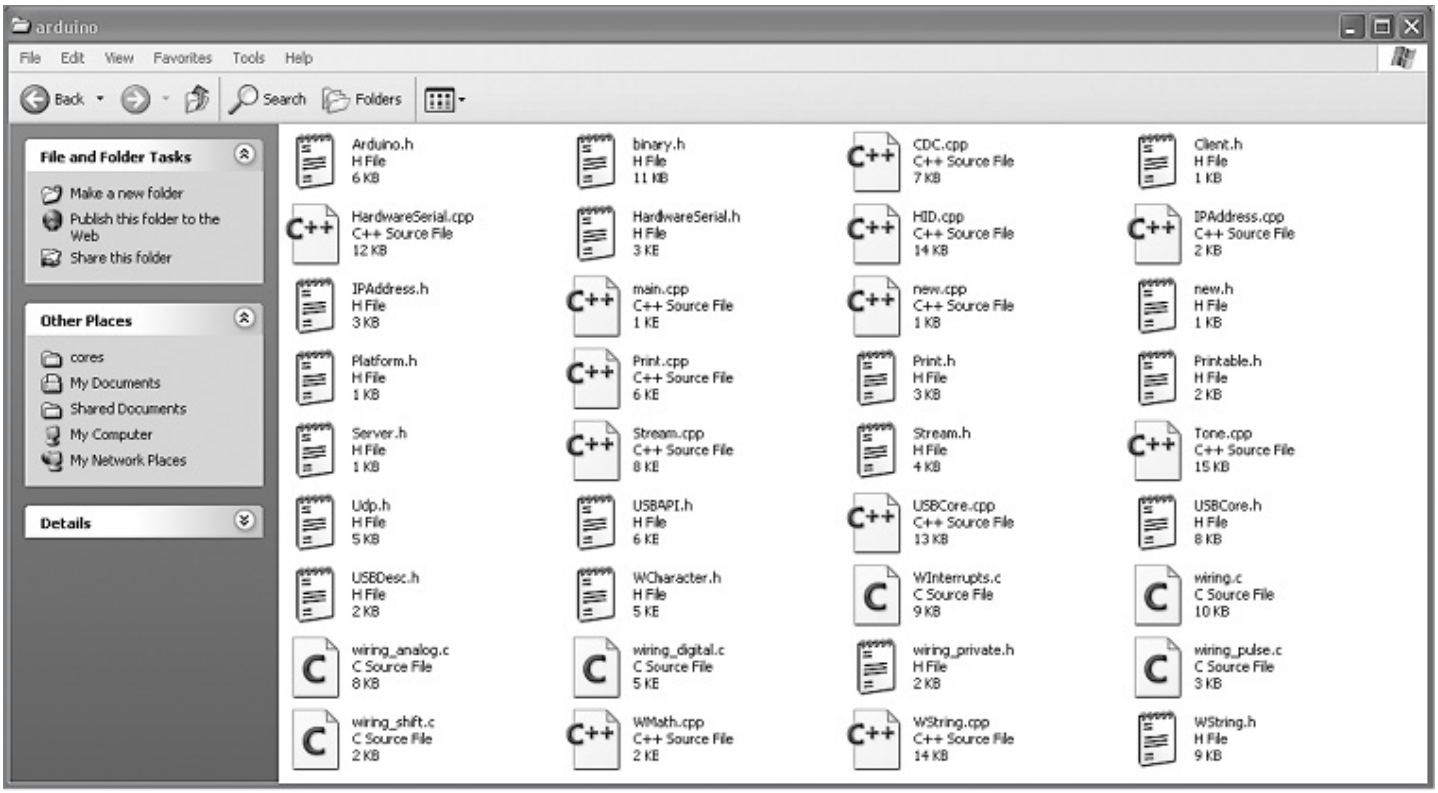


Рис. 2.3. Внутри папки cores

Открыв **Arduino.h** в текстовом редакторе, вы увидите, что он состоит из множества инструкций `#include`. Они подключают определения из других заголовочных файлов в папке **cores/arduino** в процессе компиляции (преобразования скетча в форму, пригодную для записи во флеш-память микроконтроллера).

Там же можно увидеть определения констант, например:

```
#define HIGH 0x1
#define LOW  0x0

#define INPUT 0x0
#define OUTPUT 0x1
#define INPUT_PULLUP 0x2
```

Они немного похожи на переменные в том смысле, что, обратившись к имени `HIGH`, например, программа получит значение 1. Значение определено как `0x1`, а не как `1`, потому что в этом файле все значения определяются в *шестнадцатеричном* формате (в системе счисления с основанием 16). Эти определения в действительности не являются переменными — их называют *директивами препроцессора C*, то есть когда ваш скетч будет преобразован в формат, пригодный для записи во флеш-память микроконтроллера, все слова `HIGH`, `LOW` и другие автоматически будут преобразованы в соответствующие числа. Это дает определенные преимущества перед использованием переменных, так как не требуется выделять память для их хранения.

Так как эти константы являются числовыми, вы можете, например, перевести

контакт **5** в режим OUTPUT, как показано далее, но все же лучше пользоваться символическими именами на тот случай, если разработчики Arduino решат изменить значения констант. Кроме того, использование имен упрощает чтение программного кода.

```
setMode(5, 1);
setMode(5, OUTPUT);
```

Также в файле **arduino.h** присутствует множество сигнатур функций, например таких:

```
void pinMode(uint8_t, uint8_t);
void digitalWrite(uint8_t, uint8_t);
int digitalRead(uint8_t);
int analogRead(uint8_t);
void analogReference(uint8_t mode);
void analogWrite(uint8_t, int);
```

Они предупреждают компилятор о функциях, которые фактически реализуются где-то в другом месте. Возьмем, для примера, первую сигнатуру. Она сообщает, что функция `pinMode` принимает два аргумента (которые, как вы уже знаете, представляют номер контакта и режим) типа `uint8_t`. Команда `void` говорит, что после вызова функция ничего не возвращает.

Вам может показаться странным, почему для параметров выбран тип `uint8_t`, а не `int`. Обычно, определяя номер контакта, вы указываете значение типа `int`. На самом деле `int` — это универсальный тип, широко используемый в скетчах. Он избавляет пользователей от проблемы выбора из большого разнообразия доступных типов. Но в диалекте языка C для Arduino тип `int` представляет 16-битные целые значения со знаком в диапазоне между –32 768 и 32 767. Однако номер контакта не может быть отрицательным, и вам едва ли когда-нибудь попадется плата Arduino с 32 767 контактами.

Тип `uint_8` намного точнее определяет диапазон допустимых значений, потому что вообще в языке C тип `int` может представлять значения с разрядностью от 16 до 64 битов в зависимости от конкретной реализации C. Имя типа `uint_8` читается так: символ `u` говорит, что это беззнаковый (unsigned) тип, `int` сообщает, что это целочисленный тип, и, наконец, число после символа подчеркивания (_) сообщает количество битов. То есть тип `uint_8` представляет 8-битные целые числа без знака в диапазоне между 0 и 255.

Вы свободно можете использовать в своих скетчах эти более строгие типы, что некоторые и делают. Но помните, что это сделает ваш код чуть труднее для понимания теми, кто не искушен в программировании для Arduino.

Возможность использования обычного типа `int`, представляющего 16-битные целые числа со знаком, вместо типа `unit_8`, например, объясняется способностью компилятора автоматически выполнять необходимые преобразования. Использование переменных типа `int` для хранения номеров контактов приводит к напрасному расходованию памяти. Поэтому вам придется искать компромисс между объемом памяти для хранения данных и удобочитаемостью кода. Как правило, в программировании предпочтение отдается простоте чтения кода, если только вы не собираетесь создать нечто очень сложное, способное превысить ограничения микроконтроллера.

Здесь можно провести аналогию с грузовиком, который вы собираетесь использовать для доставки чего-то кому-то. Если требуется перевезти большой объем груза, вам придется подумать, как упаковать и расположить его, чтобы он уместился весь. Если груз занимает лишь малую часть площади кузова, нет смысла тратить много времени на его упаковку и размещение.

Также в папке **arduino** можно найти файл **main.cpp**. Открыв его, вы увидите кое-что интересное.

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

#if defined(USBCON)
    USBDevice.attach();
#endif
    setup();

    for (;;) {
        loop();
        if (serialEventRun) serialEventRun();
    }

    return 0;
}
```

Если прежде вам доводилось программировать на языке C, C++ или Java, вы должны быть знакомы с идеей функции `main`. Эта функция автоматически вызывается в момент запуска программы. Функция `main` — это главная точка входа в программу. Это утверждение справедливо и для программ Arduino, только скрыто от глаз разработчиков скетчей, которые обязаны реализовать в своих скетчах две функции — `setup` и `loop`.

Если вчитаться в файл **main.cpp**, пропустив пока первые несколько строк, можно заметить, что функция `main` вызывает `setup()` и затем входит в бесконечный цикл `for`, где вызывает функцию `loop`.

Команда `for(;;)` — это, пусть и малопонятный, способ записи `while (true)`. Обратите внимание на то, что кроме вызова функции `loop` внутри цикла `for` имеется также команда `if`, которая проверяет поступление сообщений в последовательный порт и обслуживает их.

Вернувшись в начало файла **main.cpp**, можно увидеть, что в первой строке находится команда `include`, подключающая все определения из заголовочного файла **arduino.h**, о котором я говорил прежде.

Далее находится определение функции `main`, которая начинается с вызова функции `init()`. Если поискать, ее можно найти в файле **wiring.c**, она вызывает функцию `sei`, разрешающую прерывания.

Строки

```
#if defined(USBCON)
    USBDevice.attach();
#endif
```

являются еще одной директивой препроцессора C. Данный код действует подобно команде `if`, которую вы можете использовать в своих скетчах, но выполняется она не тогда, когда скетч уже работает в Arduino. Проверка условия в директиве `#if` происходит во время компиляции скетча. Данная директива дает отличную возможность включать и выключать фрагменты кода в зависимости от конкретного типа платы. В данном случае, если Arduino поддерживает интерфейс USB, в программу включается код, подключающий (инициализирующий) его, в противном случае нет никакого смысла компилировать его.

Из скетча в Arduino

Теперь, когда вы узнали, откуда берется весь этот магический код, когда пишется даже самый простой скетч для Arduino, можно посмотреть, как этот код попадает во флеш-память микроконтроллера на плате Arduino, когда вы щелкаете на кнопке **Upload** (Загрузить) в Arduino IDE.

На рис. 2.4 показано, что происходит после щелчка на кнопке **Upload** (Загрузить).

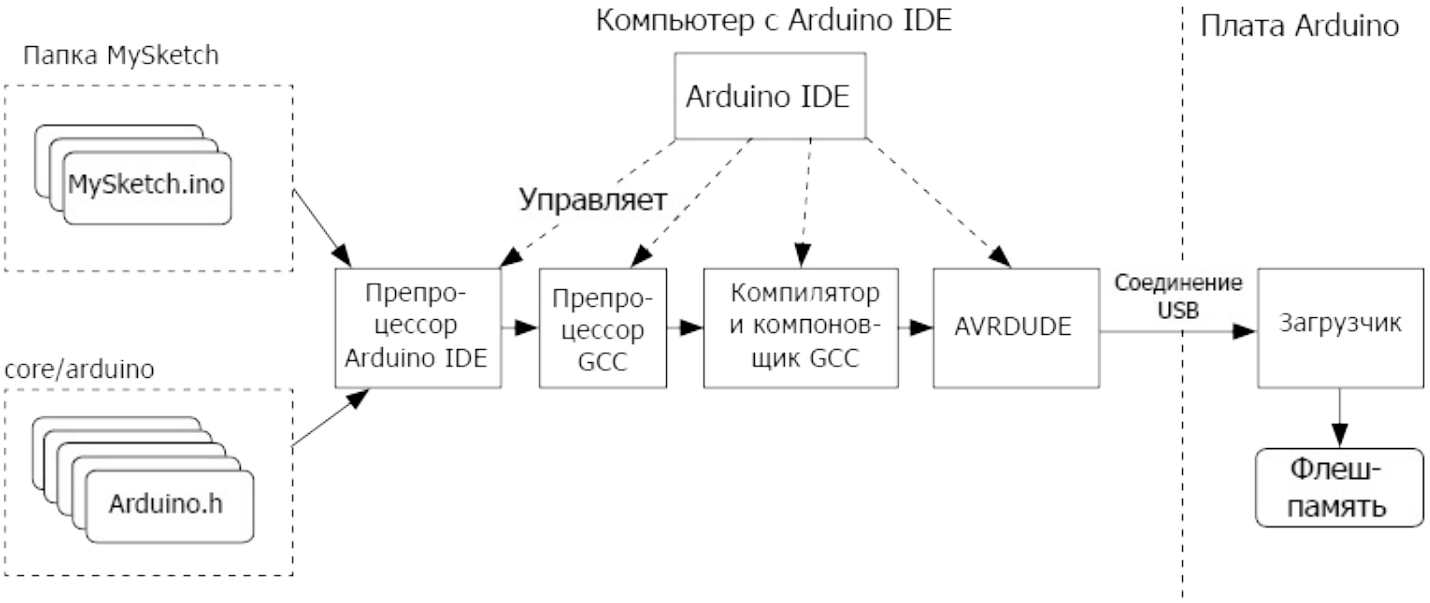


Рис. 2.4. Комплект инструментов Arduino

Скетчи для Arduino хранятся в виде текстовых файлов с расширением **.ino** в папке с тем же именем, но без расширения.

Когда пользователь пытается выгрузить скетч в плату, в дело включается среда разработки Arduino IDE, которая управляет множеством вспомогательных программ, выполняющих основную работу. Сначала компонент Arduino IDE, который я (за неимением лучшего названия) называю препроцессором (*Arduino IDE preprocessor*), собирает файлы, составляющие скетч. Обратите внимание на то, что обычно скетч состоит из единственного файла. При желании в папку скетча можно добавить другие файлы, правда, при этом для их создания придется использовать другой редактор.

Если в папке присутствуют другие файлы, они также будут включены в процесс сборки. Файлы с программным кодом на языках C и C++ компилируются отдельно друг от друга. В начало главного файла скетча добавляется строка, подключающая **arduino.h**.

Так как существует множество разных моделей плат Arduino, использующих разные микроконтроллеры с разными наименованиями контактов, Arduino IDE должна выбрать правильные их определения. Если заглянуть в папку **hard-ware/arduino/variants**, можно увидеть отдельные папки для всех моделей плат Arduino, в каждой из которых хранится свой файл **pins_arduino.h**. Этот файл содержит имена контактов для своей платформы.

После объединения файлов вызывается компилятор GCC. Это компилятор C++, распространяемый с открытым исходным кодом и входящий в состав дистрибутива Arduino. Он принимает скетч, заголовочный файл и файлы реализации с исходным кодом на C и преобразует их в код, который может выполняться микроконтроллером на плате Arduino. Этот компилятор выполняет следующие шаги.

1. Препроцессор интерпретирует все команды `#if` и `#define` и определяет, какой код должен быть скомпилирован.

2. Затем выполняются компиляция и компоновка кода в единственный файл, пригодный для выполнения процессором на плате.

3. Когда компилятор завершит свою работу, запускается открытый программный компонент с названием **avrdude**, который пересылает в плату выполняемый двоичный код в шестнадцатеричном формате через последовательный интерфейс USB.

Теперь мы в царстве Arduino. В плате Arduino имеется небольшая резидентная программа, устанавливаемая в каждый микроконтроллер. Эта программа называется *загрузчиком* (bootloader). Загрузчик выполняется каждый раз, когда происходит сброс платы Arduino. Именно поэтому, когда происходит передача данных по последовательному интерфейсу, аппаратура связи в Arduino Uno производит принудительный сброс платы, чтобы дать загрузчику возможность проверить входящие скетчи.

Если был получен скетч, плата Arduino программирует сама себя, распаковывая шестнадцатеричное представление программы в двоичное и сохраняя его во флеш-памяти. Когда в следующий раз произойдет сброс платы, после обычной проверки на наличие нового скетча загрузчик автоматически запустит программу, хранящуюся во флеш-памяти.

Возникает естественный вопрос: почему компьютер не может запрограммировать микроконтроллер напрямую, минуя такую сложную процедуру? Причина в том, что для программирования микроконтроллера требуется специальная аппаратура, использующая другой способ связи с платой Arduino (кто-нибудь из вас спрашивал себя, зачем на плате колодка с шестью контактами?). Благодаря загрузчику, постоянно прослушивающему последовательный порт, мы можем программировать Arduino через USB без использования специальной аппаратуры.

Однако если у вас есть такой программатор, например AVRISPv2, AVRDragon или USBtinyISP, с его помощью вы сможете программировать Arduino напрямую, в обход загрузчика. Фактически, как будет показано далее в этой главе, в роли такого программатора можно использовать вторую плату Arduino.

AVR Studio

Тертые инженеры-электронщики могут выразить свое презрение к Arduino IDE, заявив, что она не имеет никаких преимуществ перед инструментами компании Atmel для программирования всего семейства микроконтроллеров AVR. С технической точки зрения это верно. Но при этом не учитывается, что Arduino помогает раскрыть все тайны процесса использования микроконтроллеров и выйти из-под контроля таких экспертов. Действительно, многое из того, что делает нас поклонниками Arduino,

можно считать дилетантством, и в ответ на это я могу только сказать: «Ну и что!»

AVR Studio — это лицензионное программное обеспечение, предоставляемое производителем для программирования микроконтроллеров, на основе которых сконструированы платы Arduino. Его можно использовать вместо Arduino IDE. Однако в этом случае вам придется смириться со следующими ограничениями.

- Работа возможна только в окружении Windows.
- Программирование выполняется с использованием специального оборудования вместо USB.
- Среда разработки сложнее в использовании.

Возможно, вам захочется узнать, почему может возникнуть желание использовать эту среду. Вот некоторые веские причины.

- Чтобы избавиться от загрузчика (в модели Uno он занимает 500 байт), например из-за нехватки флеш-памяти или желания обеспечить более быстрый запуск программы после сброса.
- Чтобы использовать микроконтроллеры, отличные от тех, что применяются в стандартных платах Arduino, например маленькие и недорогие микроконтроллеры семейства ATtiny.
- Просто чтобы освоить что-то новое.

Все платы Arduino снабжаются колодкой с шестью контактами, которые можно использовать для непосредственного программирования Arduino с применением AVR Studio. Фактически некоторые модели имеют две такие колодки: одна связана с основным процессором, а другая — с интерфейсом USB, поэтому будьте внимательнее при выборе нужной.

На рис. 2.5 показано, как выглядит пользовательский интерфейс программы AVR Studio 4.

Изучение AVR Studio выходит за рамки этой книги. Однако, как можно видеть на рис. 2.5, скетч Blink не стал длиннее, но определенно выглядит намного сложнее! И в результате его компиляции выполняемый код занимает намного меньше места во флеш-памяти в сравнении с его аналогом в Arduino.

На рис. 2.6 изображена плата Arduino, подключенная к программатору AVR Dragon. Это очень гибкое и мощное устройство, позволяющее в пошаговом режиме отлаживать программы, фактически выполняющиеся на микроконтроллере ATmega.

В главе 4 мы рассмотрим непосредственное взаимодействие с портами, реализацию

которого можно видеть на рис. 2.5. Этот прием позволяет улучшить производительность ввода/вывода и продолжать пользоваться преимуществами Arduino IDE.

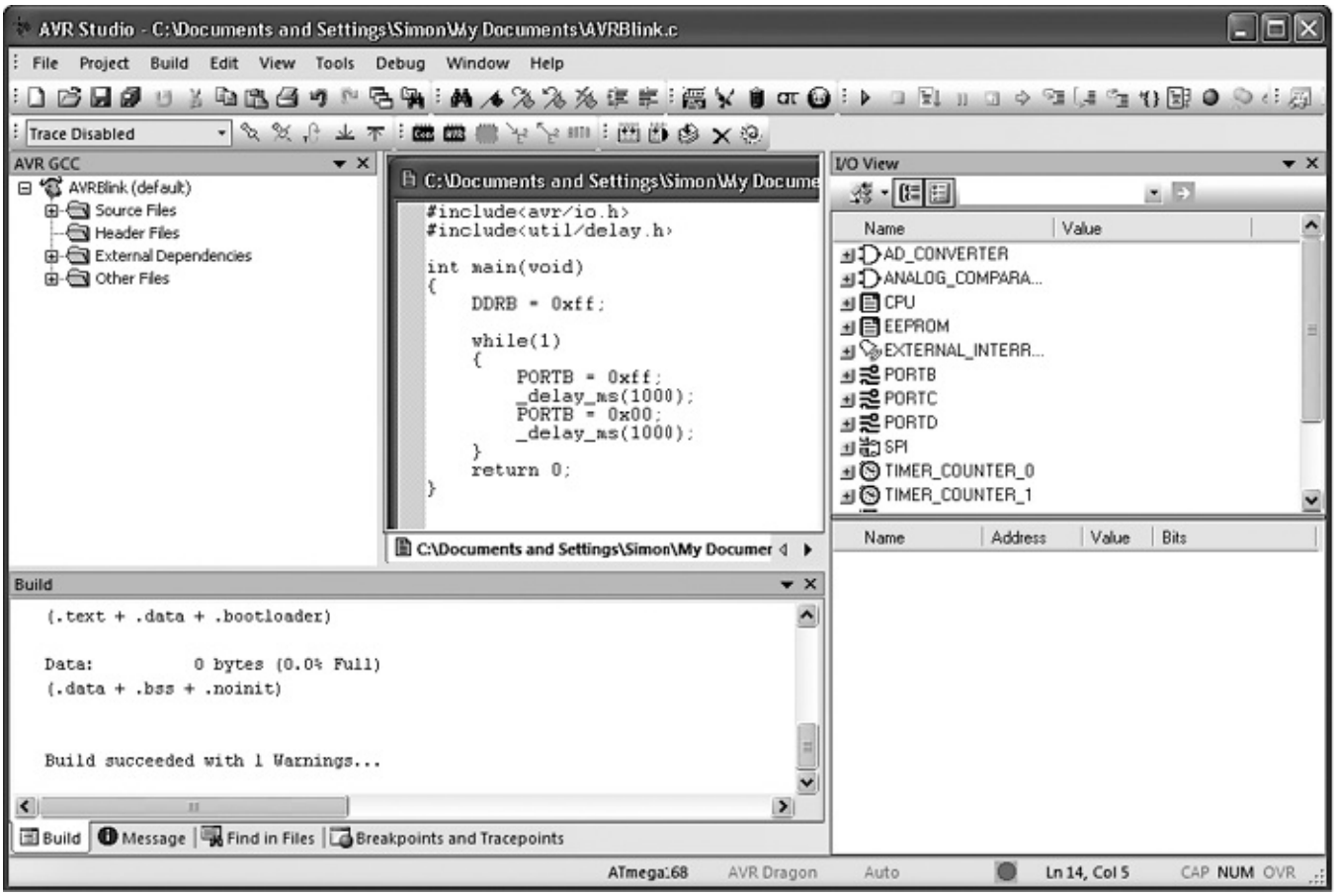


Рис. 2.5. AVR Studio

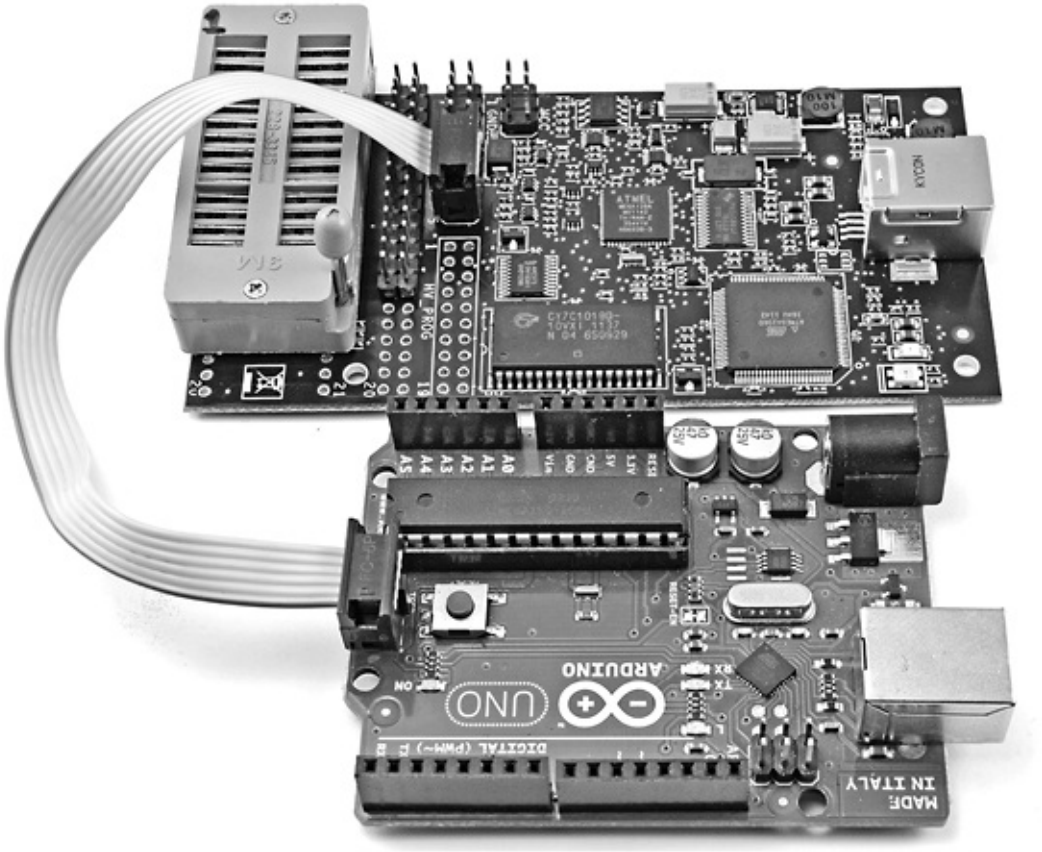


Рис. 2.6. Плата Arduino, подключенная к программатору AVR Dragon

Установка загрузчика

Установка загрузчика в плату Arduino может потребоваться по нескольким причинам. Например, вы могли по неосторожности повредить извлекаемый микроконтроллер ATmega328 на плате Arduino Uno и решили заменить его новым ATmega328, приобретенным без загрузчика. Или, занимаясь собственными разработками, вы решили взять микроконтроллер ATmega328 из платы Arduino и вставить его в плату собственной конструкции.

В любом случае имеется возможность установить загрузчик в «чистый» микроконтроллер ATmega328, используя любой из программаторов, упоминавшихся в предыдущем разделе, или с помощью второй платы Arduino.

Установка загрузчика с помощью AVR Studio и программатора

В папке установки Arduino IDE имеются шестнадцатеричные файлы загрузчиков, которые можно записать в ATmega328 с помощью AVR Studio. Эти файлы находятся в папке **hardware/arduino/bootloaders**. Там вы найдете шестнадцатеричные файлы для всех видов микроконтроллеров. Если вам требуется установить загрузчик в модель Uno, используйте **optiboot_atmega328.hex** в папке **optiboot** (рис. 2.7).

Но имейте в виду следующее: при установке загрузчика есть вероятность того, что

ваш микроконтроллер превратится в «кирпич». Микроконтроллеры имеют то, что называют битами защиты, которые можно неправильно запрограммировать без возможности восстановления. Они встраиваются для защиты коммерческих интересов, чтобы предотвратить возможность перепрограммирования. Прежде чем сделать решающий шаг, проверьте внимательно, что биты защиты установлены правильно для той платы Arduino, которую вы собираетесь запрограммировать, и подготовьтесь к худшему. На форуме Arduino (www.arduino.cc/forum)[7] вы найдете множество обсуждений этой темы наряду с советами, как избежать неприятностей.

Чтобы записать загрузчик с помощью AVR Studio и программатора AVR Dragon, подключите программатор к колодке с контактами **ICSP** (рис. 2.6).

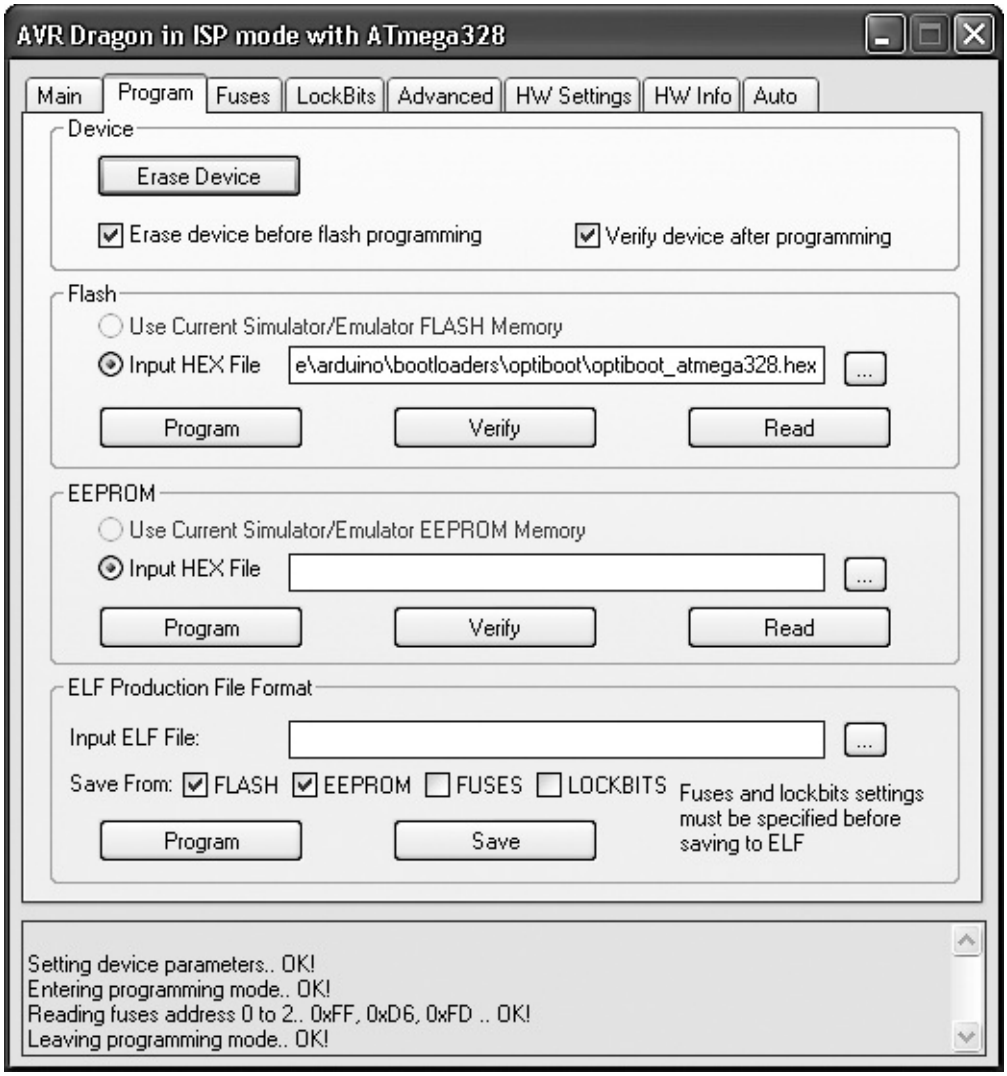


Рис. 2.7. Установка загрузчика в плату Uno в среде AVR Studio 4

Обратите внимание на то, что на плате Arduino Uno имеются две такие колодки (вторая используется для программирования интерфейса USB).

В меню **Tools** (Инструменты) выберите пункт **Program AVR** (Программировать AVR) и установите соединение с микроконтроллером ATmega328 в плате Arduino. Затем в разделе **Flash** (Флеш-память) выберите требующийся шестнадцатеричный файл и щелкните на кнопке **Program** (Программировать).

Установка загрузчика с помощью Arduino IDE и второй платы Arduino

Установка нового загрузчика с другой платы Arduino выполняется очень просто. Этот подход намного проще и безопаснее, чем использование AVR Studio. Среда разработки Arduino IDE включает необходимую функцию. Далее перечислено все, что вам потребуется:

- две платы Arduino Uno;
- 6-проводной шлейф с двумя разъемами «папа–папа» (или шесть изолированных проводов);
- один короткий изолированный провод;
- конденсатор на 10 мкФ и 10 В (также подойдет конденсатор на 100 мкФ).

Сначала нужно соединить платы проводами, как описывается в табл. 2.1.

Таблица 2.1. Соединение контактов двух плат Arduino для записи загрузчика

Плата-программатор	Программируемая плата
GND	GND
5 V	5 V
13	13
12	12
11	11
10	Reset

Конденсатор емкостью 10 мкФ включите между контактами **Reset** и **GND** на программируемой плате Arduino (то есть на плате, куда будет записан загрузчик). Более длинный положительный вывод конденсатора должен быть подключен к контакту **Reset**.

На рис. 2.8 показаны соединенные платы Arduino. Плата справа выступает в роли программатора. Обратите внимание на то, как контакт **10** платы-программатора соединен изолированным проводом с контактом **Reset** программируемой платы. Соединение выполнено так, что к одному контакту **Reset** на программируемой плате одновременно подключены провод и положительный вывод конденсатора.

Имейте в виду, что плата, выступающая в роли программатора, управляет программируемой платой, поэтому к порту USB компьютера должна быть подключена только плата-программатор.

На плату-программатор требуется установить специальный скетч. Его можно найти в меню **File—>Examples** (Файл—>Примеры). Он называется ArduinoISP и находится в

конце раздела **Examples** (Встроенные примеры).

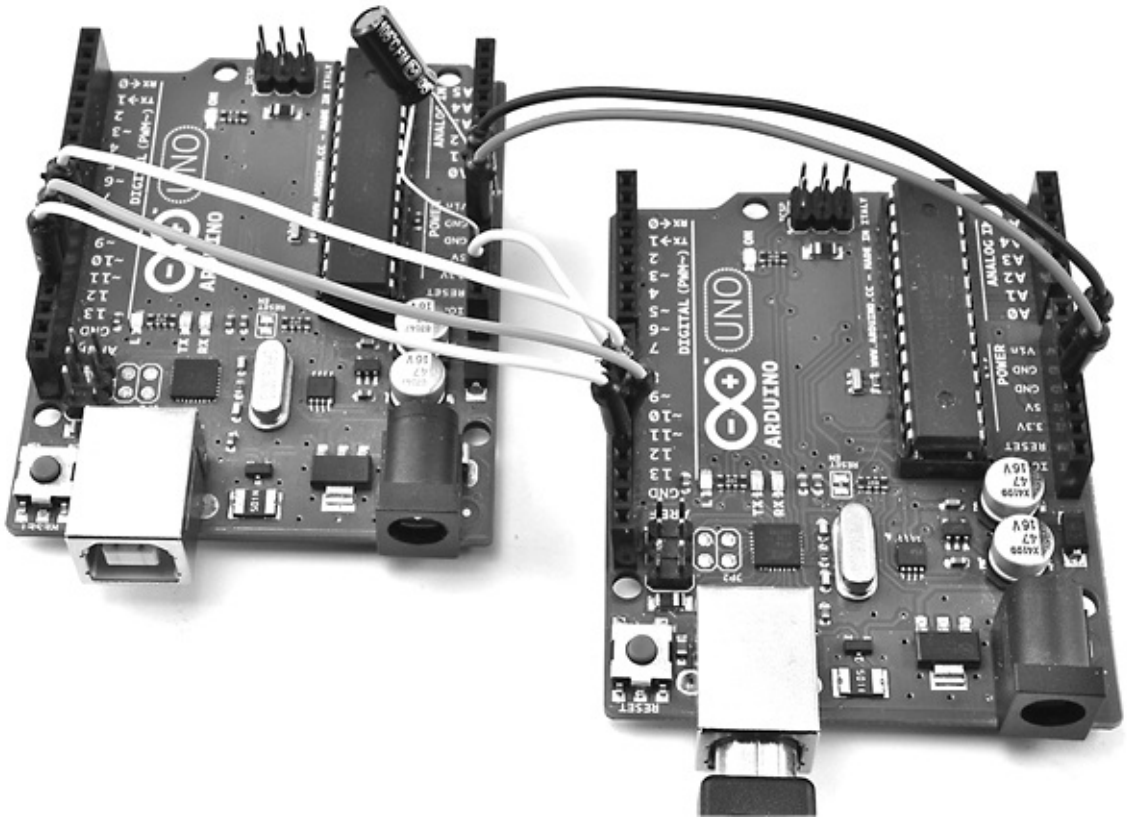


Рис. 2.8. Программирование с помощью второй платы Arduino

Как обычно, выберите тип платы и порт и выгрузите скетч ArduinoISP в плату-программатор. Затем в меню **Tools** (Инструменты) перейдите в подменю **Programmer** (Программатор) и выберите пункт **ArduinoISP**.

Наконец, выберите в меню **Tools** (Инструменты) пункт **Burn Bootloader** (Записать Загрузчик). Процедура записи займет одну-две минуты, в течение которых должны мигать светодиоды **Rx** и **Tx** на плате-программаторе и светодиод **L** на программируемой плате.

Когда процедура завершится, в программируемый микроконтроллер будет установлен новый загрузчик.

В заключение

В этой главе мы заглянули под капот платы Arduino и посмотрели, как она работает в действительности. Я показал вам, что скрывается за фасадом окружения Arduino.

В следующей главе мы посмотрим, как пользоваться прерываниями и как заставить Arduino откликаться на внешние события, вызываемые таймером, применяя для этого прерывания.

7 Аналогичные обсуждения можно найти на русскоязычном форуме **http://arduino.ru/forum**. — *Примеч. пер.*

3. Прерывания и таймеры

Прерывания позволяют микроконтроллерам откликаться на события без необходимости постоянно проверять выполнение каких-либо условий, чтобы определить момент, когда произошли важные изменения. В дополнение к возможности подключать источники прерываний к некоторым контактам можно также использовать прерывания, генерируемые таймером.

Аппаратные прерывания

Для демонстрации использования прерываний вернемся вновь к цифровым входам. Часто для определения момента некоторого входного события (например, нажатия кнопки) используется такой код:

```
void loop()
{
  if (digitalRead(inputPin) == LOW)
  {
    // Выполнить какие-то действия
  }
}
```

Этот код постоянно проверяет уровень напряжения на контакте **inputPin**, и, когда `digitalRead` возвращает `LOW`, выполняются какие-то действия, обозначенные комментарием `// Выполнить какие-то действия`. Это вполне рабочее решение, но что если внутри функции `loop` требуется выполнить массу других операций? На все эти операции требуется время, поэтому есть вероятность пропустить короткое нажатие на кнопку, пока процессор будет занят чем-то другим. На самом деле пропустить факт нажатия на кнопку почти невозможно, потому что по меркам микроконтроллера она остается нажатой очень долго.

Но как быть с короткими импульсами от датчика, которые могут длиться миллионные доли секунды? Для приема таких событий следует использовать прерывания, определяя функции, которые будут вызываться по этим событиям, независимо от того, чем занят микроконтроллер. Такие прерывания называют *аппаратными прерываниями* (hardware interrupts).

В Arduino Uno только два контакта связаны с аппаратными прерываниями, из-за чего они используются очень экономно. В Leonardo таких контактов пять, на больших платах, таких как Mega2560, их намного больше, а в Due все контакты поддерживают возможность прерывания.

Далее рассказывается, как работают аппаратные прерывания. Чтобы опробовать

представленный пример, вам понадобятся дополнительная макетная плата, кнопка, сопротивление на 1 кОм и несколько соединительных проводов.

На рис. 3.1 изображена собранная схема. Через сопротивление на контакт **D2** подается напряжение HIGH, пока кнопка не будет нажата, в этот момент произойдет заземление контакта **D2** и уровень напряжения на нем упадет до LOW.

Загрузите в плату Arduino следующий скетч:

```
// sketch 03_01_interrupts

int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
  attachInterrupt(0, stuffHapenned, FALLING);
}

void loop()
{
}

void stuffHapenned()
{
  digitalWrite(ledPin, HIGH);
}
```

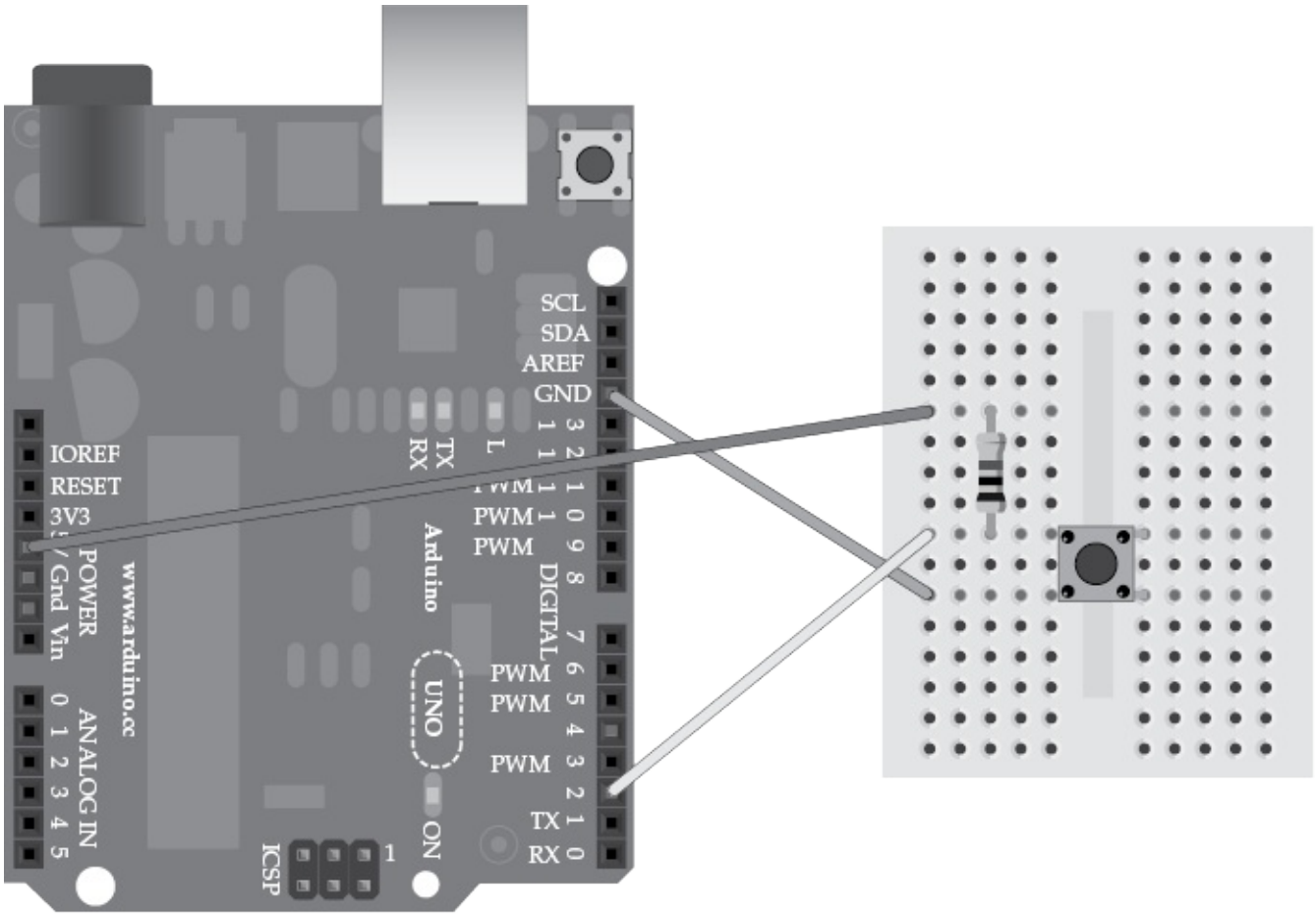


Рис. 3.1. Электрическая схема для испытания прерываний

Помимо настройки контакта **LED** на работу в режиме цифрового выхода функция `setup` с помощью еще одной строки связывает функцию с прерыванием. Теперь в ответ на каждое прерывание автоматически будет вызываться эта функция. Рассмотрим эту строку внимательнее, потому что аргументы вызываемой здесь функции выглядят несколько необычно:

```
attachInterrupt(0, stuffHapenned, FALLING);
```

Первый аргумент — `0` — это номер прерывания. Было бы понятнее, если бы номер прерывания совпадал с номером контакта, но это не так. В Arduino Uno прерывание 0 связано с контактом **D2**, а прерывание 1 — с контактом **D3**. Ситуация становится еще более запутанной из-за того, что в других моделях Arduino эти прерывания связаны с другими контактами, а кроме того, в Arduino Due нужно указывать номер контакта. На плате Arduino Due с прерываниями связаны все контакты.

Я еще вернусь к этой проблеме, а пока перейдем ко второму аргументу. Этот аргумент — `stuffHappened` — представляет имя функции, которая должна вызываться для обработки прерывания. Данная функция определена далее в скетче. К таким функциям, их называют *подпрограммами обработки прерываний* (Interrupt Service Routine, ISR), предъявляются особые требования. Они не могут иметь параметров и ничего не должны возвращать. В этом есть определенный смысл: даже при том что они вызываются в разных местах в скетче, нет ни одной строки кода, осуществляющей прямой вызов ISR, поэтому нет никакой возможности передать им

параметры или получить возвращаемое значение.

Последний параметр функции, `attachInterrupt` — это константа, в данном случае `FALLING`. Она означает, что подпрограмма обработки прерывания будет вызываться только при изменении напряжения на контакте **D2** с уровня `HIGH` до уровня `LOW` (то есть при падении — falling), что происходит в момент нажатия кнопки.

Обратите внимание на отсутствие какого-либо кода в функции `loop`. В общем случае эта функция может содержать код, выполняющийся, пока не произошло прерывание. Сама подпрограмма обработки прерываний просто включает светодиод **L**.

Когда вы будете экспериментировать, после сброса Arduino светодиод **L** должен погаснуть. А после нажатия на кнопку — сразу зажечься и оставаться зажженным до следующего сброса.

Поэкспериментировав, попробуйте изменить последний аргумент в вызове `attachInterrupt` на `RISING` и выгрузите измененный скетч. После перезапуска Arduino светодиод должен оставаться погашенным, потому что напряжение на контакте хотя и имеет уровень `HIGH`, но с момента перезапуска оставалось на этом уровне. До этого момента напряжение на контакте не падало до уровня `LOW`, чтобы потом подняться (rising) до уровня `HIGH`.

После нажатия и удержания кнопки в нажатом состоянии светодиод должен оставаться погашенным, пока вы ее не отпустите. Отпускание кнопки вызовет прерывание, связанное с контактом **D2**, потому что, пока кнопка удерживалась нажатой, уровень напряжения на контакте был равен `LOW`, а после отпускания поднялся до `HIGH`.

Если во время опробования выяснится, что происходящее у вас не соответствует описанию, приведенному ранее, это, скорее всего, обусловлено эффектом дребезга контактов в кнопке. Этот эффект вызывается тем, что кнопка не обеспечивает четкий переход между состояниями «включено»/«выключено», вместо этого в момент нажатия происходит многократный переход между этими состояниями, пока не зафиксируется состояние «включено». Попробуйте нажимать кнопку энергичнее, это должно помочь получить четкий переход между состояниями без эффекта дребезга.

Другой способ опробовать этот вариант скетча — нажать кнопку и, удерживая ее, нажать и отпустить кнопку сброса **Reset** на плате Arduino. Затем, когда скетч запустится, отпустить кнопку на макетной плате, и светодиод **L** загорится.

Контакты с поддержкой прерываний

Вернемся теперь к проблеме именования прерываний. В табл. 3.1 перечислены наиболее распространенные модели плат Arduino и приведено соответствие номеров прерываний и контактов в них.

Таблица 3.1. Контакты с поддержкой прерываний в разных моделях Arduino

Модель	Номер прерывания						Примечания
	0	1	2	3	4	5	
Uno	D2	D3	—	—	—	—	
Leonardo	D3	D2	D0	D1	D7	—	Действительно, по сравнению с Uno первые два прерывания назначены разным контактам
Mega2560	D2	D3	D21	D20	D19	D18	
Due	—	—	—	—	—	—	Вместо номеров прерываний функции attachInterrupt следует передавать номера контактов

Смена контактов первых двух прерываний в Uno и Leonardo создает ловушку, в которую легко попасть. В модели Due вместо номеров прерываний функции `attachInterrupt` следует передавать номера контактов, что выглядит более логично.

Режимы прерываний

Режимы прерываний `RISING` (по положительному перепаду) и `FALLING` (по отрицательному перепаду), использовавшиеся в предыдущем примере, чаще всего используются на практике. Однако существует еще несколько режимов. Эти режимы перечислены и описаны в табл. 3.2.

Таблица 3.2. Режимы прерываний

Режим	Действие	Описание
LOW	Прерывание генерируется при уровне напряжения LOW	В этом режиме подпрограмма обработки прерываний будет вызываться постоянно, пока на контакте сохраняется низкий уровень напряжения
RISING	Прерывание генерируется при положительном перепаде напряжения, с уровня LOW до уровня HIGH	—
FALLING	Прерывание генерируется при отрицательном перепаде напряжения, с уровня HIGH до уровня LOW	—
HIGH	Прерывание генерируется при уровне напряжения HIGH	Этот режим поддерживается только в модели Arduino Due и, подобно режиму LOW, редко используется на практике

Включение внутреннего импеданса

В схеме в предыдущем примере использовалось внешнее «подтягивающее» сопротивление. Однако на практике сигналы, вызывающие прерывания, часто заводятся с цифровых выходов датчиков, и в этом случае нет необходимости использовать «подтягивающее» сопротивление.

Но если роль датчика играет кнопка, подключенная точно так же, как макетная плата на рис. 3.1, есть возможность избавиться от сопротивления, включив внутреннее «подтягивающее» сопротивление с номиналом около 40 кОм. Для этого нужно явно настроить режим `INPUT_PULLUP` для контакта, связанного с прерыванием, как

показано в строке, выделенной жирным шрифтом:

```
void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(2, INPUT_PULLUP);
  attachInterrupt(0, stuffHapenned, FALLING);
}
```

Подпрограммы обработки прерываний

Иногда может показаться, что возможность обрабатывать прерывания, пока выполняется функция `loop`, дает простой способ обработки событий, таких как нажатия клавиш. Но в действительности накладываются очень жесткие ограничения на то, что можно или нельзя делать в подпрограммах обработки прерываний.

Подпрограммы обработки прерываний должны быть короткими и быстрыми настолько, насколько это возможно. Если во время работы подпрограммы обработки прерываний возникнет другое прерывание, эта подпрограмма не будет прервана, а полученный сигнал будет просто проигнорирован. Это, например, означает, что, если прерывания используются для измерения частоты, вы можете получить неверное значение.

Кроме того, пока выполняется подпрограмма обработки прерываний, код в функции `loop` простаивает.

На время обработки прерывания автоматически отключаются. Такое решение предохраняет от путаницы между подпрограммами, прерывающими друг друга, но имеет нежелательные побочные эффекты. Функция `delay` использует таймеры и прерывания, поэтому она не будет работать в подпрограммах обработки прерываний. То же относится к функции `millis`. Попытка использовать `millis` для получения числа миллисекунд, прошедших с момента последнего сброса платы, чтобы таким способом выполнить задержку, не приведет к успеху, так как она будет возвращать одно и то же значение, пока подпрограмма обработки прерываний не завершится. Однако вы можете использовать функцию `delayMicroseconds`, которая не использует прерываний.

Прерывания используются также во взаимодействиях через последовательные порты, поэтому не пытайтесь использовать `Serial.print` или функции чтения из последовательного порта. Впрочем, вы можете попробовать, и иногда они даже будут работать, но не ждите от такой связи высокой надежности.

Оперативные переменные

Так как подпрограмма обработки прерываний не может иметь параметров и не может

ничего возвращать, нужен какой-то способ передачи информации между ней и остальной программой. Обычно для этого используются глобальные переменные, как показано в следующем примере:

```
// sketch 03_02_interrupt_flash

int ledPin = 13;
volatile boolean flashFast = false;

void setup()
{
  pinMode(ledPin, OUTPUT);
  attachInterrupt(0, stuffHapenned, FALLING);
}

void loop()
{
  int period = 1000;
  if (flashFast) period = 100;
  digitalWrite(ledPin, HIGH);
  delay(period);
  digitalWrite(ledPin, LOW);
  delay(period);
}

void stuffHapenned()
{
  flashFast = ! flashFast;
}
```

В этом скетче функция `loop` использует глобальную переменную `flashFast`, чтобы определить период задержки. Подпрограмма обработки изменяет значение этой переменной между `true` и `false`.

Обратите внимание на то, что в объявление переменной `flashFast` включено слово volatile. Вы можете успешно разрабатывать скетч и без спецификатора `volatile`, но он совершенно необходим, потому что в отсутствие этого спецификатора компилятор C может генерировать машинный код, кэширующий значение переменной в регистре для увеличения производительности. Если, как в данном случае, кэширующий код будет прерван, он может не заметить изменения значения переменной.

В заключение о подпрограммах обработки прерываний

Когда будете писать подпрограммы обработки прерываний, помните следующие правила.

- Подпрограммы должны действовать быстро.
- Для передачи данных между подпрограммой обработки прерываний и остальной программой должны использоваться переменные, объявленные со спецификатором `volatile`.
- Не используйте `delay`, но можете использовать `delayMicroseconds`.
- Не ожидайте высокой надежности взаимодействий через последовательные порты.
- Не ожидайте, что значение, возвращаемое функцией `millis`, изменится.

Разрешение и запрет прерываний

По умолчанию прерывания в скетчах разрешены и, как упоминалось ранее, автоматически запрещаются на время работы подпрограммы обработки прерываний. Однако есть возможность явно запрещать и разрешать прерывания в программном коде, вызывая функции `noInterrupts` и `interrupts`. Эти функции не имеют параметров, и первая из них запрещает прерывания, а вторая — разрешает.

Явное управление может понадобиться, чтобы исключить возможность прерывания фрагмента кода, например, выводящего последовательность данных или генерирующего последовательность импульсов и точно выдерживающего временные интервалы с помощью функции `delayMicroseconds`.

Прерывания от таймера

Вызов подпрограмм обработки прерываний можно организовать не только по внешним событиям, но и по внутренним событиям изменения времени. Такая возможность особенно полезна, когда требуется выполнять некоторые операции через определенные интервалы времени.

Библиотека TimerOne упрощает настройку прерываний от таймера. Ее можно найти и загрузить по адресу **http://playground.arduino.cc/Code/Timer1**.

Следующий пример показывает, как с помощью TimerOne сгенерировать последовательность импульсов прямоугольной формы с частотой 1 кГц. Если в вашем распоряжении имеется осциллограф или мультиметр с возможностью измерения частоты, подключите его к контакту **12**, чтобы увидеть сигнал (рис. 3.2).

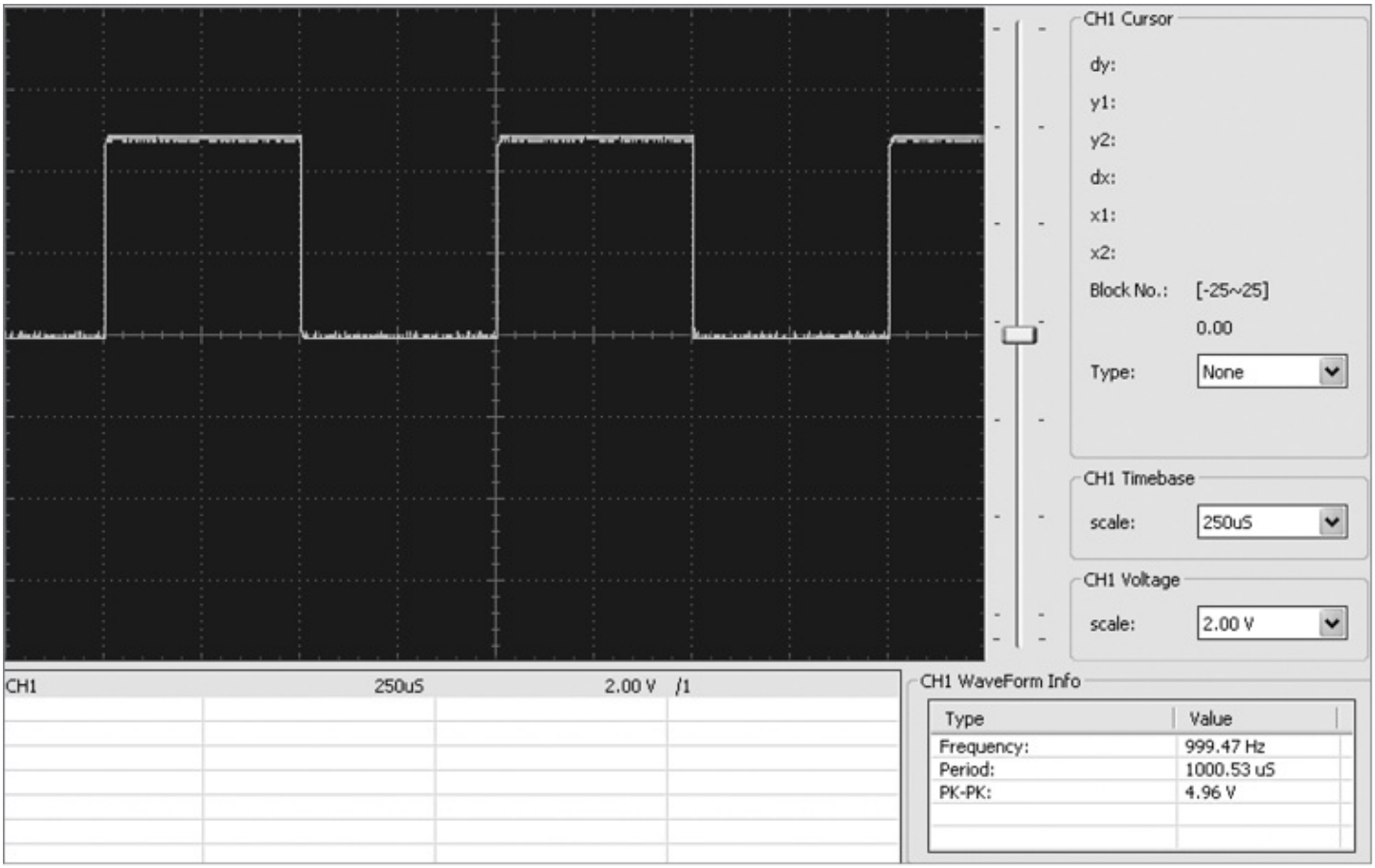


Рис. 3.2. Последовательность прямоугольных импульсов, сгенерированная с помощью таймера

```
// sketch_03_03_1kHz

#include <TimerOne.h>

int outputPin = 12;
volatile int output = LOW;

void setup()
{
  pinMode(12, OUTPUT);
  Timer1.initialize(500);
  Timer1.attachInterrupt(toggleOutput);
}

void loop()
{
}

void toggleOutput()
{
```

```
  digitalWrite(outputPin, output);
  output = ! output;
}
```

То же самое можно было бы реализовать с помощью `delay`, но применение прерываний от таймера позволяет организовать выполнение любых других операций внутри `loop`. Кроме того, использование функции `delay` не позволит добиться высокой точности, потому что время, необходимое на изменение уровня напряжения на контакте, не будет учитываться в величине задержки.

ПРИМЕЧАНИЕ

Все ограничения для подпрограмм обработки внешних прерываний, о которых рассказывалось ранее, распространяются также на подпрограммы обработки прерываний от таймера.

Представленным способом можно установить любой интервал между прерываниями в диапазоне от 1 до 8 388 480 мкс, то есть примерно до 8,4 с. Величина интервала передается функции `initialize` в микросекундах.

Библиотека TimerOne дает возможность также использовать таймер для генерирования сигналов с широтно-импульсной модуляцией (Pulse Width Modulation, PWM) на контактах **9** и **10** платы. Это может показаться излишеством, потому что то же самое делает функция `analogWrite`, но применение прерываний позволяет обеспечить более точное управление сигналом PWM. В частности, используя такой подход, можно организовать измерение протяженности положительного импульса в диапазоне 0...1023 вместо 0...255 в функции `analogWrite`. Кроме того, при использовании `analogWrite` частота следования импульсов в сигнале PWM составляет 500 Гц, а с помощью TimerOne можно эту частоту увеличить или уменьшить.

Чтобы сгенерировать сигнал PWM с применением библиотеки TimerOne, используйте функцию `Timer1.pwm`, как показано в следующем примере:

```
// sketch_03_04_pwm

#include <TimerOne.h>

void setup()
{
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  Timer1.initialize(1000);
```

```
  Timer1.pwm(9, 512);
  Timer1.pwm(10, 255);
}

void loop()
{
}
```

Здесь выбран период следования импульсов, равный 1000 мкс, то есть частота сигнала PWM составляет 1 кГц. На рис. 3.3 показана форма сигналов на контактах **10** (*вверху*) и **9** (*внизу*).

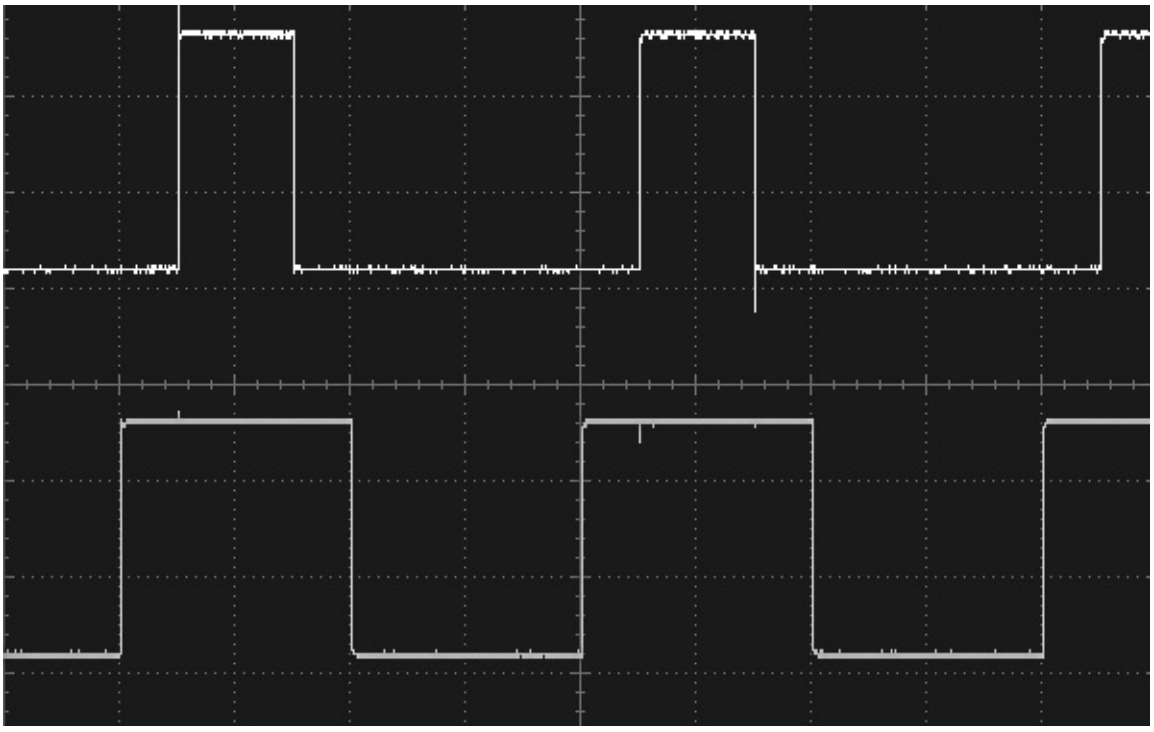


Рис. 3.3. Широтно-импульсный сигнал с частотой 1 кГц, сгенерированный с помощью TimerOne

Ради интереса давайте посмотрим, до какой степени можно увеличить частоту сигнала PWM. Если уменьшить длительность периода до 10, частота сигнала PWM должна увеличиться до 100 кГц. Форма сигналов, полученных с этими параметрами, показана на рис. 3.4.

Несмотря на наличие существенных переходных искажений, что вполне ожидаемо, протяженность положительных импульсов все же остается довольно близкой к 25 и 50% соответственно.

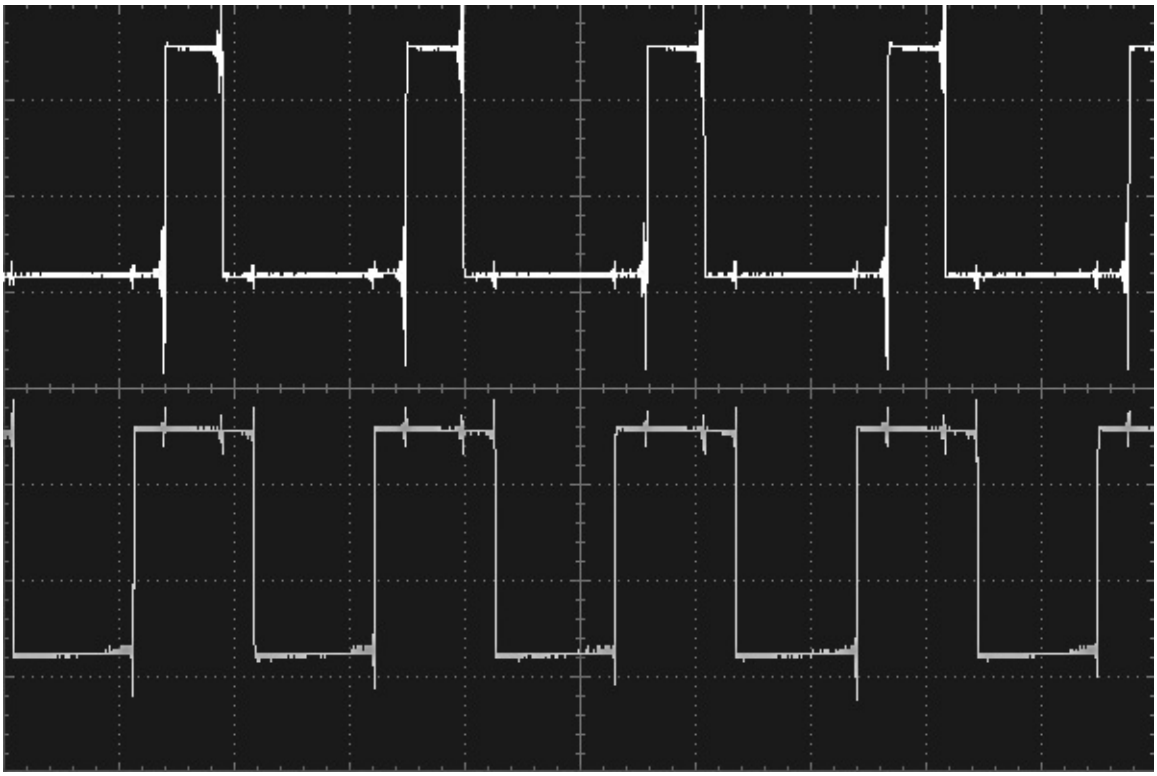


Рис. 3.4. Широтно-импульсный сигнал с частотой 100 кГц, сгенерированный с помощью TimerOne

В заключение

Прерывания, которые иногда кажутся идеальным решением для непростых проектов, могут осложнить отладку кода и не всегда оказываются лучшим способом решения трудных задач. Тщательно обдумайте возможные решения, прежде чем переходить к их использованию. В главе 14 мы познакомимся с другим приемом преодоления сложностей, связанных с тем, что Arduino не может выполнять более одной задачи одновременно.

Мы еще вернемся к прерываниям в главе 5, где рассмотрим возможность их применения для уменьшения потребления электроэнергии платой Arduino за счет периодического перевода ее в режим энергосбережения, и в главе 13, где прерывания будут применяться для увеличения точности обработки цифровых сигналов.

В следующей главе мы познакомимся с приемами увеличения производительности Arduino до максимума.

4. Ускорение Arduino

В этой главе рассказывается, как определить производительность платы Arduino и как выжать из нее дополнительную мощность, когда это необходимо.

Как определить производительность Arduino?

Прежде чем заняться изучением приемов увеличения скорости работы скетчей, потратим немного времени на тестирование Arduino, чтобы просто понять, насколько ее производительность сопоставима с производительностью компьютеров, начав с таких понятий, как мегагерц и гигагерц.

Тактовый генератор на плате Arduino Uno имеет частоту 16 МГц. Большинство инструкций (сложения или сохранения значения в переменной) выполняется за один такт. То есть Uno может выполнять до 16 млн элементарных операций в секунду. Вроде бы неплохо, не так ли? Однако все не так просто, потому что инструкции на языке C, которые вы пишете в скетчах, разворачиваются в множество машинных инструкций.

Теперь сравним плату с моим стареньким ноутбуком Mac, имеющим два процессора, работающих с тактовой частотой 2,5 ГГц. Тактовая частота моего ноутбука более чем в 150 раз выше тактовой частоты Arduino. И хотя для выполнения каждой инструкции процессору требуется несколько тактов, он все же оказывается намного быстрее.

Попробуем выполнить следующую тестовую программу на Arduino и немного измененную ее версию — на моем Mac:

```
// sketch 04_01_benchmark

void setup()
{
  Serial.begin(9600);
  Serial.println("Starting Test");
  long startTime = millis();

  // Далее следует код тестирования
  long  i = 0;
  long j = 0;
  for (i = 0; i < 20000000; i ++)
  {
    j = i + i * 10;
    if (j > 10) j = 0;
```

```
  }
  // конец кода, выполняющего тестирование
  long endTime = millis();

   Serial.println(j); // чтобы предотвратить оптимизацию цикла компилятором
  Serial.println("Finished Test");
  Serial.print("Seconds taken: ");
  Serial.println((endTime – startTime) / 1000l);
}

void loop()
{

}
```

ПРИМЕЧАНИЕ

Версию программы на C для компьютера можно найти в разделе загрузки примеров на веб-сайте книги.

Вот какие результаты получились: на MacBook Pro с процессором 2,5 ГГц тестовая программа выполнялась 0,068 с, тогда как на Arduino Uno ей понадобилось 28 с. Плата Arduino оказалась примерно в 400 раз медленнее при решении данной задачи.

Сравнение плат Arduino

В табл. 4.1 показаны результаты выполнения этого теста в нескольких разных моделях платы Arduino.

Таблица 4.1. Результаты тестирования быстродействия Arduino

Модель	Время выполнения теста, с
Uno	28
Leonardo	29
Arduino Mini Pro	28
Mega2560	28
Due	2

Как видите, большинство моделей имеют схожую производительность, и только Due показала внушительный результат — она оказалась более чем в 10 раз быстрее

остальных моделей.

Скорость арифметических операций

Для дальнейших исследований изменим только что использованный тест и вместо арифметики с длинными целыми протестируем быстродействие арифметики с вещественными числами. И те и другие занимают в памяти 32 бита, поэтому можно было бы ожидать, что время работы примера останется сопоставимым. В следующем тесте используем Arduino Uno.

```
// sketch 04_02_benchmark_float

void setup()
{
  Serial.begin(9600);
  while (! Serial) {};
  Serial.println("Starting Test");
  long startTime = millis();

  // Далее следует код тестирования
  long  i = 0;
  float j = 0.0;
  for (i = 0; i < 20000000; i ++)
  {
    j = i + i * 10.0;
    if (j > 10) j = 0.0;
  }
  // конец кода, выполняющего тестирование
  long endTime = millis();

   Serial.println(j); // чтобы предотвратить оптимизацию цикла компилятором
  Serial.println("Finished Test");
  Serial.print("Seconds taken: ");
  Serial.println((endTime – startTime) / 1000l);
}

void loop()
{
```

```
}
```

К сожалению, с использованием вещественных чисел этот скетч выполняется намного дольше. Этот пример выполнялся в Arduino около 467 с вместо 28 с. То есть простая замена длинных целых чисел вещественными уменьшила скорость выполнения более чем в 16 раз. Справедливости ради следует заметить, что отчасти ухудшение обусловлено дополнительными операциями преобразования между значениями вещественных и целочисленных типов, которые также обходятся недешево в смысле времени выполнения.

Нужны ли вещественные числа в действительности?

Многие ошибочно полагают, что если измеряется такая характеристика, как температура, ее значение обязательно следует хранить в виде вещественного числа, потому что оно часто будет выражаться дробным числом, таким как 23,5. Вещественное число действительно может понадобиться, чтобы отобразить температуру, но ее необязательно хранить именно в таком виде.

Значения, прочитанные с аналоговых входов, имеют тип `int`, и на самом деле значимыми являются только 12 бит, что соответствует целым числам в диапазоне между 0 и 1023. При желании можно, конечно, сохранить эти 12 бит в 32-битном вещественном числе, но это никак не отразится на точности данных.

Значение, читаемое с датчика, может соответствовать, например, температуре в градусах Цельсия. Широко известный температурный датчик (TMP36) выводит напряжение, пропорциональное температуре. В скетчах, как показано далее, часто можно увидеть вычисления, преобразующие значение в диапазоне 0...1023, прочитанное с аналогового входа, в температуру в градусах Цельсия:

```
int raw = analogRead(sensePin);
float volts = raw / 205.0;
float tempC = 100.0 * volts – 50;
```

Но в действительности температура в виде вещественного числа нужна только тогда, когда требуется отобразить ее на экране. Другие операции с температурой, такие как сравнение или усреднение при нескольких попытках чтения, вполне можно выполнять с непреобразованным значением типа `int`, и при этом они будут выполняться значительно быстрее.

Поиск против вычисления

Как вы уже поняли, в скетчах вещественных чисел лучше избегать. Но как быть, если

понадобится сгенерировать на аналоговом выходе сигнал синусоидальной формы, для чего, как можно догадаться, потребуется вычислять *синус* вызовом функции `sin`? Чтобы сформировать синусоидальный сигнал на аналоговом выходе, нужно обойти диапазон значений угла от 0 до 2π и вывести на аналоговый выход значение синуса этого угла. На самом деле все немного сложнее, потому что синусоиду нужно привести к диапазону значений, которые можно вывести на аналоговый выход.

Следующий пример генерирует синусоиду, разбивая каждый цикл на 64 шага, и выводит сигнал на аналоговый выход **DAC0** платы Arduino Due. Имейте в виду, что для данного эксперимента годятся только платы Arduino с истинными аналоговыми выходами, такие как Due.

```
// sketch_-4_03_sin

void setup()
{

}

float angle = 0.0;
float angleStep = PI / 32.0;

void loop()
{
  int x = (int)(sin(angle) * 127) + 127;
  analogWrite(DAC0, x);
  angle += angleStep;
  if (angle > 2 * PI)
  {
    angle = 0.0;
  }
}
```

Измерение на выходе показывает, что данный скетч действительно производит сигнал замечательной синусоидальной формы, но с частотой всего 310 Гц. Процессор на плате Arduino Due работает с тактовой частотой 80 МГц, поэтому можно было бы ожидать увидеть сигнал с большей частотой. Проблема в том, что здесь скетч снова и снова повторяет одни и те же вычисления. Но поскольку каждый раз получаются одни и те же результаты, почему бы просто не рассчитать их все сразу и не сохранить в массиве?

Следующий пример также генерирует синусоиду, разбивая цикл на 64 шага, но

использует прием поиска по таблице заранее подготовленных значений, которые выводит непосредственно в цифроаналоговый преобразователь (ЦАП).

```
byte sin64[] = {127, 139, 151, 163, 175, 186, 197,
207, 216, 225, 232, 239, 244, 248, 251, 253, 254,
253, 251, 248, 244, 239, 232, 225, 216, 207, 197, 186,
175, 163, 151, 139, 126, 114, 102, 90, 78, 67, 56, 46,
37, 28, 21, 14, 9, 5, 2, 0, 0, 0, 2, 5, 9, 14, 21, 28,
37, 46, 56, 67, 78, 90, 102, 114, 126};

void setup()
{
}

void loop()
{
  for (byte i = 0; i < 64; i++)
  {
    analogWrite(DAC0, sin64[i]);
  }
}
```

Этот пример генерирует точно такой же сигнал в форме синусоиды, но уже с частотой 4,38 кГц, то есть работает более чем в 14 раз быстрее.

Таблицу синусов можно рассчитать разными способами. Можно сгенерировать числа по обычной формуле в электронной таблице или написать скетч, который будет выводить числа в монитор последовательного порта, откуда их можно скопировать и вставить в другой скетч. Далее приводится версия скетча sketch_04_03_sin, которая выводит значения один раз в монитор последовательного порта:

```
// sketch_-4_05_sin_print

float angle = 0.0;
float angleStep = PI / 32.0;

void setup()
{
  Serial.begin(9600);
  Serial.print("byte sin64[] = {");
  while (angle < 2 * PI)
```

```
    {
      int x = (int)(sin(angle) * 127) + 127;
      Serial.print(x);
      angle += angleStep;
      if (angle < 2 * PI)
      {
        Serial.print(", ");
      }
    }
    Serial.println("};");
}

void loop()
{
}
```

Открыв окно монитора порта, вы увидите сгенерированную последовательность чисел (рис. 4.1).

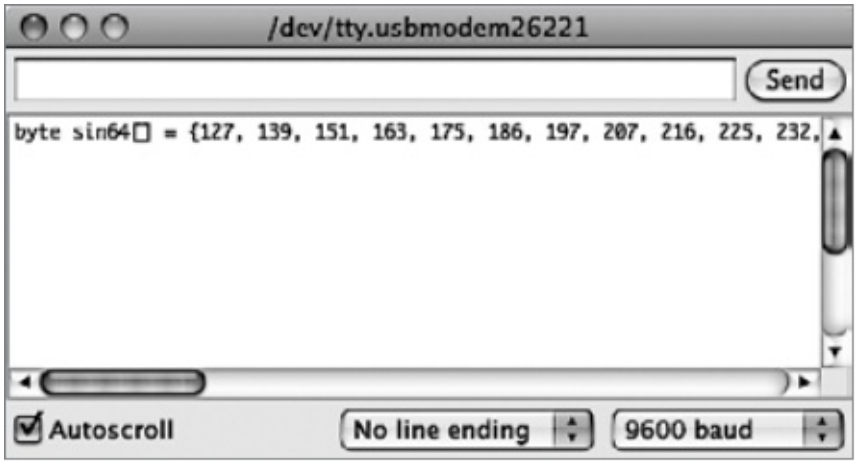


Рис. 4.1. Использование скетча для получения массива чисел

Быстрый ввод/вывод

В этом разделе мы посмотрим, как увеличить скорость включения и выключения цифровых выходов. Мы увеличим максимальную частоту с 73 кГц почти до 4 МГц.

Простая оптимизация кода

Начнем с простого кода, включающего и выключающего цифровой выход с помощью `digitalWrite`:

```
// sketch_04_05_square

int outPin = 10;
int state = 0;

void setup()
{
  pinMode(outPin, OUTPUT);
}

void loop()
{
  digitalWrite(outPin, state);
  state = ! state;
}
```

Если запустить этот скетч и подключить осциллограф или частотомер к цифровому контакту **10**, вы получите частоту чуть выше 73 кГц (мой осциллограф показал 73,26 кГц).

Прежде чем сделать большой шаг в направлении непосредственного управления портом, можно попробовать немного оптимизировать программный код скетча. Прежде всего, ни одна из переменных не обязана иметь тип `int`, их вполне можно объявить с типом `byte`. Это изменение увеличит частоту до 77,17 кГц. Далее переменную с номером контакта можно сделать константой, добавив слово `const` перед объявлением переменной. Это изменение увеличит частоту до 77,92 кГц.

В главе 2 вы узнали, что функция `loop` — это не просто цикл `while`, так как дополнительно проверяет наличие входящих данных в последовательном порте. То есть следующим шагом в направлении увеличения производительности может стать отказ от функции `loop` и перенос кода в `setup`. Скетч, в котором выполнены все описанные изменения, приводится ниже:

```
// sketch_04_08_no_loop

const byte outPin = 10;
byte state = 0;

void setup()
{
  pinMode(outPin, OUTPUT);
  while (true)
```

```
  {
    digitalWrite(outPin, state);
    state = ! state;
  }
}

void loop()
{
}
```

В результате всего этого мы получили увеличение максимальной частоты до 86,39 кГц.

В табл. 4.2 перечислены все улучшения, которые можно выполнить для увеличения производительности простого программного кода, прежде чем сделать последний шаг и заменить `digitalWrite` чем-нибудь более быстрым.

Таблица 4.2. Увеличение производительности простого программного кода

Действие	Скетч	Частота, кГц
Исходная версия	04_05	72,26
Объявление с типом byte вместо int	04_06	77,17
Использование константы с номером контакта вместо переменной	04_07	77,92
Перенос содержимого loop в setup	04_08	86,39

Байты и биты

Прежде чем переходить к непосредственному управлению портами ввода/вывода, нужно сначала разобраться с двоичным представлением, битами, байтами и целыми числами.

На рис. 4.2 показано, как связаны биты и байты.

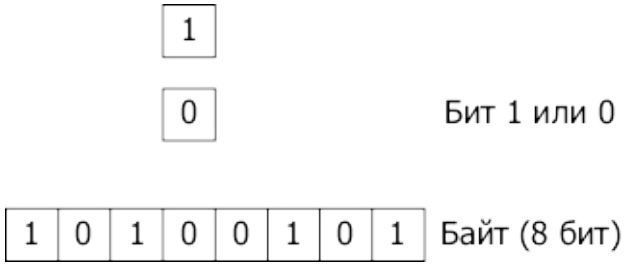


Рис. 4.2. Биты и байты

Бит (в английском языке *bit*, происходит от *binary digit* — двоичная цифра) может иметь одно из двух значений — 0 или 1. *Байт* — это коллекция из 8 битов. Так как

каждый из битов в байте может иметь значение 1 или 0, всего возможно 256 разных комбинаций битов в байте. Байт можно использовать для представления любых чисел в диапазоне от 0 до 255.

Каждый бит можно использовать также для обозначения состояния «включено» или «выключено». То есть, чтобы включить или выключить подачу напряжения на какой-то контакт, нужно установить или сбросить некоторый бит.

Порты в ATmega328

На рис. 4.3 изображены порты в микроконтроллере ATmega328 и то, как они связаны с контактами на плате Arduino Uno.

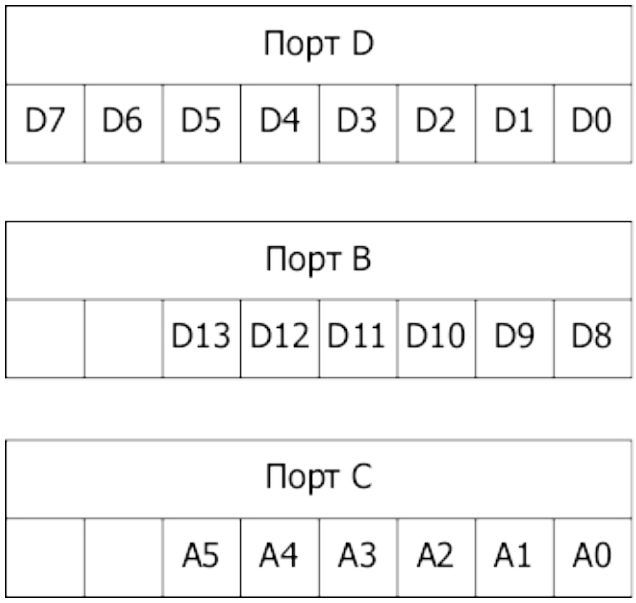


Рис. 4.3. Порты в ATmega328

Каждый порт не случайно имеет по 8 бит (байт), хотя в портах B и C используется только по 6 бит. Каждый порт управляется тремя *регистрами*. Регистр можно считать специальной переменной, позволяющей присваивать ей значения и читать значение из нее. На рис. 4.4 изображены регистры для порта D.

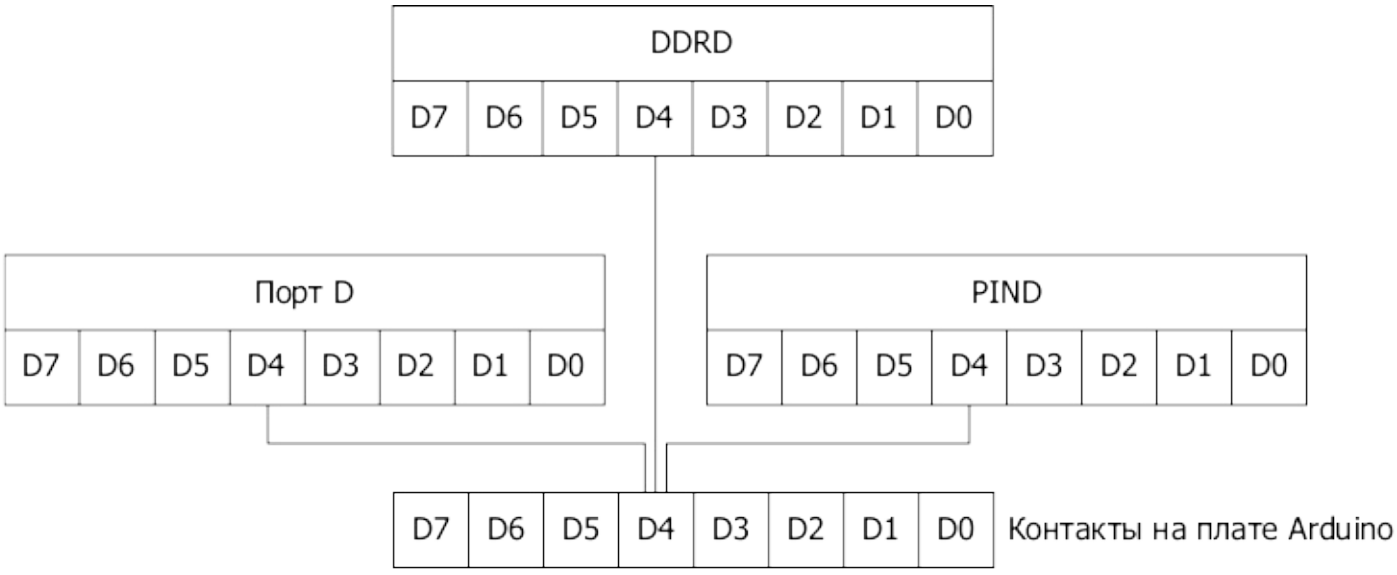


Рис. 4.4. Регистры для порта D

Регистр DDRD (Data Direction Register D — регистр D направления передачи данных) имеет 8 бит, каждый из которых определяет режим работы соответствующего контакта — вход или выход. Если бит установлен в значение 1, контакт работает как выход, в противном случае — как вход. Этим регистром управляет функция `pinMode`. Регистр *PORTD* используется для установки выходного напряжения на выходе, то есть `digitalWrite` устанавливает соответствующий бит, 1 или 0, чтобы установить на указанном контакте уровень напряжения `HIGH` или `LOW`.

Последний регистр называется *PIND* (Port Input D — вход порта D). Читая содержимое этого регистра, можно определить, на какие контакты подано напряжение `HIGH`, а на какие — `LOW`.

Каждый из трех портов имеет свои три регистра, для порта B они называются DDRB, PORTB и PINB, а для порта C — DDRC, PORTC и PINC.

Очень быстрый вывод цифровых сигналов

Следующий скетч обращается к портам напрямую, без применения `pinMode` и `digitalWrite`:

```
// sketch_04_09_square_ports

byte state = 0;

void setup()
{
  DDRB = B00000100;
  while (true)
  {
```

```
    PORTB = B00000100;
    PORTB = B00000000;
  }
}

void loop()
{
}
```

Скетч должен переключать контакт **D10**, который связан с портом B, поэтому вначале контакт настраивается на работу в режиме выхода, для чего третий бит справа в регистре DDRB устанавливается в 1. Обратите внимание на то, что `B00000100` — это двоичная константа. В главном цикле мы сначала устанавливаем тот же бит в 1, а затем сбрасываем его в 0. Установка бита производится как простое присваивание значения регистру PORTB, как если бы это была обычная переменная.

Этот скетч способен генерировать сигнал с частотой 3,97 МГц (рис. 4.5) — почти 4 млн импульсов в секунду, что почти в 46 раз быстрее, чем с использованием `digitalWrite`.

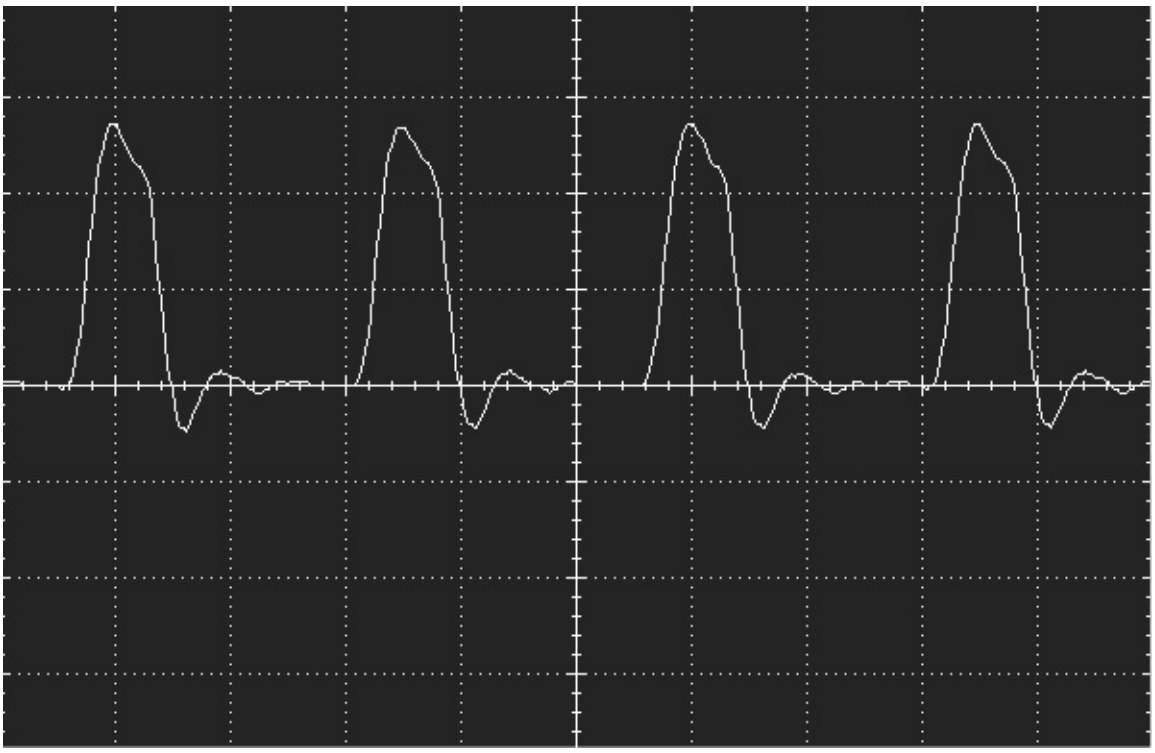


Рис. 4.5. Сигнал с частотой 4 МГц, сгенерированный платой Arduino

Сигнал далек от прямоугольной формы из-за переходных процессов, которые вполне ожидаемы на такой частоте.

Еще одно преимущество непосредственного использования регистров порта — возможность вывода сигналов сразу на восемь контактов, что может пригодиться для вывода данных в параллельную шину данных.

Быстрый ввод цифровых сигналов

Тот же прием непосредственного доступа к регистрам можно использовать для увеличения скорости ввода цифровых сигналов. Хотя, если вы предполагаете таким способом определять моменты появления очень коротких импульсов, подумайте о возможности использования прерываний, они являются лучшим решением этой задачи (см. главу 3).

Прямой доступ к порту может пригодиться, например, когда требуется прочитать состояние сразу нескольких контактов. Следующий скетч читает все контакты, связанные с портом B (с **D8** по **D13**), и выводит результат в монитор последовательного порта в виде двоичного числа (рис. 4.6).

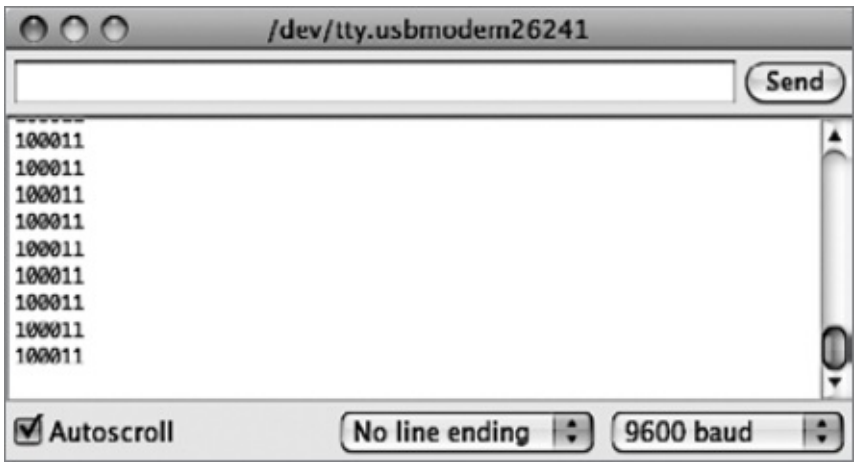


Рис. 4.6. Чтение состояния сразу восьми контактов

```
// sketch_04_010_direct_read

byte state = 0;

void setup()
{
  DDRB = B00000000; // все контакты на ввод
  Serial.begin(9600);
}

void loop()
{
  Serial.println(PINB, 2);
  delay(1000);
}
```

Сбросом всех битов в регистре DDRB в 0 соответствующие контакты на плате настраиваются на работу в режиме входов. В цикле вызывается функция

Serial.println, которая посылает число в монитор последовательного порта. Чтобы число посылалось в двоичной форме, а не в десятичной, как обычно, передается дополнительный аргумент 2.

Увеличение скорости ввода аналоговых сигналов

Давайте изменим скетч, который выполняет хронометраж, чтобы узнать, как долго работает analogRead, а потом попробуем ее ускорить:

```
// sketch 04_11_analog

void setup()
{
  Serial.begin(9600);
  while (! Serial) {};
  Serial.println("Starting Test");
  long startTime = millis();

  // Далее следует код тестирования
  long  i = 0;
  for (i = 0; i < 1000000; i ++)
  {
    analogRead(A0);
  }
  // конец кода, выполняющего тестирование
  long endTime = millis();

  Serial.println("Finished Test");
  Serial.print("Seconds taken: ");
  Serial.println((endTime – startTime) / 1000l);
}

void loop()
{
}
```

На плате Arduino Uno этот скетч выполняется 112 с. То есть Uno выполняет в секунду около 9000 операций чтения аналоговых сигналов.

Функция analogRead использует АЦП, имеющийся в микроконтроллере на плате

Arduino. В Arduino используется тип АЦП, который называют *АЦП с последовательной аппроксимацией*. Он действует методом постепенного приближения, сравнивая аналоговый сигнал с опорным напряжением. АЦП управляется таймером, поэтому есть возможность ускорить преобразование, увеличив частоту.

Следующий скетч увеличивает частоту АЦП со 128 кГц до 1 МГц, что должно увеличить скорость чтения в восемь раз:

```
// sketch 04_11_analog_fast

const byte PS_128 = (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
const byte PS_16 = (1 << ADPS2);

void setup()
{
  ADCSRA &= ~PS_128;  // сбросить масштаб 128
  ADCSRA |= PS_16;    // добавить масштаб 16 (1 МГц)
  Serial.begin(9600);
  while (! Serial) {};
  Serial.println(PS_128, 2);
  Serial.println(PS_16, 2);
  Serial.println("Starting Test");
  long startTime = millis();

  // Далее следует код тестирования
  long  i = 0;
  for (i = 0; i < 1000000; i ++)
  {
    analogRead(A0);
  }
  // конец кода, выполняющего тестирование
  long endTime = millis();

  Serial.println("Finished Test");
  Serial.print("Seconds taken: ");
  Serial.println((endTime – startTime) / 1000l);
}

void loop()
{
```

```
}
```

Теперь скетч выполняется всего 17 с, то есть, грубо, в 6,5 раза быстрее, а скорость измерений увеличилась до 58 000 в секунду. Этого вполне достаточно для оцифровки аудиосигнала, хотя при наличии всего 2 Кбайт ОЗУ вы не сможете записать большой фрагмент!

Если первоначальный вариант скетча sketch_04_11_analog запустить в Arduino Due, он справится с работой за 39 с. Однако в модели Due не получится использовать трюк с регистрами портов, так как она имеет совсем другую архитектуру.

В заключение

В этой главе мы попытались выжать все до последней капли из наших скудных 16 МГц. В следующей главе переключим внимание на снижение потребления электроэнергии платой Arduino, что очень важно для проектов, где плату предполагается питать от аккумуляторов или солнечных батарей.

5. Снижение потребления электроэнергии

Справедливости ради следует отметить, что и без применения специальных мер платы Arduino потребляют не особенно много электроэнергии. Обычно Arduino Uno потребляет ток около 40 мА, что при питании через разъем USB с напряжением 5 В составляет всего 200 мВт. Это означает, что она может благополучно работать около четырех часов, питаясь от аккумулятора 9 В (емкостью 150 мА·ч).

Потребление электроэнергии становится важным аспектом, когда плата Arduino должна работать длительное время, питаясь от аккумулятора, как в системах удаленного мониторинга или управления, когда аккумуляторы или солнечные батареи остаются единственно возможным источником питания. Например, недавно на основе платы Arduino я реализовал автоматическое управление дверью в птичник, использовав небольшую солнечную панель для зарядки аккумулятора, емкости которого достаточно только для того, чтобы открыть и закрыть дверь два раза в день.

Потребление электроэнергии платами Arduino

Прежде всего определим параметры потребления электроэнергии наиболее популярными платами Arduino. В табл. 5.1 представлены результаты непосредственных измерений амперметром силы тока, потребляемого платами. Имейте в виду, что измерение силы потребляемого тока не самая простая задача, так как он меняется, когда при выполнении периодических задач в работу включаются таймеры и другие компоненты микроконтроллера и платы Arduino.

Таблица 5.1. Потребление электроэнергии платами Arduino

Плата	Ток, мА
Uno (5 В, USB)	47
Uno (9 В, внешний источник питания)	48
Uno (5 В, с извлеченным процессором)	32
Uno (9 В, с извлеченным процессором)	40
Leonardo (5 В, USB)	42
Due (5 В, USB)	160
Due (9 В, внешний источник питания)	70
Mini Pro (9 В, внешний источник питания)	42
Mini Pro (5 В, USB)	22
Mini Pro (3,3 В, непосредственно)	8

Обратите внимание на то, как различается ток, потребляемый платами Arduino, питающимися напряжением 5 В, с процессором и без него. Разница составляет всего 15

мА, откуда получается, что остальные 32 мА потребляет сама плата. И действительно, на плате Arduino имеются интерфейс USB, светодиод **On** и стабилизатор напряжения 3,3 В, которые также потребляют некоторую мощность даже без микроконтроллера. Обратите также внимание на то, насколько меньше потребляет микроконтроллер, питающийся напряжением 3,3 В.

Приемы, описываемые далее, помогают снизить потребление электроэнергии процессором, но не самой платой. В примерах, приведенных в дальнейшем, я использовал плату Arduino Mini Pro, питающуюся непосредственно напряжением 3,3 В через контакты **VCC** и **GND** (рис. 5.1) в обход стабилизатора напряжения, чтобы кроме светодиода **On** питание подводилось только к микроконтроллеру.

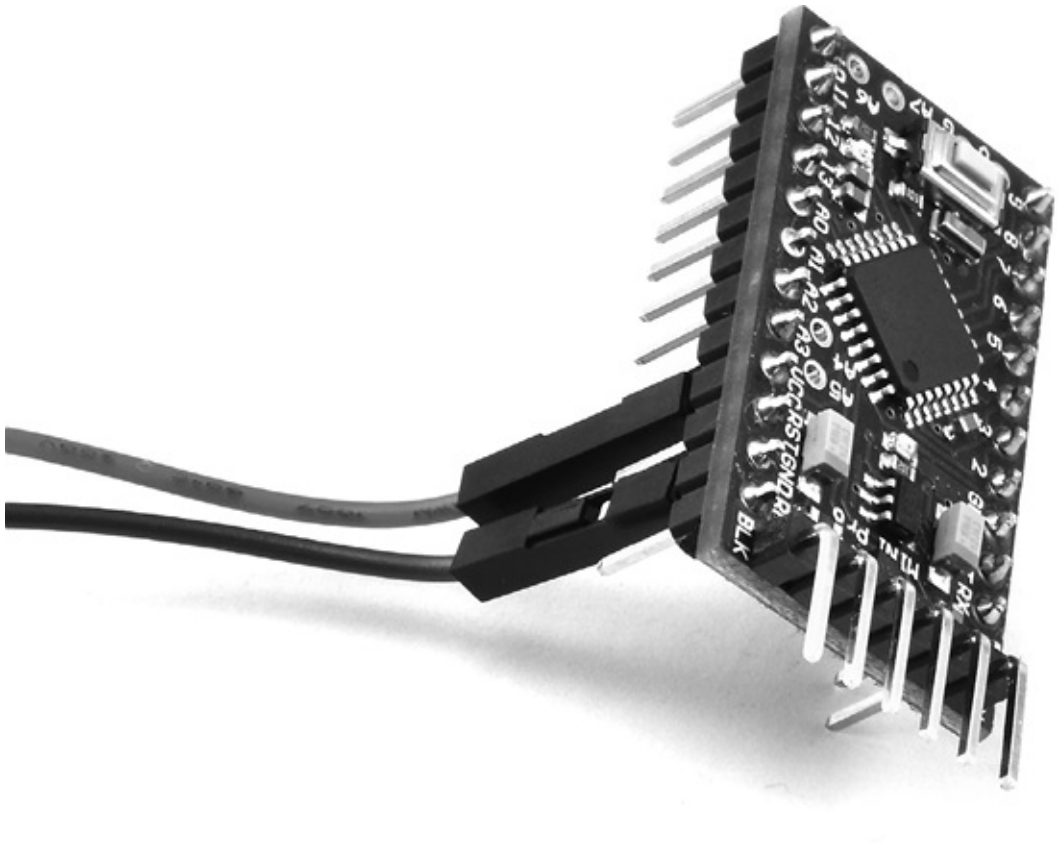


Рис. 5.1. Плата Arduino Mini Pro, запитанная непосредственно напряжением 3 В

Такая схема часто используется в системах с автономным питанием от аккумуляторов, например от единственного литий-полимерного (Lithium Polymer, LiPo) аккумулятора, дающего напряжение 2,7 В, когда почти разряжен, и 4,2 В, когда полностью заряжен, который прекрасно подходит для непосредственного питания микроконтроллера ATmega328.

Ток и аккумуляторы

Эта книга посвящена программному обеспечению, поэтому я не буду останавливаться на обсуждении аккумуляторов дольше, чем необходимо. На рис. 5.2 изображены аккумуляторы, которые можно использовать для питания плат Arduino.

Слева вверху изображен цилиндрический литий-полимерный аккумулятор емкостью 2400 мА·ч. Ниже — небольшой плоский литий-полимерный аккумулятор емкостью 850 мА·ч. Литий-полимерные аккумуляторы имеют небольшой вес, могут перезаряжаться много раз и имеют большую емкость для своих веса и размеров. Справа вверху изображен 9-вольтовый никель-металлгидридный аккумулятор емкостью 200 мА·ч. Этот

Рис. 5.2. Аккумуляторы для питания плат Arduino

аккумулятор тоже поддерживает многократную перезарядку, но создан по устаревшей технологии. Так как он имеет выходное напряжение 9 В, его можно использовать для питания плат Arduino только через встроенный стабилизатор напряжения. Вы можете приобрести специальные зажимы для подключения аккумулятора к контактам питания Arduino. Наконец, справа внизу изображена 3-вольтовая незаряжаемая литиевая батарея (CR2025) емкостью около 160 мА·ч.

Как правило, чтобы получить время в часах, в течение которого аккумулятор продержится, прежде чем полностью разрядится, достаточно разделить емкость аккумулятора в миллиампер-часах [мА·ч] на силу потребляемого тока в миллиамперах [мА]:

Время работы батареи = Емкость батареи/Потребляемый ток.

Например, если для питания 3-вольтовой платы Mini Pro использовать батарею

CR2025, можно ожидать, что ее хватит на 20 часов (160 мА·ч/8 мА). Если ту же плату запитать от литий-полимерного аккумулятора емкостью 2400 мА·ч, можно надеяться, что его хватит на 300 часов (2400 мА·ч /8 мА).

Снижение рабочей частоты

Большинство плат семейства Arduino работает с тактовой частотой 16 МГц. Основное потребление электроэнергии микроконтроллером происходит в моменты, когда тактовый сигнал переключается из состояния `HIGH` в состояние `LOW`, то есть частота, на которой работает процессор, оказывает существенное влияние на потребляемый ток. Конечно, уменьшение тактовой частоты приведет к снижению быстродействия микроконтроллера, что, впрочем, может не являться проблемой.

Снизить рабочую частоту микроконтроллера ATmega328 можно прямо из скетча. Для этой цели удобно использовать библиотеку Arduino Prescaler (**http://playground.arduino.cc/Code/Prescaler**).

Библиотека Prescaler не только позволяет уменьшить рабочую частоту микроконтроллера, но и предоставляет свои версии функций `millis` и `delay` с именами `trueMillis` и `trueDelay`. Такая замена совершенно необходима, потому что уменьшение тактовой частоты увеличивает задержки в той же пропорции.

Скетч в следующем примере включает светодиод **L** на 1 с и затем выключает на 5 с, в течение которых потребляемый ток измерялся для всех возможных значений деления частоты, поддерживаемых библиотекой Prescaler.

```
// sketch_05_01_prescale

#include <Prescaler.h>

void setup()
{
  pinMode(13, OUTPUT);
  setClockPrescaler(CLOCK_PRESCALER_256);
}

void loop()
{
  digitalWrite(13, HIGH);
  trueDelay(1000);
  digitalWrite(13, LOW);
  trueDelay(5000);
}
```

Библиотека предоставляет множество констант деления тактовой частоты. Так, константа CLOCK_PRESCALER_1 оставляет исходную тактовую частоту 16 МГц, а противоположная ей константа CLOCK_PRESCALER_256 делит исходную тактовую частоту на 256, устанавливая ее на уровне всего 62,5 кГц.

В табл. 5.2 показаны результаты измерения потребляемого тока на всех возможных частотах, а на рис. 5.3 те же данные представлены в виде графика. Как видно на графике, кривая потребления тока быстро выравнивается, поэтому частота 1 МГц выглядит наиболее оптимальным компромиссом между частотой и потребляемым током.

Таблица 5.2. Потребляемый ток в зависимости от тактовой частоты

Константа	Эквивалентная тактовая частота	Ток (светодиод выключен), мА
CLOCK_PRESCALER_1	16 МГц	7,8
CLOCK_PRESCALER_2	8 МГц	5,4
CLOCK_PRESCALER_4	4 МГц	4,0
CLOCK_PRESCALER_8	2 МГц	3,2
CLOCK_PRESCALER_16	1 МГц	2,6
CLOCK_PRESCALER_32	500 кГц	2,3
CLOCK_PRESCALER_64	250 кГц	2,2
CLOCK_PRESCALER_128	125 кГц	2,1
CLOCK_PRESCALER_256	62,5 кГц	2,1

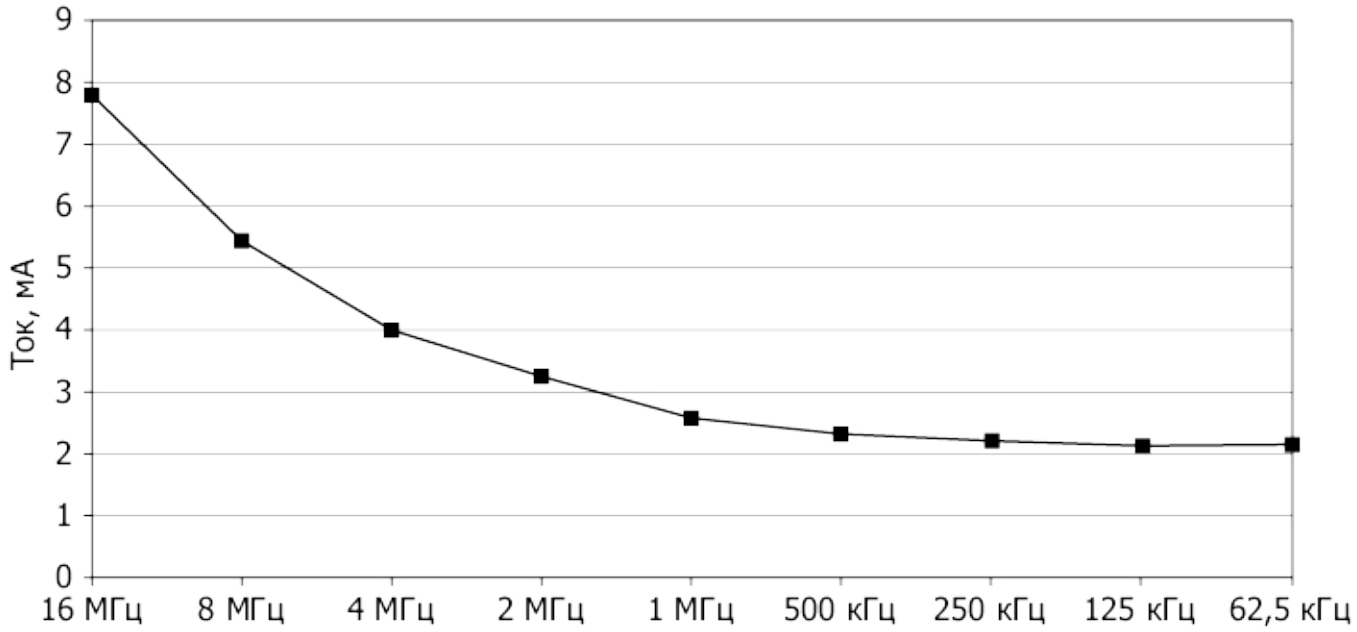


Рис. 5.3. График зависимости потребляемого тока от тактовой частоты

Помимо необходимости использовать новые версии millis и delay снижение тактовой частоты влечет за собой еще ряд следствий. Фактически любая операция,

чувствительная к изменению тактовой частоты, такая как вывод аналоговых сигналов PWM или управление сервоприводами, будет выполняться не так, как ожидается.

Большая доля тока из 2,1 мА, потребляемого на самой низкой скорости, вероятнее всего, будет поглощена светодиодом **On**, поэтому, если вас действительно заботит проблема снижения энергопотребления, вам стоит аккуратно выпаять его из платы.

Выключение электронных компонентов на плате

Контроллеры ATmega имеют широкие возможности управления электропитанием, настолько широкие, что способны отключать неиспользуемые электронные компоненты на плате, чтобы уменьшить потребляемый ток.

Более того, компоненты можно включать и выключать прямо из скетча. То есть можно, к примеру, включать АЦП непосредственно перед вызовом `analogRead` и затем выключать его.

Управление электропитанием осуществляется с помощью библиотеки avr/power.h, включающей пары функций включения/выключения. Например, вызовом функции `power_adc_disable` можно выключить АЦП, а вызовом `power_adc_enable` вновь включить.

Однако экономия электроэнергии за счет отключения компонентов будет получаться не очень большой. В ходе экспериментов с платой Mini Pro, питающейся напряжением 5 В и действующей на частоте 16 МГц, я установил, что, когда все компоненты включены, она потребляет ток 16,4 мА, а когда выключены — что-то около 14,9 мА, то есть снижение составило всего на 1,5 мА. Для измерений я использовал следующий скетч:

```
// sketch_05_02_powering_off

#include <avr/power.h>

void setup()
{
  pinMode(13, OUTPUT);
//  power_adc_disable();
  power_spi_disable();
//  power_twi_disable();
//  power_usart0_disable();
//  power_timer0_disable();
//  power_timer1_disable();
//  power_timer2_disable();
//  power_all_disable();
```

```
}

void loop()
{
}
```

Доступные функции перечислены в табл. 5.3. Каждая функция имеет пару с окончанием `enable` вместо `disable` в имени.

Таблица 5.3. Функции управления электропитанием для ATmega Arduino

Функция	Описание
power_adc_disable	Выключает аналоговые входы
power_spi_disable	Отключает интерфейс SPI
power_twi_disable	Отключает интерфейс TWI (I2C)
power_usart0_disable	Отключает УСАПП (UART, интерфейс последовательной связи через USB)
power_timer0_disable	Отключает таймер 0 (используется функциями millis и delay)
power_timer1_disable	Отключает таймер 1
power_timer2_disable	Отключает таймер 2
power_all_disable	Отключает все компоненты, перечисленные выше

Энергосберегающий режим

Самый действенный способ экономии электроэнергии — перевести плату Arduino в спящий режим на время, пока она не совершает полезной работы.

Narcoleptic

Питер Кнайт создал простую в использовании библиотеку с названием Narcoleptic, которую можно получить по адресу **https://code.google.com/p/narcoleptic/**.

Очевидно, что нет смысла переводить Arduino в энергосберегающий режим, не имея возможности вернуть ее в нормальный режим работы! Существует два способа возврата Arduino к нормальной работе. Один из них — использовать внешнее прерывание, а другой — установить таймер, который обеспечит выход в нормальный режим через определенный интервал времени. Библиотека опирается на использование таймера.

Библиотека Narcoleptic предоставляет альтернативную функцию `delay`, которая переводит Arduino в энергосберегающий режим на время, указанное в вызове `delay`. Поскольку в ходе задержки все равно ничего не происходит, этот метод действует блестяще.

Например, вернемся к старому доброму скетчу Blink. Следующий скетч включает светодиод на 1 с, затем выключает его на 10 с и повторяет эту последовательность до бесконечности:

```
// sketch_05_03_blink_standard

void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(10000);
}
```

Ниже показана версия того же скетча, использующая библиотеку Narcoleptic:

```
// sketch_05_04_narcoleptic_blink
#include <Narcoleptic.h>

void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  Narcoleptic.delay(1000);
  digitalWrite(13, LOW);
  Narcoleptic.delay(10000);
}
```

Единственное отличие этой версии в том, что она импортирует библиотеку Narcoleptic и использует ее версию функции `delay` вместо стандартной.

Запустив оба скетча на плате Mini Pro, питающейся напряжением 5 В и

действующей на частоте 16 МГц, я выяснил, что для первого скетча в момент, когда светодиод выключен, потребляемый ток составил 17,2 мА. Для версии с библиотекой Narcoleptic потребляемый ток уменьшился до 3,2 мА, из которых большую часть потребляет светодиод **On** (около 3 мА), то есть, если его выпаять, средний потребляемый ток должен упасть ниже 1 мА.

Микроконтроллер очень быстро переходит в энергосберегающий режим, поэтому, если в вашем проекте имеется кнопка, нажатие на которую вызывает некоторые действия, нет необходимости использовать внешнее прерывание, чтобы вывести микроконтроллер из энергосберегающего режима. Можно написать код (что, возможно, проще), который будет переводить плату Arduino в энергосберегающий режим и выводить ее обратно 10 раз в секунду, проверять нажатие кнопки, сравнивая цифровой вход со значением HIGH, и затем выполнять какие-то операции вместо возврата в энергосберегающий режим. Следующий скетч демонстрирует, как это можно реализовать:

```
// sketch_05_05_narcoleptic_input
#include <Narcoleptic.h>

const int ledPin = 13;
const int inputPin = 2;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(inputPin, INPUT_PULLUP);
}

void loop()
{
  if (digitalRead(inputPin) == LOW)
  {
    doSomething();
  }
  Narcoleptic.delay(100);
}

void doSomething()
{
  for (int i = 0; i < 20; i++)
  {
```

```
    digitalWrite(ledPin, HIGH);
    Narcoleptic.delay(200);
    digitalWrite(ledPin, LOW);
    Narcoleptic.delay(200);
  }
}
```

Во время выполнения этого скетча плата Mini Pro, питающаяся напряжением 5 В и действующая на частоте 16 МГц, потребляла мизерные 3,25 мА, ожидая, пока что-то произойдет. После замыкания контакта **2** на «землю» светодиод **L** мигнул 20 раз, но, так как для задержки между включением и выключением светодиода скетч использует все ту же версию `delay` из библиотеки Narcoleptic, потребляемый ток увеличился в среднем всего лишь до 4–5 мА.

Если изменить вызов `delay` внутри функции `loop`, чтобы выводить Arduino из энергосберегающего режима, скажем, 100 раз в секунду, потребляемый ток увеличится, потому что для перевода Arduino в энергосберегающий режим действительно требуется некоторое время. Однако задержка на 50 мс (20 раз в секунду) дает довольно хорошие результаты.

Вывод из энергосберегающего режима внешними прерываниями

Только что описанный подход можно с успехом использовать в разных ситуациях, однако если требуется получить более быстрый отклик на внешнее событие, можно реализовать вывод микроконтроллера из энергосберегающего режима с помощью внешнего прерывания.

Чтобы переделать предыдущий пример и использовать контакт **D2** как приемник внешних прерываний, требуется приложить дополнительные усилия, но результаты получаются немного лучше, так как отпадает необходимость периодически проверять состояние контакта. Код скетча получился сложным, поэтому сначала я покажу сам код, а потом расскажу, как он работает. Если вы пропустили главу 3 о прерываниях, то вам стоит прочитать ее перед изучением примера.

```
// sketch_05_06_sleep_external_wake
#include <avr/sleep.h>

const int ledPin = 13;
const int inputPin = 2;

volatile boolean flag;

void setup()
```

```
{
  pinMode(ledPin, OUTPUT);
  pinMode(inputPin, INPUT_PULLUP);
  goToSleep();
}

void loop()
{
  if (flag)
  {
    doSomething();
    flag = false;
    goToSleep();
  }
}

void setFlag()
{
  flag = true;
}

void goToSleep()
{
   set_sleep_mode(SLEEP_MODE_PWR_DOWN);
   sleep_enable();
   attachInterrupt(0, setFlag, LOW); // контакт D2
   sleep_mode(); // включить энергосберегающий режим
   // Теперь микроконтроллер простаивает, пока уровень напряжения
   // на контакте прерывания не упадет до LOW, затем...
   sleep_disable();
   detachInterrupt(0);
}

void doSomething()
{
  for (int i = 0; i < 20; i++)
  {
    digitalWrite(ledPin, HIGH);
    delay(200);
```

```
    digitalWrite(ledPin, LOW);
    delay(200);
  }
}
```

Первое, на что следует обратить внимание, — в примере используются несколько функций из библиотеки avr/sleep.h. Подобно библиотеке avr/power.h, использовавшейся в предыдущих примерах, эта библиотека не является частью ядра Arduino — она поддерживает семейство микроконтроллеров AVR. То есть она не будет работать в модели Arduino Due, но в то же время, если вы разрабатываете проект с низким энергопотреблением на основе Arduino, модель Due должна быть последней в списке для выбора.

После выбора контактов для использования я определяю оперативную (со спецификатором `volatile`) переменную, чтобы подпрограмма обработки прерываний могла взаимодействовать с остальным скетчем.

Функция `setup` выполняет настройку контактов и вызывает `goToSleep`. Эта функция устанавливает вид режима энергосбережения — в данном случае `SLEEP_MODE_PWR_DOWN`. В этом режиме энергопотребление снижается до минимума, поэтому есть смысл использовать его.

Далее вызывается `sleep_enable`. Этот вызов еще не переводит микроконтроллер в режим энергосбережения. Прежде чем сделать это, нужно настроить прерывание 0 (контакт **D2**), чтобы плату можно было вернуть в нормальный режим функционирования.

ПРИМЕЧАНИЕ

Обратите внимание на то, что выбран тип прерывания LOW. Это единственный тип прерывания, который можно использовать в данном примере. Типы RISING, FALLING и CHANGE не будут работать.

Вызов `sleep_mode()` после настройки прерывания фактически переводит микроконтроллер в энергосберегающий режим. Когда позднее произойдет возврат в нормальный режим работы, будет вызвана подпрограмма обработки прерываний и скетч продолжит выполнение со следующей строки в функции `goToSleep`. В этой строке сразу же выполняется вызов `disable_sleep`, и прерывание отключается, поэтому подпрограмма обработки прерываний не будет вызвана снова, пока скетч вновь не переведет микроконтроллер в энергосберегающий режим.

Когда падение напряжения на контакте **D2** вызовет прерывание, подпрограмма-обработчик (`setFlag`) просто установит флаг, который проверяется функцией `loop`. Не забывайте, что в подпрограммах обработки прерываний нельзя использовать

функцию `delay` и подобные ей. Поэтому функция `loop` должна проверить флаг и, если он установлен, вызвать ту же функцию `doSomething`, которая использовалась в примере с библиотекой Narcoleptic. После выполнения операции флаг сбрасывается, и Arduino вновь переводится в энергосберегающий режим.

По величине потребляемого тока этот скетч практически совпадает с примером на основе библиотеки Narcoleptic, с той лишь разницей, что во время, когда светодиод мигает, уровень потребляемого тока в данном примере выше из-за того, что используется обычная функция `delay`.

Использование цифровых выходов для управления питанием

Хотя в этой главе обсуждается проблема снижения энергопотребления программным способом, здесь нелишне будет дать полезный совет по уменьшению энергопотребления аппаратным способом.

На рис. 5.4 изображена схема датчика освещенности на основе фоторезистора (изменяет сопротивление в зависимости от освещенности) и постоянного сопротивления, подключенных к аналоговому входу Arduino, посредством которого измеряется степень освещенности.

Проблема данной реализации в том, что через постоянное сопротивление и фоторезистор течет постоянный ток напряжением 5 В. Если при полной освещенности она имеет сопротивление 500 Ом, то согласно закону

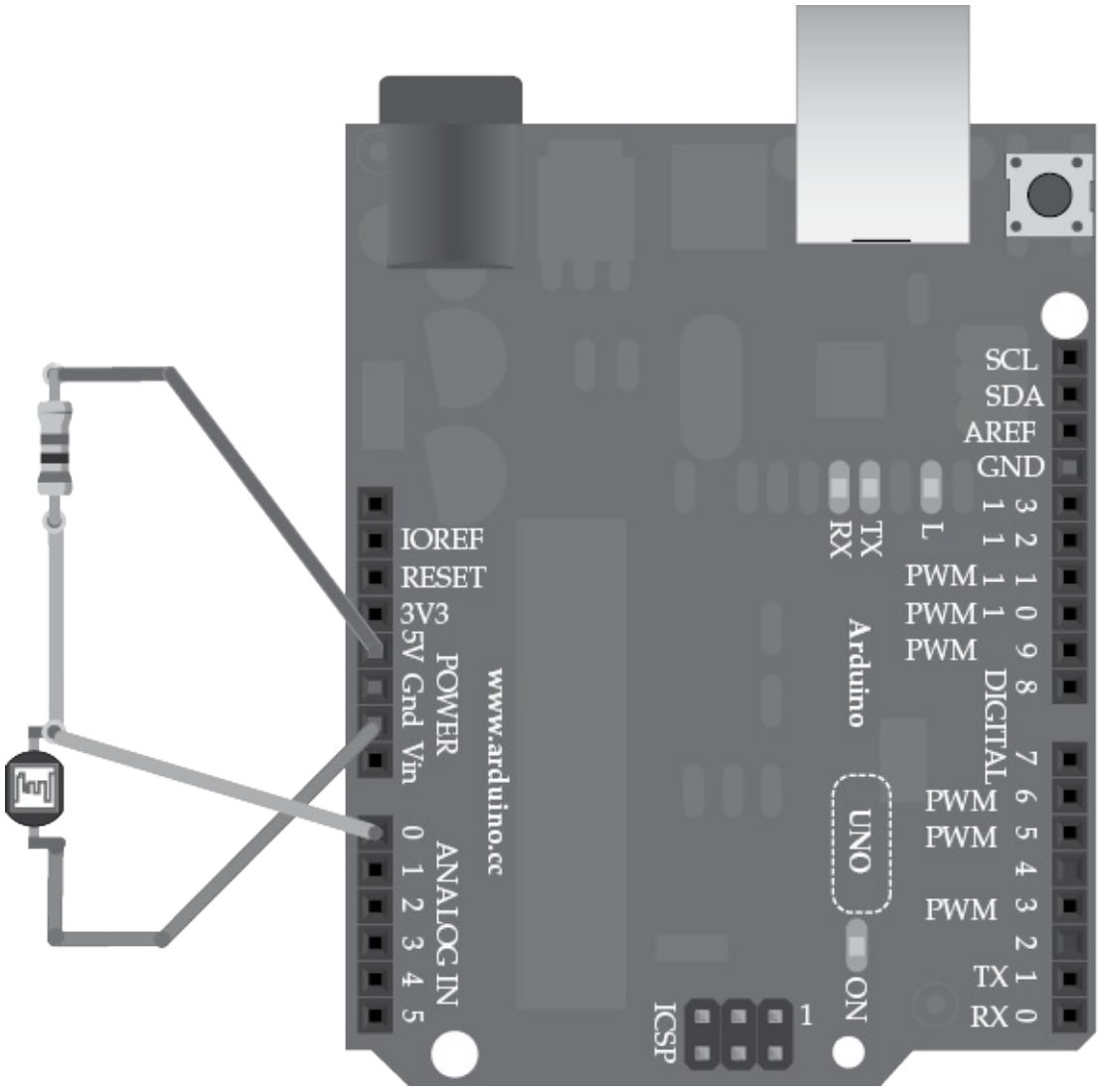


Рис. 5.4. Измерение освещенности с применением фоторезистора

Ома протекающий ток будет иметь значение $I = V/R$ = 5 В/(1000 Ом + + 500 Ом) = 3,3 мА.

Вместо источника постоянного напряжения 5 В на плате Arduino можно использовать цифровой выход (рис. 5.5) и подавать на него уровень напряжения `HIGH` только в момент чтения значения с аналогового входа, а затем устанавливать на нем уровень `LOW`. В этом случае ток 3,3 мА будет протекать только в течение очень короткого промежутка времени, когда выполняется чтение, благодаря чему можно снизить общий уровень энергопотребления.

Это решение иллюстрирует следующий скетч:

```
// sketch_05_07_light_sensing

const int inputPin = A0;
const int powerPin = 12;

void setup()
```

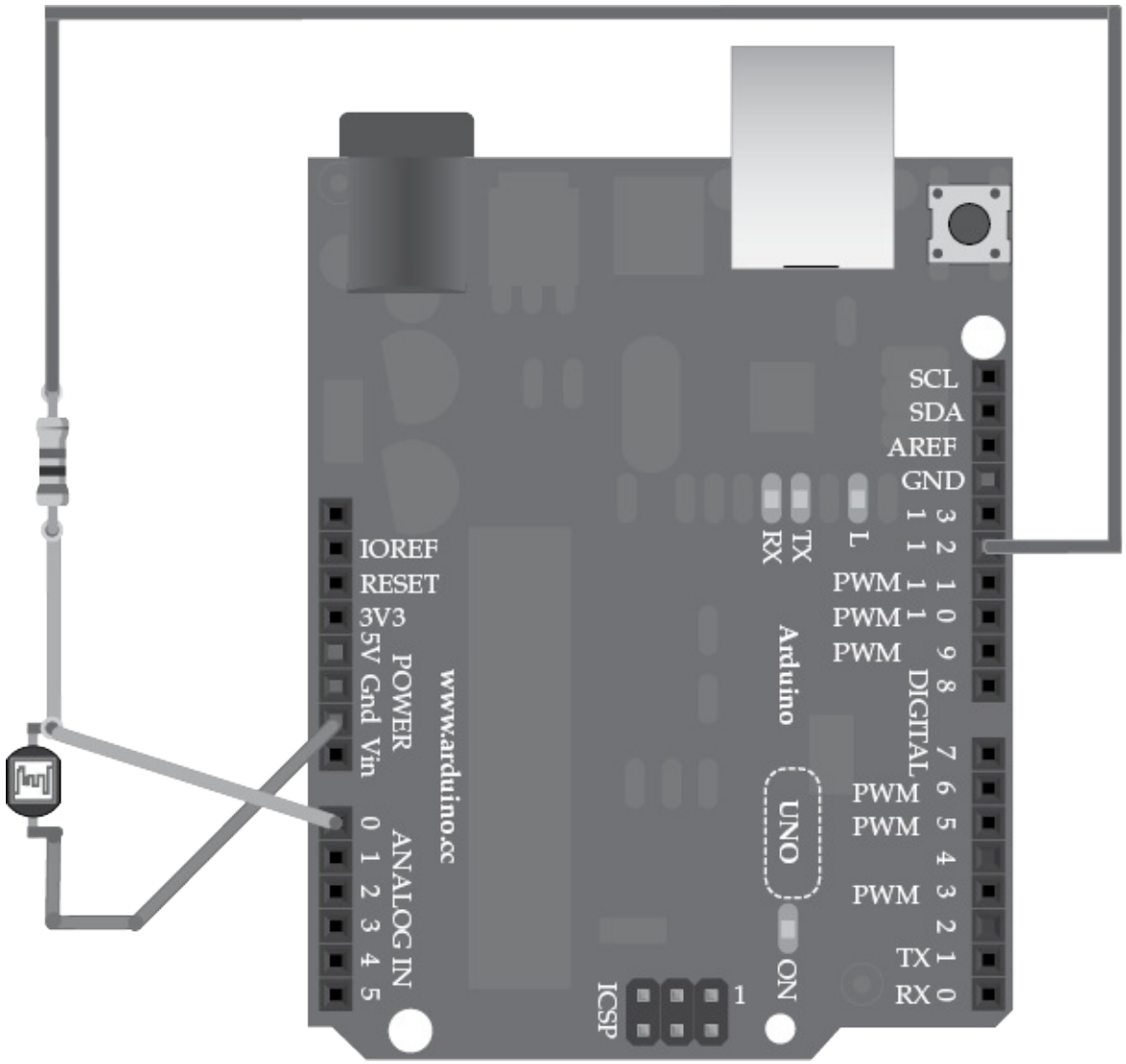


Рис. 5.5. Экономичная схема измерения освещенности

```
{
  pinMode(powerPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  Serial.println(takeReading());
  delay(500);
}

int takeReading()
{
  digitalWrite(powerPin, HIGH);
  delay(10); // фоторезистору требуется некоторое время
  int reading = analogRead(inputPin);
```

```
  digitalWrite(powerPin, LOW);
  return reading;
}
```

Этот подход можно использовать не только при измерении освещенности. Можно, например, с помощью цифрового выхода управлять полевым транзистором, включающим и выключающим мощные потребители электроэнергии в вашем проекте.

В заключение

Лучшие способы уменьшить потребление электроэнергии:

- переводить микроконтроллер в режим энергосбережения, когда не требуется выполнять никаких действий;
- использовать для питания Arduino пониженное напряжение;
- уменьшать тактовую частоту Arduino.

6. Память

Объем памяти в большинстве компьютеров исчисляется гигабайтами, но в Arduino Uno ее всего 2 Кбайт. То есть более чем в миллион раз меньше, чем в обычном компьютере. Однако ограниченный объем памяти удивительным образом способствует концентрации мысли в процессе программирования. Здесь нет места для расточительства, которым страдает большинство компьютеров.

Писать эффективный код, конечно, важно, но необязательно делать это за счет усложнения чтения и сопровождения. Даже при таких ограниченных ресурсах, как в Arduino, большинство скетчей оказываются далеки от использования всего объема оперативного запоминающего устройства (ОЗУ). Беспокоиться о нехватке памяти приходится, только когда создается действительно очень сложный скетч, использующий массу данных.

Память в Arduino

Сравнивать объем памяти в Arduino и в обычных компьютерах не совсем корректно, так как в них память ОЗУ используется для разных целей. На рис. 6.1 показано, как используется память в компьютере, когда запускается программа.

Когда компьютер запускает программу, он сначала копирует ее целиком с жесткого диска в ОЗУ, а затем запускает эту копию. Переменные в программе занимают дополнительный объем ОЗУ. Для сравнения на рис. 6.2 показано, как используется память в Arduino, когда запускается программа. Сама программа действует, находясь во флеш-памяти. Она не копируется в ОЗУ.

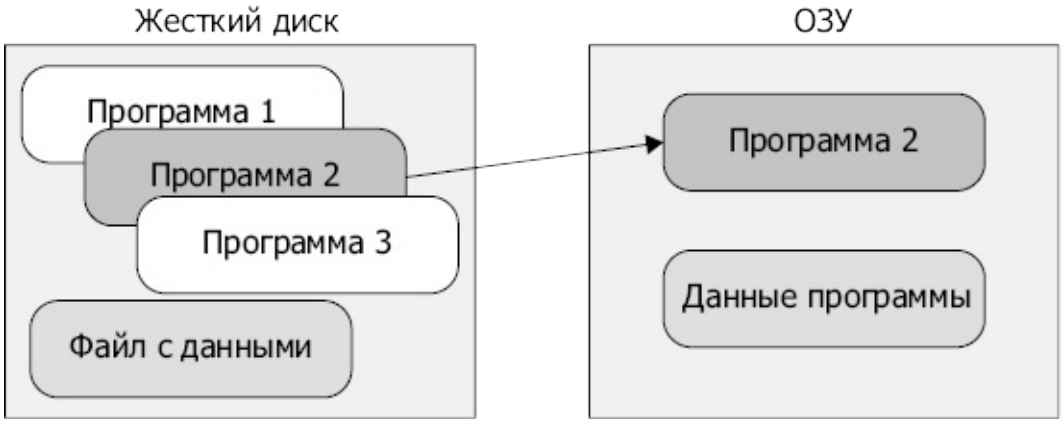


Рис. 6.1. Как используется память в компьютере

Рис. 6.2. Как используется память в Arduino

ОЗУ в Arduino используется только для хранения переменных и других данных, имеющих отношение к выполняющейся программе. ОЗУ является энергозависимой памятью, то есть после отключения питания оно очищается. Чтобы сохранить данные надолго, программа должна записать их в ЭСППЗУ. После этого скетч сможет считать данные в момент повторного запуска.

При приближении к границам возможностей Arduino придется позаботиться о рациональном использовании ОЗУ и, в меньшей степени, о размере программы внутри флеш-памяти. Так как в Arduino Uno имеется 32 Кбайт флеш-памяти, этот предел достигается нечасто.

Уменьшение используемого объема ОЗУ

Как вы уже видели, чтобы уменьшить используемый объем ОЗУ, следует уменьшить объем памяти, занимаемой переменными.

Используйте правильные структуры данных

Самым широко используемым типом данных в Arduino C, бесспорно, является тип `int`. Каждая переменная типа `int` занимает 2 байта, но часто такие переменные используются для представления чисел из намного более узкого диапазона, чем –32 768...+32 767, и нередко типа `byte` с его диапазоном 0...255 для них оказывается вполне достаточно. Большинство встроенных методов, принимающих аргументы типа `int`, с таким же успехом могут принимать однобайтовые аргументы.

Типичным примером могут служить переменные с номерами контактов. Они часто объявляются с типом `int`, как показано в следующем примере:

```
// sketch_06_01_int
int ledPins[] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

void setup()
{
  for (int i = 0; i < 12; i++)
  {
    pinMode(ledPins[i], OUTPUT);
    digitalWrite(ledPins[i], HIGH);
  }
}

void loop()
{
}
```

Массив типа `int` без всяких последствий можно преобразовать в массив байтов. В этом случае функции в программе будут выполняться с той же скоростью, зато массив будет занимать в два раза меньше памяти.

По-настоящему отличный способ экономии ОЗУ — объявление неизменяемых переменных константами. Для этого достаточно добавить слово `const` в начало объявления переменной. Зная, что значение никогда не изменится, компилятор сможет подставлять значение переменной в местах обращения к ней и тем самым экономить ОЗУ. Например, массив из предыдущего примера можно объявить так:

```
const byte ledPins[] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

Не злоупотребляйте рекурсией

Рекурсией называется вызов функцией самой себя. Рекурсия может быть мощным инструментом выражения и решения задач. В языках функционального программирования, таких как LISP и Scheme, рекурсия используется чуть ли не повсеместно.

Когда происходит вызов функции, в области памяти, называемой *стеком*, выделяется фрагмент. Представьте подпружиненный дозатор для леденцов, например Pez™, но позволяющий вталкивать леденцы и выталкивать их сверху (рис. 6.3). Под термином «вталкивать» понимается добавление чего-то на стек, а под термином «выталкивать» — извлечение со стека.

Каждый раз, когда вызывается функция, создается *кадр стека*. Кадр стека — это небольшой объем памяти, где сохраняются параметры и локальные переменные функции, а также адрес возврата, указывающий точку в программе, откуда должно

быть продолжено выполнение после завершения функции.

Первоначально стек пуст, но, когда скетч вызовет функцию (пусть это будет функция А), на стеке выделяется пространство под кадр. Если функция А вызовет другую функцию (функцию Б), на вершину стека будет добавлен еще один кадр и теперь в стеке будет храниться две записи. Когда функция Б завершится, ее кадр будет вытолкнут со стека. Затем, когда завершится функция А, ее кадр также будет вытолкнут со стека. Поскольку локальные переменные функции находятся в кадре стека, они не сохраняются между вызовами функции.

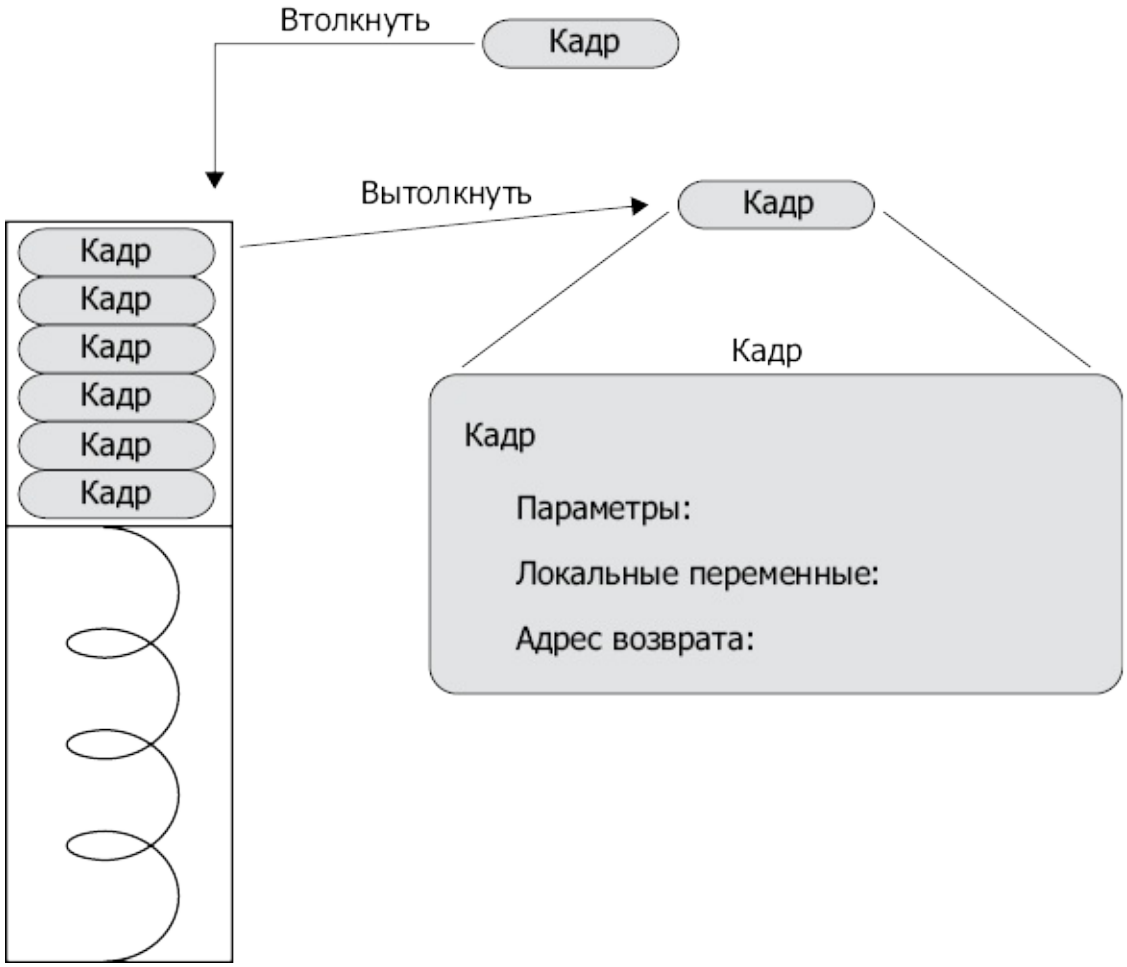


Рис. 6.3. Стек

Под стек используется некоторый объем ценной памяти, и большую часть времени на стеке находятся не более трех-четырех кадров. Исключение составляют ситуации, когда функции вызывают сами себя или в цикле вызывают друг друга. В таких случаях есть опасность, что программа исчерпает память для стека.

Например, математическая функция вычисления факториала находит произведение всех целых чисел, предшествующих указанному числу, включая его. Факториал числа 6 равен 6 x 5 x 4 x 3 x 2 x 1 = 720.

Рекурсивный алгоритм вычисления факториала определяется так.

- Если $n = 0$, факториал числа n равен 1.

• Иначе факториал числа *n* равен произведению *n* на факториал (*n* – 1).

Далее показана реализация этого алгоритма на языке Arduino C:

```
long factorial(long n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n* factorial(n – 1);
  }
}
```

Полную версию кода, который вычисляет факториалы чисел и выводит результаты, вы найдете в скетче sketch_06_02_factorial. Люди с математическим складом ума находят такую реализацию весьма искусной. Но обратите внимание на то, что глубина стека в вызове такой функции равна числу, факториал которого требуется найти. Совсем нетрудно догадаться, как реализовать нерекурсивную версию функции `factorial`:

```
long factorial(long n)
{
  long result = 1;
  while (n > 0)
  {
    result = result * n;
    n--;
  }
  return result;
}
```

С точки зрения удобочитаемости этот код, возможно, выглядит понятнее, а кроме того, он расходует меньше памяти и работает быстрее. Вообще старайтесь избегать рекурсии или хотя бы ограничивайтесь высокоэффективными рекурсивными алгоритмами, такими как Quicksort (**http://ru.wikipedia.org/wiki/Быстрая_сортировка**), который очень эффективно упорядочивает массив чисел.

Сохраняйте строковые константы во флеш-памяти

По умолчанию строковые константы, как в следующем примере, сохраняются в ОЗУ и во флеш-памяти — один экземпляр хранится в коде программы, а второй экземпляр создается в ОЗУ во время выполнения скетча:

```
Serial.println("Program Started");
```

Но если использовать код, как показано далее, строковая константа будет храниться только во флеш-памяти:

```
Serial.println(F("Program Started"));
```

В разделе «Использование флеш-памяти» далее в этой главе вы познакомитесь с другими способами использования флеш-памяти.

Типичные заблуждения

Многие заблуждаются, полагая, что использование более коротких имен переменных позволяет экономить память. В действительности это не так. Компилятор сам заботится об этом и не включает имена переменных в скомпилированный скетч. Другое распространенное заблуждение: комментарии увеличивают размер программы или объем потребляемой ею оперативной памяти. Это не так.

Некоторые также считают, что организация программного кода в виде множества маленьких функций увеличивает размер скомпилированного кода. Обычно этого не происходит, потому что компилятор достаточно сообразителен для того, чтобы в ходе оптимизации кода заменить вызовы функций их фактическими реализациями. Это обстоятельство помогает писать более удобочитаемый код.

Измерение объема свободной памяти

Узнать, какой объем ОЗУ занимает скетч во время выполнения, можно с помощью библиотеки MemoryFree, доступной по адресу **http://playground.arduino.cc/Code/AvailableMemory**.

Пользоваться этой библиотекой совсем не сложно: в ней имеется функция `freeMemory`, возвращающая число доступных байтов. Следующий скетч иллюстрирует ее использование:

```
#include <MemoryFree.h>

void setup()
{
  Serial.begin(115200);
```

```
}
void loop()
{
  Serial.print("freeMemory()=");
  Serial.println(freeMemory());
  delay(1000);
}
```

Эта библиотека может пригодиться при диагностике неожиданных проблем, которые, по вашему мнению, могут быть вызваны нехваткой памяти. Конечно же, использование библиотеки ведет к небольшому увеличению потребления памяти.

Уменьшение используемого объема флеш-памяти

По окончании процедуры компиляции скетча в нижней части окна Arduino IDE появится примерно такое сообщение:

```
Скетч использует 1344 байт (4%) памяти устройства. Всего доступно 32 256 байт.
```

Эта строка сообщает точный объем флеш-памяти в Arduino, который будет занят скетчем, благодаря чему вы всегда будете знать, насколько близко подошли к пределу в 32 Кбайт. Оказавшись близко к предельному значению, нужно позаботиться об оптимизации использования флеш-памяти. В этом вам помогут рассматриваемые далее рекомендации.

Используйте константы

Многие, стараясь дать имена контактам, определяют для этого переменные, как показано ниже:

```
int ledPin = 13;
```

Если вы не собираетесь изменять номер контакта с именем `ledPin` в процессе выполнения скетча, то вместо переменной можно использовать константу. Просто добавьте слово `const` в начало объявления:

```
const int ledPin = 13;
```

Это поможет сэкономить 2 байта ОЗУ плюс 2 байта флеш-памяти при каждом использовании константы. Для часто используемых переменных экономия может достигать нескольких десятков байтов.

Удалите ненужные трассировочные вызовы

В процессе отладки скетчей для Arduino принято вставлять в код команды `Serial.println`, помогающие увидеть значения переменных в разных точках программы и определить источники ошибок. Эти команды потребляют значительный объем флеш-памяти. Любое использование `Serial.println` требует включения в скетч примерно 500 байт библиотечного кода. Поэтому, убедившись в безупречной работе скетча, удалите или закомментируйте все такие команды.

Откажитесь от использования загрузчика

В главе 2 рассказывалось, как запрограммировать микроконтроллер непосредственно через контакты ICSP на плате Arduino с применением аппаратных программаторов. Такой подход поможет сэкономить пару килобайт, так как не требует установки загрузчика.

Статическое и динамическое размещение в памяти

Если вы, подобно автору книги, имеете опыт разработки крупномасштабных систем на таких языках, как Java или C#, вам наверняка приходилось создавать объекты во время выполнения и позволять сборщику мусора освобождать занимаемую ими память без вашего участия. Этот подход к программированию в принципе непригоден для программ, выполняющихся на микропроцессорах, которые имеют всего 2 Кбайт памяти. Ведь в Arduino просто нет никакого сборщика мусора, и, что более важно, в программах, которые пишутся для Arduino, выделение и освобождение памяти во время выполнения редко бывают необходимы.

Ниже приводится пример объявления статического массива, как это обычно делается в скетчах:

```
// sketch_06_04_static

int array[100];

void setup()
{
  array[0] = 1;
  array[50] = 2;
  Serial.begin(9600);
  Serial.println(array[50]);
}
```

```
void loop()
{
}
```

Объем памяти, занимаемой массивом, известен уже на этапе компиляции скетча, поэтому компилятор может зарезервировать для массива необходимый объем памяти. Второй пример, приведенный ниже, также создает массив того же размера, но выделяет память для него во время выполнения из пула доступной памяти. Обратите внимание на то, что версии Arduino IDE ниже 1.0.4 не поддерживают `malloc`.

```
// sketch_06_03_dynamic

int *array;

void setup()
{
  array = (int *)malloc(sizeof(int) * 100);
  array[0] = 1;
  array[50] = 2;
  Serial.begin(9600);
  Serial.println(array[50]);
}

void loop()
{
}
```

В начале скетча определяется переменная `int *array`. Символ * сообщает, что это указатель на целочисленное значение (или в данном случае массив целых чисел), а не простое значение. Объем памяти, занимаемой массивом, неизвестен, пока не будет выполнена следующая строка в функции `setup`:

```
  array = (int *)malloc(sizeof(int) * 100);
```

Команда `malloc` (*memory allocate — выделить память*) выделяет память в области ОЗУ, которую называют *кучей* (heap). В качестве аргумента ей передается объем памяти в байтах, который следует выделить. Так как массив хранит 100 значений типа `int`, требуется выполнить некоторые расчеты, чтобы определить размер массива в байтах. В действительности можно было бы просто передать функции `malloc` число `200` в аргументе, потому что известно, что каждое значение типа `int` занимает 2 байта памяти, но использование функции `sizeof` гарантирует получение правильного числа

в любом случае.

После выделения памяти массивом можно пользоваться точно так же, как если бы память для него была выделена статически. Динамическое распределение памяти позволяет отложить принятие решения о размере массива до фактического запуска скетча, и это единственное преимущество данного подхода.

Однако, используя прием динамического распределения памяти, легко оказаться в ситуации, когда память выделяется, но не освобождается, из-за чего скетч может быстро исчерпать имеющуюся память. Исчерпание памяти может вызвать зависание Arduino. Но если вся память выделяется статически, такого не происходит.

Обратите внимание на то, что даже мне, разработавшему не одну сотню проектов на Arduino, сложно найти вескую причину, оправдывающую прием динамического выделения памяти в Arduino.

Строки

Строки (текста) намного реже используются в скетчах для Arduino, чем в обычных программах. В обычных программах строки чаще всего применяются для взаимодействий с пользователями или базами данных, где текст является естественным средством передачи информации.

Многие программы для Arduino вообще не нуждаются в текстовом представлении данных или используют его только в командах `Serial.println` для нужд отладки.

В Arduino поддерживаются два основных метода использования строк: старый метод — массивы элементов типа `char` и новый метод с применением библиотеки String Object.

Массивы элементов типа char

Когда в скетче определяется строковая константа, такая как

```
char message[] = "Hello World";
```

создается статический массив элементов типа `char`, содержащий 12 символов. Именно 12, а не 11, по числу букв в строке «Hello World», потому что в конец добавляется заключительный нулевой символ (`\0`), отмечающий конец строки. Такое соглашение для строк символов, принятое в языке C, позволяет использовать массивы символов большего размера, чем предполагалось вначале (рис. 6.4). Каждая буква, цифра или другой символ имеет код, который называют значением ASCII.

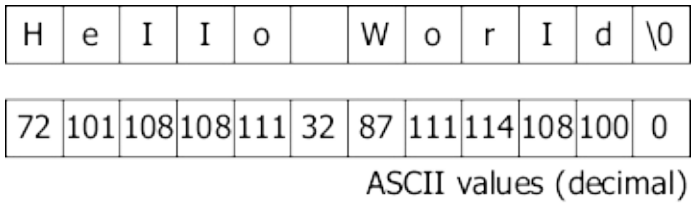


Рис. 6.4. Массив элементов типа char в стиле языка C с завершающим нулевым символом

Обратите внимание на то, что часто используется немного иной синтаксис записи строковых констант:

```
char *message = "Hello World";
```

Этот синтаксис действует подобным образом, но определяет `message` как указатель на символ (первый символ в массиве).

Форматированный вывод строк несколькими командами print

Часто строки необходимы, только чтобы вывести сообщение на жидкокристаллический дисплей или в качестве параметра `Serial.println`. Многие могут подумать, что в основном требуется только возможность объединения строк и преобразования чисел в строки. Например, рассмотрим конкретную проблему — как на жидкокристаллическом дисплее отобразить сообщение «Temp: 32 C». Вы могли бы предположить, что для этого нужно объединить число 32 со строкой `"Temp: "` и затем добавить в конец строку `" C"`. И действительно, программисты с опытом использования языка Java могли бы попытаться написать на C следующий код:

```
String text = "Temp: " + tempC + " C";
```

Увы, в C этот прием не работает. В данном случае сообщение можно вывести несколькими инструкциями `print`, как показано далее:

```
lcd.print("Temp: "); lcd.print(tempC); lcd.print(" C");
```

Этот подход устраняет необходимость закулисного копирования данных в процессе конкатенации (объединения) строк, как происходит в других современных языках.

Аналогичный подход с применением нескольких инструкций вывода можно использовать при работе с монитором последовательного порта и инструкциями `Serial.print`. В подобных случаях последней в строке обычно используется команда `println`, добавляющая в конец символ перевода строки.

Форматирование строк с помощью sprintf

Стандартная библиотека строковых функций для языка C (не путайте с библиотекой Arduino String Object, которая обсуждается в следующем разделе) включает очень

удобную функцию sprintf, выполняющую форматирование массивов символов. Она вставляет значения переменных в строку шаблона, как показано в следующем примере:

```
char line1[17];
int tempC = 30;
sprint(line1, "Temp: %d C", tempC);
```

Массив символов line1 — это строковый буфер, содержащий форматированный текст. Как указано в примере, он имеет емкость 17 символов, включая дополнительный нулевой символ в конце. Имя line1 я выбрал потому, что собираюсь показать, как сформировать содержимое верхней строки для жидкокристаллического дисплея с двумя строками по 16 символов в каждой.

В первом параметре команде sprintf передается массив символов, в который должен быть записан результат. Следующий аргумент — строка формата, содержащая смесь простого текста, такого как Temp:, и команд форматирования, например %d. В данном случае %d означает «десятичное целое со знаком». Остальные параметры будут подставлены в строку формата в порядке их следования на место команд форматирования.

Чтобы во вторую строку на жидкокристаллическом дисплее вывести время, его можно сформировать из отдельных значений часов, минут и секунд, как показано далее:

```
char line2[17];
int h = 12;
int m = 30;
int s = 5;
sprintf(line2, "Time: %2d:%02d:%02d", h, m, s);
```

Если попробовать вывести строку line2 в монитор последовательного порта или на экран жидкокристаллического дисплея, вы увидите текст

```
Time: 12:30:05
```

Команда sprintf не только подставила числа в нужные места, но и добавила ведущий ноль перед цифрой 5. В примере между символами : находятся команды форматирования трех компонентов времени. Часам соответствует команда %2d, которая выводит двузначное десятичное число. Команды форматирования для минут и секунд немного отличаются (%02d). Эти команды также выводят двузначные десятичные числа, но добавляют ведущий ноль, если это необходимо.

Однако имейте в виду, что этот прием предназначен для значений типа int. К сожалению, разработчики Arduino не реализовали в стандартной библиотеке C

поддержку других типов, таких как `float`.

Определение длины строки

Так как строки, хранящиеся в массивах символов, часто оказываются короче самих массивов, в библиотеке предусмотрена удобная функция с именем `strlen`. Эта функция подсчитывает число символов в массиве, предшествующих нулевому символу, отмечающему конец строки.

Функция принимает массив символов в своем единственном параметре и возвращает размер строки (исключая пустой символ), хранящейся в нем, например, команда

```
strlen("abc");
```

вернет число 3.

Библиотека Arduino String Object

В Arduino IDE, начиная с версии 019, вышедшей несколько лет тому назад, включается библиотека String, более понятная и дружественная разработчикам, использующим Java, Ruby, Python и другие языки, где конкатенацию строк допускается выполнять простым оператором +. Эта библиотека также предлагает массу вспомогательных функций для работы со строками.

Конечно, данная библиотека добавляет к скетчу несколько килобайт кода. Кроме того, она использует механизм динамического распределения памяти со всеми сопутствующими проблемами, такими как исчерпание памяти. Поэтому подумайте хорошенько, прежде чем принять решение о ее использовании. Многие пользователи Arduino предпочитают применять обычные массивы символов.

Эта библиотека удивительно проста в использовании, и, если вам приходилось работать со строками в Java, благодаря библиотеке Arduino String Object вы будете чувствовать себя как дома.

Создание строк

Создать строку можно из массива элементов типа `char`, а также из значения типа `int` или `float`, как показано в следующем примере:

```
String message = "Temp: ";
String temp = String(123);
```

Конкатенация строк

Строки типа `String` можно объединять друг с другом и с данными других типов с помощью оператора +. Попробуйте добавить следующий код в функцию `setup` пустого скетча:

```
Serial.begin(9600);
String message = "Temp: ";
String temp = String(123);
Serial.println(message + temp + " C");
```

Обратите внимание на то, что последнее значение, добавляемое в строку, в действительности является массивом символов. Если первый элемент в последовательности значений между операторами + является строкой, остальные элементы автоматически будут преобразованы в строки перед объединением.

Другие строковые функции

В табл. 6.1 перечислены еще несколько удобных функций из библиотеки String. Полный список доступных функций можно найти по адресу **http://arduino.cc/en/Reference/StringObject**.

Таблица 6.1. Некоторые полезные функции в библиотеке String

Функция	Пример	Описание
[]	char ch = String("abc")[0]	Переменная ch получит значение "a"
trim	String s = " abc "; s.trim();	Удалит пробелы с обеих сторон от группы символов abc. Переменная s получит значение "abc"
toInt	String s = "123"; int x = s.toInt();	Преобразует строковое представление числа в значение типа int или long
substring	String s = "abcdefg"; String s2 = s.substring(1, 3);	Возвращает фрагмент исходной строки. Переменная s2 получит значение "bc". В параметрах передаются: индекс первого символа фрагмента и индекс символа, следующего за последним символом фрагмента
replace	String s = "abcdefg"; s.replace("de", "DE");	Заменит все вхождения "de" в строке на "DE". Переменная s2 получит значение "abcDEfg"

Использование ЭСППЗУ

Содержимое всех переменных, используемых в скетче Arduino, теряется при выключении питания или выполнении сброса. Чтобы сохранить значения, их нужно записать байт за байтом в память ЭСППЗУ. В Arduino Uno имеется 1 Кбайт памяти ЭСППЗУ.

ПРИМЕЧАНИЕ

Это не относится к плате Arduino Due, не имеющей ЭСППЗУ. В этой модели данные следует сохранять на карту microSD.

Для чтения и записи данных в ЭСППЗУ требуется использовать библиотеку, входящую в состав Arduino IDE. Следующий пример демонстрирует, как записать единственный байт в ЭСППЗУ, в данном случае операция выполняется в функции `setup`:

```
#include <EEPROM.h>
void setup()
{
  byte valueToSave = 123
  EEPROM.write(0, valueToSave);
}
```

В первом аргументе функции `write` передается адрес в ЭСППЗУ, куда должен быть записан байт данных, а во втором — значение для записи в этот адрес.

Для чтения данных из ЭСППЗУ используется команда `read`. Чтобы прочитать единственный байт, достаточно выполнить следующую команду:

```
EEPROM.read(0);
```

где `0` — это адрес в ЭСППЗУ.

Пример использования ЭСППЗУ

Следующий пример демонстрирует типичный сценарий записи значения в процессе нормального выполнения программы и его чтения в момент запуска. Приложение реализует кодовый замок двери и дает возможность вводить и изменять шифр с помощью монитора последовательного порта. Шифр хранится в ЭСППЗУ, поэтому его можно менять. Если бы шифр должен был сбрасываться при каждом запуске Arduino, не было бы смысла давать пользователю возможность изменять его.

В дискуссии, приведенной далее, будут обсуждаться отдельные фрагменты скетча. Желающие увидеть полный код скетча могут открыть скетч sketch_06_06_EEPROM_example в Arduino IDE, доступный в пакете примеров для этой

книги на сайте www.simonmonk.org. Опробуйте этот скетч у себя, чтобы получить более полное представление о его работе. Он не требует подключения дополнительного аппаратного обеспечения к Arduino.

Функция `setup` содержит вызов функции `initializeCode`.

```
void initializeCode()
{
  byte codeSetMarker = EEPROM.read(0);
  if (codeSetMarker == codeSetMarkerValue)
  {
    code = readSecretCodeFromEEPROM();
  }
  else
  {
    code = defaultCode;
  }
}
```

Задача этой функции — записать значение в переменную `code` (шифр). Это значение обычно читается из ЭСППЗУ, но при этом возникает несколько сложностей.

Содержимое ЭСППЗУ может быть не очищено в момент выгрузки нового скетча; значение, однажды записанное в ЭСППЗУ, может измениться только в результате записи нового значения поверх старого. То есть при первом запуске скетча нет никакой возможности узнать, не было ли значение оставлено в ЭСППЗУ предыдущим скетчем. В результате можно оказаться перед закрытой дверью, не зная, какой шифр хранится в ЭСППЗУ.

Для решения этой проблемы можно написать отдельный скетч, устанавливающий шифр по умолчанию. Этот скетч потребовалось бы установить в плату Arduino перед основным скетчем.

Второй, менее надежный, но более удобный способ — использовать специальный признак, который записывается в ЭСППЗУ и указывает, что шифр действительно был записан. Недостатком этого решения является малая вероятность того, что в ячейке ЭСППЗУ, где должен храниться признак, уже будет записано его значение. Из-за этого обстоятельства данное решение неприемлемо для коммерческих продуктов, но в данном случае мы можем так рискнуть.

Функция `initializeCode` читает первый байт из ЭСППЗУ, и, если он равен переменной `codeMarkerValue`, которой где-то в другом месте присваивается значение 123, она считает, что ЭСППЗУ содержит установленный пользователем шифр, и вызывает функцию `readSecretCodeFromEEPROM`:

```
int readSecretCodeFromEEPROM()
{
  byte high = EEPROM.read(1);
  byte low = EEPROM.read(2);
  return (high << 8) + low;
}
```

Эта функция читает двухбайтный шифр типа `int` из байтов с адресами 1 и 2 в ЭСППЗУ (рис. 6.5).

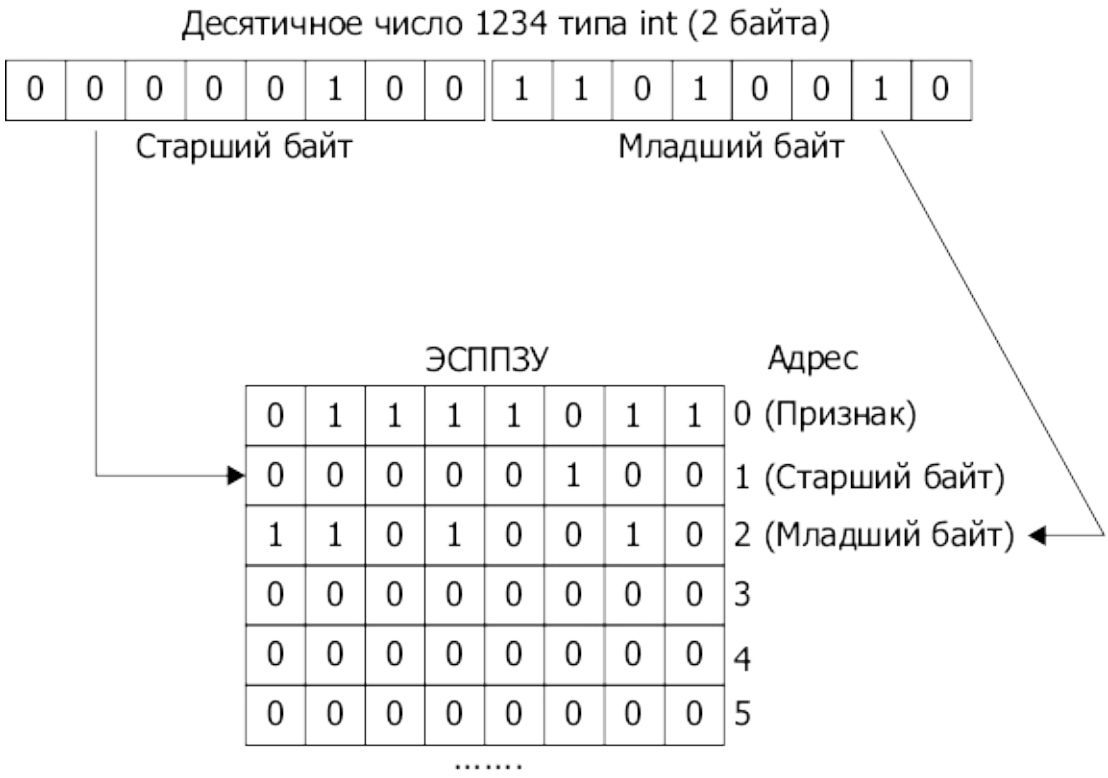


Рис. 6.5. Хранение значения типа int в ЭСППЗУ

Чтобы из двух отдельных байтов получить одно значение `int`, нужно сдвинуть старший байт влево на 8 двоичных разрядов (`high << 8`) и затем прибавить младший байт.

Чтение хранимого кода из ЭСППЗУ выполняется только в случае сброса платы Arduino. Но запись шифра в ЭСППЗУ должна выполняться при каждом его изменении, чтобы после выключения или сброса Arduino шифр сохранился в ЭСППЗУ и мог быть прочитан в момент запуска скетча.

За запись отвечает функция `saveSecretCodeToEEPROM`:

```
void saveSecretCodeToEEPROM()
{
  EEPROM.write(0, codeSetMarkerValue);
  EEPROM.write(1, highByte(code));
  EEPROM.write(2, lowByte(code));
```

```
}
```

Она записывает признак в ячейку ЭСППЗУ с адресом 0, указывающим, что в ЭСППЗУ хранится действительный шифр, и затем записывает два байта шифра. Для получения старшего и младшего байтов шифра типа `int` используются вспомогательные функции `highByte` и `lowByte` из стандартной библиотеки Arduino.

Использование библиотеки avr/eeprom.h

Библиотека EEPROM позволяет писать и читать данные только по одному байту. В предыдущем разделе мы обошли это ограничение, разбивая значение `int` на два байта перед сохранением и объединяя два байта в значение `int` после чтения. В качестве альтернативы, однако, можно использовать библиотеку EEPROM, предоставляемую компанией AVR, производящей микроконтроллеры. Она обладает более широкими возможностями, включая чтение и запись целых слов (16 бит) и даже блоков памяти произвольного размера.

Следующий скетч использует эту библиотеку для сохранения и чтения значения `int` непосредственно, увеличивая его при каждом перезапуске Arduino:

```
// sketch_06_07_avr_eeprom_int

#include <avr/eeprom.h>

void setup()
{
  int i = eeprom_read_word((uint16_t*)10);
  i++;
  eeprom_write_word((uint16_t*)10, i);
  Serial.begin(9600);
  Serial.println(i);
}

void loop()
{
}
```

Аргумент в вызове `eeprom_read_word (10)` и первый аргумент в вызове `eeprom_write_word` — это начальный адрес слова. Обратите внимание на то, что слово состоит из двух байтов, поэтому, если понадобится записать еще одно значение `int`, нужно будет указать адрес 12, а не 11. Конструкция `(uint16_t*)` перед `10` необходима, чтобы привести адрес (или индекс) к типу, ожидаемому библиотечной

функцией.

Еще одна полезная пара функций в этой библиотеке — eeprom_read_block и eeprom_write_block. Эти функции позволяют сохранять и извлекать произвольные структуры данных (допустимого размера).

Например, далее приводится скетч, записывающий строку символов в ЭСППЗУ, начиная с адреса 100:

```
// sketch_06_07_avr_eeprom_string

#include <avr/eeprom.h>

void setup()
{
  char message[] = "I am written in EEPROM";
  eeprom_write_block(message, (void *)100,
         strlen(message) + 1);
}

void loop()
{
}
```

В первом аргументе функции eeprom_write_block передается указатель на массив символов, подлежащий записи, во втором — адрес первой ячейки в ЭСППЗУ (100). В последнем аргументе передается число байтов, которые требуется записать. Здесь это число вычисляется как длина строки плюс один байт для завершающего нулевого символа.

Ниже демонстрируется скетч, который читает строку из ЭСППЗУ и выводит ее в монитор последовательного порта вместе с числом, обозначающим длину строки:

```
// sketch_06_07_avr_eeprom_string_read
#include <avr/eeprom.h>
void setup()
{
char message[50]; // буфер достаточно большого размера
eeprom_read_block(&message, (void *)100, 50);
Serial.begin(9600);
Serial.println(message);
Serial.println(strlen(message));
}
```

```
void loop()
{
}
```

Для чтения строки создается массив емкостью 50 символов. Затем вызывается функция `eeprom_read_block`, которая читает 50 символов в `message`. Знак & перед `message` указывает, что функции передается адрес массива `message` в ОЗУ.

Так как текст завершается нулевым символом, в монитор последовательного порта выводится только ожидаемый текст, а не все 50 символов.

Ограничения ЭСППЗУ

Операции чтения/записи с памятью ЭСППЗУ выполняются очень медленно — около 3 мс. Кроме того, надежность хранения гарантируется только для 100 000 циклов записи, после чего появляется вероятность искажения записанных данных. По этой причине старайтесь не выполнять запись в цикле.

Использование флеш-памяти

Объем флеш-памяти в Arduino намного больше, чем объем любой другой памяти. В Arduino Uno, например, объем флеш-памяти составляет 32 Кбайт против 2 Кбайт ОЗУ. Это делает флеш-память привлекательным местом для хранения данных, особенно если учесть, что она сохраняет данные после выключения питания.

Однако есть несколько препятствий, мешающих использованию флеш-памяти для хранения данных.

- Флеш-память в Arduino гарантирует сохранность данных только для 100 000 циклов записи, после чего она становится бесполезной.
- Флеш-память хранит программу, поэтому, если в расчеты вкрадется ошибка и часть программы окажется затертой данными, в скетче могут произойти невероятные события.
- Флеш-память содержит также загрузчик, уничтожение или искажение которого может превратить плату Arduino в «кирпич», после чего восстановить ее можно будет только с помощью аппаратного программатора (как описывалось в главе 2).
- Записывать данные во флеш-память можно только блоками по 64 байта.

Несмотря на все сказанное, в целом довольно безопасно использовать флеш-память для хранения постоянных данных, не изменяющихся в процессе выполнения скетча.

Для платы Arduino Due была создана сторонняя библиотека, позволяющая выполнять операции чтения/записи с флеш-памятью, чтобы компенсировать отсутствие ЭСППЗУ в этой модели. Более полную информацию об этом проекте можно получить по адресу **http://pansenti.wordpress.com/2013/04/19/simple-flash-library-for-arduino-due/**.

Самый простой способ создать строковую константу, хранящуюся во флеш-памяти, — использовать функцию `F`, упоминавшуюся в одном из предыдущих разделов. Напомню ее синтаксис:

```
Serial.println(F("Program Started"));
```

Этот прием работает только при использовании строковых констант непосредственно в вызове функции вывода. Нельзя, например, присвоить результат указателю на тип `char`.

Более гибкий, но более сложный способ заключается в использовании директивы `PROGMEM` (Program Memory — память программы) для сохранения любых структур данных. Однако данные должны быть постоянными — они не могут изменяться в процессе выполнения сценария.

Следующий пример иллюстрирует, как можно определить массив целых чисел (`int`), хранящийся во флеш-памяти:

```
// sketch_06_10_PROGMEM_array

#include <avr/pgmspace.h>

PROGMEM int value[] = {10, 20, 25, 25, 20, 10};

void setup()
{
  Serial.begin(9600);
  for (int i = 0; i < 6; i++)
  {
    int x = pgm_read_word(&value[i]);
    Serial.println(x);
  }
}

void loop()
{
}
```

Директива PROGMEM перед объявлением массива гарантирует, что он будет храниться только во флеш-памяти. Но прочитать значение элемента из такого массива можно только с помощью функции pgm_read_word из библиотеки avr/pgmspace:

```
int x = pgm_read_word(&value[i]);
```

Символ & перед именем массива в параметре указывает, что функции передается адрес данного элемента массива во флеш-памяти, а не его значение.

Функция pgm_read_word читает из флеш-памяти слово (2 байта). В библиотеке имеются также функции pgm_read_byte и pgm_read_dword, возвращающие 1 и 4 байта соответственно.

Использование SD-карты

Несмотря на то что сами платы Arduino не имеют слота для SD-карт, некоторые платы расширения, включая Ethernet и MP3 (рис. 6.6), имеют слоты для карт SD или microSD.

Для подключения карт SD используется интерфейс SPI (обсуждается в главе 9). К счастью, чтобы использовать карту SD с платой Arduino, не требуется писать низкоуровневый код для взаимодействия с интерфейсом SPI, так как в состав Arduino IDE входит специализированная библиотека с простым названием SD.

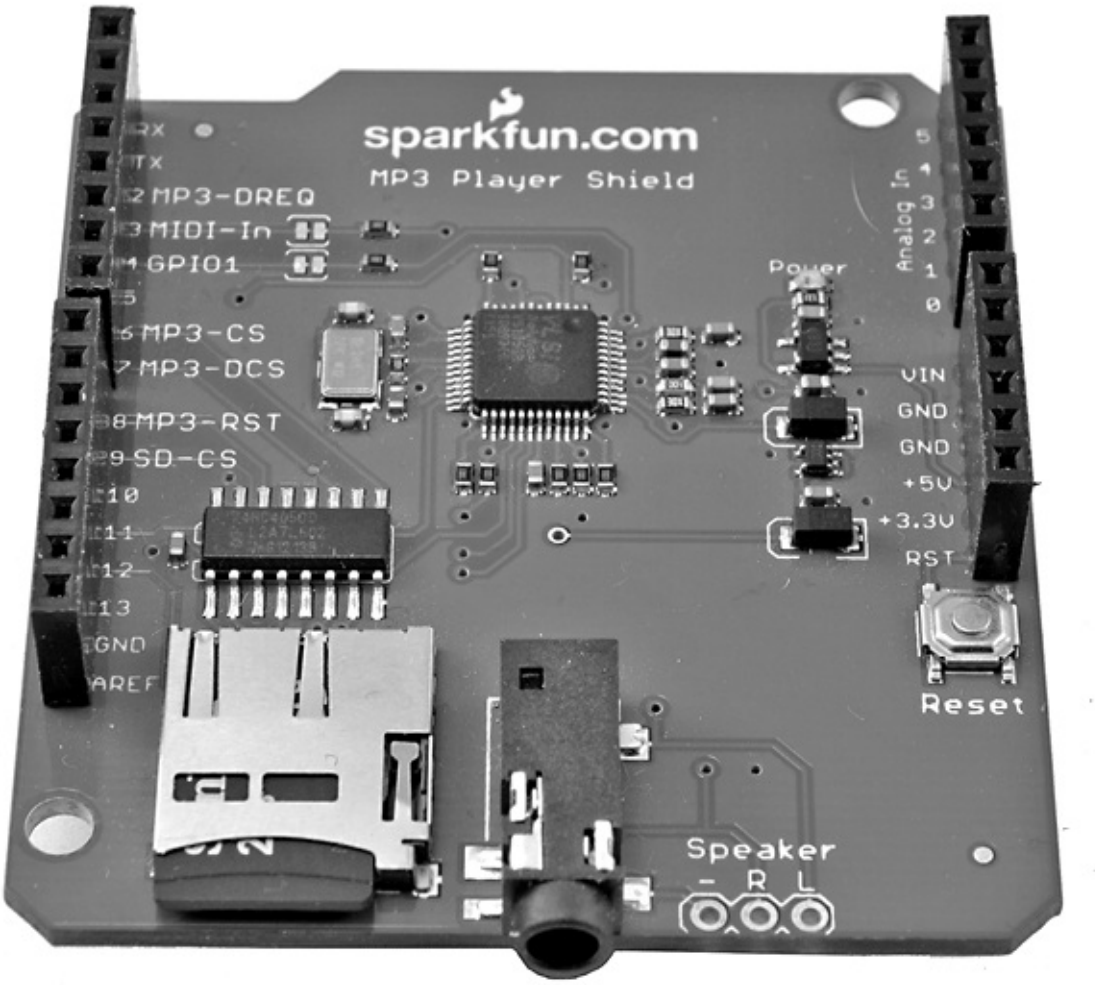


Рис. 6.6. Плата расширения MP3 со слотом для карты microSD

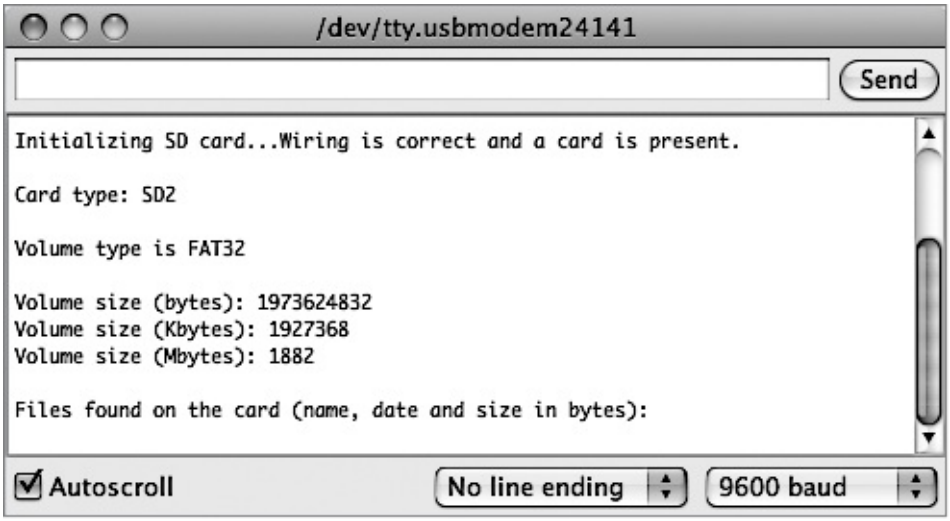


Рис. 6.7. Результат работы примера Cardinfo

Эта библиотека включает множество примеров скетчей, выполняющих разные операции с картой SD, включая поиск информации о карте SD и ее вывод в монитор последовательного порта (рис. 6.7).

Запись на карту SD выполняется очень просто, как показано в следующем фрагменте кода:

```
File dataFile = SD.open("datalog.txt", FILE_WRITE);
// Если файл существует, записать в него
if(dataFile) {
  dataFile.println(dataString);
  dataFile.close();
  // вывести также в монитор последовательного порта
  Serial.println(dataString);
}
```

В заключение

В этой главе вы познакомились со всеми аспектами использования памяти и хранения данных в Arduino. В следующих главах мы займемся исследованием приемов программирования различных последовательных интерфейсов в Arduino, начав с шины I2C.

7. Интерфейс I2C

Интерфейсная шина I2C (произносится «и квадрат си») — стандартный способ подключения периферийных устройств к микроконтроллерам. Иногда интерфейс I2C называют двухпроводным интерфейсом (Two Wire Interface, TWI). Все платы Arduino имеют хотя бы один интерфейс I2C, к которому можно подключать широкий диапазон периферийных устройств. Некоторые из таких устройств представлены на рис. 7.1.

Все три устройства в верхнем ряду на рис. 7.1 являются модулями отображения информации, выпускаемыми компанией Adafruit. В нижнем ряду слева находится модуль УКВ-приемника TEA5767. Эти модули можно приобрести на сайте eBay или где-то в другом месте за несколько долларов. Приобретая модуль TEA5767, вы получаете полноценный приемник УКВ, который можно настроить на определенную частоту командами через интерфейс I2C. В центре находится модуль часов реального времени (Real-Time Clock, RTC), включающий микросхему обслуживания шины I2C и кварцевый резонатор, обеспечивающий высокую точность измерения времени. Установив текущие время и дату через интерфейс I2C, вы сможете в любой момент прочитать текущие время и дату через тот же интерфейс I2C. Этот модуль включает также литиевую батарейку с длительным сроком службы, обеспечивающую работу модуля даже в отсутствие электропитания от внешнего источника. Наконец, справа находится 16-канальный ШИМ/сервопривод, добавляющий к вашей плате Arduino 16 дополнительных аналоговых выходов.

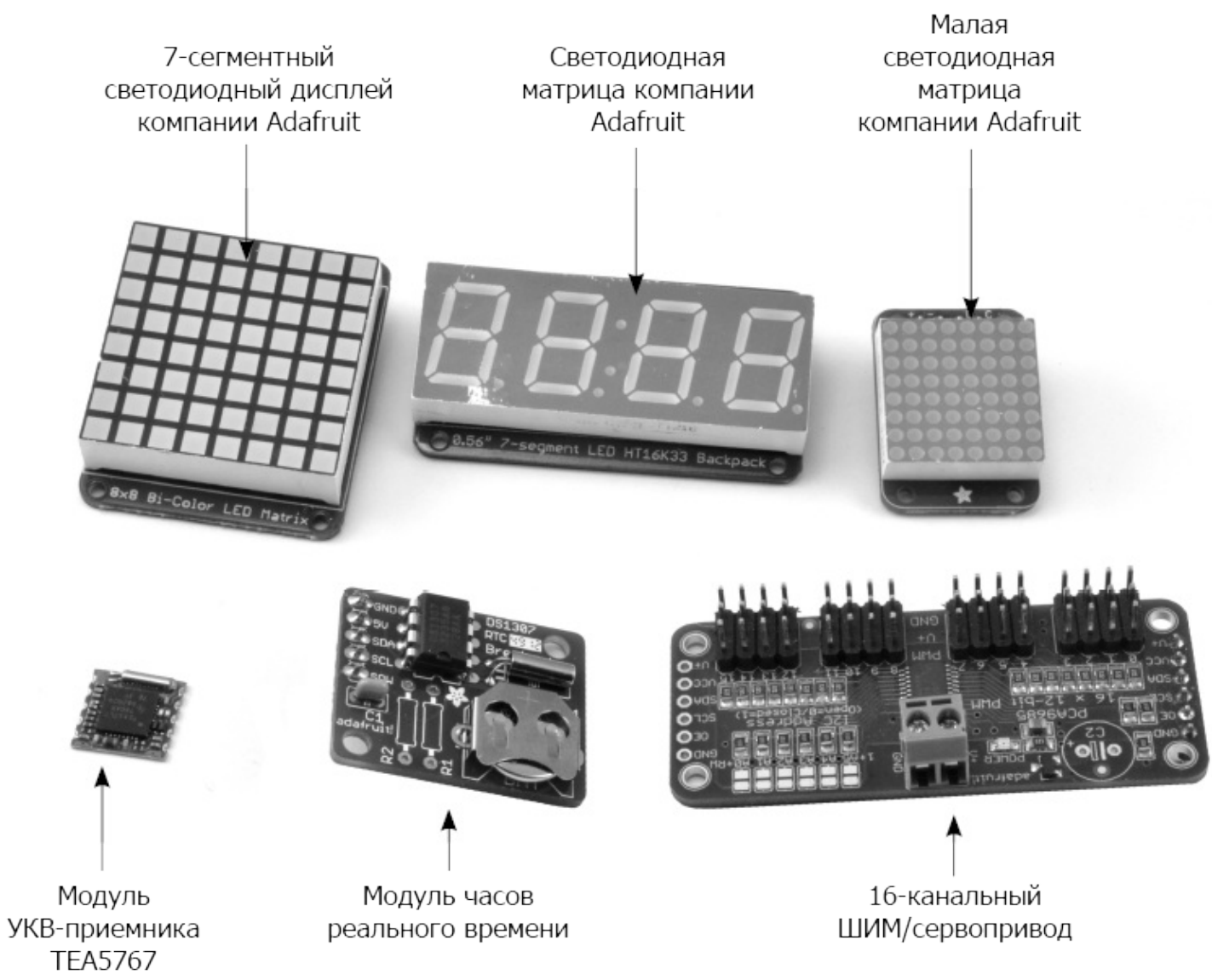


Рис. 7.1. Устройства с интерфейсом I2C

Стандарт I2C определяется как стандарт шины, потому что допускает подключение множества устройств друг к другу. Например, если вы уже подключили дисплей к микроконтроллеру, к той же паре контактов на «ведущем» устройстве можно подключить целое множество «ведомых» устройств. Плата Arduino выступает в роли «ведущего» устройства, а все «ведомые» устройства имеют уникальные адреса, идентифицирующие устройства на шине.

На рис. 7.2 изображена возможная схема подключения к плате Arduino двух компонентов I2C: часов реального времени и модуля дисплея.

Через интерфейс I2C можно также соединить две платы Arduino и организовать обмен данными между ними. В этом случае одна из плат должна быть настроена как ведущее устройство, а другая — как ведомое.

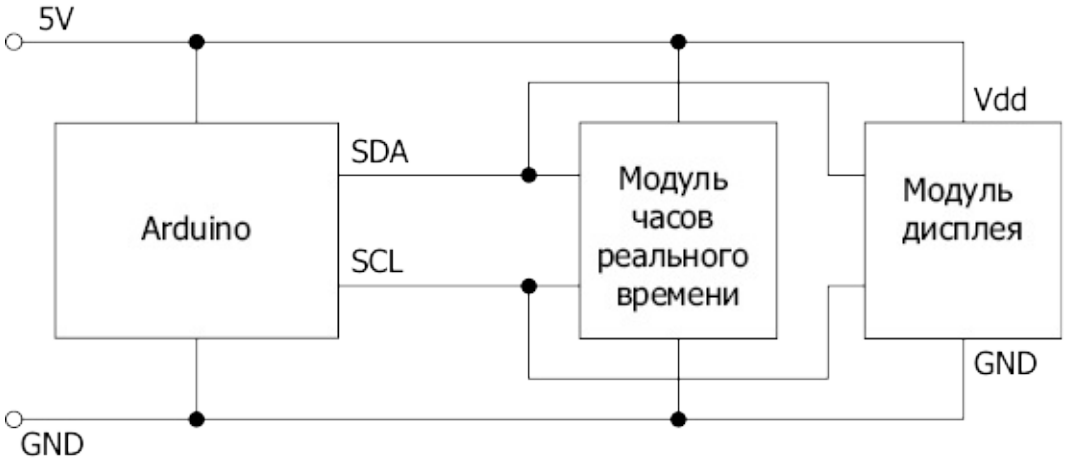


Рис. 7.2. Arduino управляет двумя устройствами I2C

Аппаратная часть I2C

Электрически линии соединения интерфейса I2C могут действовать подобно цифровым выходам или входам (их также называют выводами *с тремя состояниями*). В третьем состоянии линии соединения не находятся ни в одном из состояний, `HIGH` или `LOW`, а имеют плавающий уровень напряжения. Кроме того, выходы являются логическими элементами с *открытым коллектором*, то есть они требуют использования подтягивающего сопротивления. Эти сопротивления должны иметь номинал 4,7 кОм, и только одна пара контактов на всей шине I2C должна подключаться через подтягивающее сопротивление к шине питания 3,3 В или 5 В в зависимости от уровня напряжения, на котором действует шина. Если какое-то устройство на шине имеет другое напряжение питания, для его подключения необходимо использовать преобразователь уровня напряжения. Для шины I2C можно использовать модули двунаправленного преобразования, такие как BSS138, выпускаемые компанией Adafruit (**www.adafruit.com/products/757**).

На разных моделях Arduino интерфейс I2C подключается к разным контактам. Например, в модели Uno используются контакты **A4** и **A5** — линии SDA и SCL соответственно, а в модели Leonardo используются контакты **D2** и **D3**. (Подробнее о линиях SDA и SCL рассказывается в следующем разделе.) На обеих моделях линии SDA и SCL выводятся также на колодку, находящуюся рядом с контактом **AREF** (рис. 7.3).

В табл. 7.1 перечисляются наиболее распространенные модели платы Arduino и контакты, соответствующие интерфейсу I2C.

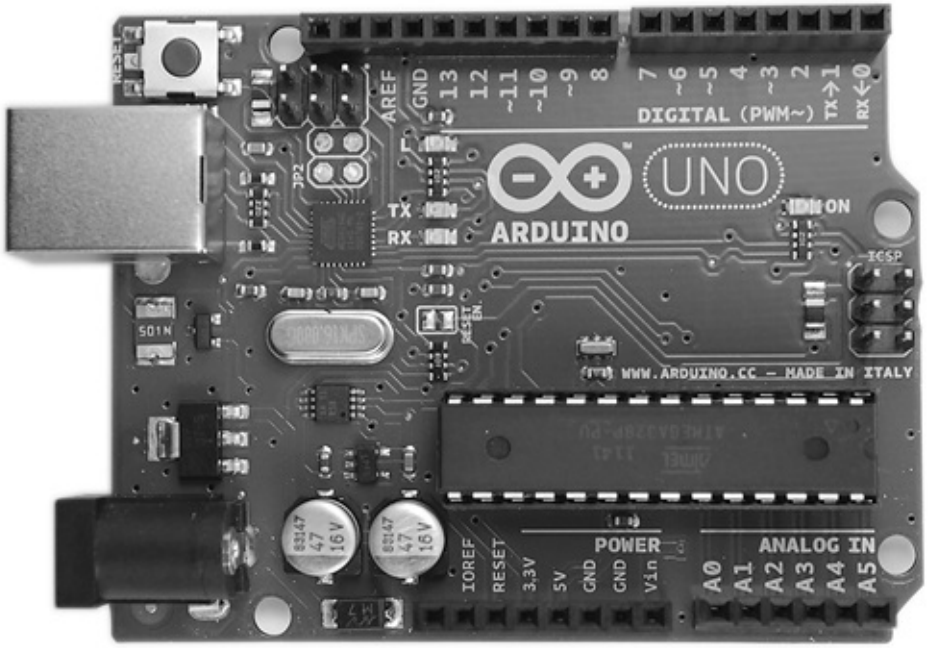


Рис. 7.3. Контакты I2C на плате Arduino Uno

Таблица 7.1. Контакты I2C в разных моделях Arduino

Модель	Контакты	Примечания
Uno	**A4** (**SDA**) и **A5** (**SCL**)	Контакты подписаны **SCL** и **SDA** и находятся рядом с контактом **AREF**. Эти линии интерфейса выводятся также на контакты **A4** и **A5**
Leonardo	**D2** (**SDA**) и **D3** (**SCL**)	Контакты подписаны **SCL** и **SDA** и находятся рядом с контактом **AREF**. Эти линии интерфейса выводятся также на контакты **D2** и **D3**
Mega2560	**D20** (**SDA**) и **D21** (**SCL**)	—
Due	**D20** (**SDA**) и **D21** (**SCL**)	Модель Due имеет вторую пару контактов I2C, подписанных **SDA1** и **SCL1**

Протокол I2C

Для передачи и приема данных через интерфейс I2C используются две линии (отсюда второе название — двухпроводной интерфейс, Two Wire Interface). Эти две линии называют также тактовой линией (Serial Clock Line, SCL) и линией данных (Serial Data Line, SDA). На рис. 7.4 изображена временная диаграмма сигнала, передаваемого через интерфейс I2C.

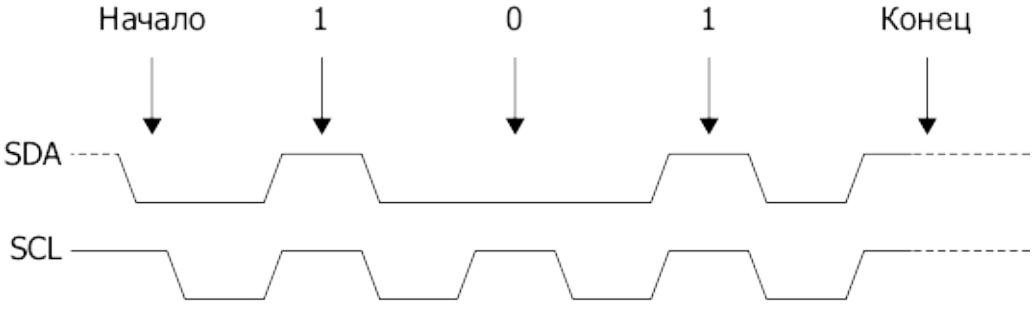


Рис. 7.4. Временная диаграмма сигнала, передаваемого через интерфейс I2C

Ведущее устройство генерирует тактовые импульсы на линии SCL, и, когда имеются данные для передачи, отправитель (ведущее или ведомое устройство) выводит линию SDA из третьего состояния (в режим цифрового выхода) и посылает данные в виде логических нулей и единиц в моменты положительных импульсов тактового сигнала. По окончании передачи вывод тактовых импульсов может быть остановлен, и линия SDA возвращается в третье состояние.

Библиотека Wire

Можно, конечно, генерировать описанные ранее импульсы самостоятельно, управляя битами, то есть включая и выключая цифровые выходы программно. Но чтобы упростить нам жизнь, в составе программного обеспечения для Arduino имеется библиотека Wire, принимающая на себя все сложности, связанные с синхронизацией, и позволяющая нам просто посылать и принимать байты данных.

Чтобы задействовать библиотеку Wire, ее сначала нужно подключить командой

```
#include <Wire.h>
```

Инициализация I2C

В большинстве случаев плата Arduino играет роль ведущего устройства на любой шине I2C. Чтобы инициализировать Arduino как ведущее устройство, нужно выполнить команду `begin` в функции `setup`, как показано далее:

```
void setup()
{
  Wire.begin();
}
```

Обратите внимание: поскольку в данном случае плата Arduino действует как ведущее устройство, ей не нужно присваивать адрес. Если бы плата настраивалась на работу в режиме ведомого устройства, нам пришлось бы присвоить адрес в диапазоне от 0 до 127, передав его как параметр, чтобы уникально идентифицировать плату на шине I2C.

Отправка данных ведущим устройством

Чтобы отправить данные устройству на шине I2C, сначала нужно выполнить функцию `beginTransmission` и передать ей адрес устройства-получателя:

```
Wire.beginTransmission(4);
```

Отправка данных устройствам на шине I2C может производиться побайтно или целыми массивами типа char, как показано в следующих двух примерах:

```
Wire.send(123); // передача байта со значением 123
Wire.send("ABC"); // передача строки символов "ABC"
```

По окончании передачи должна вызываться функция endTransmission:

```
Wire.endTransmission();
```

Прием данных ведущим устройством

Чтобы принять данные от ведомого устройства, сначала нужно указать количество ожидаемых байтов вызовом функции requestFrom:

```
Wire.requestFrom(4, 6); // запросить 6 байт у устройства с адресом
4
```

В первом аргументе этой функции передается адрес ведомого устройства, от которого ведущее устройство желает получить данные, а во втором аргументе — количество байтов, которое ожидается получить. Ведомое устройство может передать меньшее количество байтов, поэтому, чтобы определить, были ли получены данные и сколько байтов действительно получено, необходимо использовать функцию available. Следующий пример (взят из пакета примеров, входящих в состав библиотеки Wire) демонстрирует, как ведущее устройство принимает все полученные данные и выводит их в монитор последовательного порта:

```
#include <Wire.h>

void setup() {
  Wire.begin();        // подключиться к шине i2c (для ведущего
                       // устройства адрес не указывается)
       Serial.begin(9600);       //   инициализировать   монитор
последовательного порта
}

void loop() {
   Wire.requestFrom(8, 6);       // запросить 6 байт у ведомого
устройства #8

  while (Wire.available()) { // ведомое устройство может прислать
меньше
```

```
    char c = Wire.read();    // принять байт как символ
    Serial.print(c);         // вывести символ
  }

  delay(500);
}
```

Библиотека Wire автоматически буферизует входящие данные.

Примеры использования I2C

Любое устройство I2C должно иметь сопроводительное техническое описание, где перечисляются поддерживаемые им сообщения. Такие описания необходимы, чтобы конструировать сообщения для отправки ведомым устройствам и интерпретировать их ответы. Однако для многих устройств I2C, которые можно подключить к плате Arduino, существуют специализированные библиотеки, обертывающие сообщения I2C в простые и удобные функции. Фактически, если вам придется работать с каким-то устройством, для которого отсутствует специализированная библиотека, опубликуйте собственную библиотеку для всеобщего использования и заработайте себе несколько очков в карму.

Даже если полноценная библиотека поддержки того или иного устройства отсутствует, часто в Интернете можно найти полезные фрагменты кода, демонстрирующие работу с устройством.

УКВ-радиоприемник TEA5767

В первом примере, демонстрирующем взаимодействие с устройством I2C, библиотека не используется. Здесь осуществляется обмен фактическими сообщениями между Arduino и модулем TEA5767. Данный модуль можно купить в Интернете очень недорого, он легко подключается к плате Arduino и используется как УКВ-приемник, управляемый Arduino.

Самый сложный этап — подключение устройства. Контактные площадки очень маленькие и расположены очень близко друг к другу, поэтому многие предпочитают смастерить или купить адаптер для подключения к плате.

На рис. 7.5 изображена схема подключения модуля к Arduino.

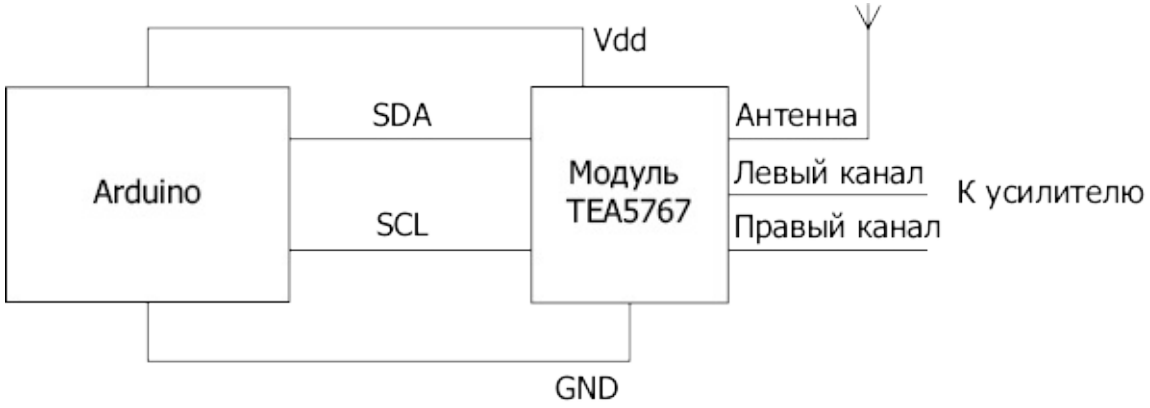


Рис. 7.5. Подключение модуля TEA5767 к плате Arduino Uno через интерфейс I2C

Техническое описание модуля TEA5767 можно найти по адресу www.sparkfun.com/datasheets/Wireless/General/TEA5767.pdf. Описание содержит массу технической информации, но, если пробежать взглядом по документу, можно заметить раздел с подробным описанием сообщений, распознаваемых устройством. В документации указывается, что TEA5767 принимает сообщения длиной 5 байт. Далее приводится полностью работоспособный пример, выполняющий настройку частоты сразу после запуска. На практике же обычно требуется несколько иной механизм настройки, например на основе кнопок и жидкокристаллического дисплея.

```
// sketch_07_01_I2C_TEA5767

#include <Wire.h>

void setup()
{
  Wire.begin();
  setFrequency(93.0); // МГц
}

void loop()
{
}

void setFrequency(float frequency)
{
  unsigned int frequencyB = 4 * (frequency * 1000000 + 225000) /
32768;
  byte frequencyH = frequencyB >> 8;
  byte frequencyL = frequencyB & 0XFF;
```

```
  Wire.beginTransmission(0x60);
  Wire.write(frequencyH);
  Wire.write(frequencyL);
  Wire.write(0xB0);
  Wire.write(0x10);
  Wire.write(0x00);
  Wire.endTransmission();
  delay(100);
}
```

Весь код, представляющий для нас интерес в этом примере, находится в функции `setFrequency`. Она принимает вещественное число — частоту в мегагерцах. То есть, если вы пожелаете собрать и опробовать этот проект, узнайте частоту, на которой вещает местная радиостанция с хорошим сильным сигналом, и вставьте ее значение в вызов `setFrequency` в функции `setup`.

Чтобы преобразовать вещественное значение частоты в двухбайтное представление, которое можно послать в составе пятибайтного сообщения, нужно выполнить некоторые арифметические операции. Эти операции выполняет следующий фрагмент:

```
unsigned int frequencyB = 4 * (frequency * 1000000 + 225000) / 32768;
byte frequencyH = frequencyB >> 8;
byte frequencyL = frequencyB & 0XFF;
```

Команда >> сдвигает биты вправо, то есть операция >> 8 сдвинет старшие 8 бит в сторону младших на 8 двоичных разрядов. Оператор & выполняет поразрядную операцию И (AND), которая в данном случае сбросит старшие 8 бит и оставит только младшие. Более полное обсуждение операций с битами вы найдете в главе 9.

Остальной код в функции `setFrequency` инициализирует передачу сообщения I2C ведомому устройству с адресом 0x60, который закреплен за приемником TEA5767. Затем осуществляется последовательная передача 5 байт, начиная с 2 байт частоты.

Прочитав документацию, вы узнаете о множестве других возможностей, доступных посредством разных сообщений, например о сканировании диапазона, выключении одного или двух каналов вывода звука и выборе режима моно/стерео.

В приложении мы еще вернемся к этому примеру и создадим библиотеку для Arduino, чтобы упростить работу с модулем TEA5767.

Взаимодействие между двумя платами Arduino

Во втором примере используются две платы Arduino, одна действует как ведущее устройство I2C, а другая — как ведомое. Ведущее устройство будет посылать сообщения ведомому, которое, в свою очередь, будет выводить их в монитор последовательного порта, чтобы можно было наглядно убедиться, что схема работает.

Схема соединения плат для этого примера показана на рис. 7.6. Обратите внимание на то, что модуль TEA5767 имеет встроенные подтягивающие сопротивления на линиях I2C. Однако в данном случае, когда друг к другу подключаются две платы Arduinos, такие резисторы отсутствуют, поэтому понадобится включить свои сопротивления с номиналом 4,7 кОм (рис. 7.6).

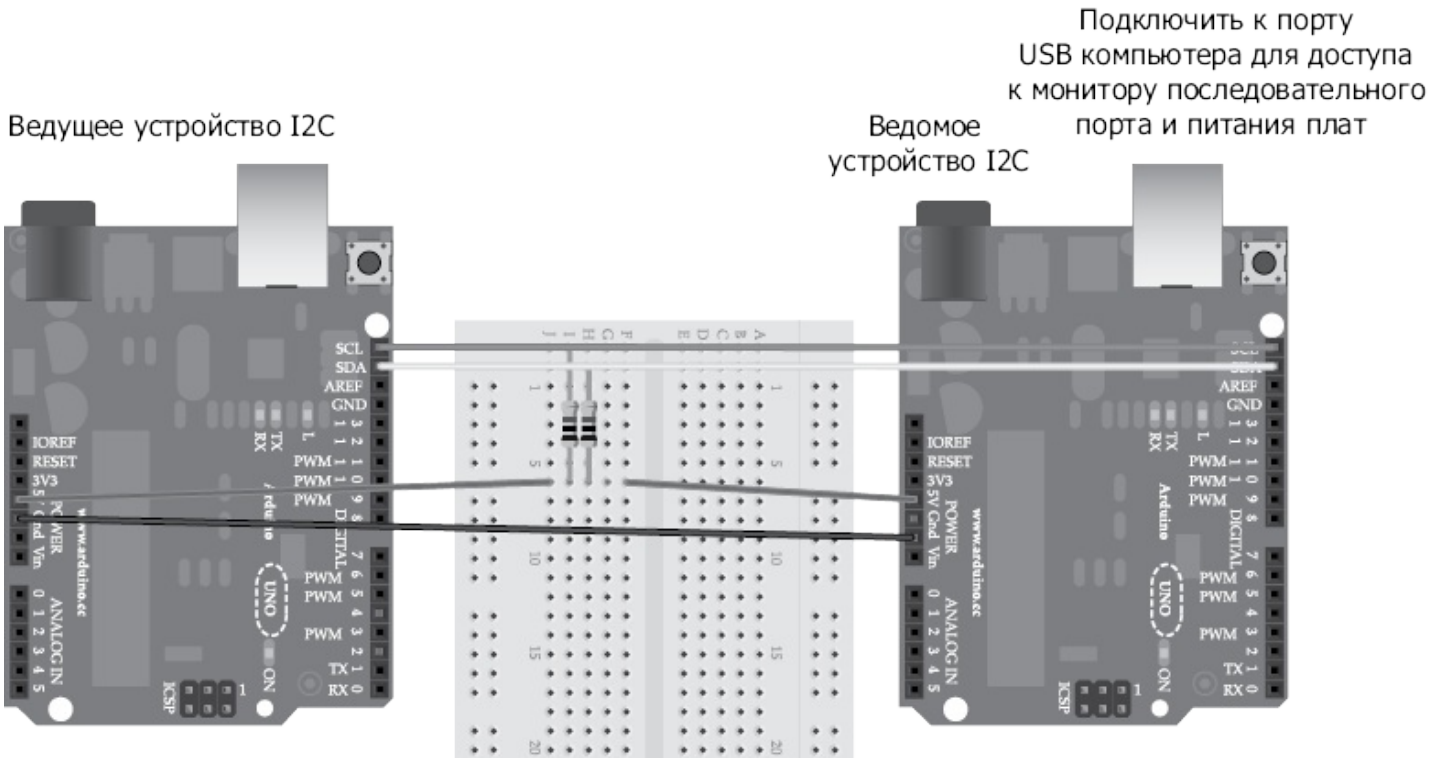


Рис. 7.6. Соединение двух плат Arduino через интерфейс I2C

В платы должны быть загружены разные скетчи. Оба скетча включены в состав примеров для библиотеки Wire. Программа для ведущей платы Arduino находится в меню **File—>Example—>Wire—>master_writer** (Файл—> Примеры—>Wire—>master_writer), а для ведомой платы — в меню **File—> Example—>Wire—>slave_receiver** (Файл—>Примеры—>Wire—>slave_receiver).

Запрограммировав обе платы, оставьте ведомую подключенной к компьютеру, чтобы увидеть вывод с этой платы в монитор последовательного порта и обеспечить питание ведущей платы Arduino.

Начнем со скетча в ведущей плате:

```
#include <Wire.h>

void setup() {
    Wire.begin();  // подключиться к шине i2c (для ведущего
```

```
устройства
                // адрес не указывается)
}

byte x = 0;

void loop() {
     Wire.beginTransmission(4);   //   инициализировать   передачу
устройству #4
  Wire.write("x is ");        // послать 5 байт
  Wire.write(x);              // послать 1 байт
  Wire.endTransmission();     // остановить передачу
  x++;
  delay(500);
}
```

Этот скетч генерирует сообщение вида `x is 1`, где `1` — число, увеличивающееся каждые полсекунды. Затем сообщение посылается ведомому устройству с адресом 4, как определено в вызове `beginTransmission`.

Задача ведомого скетча — принять сообщение от ведущего устройства и вывести его в монитор последовательного порта:

```
#include <Wire.h>

void setup() {
   Wire.begin(4);                    // подключиться к шине i2c с
адресом #4
    Wire.onReceive(receiveEvent); // зарегистрировать обработчик
события
    Serial.begin(9600);                        // открыть монитор
последовательного порта
}

void loop() {
  delay(100);
}

// эта функция вызывается всякий раз, когда со стороны ведущего
устройства
// поступают очередные данные, эта функция зарегистрирована как
```

```
обработчик
// события, см. setup()
void receiveEvent(int howMany) {
  while (1 < Wire.available()) { // цикл по всем принятым байтам,
кроме
                                 // последнего
    char c = Wire.read();        // прочитать байт как символ
    Serial.print(c);             // вывести символ
  }
  int x = Wire.read();           // прочитать байт как целое число
  Serial.println(x);             // вывести целое число
}
```

Первое, на что следует обратить внимание в этом скетче, — функции `Wire.begin` передается параметр 4. Он определяет адрес ведомого устройства на шине I2C, в данном случае 4. Он должен соответствовать адресу, который используется ведущим устройством для отправки сообщений.

СОВЕТ

К одной двухпроводной шине можно подключить множество ведомых плат Arduino при условии, что все они будут иметь разные адреса I2C.

Скетч для ведомой платы отличается от скетча для ведущей платы, потому что использует прерывания для приема сообщений, поступающих от ведущего устройства. Установка обработчика сообщений выполняется функцией `onReceive`, которая вызывается подобно подпрограммам обработки прерываний (глава 3). Вызов этой функции нужно поместить в функцию `setup`, чтобы обеспечить вызов пользовательской функции `receiveEvent` при получении любых поступающих сообщений.

Функция `receiveEvent` принимает единственный параметр — количество байт, готовых для чтения. В данном случае это число игнорируется. Цикл `while` читает по очереди все доступные символы и выводит их в монитор последовательного порта. Затем выполняются чтение единственного однобайтного числа в конце сообщения и его вывод в монитор порта. Использование `println` вместо `write` гарантирует, что значение байта будет выведено как число, а не символ с соответствующим числовым кодом (рис. 7.7).

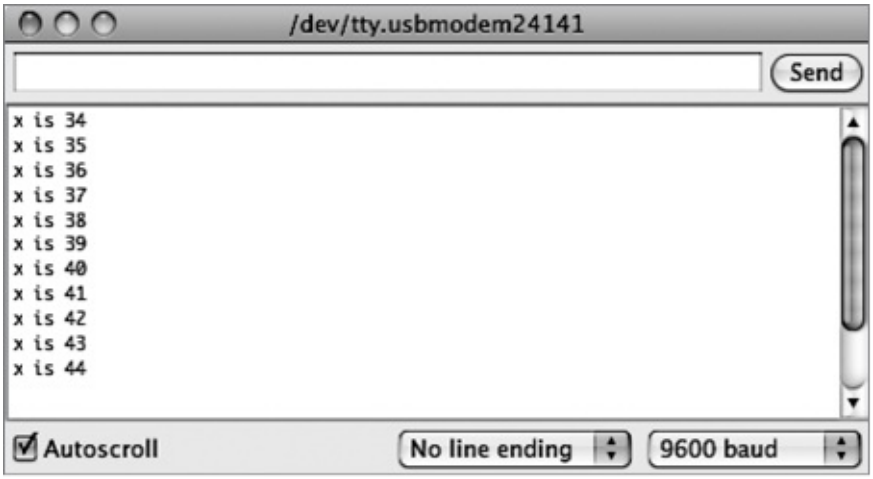


Рис. 7.7. Вывод в монитор порта сообщений, получаемых одной платой Arduino от другой через интерфейс I2C

Платы со светодиодными индикаторами

Еще один широкий спектр устройств I2C — разного рода дисплеи. Наиболее типичными представителями этих устройств являются светодиодные матрицы и семисегментные индикаторы, производимые компанией Adafruit. Они содержат светодиодные дисплеи, смонтированные на печатной плате, и управляющие микросхемы с поддержкой интерфейса I2C. Такое решение избавляет от необходимости использовать большое число контактов ввода/вывода на плате Arduino для управления светодиодным дисплеем и позволяет обойтись всего двумя контактами, **SDA** и **SCL**.

Эти устройства (верхний ряд на рис. 7.1) используются вместе с библиотеками, имеющими исчерпывающий набор функций для отображения графики и текста на светодиодных дисплеях компании Adafruit. Больше информации об этих красочных и интересных устройствах можно найти на странице www.adafruit.com/products/902.

Библиотеки скрывают все взаимодействия через интерфейс I2C за своим фасадом, давая возможность пользоваться высокоуровневыми командами, как демонстрирует следующий фрагмент, взятый из примера, входящего в состав библиотеки:

```
#include <Wire.h>
#include "Adafruit_LEDBackpack.h"
#include "Adafruit_GFX.h"
Adafruit_8x8matrix matrix = Adafruit_8x8matrix();
void setup()
{
  matrix.begin(0x70);
  matrix.clear();
  matrix.drawLine(0, 0, 7, 7, LED_RED);
  matrix.writeDisplay();
}
```

Часы реального времени DS1307

Еще одно распространенное устройство I2C — модуль часов реального времени DS1307. Для этого модуля также имеется удобная и надежная библиотека, упрощающая взаимодействие с модулем и избавляющая от необходимости иметь дело с фактическими сообщениями I2C. Библиотека называется RTClib и доступна по адресу **https://github.com/adafruit/RTClib**.

Следующие фрагменты кода тоже взяты из примеров, поставляемых с библиотекой:

```
#include <Wire.h>
#include "RTClib.h"
RTC_DS1307 RTC;

void setup ()
{
  Serial.begin(9600);
  Wire.begin();
  RTC.begin()

  if (! RTC.isrunning()) {
    Serial.println("RTC is NOT running!");
    // записать в модуль дату и время компиляции скетча
    RTC.adjust(DateTime(__DATE__, __TIME__));
  }
}

void loop () {
    DateTime now = RTC.now();
    Serial.print(now.year(), DEC);
    Serial.print('/');
    Serial.print(now.month(), DEC);
    Serial.print('/');
    Serial.print(now.day(), DEC);
    Serial.print(" (");
    Serial.print(daysOfTheWeek[now.dayOfTheWeek()]);
    Serial.print(") ");
    Serial.print(now.hour(), DEC);
    Serial.print(':');
    Serial.print(now.minute(), DEC);
    Serial.print(':');
```

```
    Serial.print(now.second(), DEC);
    Serial.println();
    delay(1000);
}
```

Если вам интересно увидеть, как в действительности выполняются взаимодействия через интерфейс I2C, просто загляните в файлы библиотеки. Например, исходный код библиотеки RTClib хранится в файлах **RTClib.h** и **RTClib.cpp**. Эти файлы находятся в папке **libraries/RTClib**.

Например, в файле **RTClib.cpp** можно найти определение функции now:

```
DateTime RTC_DS1307::now() {
  Wire.beginTransmission(DS1307_ADDRESS);
  Wire.write(i);
  Wire.endTransmission();
  Wire.requestFrom(DS1307_ADDRESS, 7);
  uint8_t ss = bcd2bin(Wire.read() & 0x7F);
  uint8_t mm = bcd2bin(Wire.read());
  uint8_t hh = bcd2bin(Wire.read());
  Wire.read();
  uint8_t d = bcd2bin(Wire.read());
  uint8_t m = bcd2bin(Wire.read());
  uint16_t y = bcd2bin(Wire.read()) + 2000;

  return DateTime (y, m, d, hh, mm, ss);
}
```

Функция `Wire.read` возвращает значения в двоично-десятичном формате (Binary-Coded Decimal, BCD), поэтому они преобразуются в байты с помощью библиотечной функции `bcd2bin`.

В формате BCD байт делится на два 4-битных полубайта. Каждый полубайт представляет одну цифру двузначного десятичного числа. Так, число 37 в формате BCD будет представлено как 0011 0111. Первые четыре бита соответствуют десятичному значению 3, а вторые четыре бита — значению 7.

В заключение

В этой главе вы познакомились с интерфейсом I2C и приемами его использования для организации взаимодействий плат Arduino с периферийными устройствами и другими платами Arduino.

В следующей главе мы исследуем еще одну разновидность последовательного интерфейса, используемого для взаимодействий с периферией. Он называется *1-Wire*. Этот интерфейс не получил такого широкого распространения, как I2C, но он используется в популярном датчике температуры DS18B20.

8. Взаимодействие с устройствами 1-Wire

Шина 1-Wire служит целям, похожим на цели шины I2C (глава 7), то есть она обеспечивает возможность взаимодействий микроконтроллеров с периферийными устройствами посредством минимального количества линий передачи данных. Стандарт 1-Wire, разработанный в компании Dallas Semiconductor, свел потребность в линиях до логического минимума — всего одной. Шина имеет более низкое быстродействие, чем I2C, но обладает интересной особенностью — *паразитным питанием* (parasitic power), позволяющее подключать периферийные устройства к микроконтроллеру всего двумя проводами: GND (ground — земля) и комбинированным проводом питания и передачи данных.

Шина 1-Wire поддерживается более узким диапазоном устройств, чем I2C. Большинство из них производят компании Dallas Semiconductor и Maxim. К их числу относятся устройства идентификации картриджей для принтеров, флеш-память и ЭСППЗУ, а также АЦП. Однако наибольшую популярность среди устройств 1-Wire у радиолюбителей завоевал температурный датчик DS18B20 компании Dallas Semiconductor.

Аппаратная часть 1-Wire

На рис. 8.1 показано, как подключить датчик DS18B20 к плате Arduino, используя всего два контакта и режим паразитного питания DS18B20.

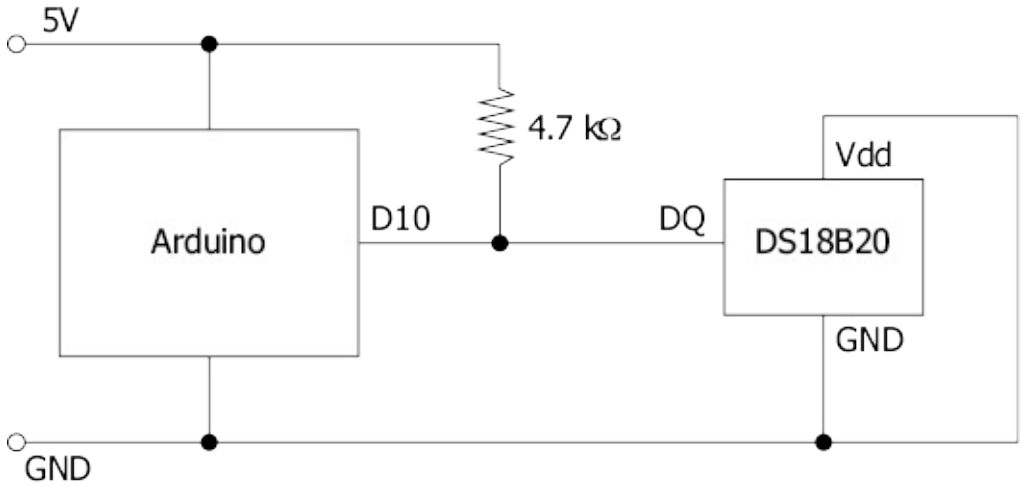


Рис. 8.1. Подключение устройства 1-Wire к плате Arduino

1-Wire — это именно шина, а не соединение «точка–точка». К ней можно подключить до 255 устройств, взяв за основу схему, изображенную на рис. 8.1. Если вы пожелаете использовать устройство в режиме нормального питания, то сопротивление 4,7 кОм можно убрать, а вывод **Vdd** датчика DS18B20 вместо GND соединить непосредственно с контактом **5 В** на плате Arduino.

Протокол 1-Wire

Так же как I2C, интерфейс 1-Wire использует понятия ведущего и ведомого устройств. Микроконтроллер играет роль ведущего, а периферийные устройства — ведомых. Каждое ведомое устройство еще на заводе получает уникальный идентификационный номер, который часто называют адресом, чтобы его можно было идентифицировать на шине, к которой подключено множество ведомых. Адрес имеет размер 64 бита, что позволяет иметь примерно $1{,}8 \times 10^{19}$ разных идентификационных номеров.

Подобно I2C, протокол 1-Wire предусматривает переключение режима работы шины ведущим устройством на ввод и вывод, чтобы иметь возможность двусторонних взаимодействий. Однако в шине 1-Wire отсутствует отдельная линия передачи тактовых сигналов, поэтому нули и единицы передаются длинными и короткими импульсами. Импульс длительностью 60 мкс обозначает 0, а длительностью 15 мкс — 1.

Обычно линия данных находится под напряжением с уровнем `HIGH`, но, когда микроконтроллеру (ведущему) требуется послать команду устройству, он генерирует специальный импульс сброса с уровнем `LOW` длительностью не менее 480 мкс. Вслед за ним следует последовательность импульсов 1 и 0.

Библиотека OneWire

Работу с интерфейсом 1-Wire здорово упрощает библиотека OneWire, которая доступна по адресу **http://playground.arduino.cc/Learning/OneWire**.

Инициализация 1-Wire

Чтобы инициализировать Arduino как ведущее устройство на шине 1-Wire, сначала нужно подключить библиотеку OneWire:

```
#include <OneWire.h>
```

Затем создать экземпляр OneWire и указать, какой контакт Arduino будет использоваться как линия данных на шине 1-Wire. Эти два действия можно объединить в одну команду, а в роли линии данных использовать любой контакт на плате Arduino — достаточно просто передать номер контакта в виде параметра:

```
OneWire bus(10);
```

В данном случае роль линии данных шины будет играть контакт **D10**.

Сканирование шины

Поскольку каждое ведомое устройство, подключенное к шине, имеет уникальный идентификационный номер, присвоенный на заводе, нужен какой-то способ определить адреса устройств, подключенных к шине. Было бы неблагоразумно «зашивать» адреса устройств в скетч, потому что в случае замены новое ведомое устройство будет иметь уже другой адрес и скетч не сможет обращаться к нему. Поэтому ведущее устройство (Arduino) должно создать своеобразную опись устройств на шине. Здесь следует отметить, что первые 8 бит в адресе определяют «семейство», которому принадлежит устройство, то есть по ним можно определить, является ли устройство, например, датчиком DS18B20 или относится к какому-то другому типу.

В табл. 8.1 перечислены некоторые из наиболее известных кодов семейств для шины 1-Wire. Полный список можно найти на странице **http://owfs.sourceforge.net/family.html**.

Таблица 8.1. Коды семейств устройств для шины 1-Wire

Код семейства (шестнадцатеричный)	Семейство	Описание
06	iButton 1993	Идентификационный ключ
10	DS18S20	Высокоточный температурный датчик с разрешающей способностью 9 бит
28	DS18B20	Высокоточный температурный датчик с разрешающей способностью 12 бит
1C	DS28E04-100	ЭСППЗУ емкостью 4 Кбайт

В библиотеке OneWire имеется функция `search`, которую можно использовать для поиска всех ведомых устройств на шине. Следующий пример выводит адреса всех устройств на шине в монитор последовательного порта:

```
// sketch_08_01_OneWire_List

#include <OneWire.h>

OneWire bus(10);

void setup()
{
  Serial.begin(9600);
  byte address[8]; // 64 бита
  while (bus.search(address))
  {
    for(int i = 0; i < 7; i++)
    {
```

```
      Serial.print(address[i], HEX);
      Serial.print(" ");
    }
    // проверить контрольную сумму
    if (OneWire::crc8(address, 7) == address[7])
    {
      Serial.println(" CRC OK");
    }
    else
    {
      Serial.println(" CRC FAIL");
    }
  }
}

void loop()
{
}
```

На рис. 8.2 показан результат выполнения этого скетча при наличии двух температурных датчиков DS18B20, подключенных к Arduino. Обратите внимание на то, что оба устройства имеют один и тот же код семейства в первом байте, равный 28 (в шестнадцатеричном формате).

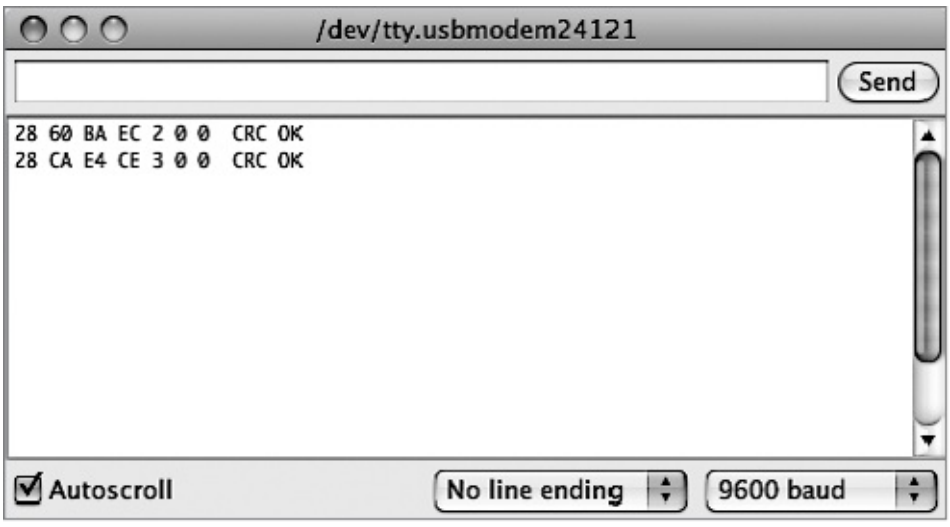


Рис. 8.2. Список ведомых устройств 1-Wire

Для работы функции `search` требуется массив размером 8 байт, куда она могла бы поместить следующий найденный адрес. После последнего обнаруженного устройства она возвращает 0. Это позволяет выполнять итерации в цикле `while`, как в предыдущем примере, пока не будут определены все адреса. Последний байт адреса в

действительности является циклической контрольной суммой (Cyclic Redundancy Check, CRC), позволяющей проверить целостность адреса. Библиотека OneWire включает специальную функцию для проверки контрольной суммы CRC.

Использование DS18B20

Следующий пример иллюстрирует использование библиотеки OneWire с температурным датчиком DS18B20. На рис. 8.3 изображена схема подключения DS18B20 к плате Arduino. Обратите внимание на то, что у самого датчика всего три контакта и он имеет вид обычного транзистора.

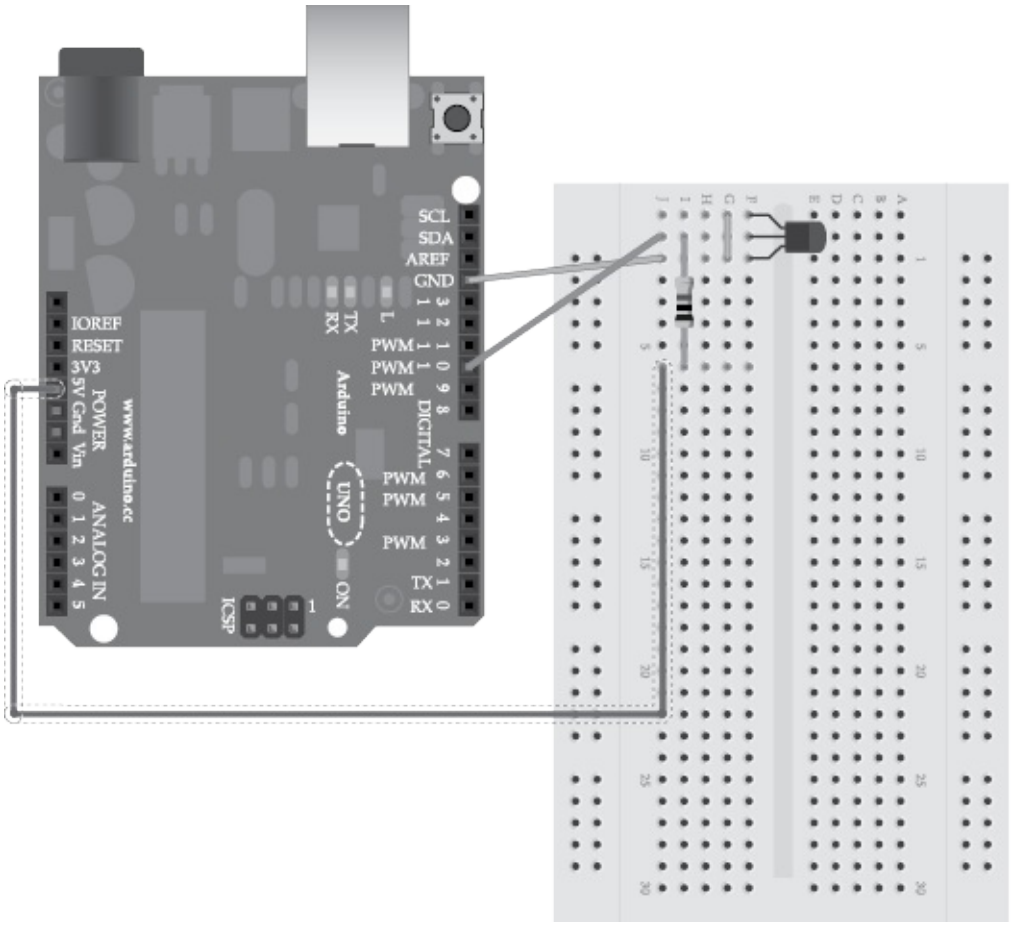


Рис. 8.3. Схема подключения DS18B20 к Arduino

Для датчика температуры компании Dallas Semiconductor имеется собственная библиотека, упрощающая операцию запроса температуры и декодирования результата. Библиотека DallasTemperature доступна для загрузки по адресу **https://github.com/milesburton/Arduino-Temperature-Control-Library**.

```
// sketch_08_02_OneWire_DS18B20

#include <OneWire.h>
#include <DallasTemperature.h>
```

```
const int busPin = 10;

OneWire bus(busPin);
DallasTemperature sensors(&bus);
DeviceAddress sensor;

void setup()
{
  Serial.begin(9600);
  sensors.begin();
  if (!sensors.getAddress(sensor, 0))
  {
    Serial.println("NO DS18B20 FOUND!");
  }
}

void loop()
{
  sensors.requestTemperatures();
  float tempC = sensors.getTempC(sensor);
  Serial.println(tempC);
  delay(1000);
}
```

Этот скетч выводит в окно монитора последовательного порта температуру в градусах Цельсия, прочитанную с единственного датчика температуры (рис. 8.4).

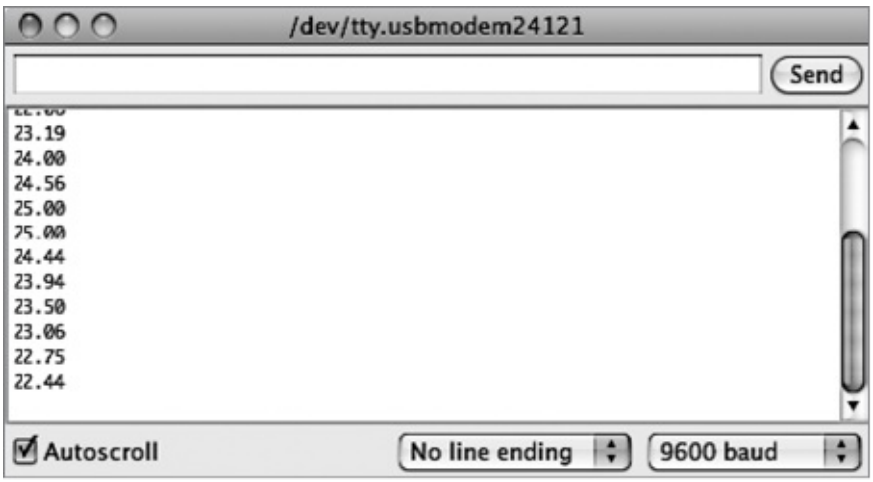


Рис. 8.4. Вывод температуры, прочитанной с датчика DS18B20

В этом примере используется только один датчик температуры, но его легко можно распространить на случай с несколькими датчиками. Библиотека DallasTemperature

сама определяет адреса устройств с помощью OneWire в функции `getAddress`, вторым параметром которой передается номер позиции датчика. Чтобы добавить второй датчик, нужно определить новую переменную для хранения его адреса и затем определить адрес вызовом `getAddress`. Пример с использованием двух датчиков можно загрузить с сайта книги, где он хранится под именем sketch_08_03_OneWire_DS18B20_2.

В заключение

В этой главе вы познакомились с шиной 1-Wire и узнали, как использовать ее для подключения популярного температурного датчика DS18B20.

В следующей главе мы исследуем еще одну разновидность последовательного интерфейса с названием SPI.

9. Взаимодействие с устройствами SPI

Последовательный периферийный интерфейс (Serial Peripheral Interface, SPI) — еще одна последовательная шина для подключения периферийных устройств к плате Arduino. Это быстрая шина, но для передачи данных в ней используются четыре линии против двух, используемых интерфейсом I2C. В действительности SPI не является истинной шиной, так как четвертая линия в нем называется «выбор ведомого» (Slave Select, SS). Каждое периферийное устройство на шине должно быть соединено своей линией SS с отдельным контактом на плате Arduino. Такая схема подключения эффективно выбирает нужное периферийное устройство на шине, отключая все остальные.

Интерфейс SPI поддерживается широким спектром устройств, включая многие типы тех же устройств, что поддерживают I2C. Нередко периферийные устройства поддерживают оба интерфейса, I2C и SPI.

Операции с битами

Взаимодействие по интерфейсу SPI часто связано с выполнением большого объема операций с отдельными битами. Первый пример, демонстрирующий использование АЦП на основе микросхемы MCP3008, в частности, требует хорошего понимания битовых операций и того, как маскировать ненужные биты, чтобы получить целое значение при чтении аналогового сигнала. По этой причине, прежде чем погружаться в особенности работы SPI, я хочу подробно поговорить об операциях с битами.

Двоичное и шестнадцатеричное представление

Впервые с битами мы встретились в главе 4 (см. рис. 4.2). Оперируя битами в байте или в слове (два байта), можно использовать их десятичные значения, но выполнять мысленно преобразования между двоичным и десятичным представлениями очень неудобно. Поэтому в скетчах для Arduino значения часто выражаются в виде двоичных констант, для чего поддерживается специальный синтаксис:

```
byte x = 0b00000011; // 3
unsigned int y = 0b0000000000000011; // 3
```

В первой строке определяется байт с десятичным значением 3 (2 + 1). Ведущие нули при желании можно опустить, но они служат отличным напоминанием о том, что определяется 8-битное значение.

Во второй строке определяется значение типа `int`, состоящее из 16 бит. Квалификатор `unsigned` перед именем типа `int` указывает, что определяемая

переменная может хранить только положительные числа. Этот квалификатор имеет значение лишь для операций с переменной, таких как +, –, * и др., которые не должны применяться, если переменная предназначена для манипуляций с битами. Но добавление слова `unsigned` в определения таких переменных считается хорошей практикой.

Когда дело доходит до 16-битных значений, двоичное представление становится слишком громоздким. По этой причине многие предпочитают использовать *шестнадцатеричную* форму записи.

Шестнадцатеричные числа — это числа в системе счисления с основанием 16, для обозначения цифр в этой системе используются не только десятичные цифры от 0 до 9, но и буквы от A до F, представляющие десятичные значения от 10 до 15. В этом представлении каждые четыре бита числа можно представить единственной шестнадцатеричной цифрой. В табл. 9.1 перечислены десятичные значения от 0 до 15 и показаны их двоичные и шестнадцатеричные представления.

Таблица 9.1. Двоичные и шестнадцатеричные числа

Десятичное значение	Двоичное значение	Шестнадцатеричное значение
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Шестнадцатеричные константы, как и двоичные, имеют специальную форму записи:

```
int x = 0x0003; // 3
int y = 0x010F; // 271 (256 + 15)
```

Эту форму записи используют не только в программном коде на C, но и в документации, чтобы показать, что число является шестнадцатеричным, а не десятичным.

Маскирование битов

Нередко при приеме данных от периферийных устройств, независимо от вида связи, данные поступают упакованными в байты, в которых часть битов может нести служебную информацию. Создатели периферийных устройств часто стараются втолкнуть как можно больше информации в минимальное число бит, чтобы добиться максимальной скорости передачи, но это усложняет программирование взаимодействий с такими устройствами.

Операция маскирования битов позволяет игнорировать некоторую часть данных в байте или в большой структуре данных. На рис. 9.1 показано, как выполнить маскирование байта, содержащего разнородные данные, и получить число, определяемое тремя младшими битами.

Рис. 9.1. Маскирование битов

В описаниях двоичных чисел вы обязательно столкнетесь со словосочетаниями «самый младший» и «самый старший». В двоичных числах, записанных с соблюдением правил, принятых в математике, самым старшим битом является крайний левый бит, а младшим значащим — крайний правый. Крайний правый бит может иметь ценность только 1 или 0. Вам также встретятся термины *самый старший бит* (Most Significant Bit, MSB) и *самый младший бит* (Least Significant Bit, LSB). Самый младший бит иногда называют также нулевым битом (бит 0), первый бит (бит 1) — следующий по старшинству и т.д.

В примере, изображенном на рис. 9.1, байт включает несколько значений, но нас

интересуют только три младших бита, которые нужно извлечь как число. Для этого можно выполнить поразрядную операцию И (AND) данных с маской, в которой три младших бита имеют значение 1. Поразрядная операция И (AND) для двух байт в свою очередь выполняет операцию И (AND) для каждой пары соответствующих битов и конструирует общий результат. Операция И (AND) для двух битов вернет 1, только если оба бита имеют значение 1.

Далее показана реализация этого примера на Arduino C с использованием оператора &. Обратите внимание на то, что поразрядная операция И (AND) обозначается единственным символом &, а логическая операция И (AND) — двумя: &&.

```
byte data = 0b01100101;
byte result = (data & 0b00000111);
```

Переменная `result` в данном случае получит десятичное значение 5.

Сдвиг битов

Часто необходимые биты в принимаемых данных могут занимать не самые младшие разряды в байте. Например, если из данных, изображенных на рис. 9.1, потребуется извлечь число, определяемое битами с 5-го по 3-й (рис. 9.2), то вам придется сначала применить маску, чтобы оставить интересующие биты, как в предыдущем примере, а затем сдвинуть биты на три позиции вправо.

Сдвиг вправо в языке C выполняется оператором >>, за которым следует число, определяющее количество разрядов, на которое производится сдвиг. В результате часть битов будет сдвинута за границу байта. Далее приводится реализация примера из предыдущего раздела на языке C:

```
byte data = 0b01101001;
byte result = (data & 0b00111000) >> 3;
```

Представьте, что вы получили два 8-битных байта и должны собрать из них одно 16-битное значение типа `int`. Для этого можно сдвинуть биты старшего байта в один конец значения `int`, а затем прибавить второй байт. Этот процесс иллюстрирует рис. 9.3.

Рис. 9.2. Маскирование и сдвиг битов

Рис. 9.3. Объединение двух байтов в значение типа int

Чтобы реализовать это в Arduino C, нужно сначала записать `highByte` в переменную результата типа `int`, сдвинуть влево на восемь позиций, а потом прибавить `lowByte`:

```
byte highByte = 0x6A;
byte lowByte = 0x0F;
int result = (highByte << 8) + lowByte;
```

Аппаратная часть SPI

На рис. 9.4 изображена типичная схема подключения к Arduino двух ведомых устройств.

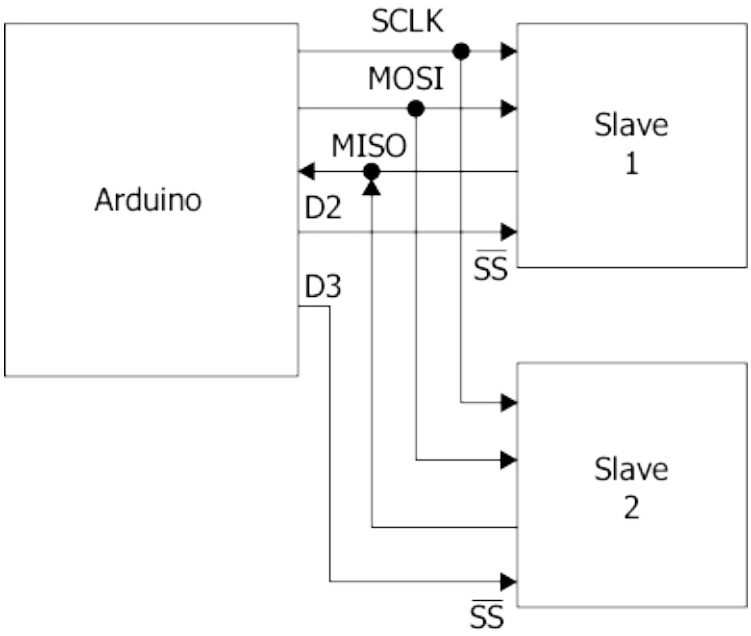


Рис. 9.4. Плата Arduino и два ведомых устройства SPI

Линии тактового сигнала системы (System Clock, SCLK), выход ведущего/вход ведомого (Master Out Slave In, MOSI) и вход ведущего/выход ведомого (Master In Slave Out, MISO) подключаются к контактам на плате Arduino с теми же именами, которые в модели Uno соответствуют контактам **D13**, **D11** и **D12**. В табл. 9.2 перечислены наиболее распространенные модели плат и соответствие контактов линиям интерфейса SPI.

Таблица 9.2. Контакты интерфейса SPI на плате Arduino

Модель	SCLK	MOSI	MISO
Uno	13 (ICSP3)	11 (ICSP4)	12 (ICSP1)
Leonardo	ICSP3	ICSP4	ICSP1
Mega2560	52 (ICSP3)	51 (ICSP4)	50 (ICSP1)
Due	ICSP3	ICSP4	ICSP1

Линиями выбора ведомого могут быть любые контакты на плате Arduino. Они используются для выбора определенного ведомого устройства непосредственно перед передачей данных и его отключения по завершении обмена данными.

Ни к одной из линий не требуется подключать подтягивающее сопротивление.

Поскольку в некоторых моделях Arduino, в том числе Leonardo, контакты интерфейса SPI имеются только на колодке ICSP, многие платы расширений часто имеют гнезда SPI для подключения к колодке контактов ICSP. На рис. 9.5 изображена

колодка ICSP с подписанными контактами.

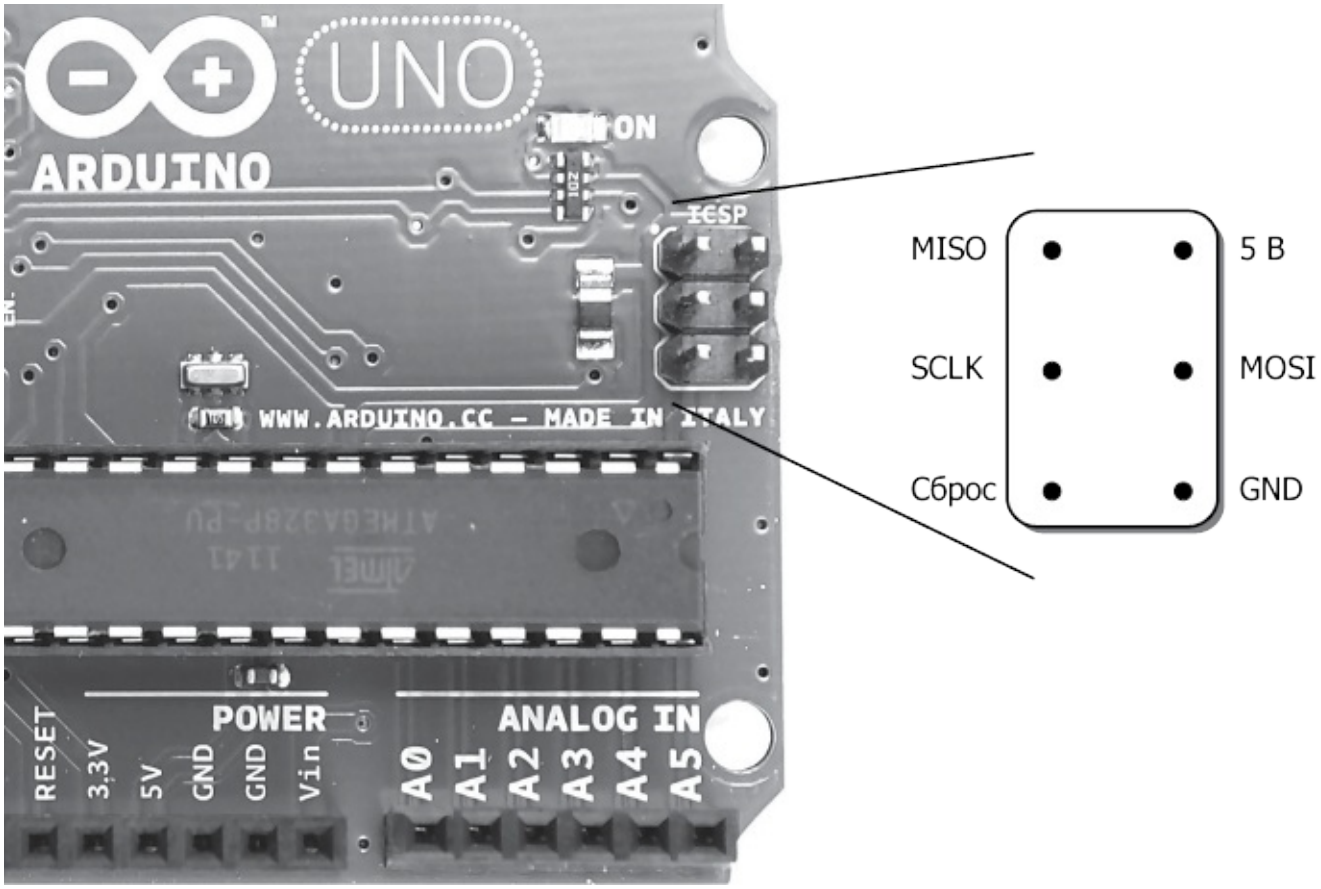


Рис. 9.5. Контакты ICSP на плате Arduino Uno

Обратите внимание на то, что на плате Arduino Uno имеется вторая колодка ICSP, рядом с кнопкой сброса. Она предназначена для программирования интерфейса USB.

Протокол SPI

Протокол SPI на первый взгляд кажется сложным и запутанным, потому что данные передаются и принимаются обеими сторонами, ведущим и выбранным ведомым, параллельно. Одновременно с передачей ведущим устройством (Arduino) бита по линии MOSI ведомое устройство посылает другой бит по линии MISO плате Arduino.

Обычно Arduino посылает байт данных и затем восемь нулей, одновременно принимая результат от ведомого устройства. Так как частота передачи устанавливается ведущим устройством, убедитесь в том, что она не слишком высока для ведомого устройства.

Библиотека SPI

Библиотека SPI входит в состав Arduino IDE, поэтому вам не придется ничего устанавливать, чтобы воспользоваться ею. Но она поддерживает только сценарии, когда плата Arduino действует в роли ведущего устройства. Кроме того, библиотека

поддерживает передачу данных только целыми байтами. Для большинства периферийных устройств этого вполне достаточно, однако некоторые устройства предполагают обмен 12-битными сообщениями, что несколько осложняет обмен из-за необходимости манипуляций с битами, как будет показано в примере в следующем разделе.

Прежде всего, как обычно, необходимо подключить библиотеку SPI:

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

Затем инициализировать ее командой `SPI.begin` в функции запуска передачи:

```
void setup()
{
  SPI.begin();
  pinMode(chipSelectPin, OUTPUT);
  digitalWrite(chipSelectPin, HIGH);
}
```

Для моделей платы Arduino, кроме Due, нужно также настроить цифровые выходы для всех линий выбора ведомых устройств. Роль таких выходов могут играть любые контакты на плате Arduino. После настройки их на работу в режиме выходов требуется сразу же установить на них уровень напряжения `HIGH` из-за инвертированной логики выбора ведомого, согласно которой напряжение `LOW` означает, что данное устройство выбрано.

Для модели Due имеется расширенная версия библиотеки SPI, поэтому достаточно определить контакт для выбора ведомого — и библиотека автоматически будет устанавливать на нем уровень `LOW` перед передачей и возвращать уровень `HIGH` по ее окончании. Для этого нужно передать команде `SPI.begin` аргумент с номером контакта. Недостаток такого подхода заключается в нарушении совместимости с другими моделями Arduino. В примерах, приводимых далее, все ведомые устройства выбираются вручную, и потому эти примеры будут работать на всех платах Arduino.

Для настройки соединения через интерфейс SPI имеется множество вспомогательных функций. Однако параметры по умолчанию вполне подходят для большинства случаев, поэтому изменять их нужно, только если документация с описанием ведомого устройства требует их изменения. Эти функции перечислены в табл. 9.3.

Таблица 9.3. Вспомогательные функции

Функция	Описание
SPI.setClockDivider(SPI_CLOCK_DIV64)	Выполняет деление тактовой частоты (по умолчанию равна 4 МГц) на 2, 4, 8, 16, 32, 64 или 128

SPI.setBitOrder(LSBFIRST)	Устанавливает порядок передачи битов **LSBFIRST** (от младшего к старшему) или **MSBFIRST** (от старшего к младшему). По умолчанию используется порядок **MSBFIRST**
SPI.setDataMode(SPI_MODE0)	Возможные значения аргументов этой функции от **SPI_MODE0** до **SPI_MODE3**. Определяют полярность и фазу тактового сигнала. Обычно нет необходимости изменять эту настройку, если только документация не требует установить какой-то определенный режим работы для организации обмена с ведомым устройством

Объединенные передача и прием происходят в функции `transfer`. Эта функция посылает байт данных и возвращает байт данных, принятый в процессе передачи:

```
byte sendByte = 0x23;
byte receiveByte = SPI.transfer(sendByte);
```

Поскольку диалог с периферией обычно имеет форму запроса со стороны ведущего и ответа со стороны ведомого, часто последовательно выполняются две передачи данных: одна — запрос, другая (возможно, во время передачи нулей) — ответ периферийного устройства. Вы увидите, как это происходит, в следующем примере.

Пример SPI

Этот пример демонстрирует взаимодействие платы Arduino с интегральной микросхемой восьмиканального АЦП MCP3008, добавляющего еще восемь 10-битных аналоговых входов. Эта микросхема стоит очень недорого и легко подключается к плате. На рис. 9.6 изображена схема подключения микросхемы к плате Arduino с использованием макетной платы и нескольких проводов. Переменное сопротивление служит для изменения уровня напряжения на аналоговом входе **0** между 0 и 5 В.

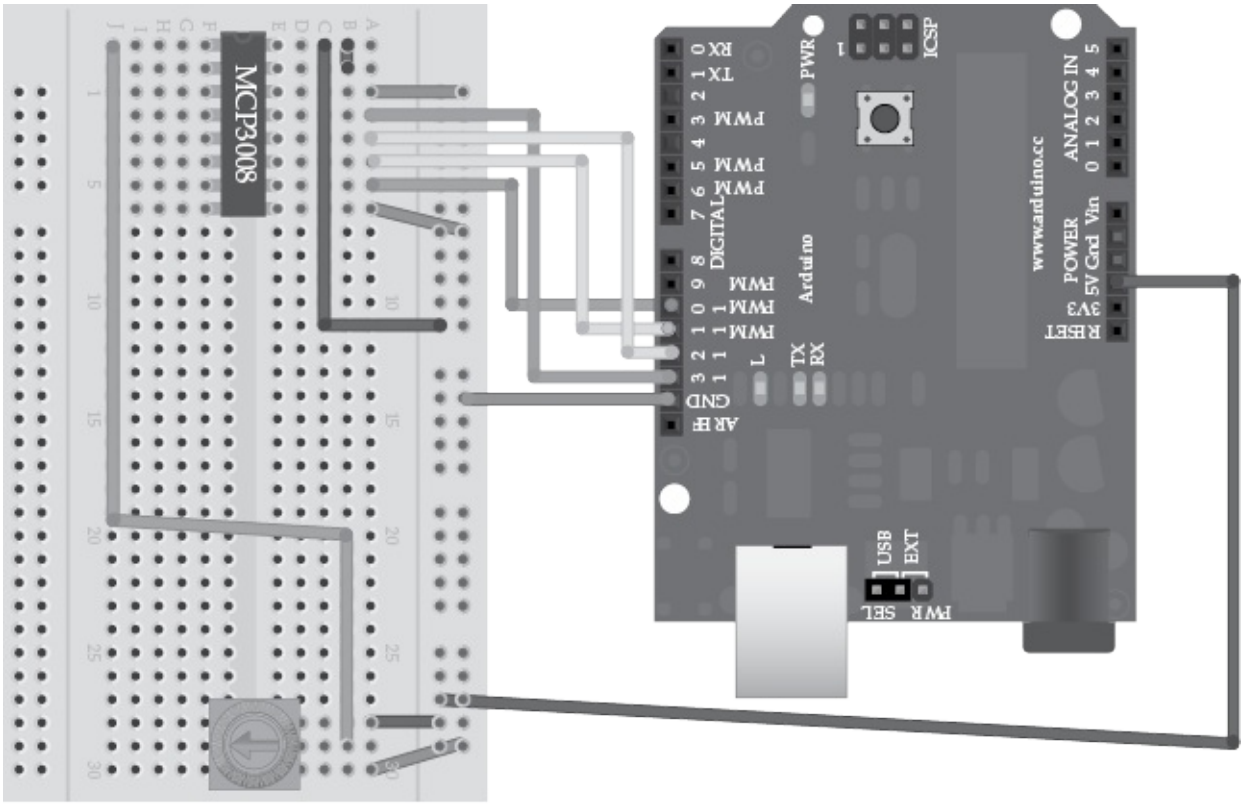


Рис. 9.6. Схема соединения компонентов для примера SPI

Ниже приводится скетч для этого примера:

```
// sketch_09_01_SPI_ADC

#include <SPI.h>

const int chipSelectPin = 10;

void setup()
{
  Serial.begin(9600);
  SPI.begin();
  pinMode(chipSelectPin, OUTPUT);
  digitalWrite(chipSelectPin, HIGH);
}

void loop()
{
  int reading = readADC(0);
  Serial.println(reading);
  delay(1000);
}

int readADC(byte channel)
{
  unsigned int configWord = 0b11000 | channel;
  byte configByteA = (configWord >> 1);
  byte configByteB = ((configWord & 1) << 7);
  digitalWrite(chipSelectPin, LOW);
  SPI.transfer(configByteA);
  byte readingH = SPI.transfer(configByteB);
  byte readingL = SPI.transfer(0);
  digitalWrite(chipSelectPin, HIGH);

//  printByte(readingH);
//  printByte(readingL);

  int reading = ((readingH & 0b00011111) << 5)
```

```
                + ((readingL & 0b11111000) >> 3);

  return reading;
}

void printByte(byte b)
{
  for (int i = 7; i >= 0; i--)
  {
     Serial.print(bitRead(b, i));
  }
  Serial.print(" ");
}
```

Функция `printByte` использовалась на этапе разработки для вывода двоичных данных. Несмотря на то что `Serial.print` может выводить двоичные значения, она не добавляет ведущие нули, что немного усложняет интерпретацию данных. Функция `printByte`, напротив, всегда выводит все 8 бит.

Чтобы увидеть данные, поступающие от микросхемы MCP3008, уберите символы `//` перед двумя вызовами `printByte`, и данные появятся в окне монитора последовательного порта.

Наиболее интересный для нас код сосредоточен в функции `readADC`, которая принимает номер канала АЦП (от 0 до 7). Прежде всего нужно выполнить некоторые манипуляции с битами, чтобы создать конфигурационный байт, определяющий вид преобразования аналогового сигнала и номер канала.

Микросхема поддерживает два режима работы АЦП. В одном выполняется сравнение двух аналоговых каналов, а во втором, несимметричном режиме (который используется в этом примере), возвращается значение, прочитанное из указанного канала, как в случае с аналоговыми входами на плате Arduino. В документации к MCP3008 (http://ww1.microchip.com/downloads/en/DeviceDoc/21295d.pdf) указывается, что в настроечной команде должны быть установлены четыре бита: первый бит должен быть установлен в 1, чтобы включить несимметричный режим, следующие три бита определяют номер канала (от 0 до 7).

Микросхема MCP3008 не поддерживает режим побайтовой передачи, в котором действует библиотека SPI. Чтобы MCP3008 распознала эти четыре бита, их нужно разбить на два байта. Далее показано, как это делается:

```
unsigned int configWord = 0b11000 | channel;
byte configByteA = (configWord >> 1);
byte configByteB = ((configWord & 1) << 7);
```

Первый байт конфигурационного сообщения содержит две единицы, первая из которых может не понадобиться, а вторая — это бит режима (в данном случае несимметричного). Другие два бита в этом байте — старшие два бита номера канала. Оставшийся бит из этого номера передается во втором конфигурационном байте как самый старший бит.

Следующая строка устанавливает уровень `LOW` в линии выбора ведомого устройства, чтобы активировать его:

```
digitalWrite(chipSelectPin, LOW);
```

После этого отправляется первый конфигурационный байт:

```
SPI.transfer(configByteA);
byte readingH = SPI.transfer(configByteB);
byte readingL = SPI.transfer(0);
digitalWrite(chipSelectPin, HIGH);
```

Аналоговые данные не будут передаваться обратно, пока не будет отправлен второй байт. 10 битов данных из АЦП разбиты на два байта, поэтому, чтобы подтолкнуть отправку оставшихся данных, выполняется передача нулевого байта.

Затем в линии выбора ведомого устанавливается уровень `HIGH` как признак того, что передача завершена.

Полученное 10-битное значение пересчитывается, как показано в следующей строке:

```
int reading = ((readingH & 0b00011111) << 5)
            + ((readingL & 0b11111000) >> 3);
```

Каждый из двух байт содержит пять бит данных из десяти. Первый байт содержит данные в пяти младших битах. Все остальные биты, кроме этих пяти, маскируются, и затем выполняется сдвиг 16-битного значения `int` влево на пять разрядов. Младший байт содержит остальные данные в пяти старших битах. Они также выделяются маской, сдвигаются вправо на три разряда и прибавляются к 16-битному значению `int`.

Для проверки откройте монитор последовательного порта. Вы должны увидеть, как в нем появляются некоторые данные. Если повернуть шток переменного сопротивления по часовой стрелке, чтобы увеличить напряжение на аналоговом входе с 0 до 5 В, вы должны увидеть картину, похожую на рис. 9.7. Первые два двоичных числа — это два байта, полученных от MCP3008, а последнее десятичное число — это аналоговое значение между 0 и 1023.

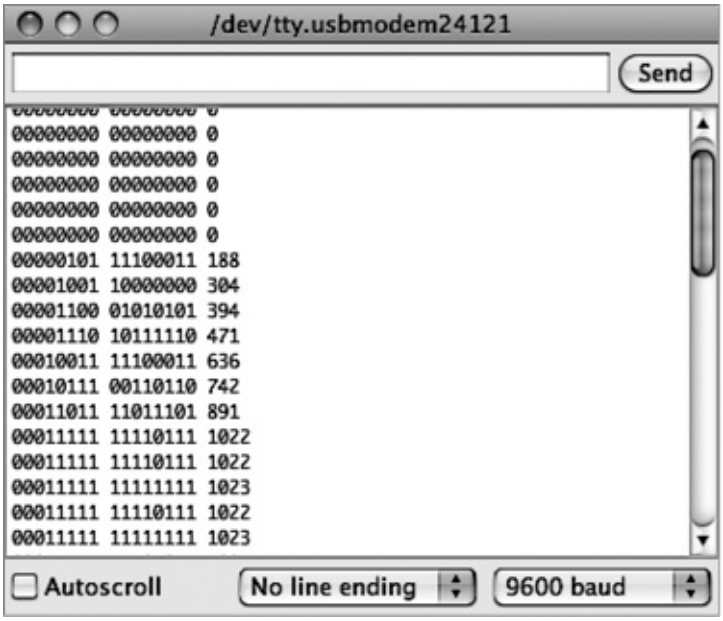


Рис. 9.7. Просмотр сообщений в двоичном виде

В заключение

Организовать взаимодействие через интерфейс SPI без применения библиотеки очень непросто. Вам придется пройти тернистый путь проб и ошибок, чтобы добиться нужного результата. Занимаясь отладкой любого кода, всегда начинайте со сбора информации и исследования принимаемых данных. Шаг за шагом вы нарисуете полную картину происходящего и затем сможете сконструировать код, помогающий достичь желаемого результата.

В следующей главе мы исследуем последний стандартный интерфейс, поддерживаемый Arduino, — последовательный порт ТТЛ. Это стандартный вид связи «точка–точка», а не шина, но тем не менее очень удобный и широко используемый механизм обмена данными.

10. Программирование последовательного интерфейса

Последовательный интерфейс вам должен быть хорошо знаком. Он используется для программирования платы Arduino, а также для взаимодействий с монитором последовательного порта, посредством которого можно организовать обмен данными между платой и компьютером. Это можно делать через адаптер, связывающий порт USB с последовательным портом на плате Arduino, или непосредственно через адаптер последовательного порта. Адаптер последовательного порта часто называют последовательным портом ТТЛ или просто последовательным портом. Аббревиатура ТТЛ означает «транзистор-транзисторная логика» — это редко используемая в настоящее время технология, основанная на 5-вольтовой логике.

Последовательный интерфейс этого вида не является шиной. Он поддерживает взаимодействия вида «точка–точка», в которые вовлечены только два устройства — обычно сама плата Arduino и периферийное устройство.

Последовательный интерфейс ТТЛ вместо I2C или SPI обычно поддерживают большие периферийные устройства или устройства, разработанные довольно давно и традиционно использующие последовательный интерфейс ТТЛ. К их числу относятся также устройства, изначально предназначенные для подключения к последовательному порту персонального компьютера. Примерами могут служить модули GPS, мультиметры с возможностью передачи данных, а также устройства чтения штрихкодов и радиометок.

Аппаратная часть последовательного интерфейса

На рис. 10.1 изображена схема последовательного интерфейса на плате Arduino Uno.

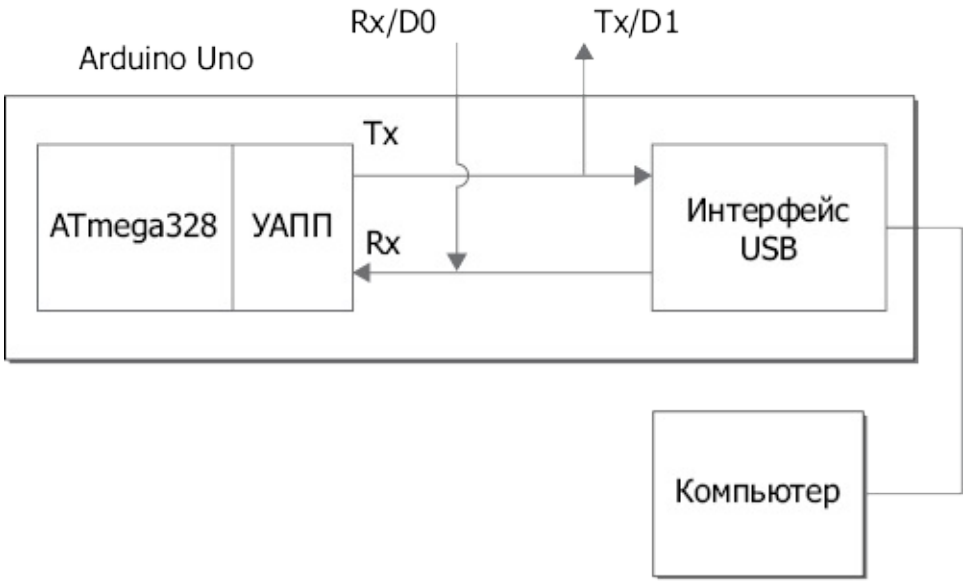


Рис. 10.1. Последовательный интерфейс на плате Arduino Uno

Микроконтроллер ATmega328 на плате Arduino Uno имеет два контакта: **Rx** и **Tx**

(*прием* и *передача* соответственно). Они дополнительно выводятся на контакты **D0** и **D1**, но, если вы решите использовать их как обычные входы/выходы, имейте в виду, что не сможете запрограммировать Arduino, пока к ним подключены внешние устройства.

Контакты **Rx** и **Tx** составляют последовательный интерфейс аппаратного универсального асинхронного приемопередатчика (УАПП) (Universal Asynchronous Receiver Transmitter, UART) в ATmega328. Этот компонент микроконтроллера отвечает за передачу байтов данных в микроконтроллер и их прием из него.

Модель Uno имеет отдельный процессор, действующий как адаптер между портом USB и последовательным портом. Помимо различий в уровнях сигналов, шина USB имеет также более сложный протокол, чем последовательный порт, поэтому за кулисами выполняется масса преобразований разного рода, чтобы создавалось ощущение, что последовательный порт микроконтроллера ATmega328 взаимодействует с компьютером напрямую.

Модель Arduino Leonardo не имеет отдельной микросхемы интерфейса USB, вместо этого в ней используется микроконтроллер ATmega, включающий два кристалла УАПП и один интерфейс USB (рис. 10.2).

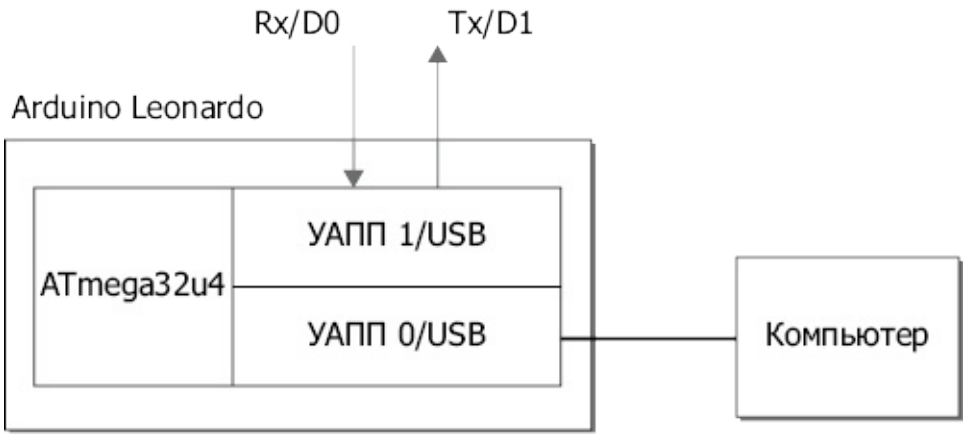


Рис. 10.2. Аппаратная поддержка последовательного интерфейса на плате Arduino Leonardo

Один из приемопередатчиков УАПП обслуживает интерфейс USB, а другой соединен с контактами **Rx** и **Tx** (**D0** и **D1**). Это позволяет подключать внешние устройства к контактам **Tx** и **Rx** и сохранить возможность программирования платы Arduino и обмена данными с монитором последовательного порта.

Другие модели Arduino имеют иное количество и схему подключения последовательных портов, как показано в табл. 10.1. Обратите внимание на то, что Due является единственной моделью Arduino, в которой последовательные порты работают с уровнями сигналов 3,3 В, а не 5 В.

Последовательный интерфейс ТТЛ способен поддерживать связь лишь по относительно коротким линиям (в пределах нескольких метров), и чем выше скорость обмена, тем короче должна быть линия. Для передачи данных на большие расстояния был разработан электрический стандарт RS232. Персональные компьютеры,

выпускавшиеся до недавнего прошлого, часто снабжались последовательными портами RS232. Стандарт RS232 изменил уровни сигналов, чтобы обеспечить передачу данных на большие расстояния, чем позволяет последовательный интерфейс ТТЛ.

Таблица 10.1. Последовательные интерфейсы УАПП в разных моделях Arduino

Модель	Число последовательных портов	Подробности
Uno	1	Линия **Rx** подключена к контакту **D0**, а линия **Tx** — к контакту **D1**. Этот порт используется также интерфейсом USB
Leonardo	2	Отдельный порт для интерфейса USB. Линия **Rx** подключена к контакту **D0**, а линия **Tx** — к контакту **D1**
Mega2560	4	Интерфейс USB подключен к контактам **D0** и **D1**. Три других порта: Serial1 — к контактам 19 (**Rx**) и 18 (**Tx**), Serial2 — к контактам 17 (**Rx**) и 16 (**Tx**), Serial3 — к контактам 15 (**Rx**) и 14 (**Tx**)
Due	4	Отдельный порт для интерфейса USB. Последовательный порт 0 использует контакты **D0** (**Rx**) и D1 (**Tx**). Три других порта: Serial1 — к контактам 19 (**Rx**) и 18 (**Tx**), Serial2 — к контактам 17 (**Rx**) и 16 (**Tx**), Serial3 — к контактам 15 (**Rx**) и 14 (**Tx**)

Протокол последовательного интерфейса

Протокол последовательного интерфейса и большая часть терминологии появились еще на заре развития компьютерных сетей. Отправитель и получатель должны были договориться о скорости обмена данными. Эта скорость, измеряемая в *бодах*, устанавливалась на обоих концах соединения перед началом обмена. Скорость в бодах определяет частоту переходов между уровнями сигнала. Она совпадала бы с количеством бит в секунду, если бы байт данных имел стартовый бит, стоповый бит и бит четности. То есть, чтобы получить грубую оценку скорости передачи в байтах в секунду, нужно скорость в бодах разделить на 10.

Скорость в бодах выбиралась из числа предопределенных стандартом значений. Увидеть эти значения можно в раскрывающемся списке в окне монитора последовательного порта. Программное обеспечение Arduino поддерживает следующие скорости: 300, 1200, 4800, 9600, 14 400, 19 200, 28 800, 38 400, 57 600 и 115 200 бод.

Чаще всего для связи с Arduino используется скорость 9600 бод, которая выбирается по умолчанию. Этот выбор не обусловлен какими-то серьезными причинами, так как связь с платами Arduino действует вполне надежно даже на скорости 115 200 бод. Эту скорость можно использовать в проектах, где требуется высокая скорость передачи. Также часто используется скорость 2400 бод. Некоторые периферийные устройства, такие как адаптеры Bluetooth и аппаратура GPS, работают на этой скорости.

Другой довольно запутанный параметр настройки последовательного интерфейса, который может вам встретиться, — это строка вида 8N1. В данном случае она означает:

размер пакета 8 бит, отсутствие контроля четности и 1 стоповый бит. Несмотря на возможность других комбинаций, практически все устройства, которые вам попадутся, будут использовать параметр 8N1.

Команды последовательного порта

Команды последовательного порта включены в стандартную библиотеку Arduino, поэтому нет необходимости использовать в скетчах команду `include`.

Запуск взаимодействий по последовательному порту осуществляется командой `Serial.begin`, которая принимает параметр со скоростью в бодах:

```
Serial.begin(9600);
```

Обычно она вызывается только один раз, в функции `setup`.

Если используется плата, имеющая несколько последовательных портов, и вы собираетесь организовать обмен через порт по умолчанию (порт 0), достаточно вызвать простую команду `Serial.begin`. Но для других портов нужно указать в команде номер порта после слова `Serial`. Например, чтобы запустить взаимодействия по последовательному порту 3 на плате Arduino Due, скетч должен выполнить следующую команду:

```
Serial3.begin(9600);
```

После вызова команды `Serial.begin` приемопередатчик УАПП переходит в режим приема входящих данных и автоматически сохраняет их в буфере, поэтому, даже если процессор занят в это время чем-то другим, данные не будут теряться, пока буфер не переполнится.

Функция `loop` может проверить наличие входящих данных с помощью функции `Serial.available`. Она возвращает число байтов, доступных для чтения. Если в буфере нет ни одного байта, она вернет 0. В языке C это равносильно значению `false`, поэтому часто можно видеть такой код, проверяющий доступность данных:

```
void loop()
{
  if (Serial.available())
  {
    // прочитать и обработать следующий байт
  }
}
```

Команда `read` не имеет параметров и просто читает следующий доступный байт из

буфера. Функция `readBytes` читает доступные байты в буфер, организованный внутри скетча. Она принимает два аргумента: буфер (это должна быть ссылка на массив байтов) и максимальное число байтов для чтения. Эта команда может пригодиться в проектах для пересылки в плату Arduino строк переменной длины. Но вообще лучше избегать этого и стараться осуществлять обмен максимально простыми данными фиксированного размера.

Также могут пригодиться функции `parseInt` и `parseFloat`, позволяющие сохранять строки, пересылаемые в плату Arduino, как числа в переменных типа `int` и `float` соответственно.

```
void loop()
{
  if (Serial.available())
  {
    int x = parseInt();
  }
}
```

Обе функции читают символы, пока не достигнут конца строки или не встретят пробел или другой нецифровой символ, и затем преобразуют прочитанную последовательность цифр в числовое значение.

Перед использованием функций, таких как `parseInt` и `parseFloat`, убедитесь, что понимаете, зачем вы это делаете. Мне приходилось видеть код, написанный другими, который преобразовывал значение `int` в массив символов и посылал его второй плате Arduino, которая преобразовывала массив обратно в значение `int`. Такое решение нельзя назвать удачным по следующим причинам.

- В этом нет необходимости. Двоичные данные передаются через последовательный интерфейс ничуть не хуже. Достаточно просто передать старший и младший байты значения `int` и затем собрать их обратно в значение `int` на стороне получателя.

- Преобразование чисел в строки и обратно выполняется медленно.

- Вместо шести символов (включая завершающий нулевой символ) по линии связи можно передать всего два байта, составляющие значение `int`.

Если устройство, с которым вы взаимодействуете, вам неподконтрольно и протоколом предполагается передача чисел в виде строк или полей данных переменной длины, то применение этих функций вполне оправданно. Но, если реализация протокола полностью находится в ваших руках, облегчите себе жизнь и откажитесь от ненужных сложностей, связанных с преобразованием типов и передачей

сообщений в разных форматах.

Примеры, приведенные в разделе «Примеры использования последовательного интерфейса» далее в этой главе, можно использовать в качестве шаблонов при разработке своего кода, осуществляющего обмен данными.

Поддержка последовательного интерфейса включает массу функций, многие из которых вам никогда не понадобятся. Мы охватили здесь только самые необходимые. Информацию об остальных ищите в документации с описанием последовательного интерфейса Arduino по адресу **http://arduino.cc/en/Reference/Serial**[8].

Библиотека SoftwareSerial

Иногда, особенно при использовании модели Arduino Uno, единственного последовательного порта оказывается недостаточно. Библиотека SoftwareSerial позволяет использовать для последовательных взаимодействий практически любую пару контактов, хотя и с некоторыми ограничениями.

- С помощью SoftwareSerial невозможно принимать данные одновременно по нескольким портам.
- Если скетч использует таймеры или внешние прерывания, могут возникать проблемы.

Функции в библиотеке имеют те же имена, что и команды Serial, но продуманы лучше. Библиотека SoftwareSerial поддерживает последовательные взаимодействия с устройствами, использующими инвертированные сигналы, такими как дальномеры MaxSonar. Кроме того, создание объектов SoftwareSerial для соединений выполняется более ясным способом, чем стандартный подход с использованием номеров после слова `Serial`.

В табл. 10.2 перечислены контакты на платах Uno и Leonardo, которые может использовать библиотека SoftwareSerial. Если вы работаете с платой, имеющей четыре аппаратных последовательных порта, библиотека SoftwareSerial едва ли вам понадобится. Номера контактов без префикса **A** соответствуют цифровым входам/выходам.

Таблица 10.2. Контакты, доступные библиотеке SoftwareSerial

Модель	Контакты для линии Rx	Контакты для линии Tx
Uno	Любые, кроме 0 и 1	Любые, кроме 0 и 1
Leonardo	Любые, кроме 0 и 1	8, 9, 10, 11, 14 (**MISO**), 15 (**SCK**), 16 (**MOSI**)

При создании объекта SoftwareSerial нужно передать два параметра с номерами

контактов для линий **Rx** и **Tx**. Чтобы запустить взаимодействия, нужно вызвать функцию `begin` и передать ей скорость в бодах:

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX, TX

void setup()
{
  mySerial.begin(9600);
  mySerial.println("Hello, world?");
}
```

Полное описание библиотеки SoftwareSerial можно найти по адресу **http://arduino.cc/en/Reference/SoftwareSerial**[9].

Примеры использования последовательного интерфейса

В этом разделе демонстрируется несколько примеров использования УАПП и библиотеки SoftwareSerial.

Передача из компьютера в Arduino через USB

В первом примере демонстрируется применение монитора последовательного порта для передачи команд в плату Arduino. Раз в секунду Arduino будет посылать значение, прочитанное с аналогового входа **A0**, и одновременно ждать получения односимвольных команд `g` (go — вперед) и `s` (stop — стоп), управляющих передачей прочитанных значений. На рис. 10.3 изображено окно монитора последовательного порта с данными, полученными во время работы скетча.

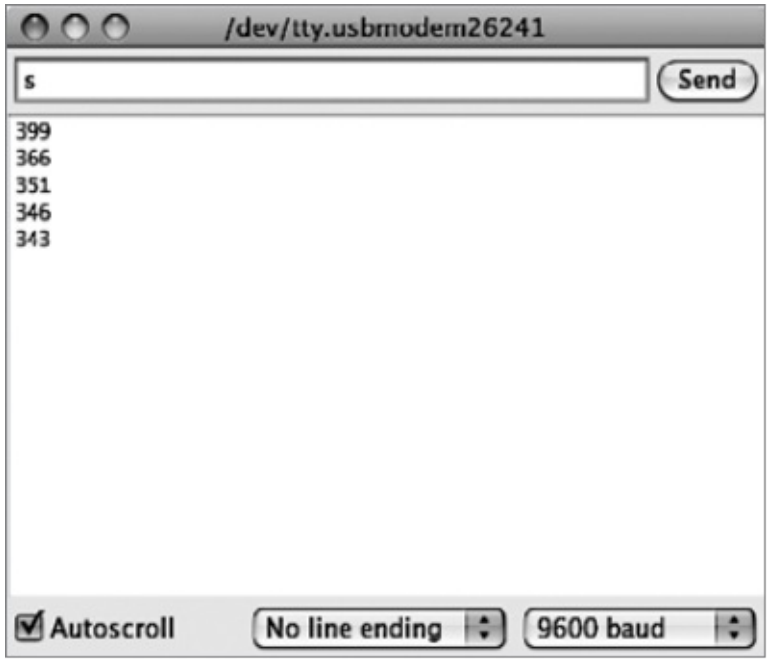


Рис. 10.3. Окно монитора последовательного порта с данными, полученными от платы Arduino

В данном случае из-за того, что вывод производится непосредственно в окно монитора последовательного порта, данные, прочитанные с аналогового входа, передаются не в двоичном, а в текстовом виде.

Далее приводится скетч для этого примера:

```
// sketch_10_01_PC_to_Arduino

const int readingPin = A0;

boolean sendReadings = true;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  if (Serial.available())
  {
    char ch = Serial.read();
    if (ch == 'g')
    {
      sendReadings = true;
    }
```

```
    else if (ch == 's')
    {
      sendReadings = false;
    }
  }
  if (sendReadings)
  {
    int reading = analogRead(readingPin);
    Serial.println(reading);
    delay(1000);
  }
}
```

Функция `loop` проверяет получение данных и, если они имеются, читает их по одному байту как символы. После полученный байт сравнивается с командами `'s'` и `'g'`, и переменной `sendReadings` присваивается соответствующее значение.

Затем по значению переменной `sendReadings` определяется необходимость чтения аналогового входа и вывода результатов. Если флаг `sendReadings` имеет значение `true`, перед отправкой следующего значения выполняется задержка на одну секунду.

Использование функции `delay` означает, что значение `sendReadings` сможет измениться только в следующей итерации функции `loop`. В данном скетче это не является проблемой, но в других ситуациях может потребоваться использовать другое решение, не блокирующее работу функции `loop`. Подробнее о подобных решениях рассказывается в главе 14.

Передача из Arduino в Arduino

Второй пример иллюстрирует передачу данных из одной платы Arduino Uno в другую. В данном случае одна плата Arduino передает значение, прочитанное с входа **A1**, другой плате, которая затем по этому значению определяет частоту мигания встроенного светодиода **L**.

На рис. 10.4 изображена схема соединения плат.

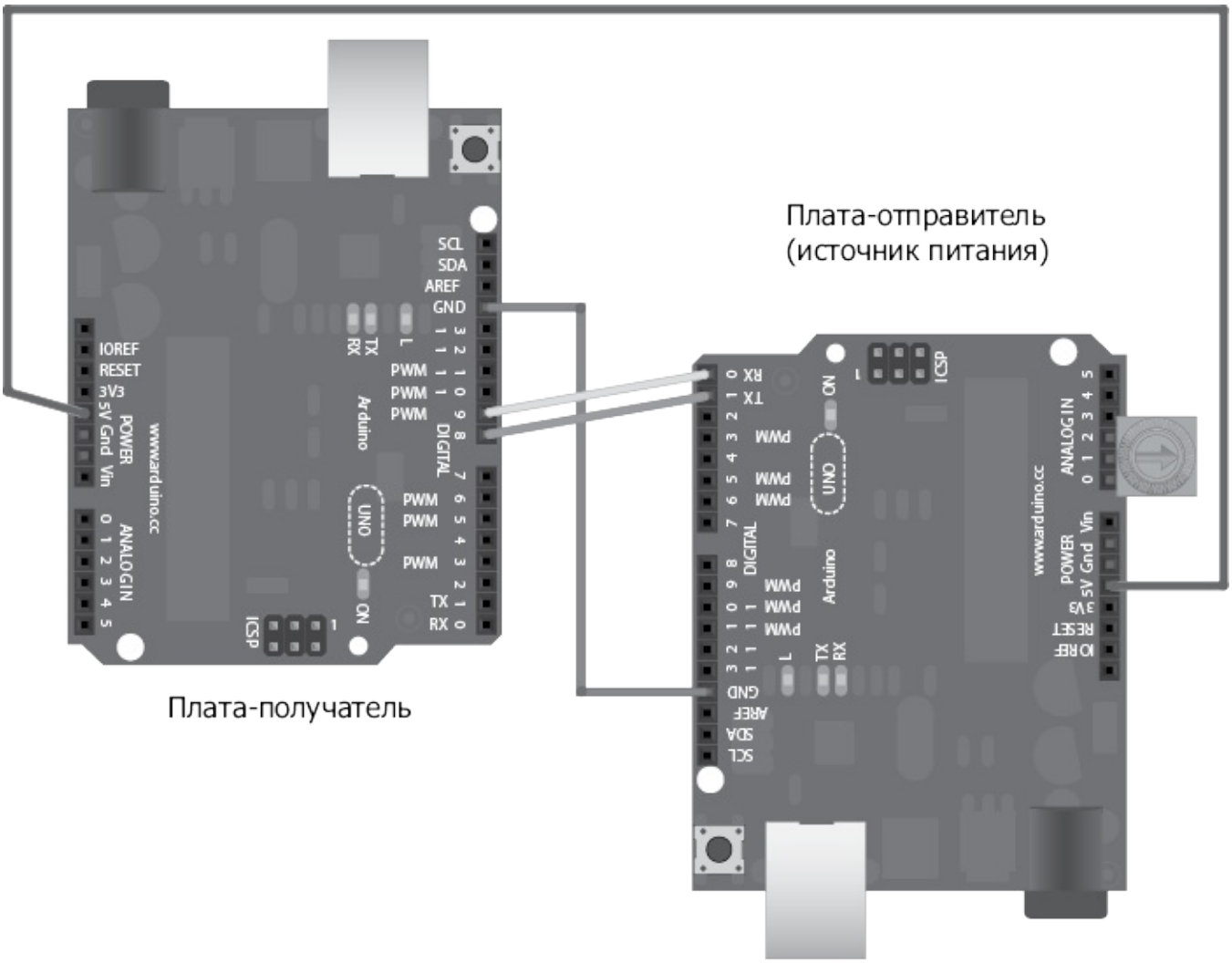


Рис. 10.4. Две платы Ardiono Uno взаимодействуют через последовательный порт

Контакт **Tx** на одной плате Arduino должен быть подключен к контакту **Rx** на другой и наоборот. В этом примере обе платы используют библиотеку SoftwareSerial, контакты **8** служат линией **Rx**, а контакты **9** — линией **Tx**.

Контакты GND обеих плат должны быть соединены. Чтобы плата-отправитель могла служить источником питания для платы-получателя, необходимо также соединить их контакты **5V**. К плате-отправителю подключено переменное сопротивление, соединяющее гнезда **A0** и **A2**. Настроив контакты **A0** и **A2** на работу в режиме цифровых выходов и установив на выходе **A2** уровень `HIGH`, можно изменять уровень напряжения на контакте **A1** в диапазоне от 0 до 5 В, вращая шток резистора, и тем самым управлять частотой мигания светодиода на другой плате Arduino.

Далее приводится скетч для платы-отправителя:

```
// sketch_10_02_Adruino_Sender

#include "SoftwareSerial.h"

const int readingPin = A1;
const int plusPin = A2;
```

```
const int gndPin = A0;

SoftwareSerial sender(8, 9); // RX, TX

void setup()
{
  pinMode(gndPin, OUTPUT);
  pinMode(plusPin, OUTPUT);
  digitalWrite(plusPin, HIGH);
  sender.begin(9600);
}

void loop()
{
  int reading = analogRead(readingPin);
  byte h = highByte(reading);
  byte l = lowByte(reading);
  sender.write(h);
  sender.write(l);
  delay(1000);
}
```

Перед отправкой прочитанное 16-битное значение (`int`) разбивается на старший и младший байты, затем оба байта посылаются в последовательный интерфейс командой `write`. Команды `print` и `println` преобразуют свой аргумент в строку, но команда `write` посылает байт в двоичном виде.

Далее приводится скетч для платы-получателя:

```
// sketch_10_03_Adruino_Receiver

#include "SoftwareSerial.h"

const int ledPin = 13;
int reading = 0;
SoftwareSerial receiver(8, 9); // RX, TX

void setup()
{
  pinMode(ledPin, OUTPUT);
```

```
  receiver.begin(9600);
}

void loop()
{
  if (receiver.available() > 1)
  {
    byte h = receiver.read();
    byte l = receiver.read();
    reading = (h << 8) + l;
  }
  flash(reading);
}

void flash(int rate)
{
  // 0 – редко, 1023 – очень часто
  int period = (50 + (1023 – rate) / 4);
  digitalWrite(ledPin, HIGH);
  delay(period);
  digitalWrite(ledPin, LOW);
  delay(period);
}
```

Принимающий скетч ждет, пока будет получено не менее 2 байт, и затем восстанавливает значение типа `int`, сдвигая старший байт на 8 бит влево и прибавляя к результату младший байт.

Для передачи данных с более сложной структурой можно использовать библиотеку EasyTransfer: **www.billporter.info/2011/05/30/easytransfer-arduino-library/**.

Несмотря на то что в этом примере контакт **Tx** одной платы Arduino связан проводом с контактом **Rx** другой, для взаимодействий точно так же можно использовать беспроводные соединения. Многие модули беспроводной связи действуют прозрачно, то есть как если бы связь осуществлялась по проводам.

Модуль GPS

В заключительном примере демонстрируется использование последовательного интерфейса ТТЛ для чтения географических координат (широты и долготы) из модуля глобальной навигационной системы (Global Positioning System, GPS), которые затем форматируются и выводятся в монитор последовательного порта (рис. 10.5).

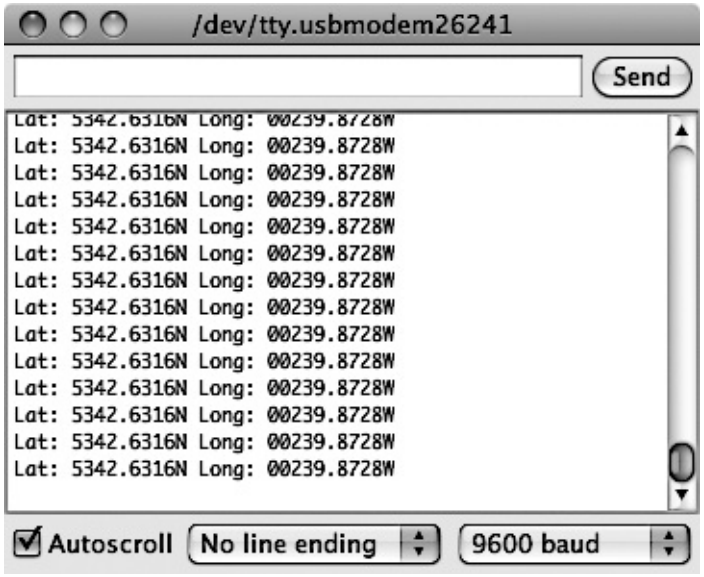


Рис. 10.5. Чтение данных из модуля GPS в плату Arduino

Связь с модулем GPS возможна только в одну сторону, поэтому достаточно соединить вывод **Tx** модуля с выводом **Rx** на плате Arduino. В примере используется модуль GPS, выпускаемый компанией Sparkfun Venus (www.sparkfun.com/products/11058). Подобно большинству других модулей GPS, он имеет последовательный интерфейс ТТЛ и раз в секунду посылает сообщения на скорости 9600 бод.

Формат сообщений соответствует стандарту национальной ассоциации морской электроники (National Marine Electronics Association, NMEA). Сообщение — это текстовая строка, завершающаяся символом перевода строки, с полями, разделенными запятыми. Далее показано, как выглядит типичное сообщение:

```
$GPRMC,081019.548,A,5342.6316,N,00239.8728,W,000.0,079.7,110613,,,A
```

Поля в данном примере имеют следующие значения:

- `$GPRMC` — тип сообщения;
- `081019.548` — время (очень точное) в 24-часовом формате, 8:10:19.548;
- `5342.6316, N` — широта, умноженная на 100, то есть 53,426316 градуса северной широты;
- `00239.8728,W` — долгота, умноженная на 100, то есть 0,2398728 градуса западной долготы;
- `000.0` — скорость;
- `079.7` — курс 79,7 градуса;
- `110613` — дата, 11 июня 2013.

Остальные поля для данного примера не имеют значения.

ПРИМЕЧАНИЕ

Полный список сообщений NMEA GPS можно найти по адресу http://aprs.gids.nl/nmea/.

Далее приводится скетч для этого примера:

```
#include <SoftwareSerial.h>

SoftwareSerial gpsSerial(10, 11); // RX, TX (TX не используется)
const int sentenceSize = 80;

char sentence[sentenceSize];

void setup()
{
  Serial.begin(9600);
  gpsSerial.begin(9600);
}

void loop()
{
  static int i = 0;
  if (gpsSerial.available())
  {
    char ch = gpsSerial.read();
    if (ch != '\n' && i < sentenceSize)
    {
      sentence[i] = ch;
      i++;
    }
    else
    {
     sentence[i] = '\0';
     i = 0;
     //Serial.println(sentence);
     displayGPS();
    }
```

```
  }
}

void displayGPS()
{
  char field[20];
  getField(field, 0);
  if (strcmp(field, "$GPRMC") == 0)
  {
    Serial.print("Lat: ");
    getField(field, 3);  // число
    Serial.print(field);
    getField(field, 4); // широта N/S
    Serial.print(field);

    Serial.print(" Long: ");
    getField(field, 5);  // число
    Serial.print(field);
    getField(field, 6);  // долгота E/W
    Serial.println(field);
  }
}

void getField(char* buffer, int index)
{
  int sentencePos = 0;
  int fieldPos = 0;
  int commaCount = 0;
  while (sentencePos < sentenceSize)
  {
    if (sentence[sentencePos] == ',')
    {
      commaCount ++;
      sentencePos ++;
    }
    if (commaCount == index)
    {
      buffer[fieldPos] = sentence[sentencePos];
      fieldPos ++;
```

```
    }
    sentencePos ++;
  }
  buffer[fieldPos] = '\0';
}
```

Сообщения, посылаемые модулем GPS, имеют разную длину, но не более 80 символов, поэтому в скетче используется буфер емкостью 80 символов, который заполняется данными, пока не заполнится или не будет получен признак конца строки.

После того как полное сообщение будет прочитано, в конец буфера вставляется нулевой символ, завершающий строки в языке C. Это необходимо, только если вы пожелаете «напечатать» сообщение в его исходном виде.

Остальная часть скетча реализует извлечение полей и форматирование выходной строки для записи в монитор последовательного порта. Функция `getField` извлекает текст из поля с указанным индексом.

Функция `displayGPS` игнорирует любые сообщения, тип которых отличается от `"$GPRMC"`, и извлекает широту и долготу, а также односимвольные названия полушарий для отображения.

В заключение

В этой главе мы исследовали несколько способов программирования взаимодействий через последовательный интерфейс между платами Arduino, периферийными устройствами и компьютерами.

В следующей главе мы обратим внимание на одну интересную особенность Arduino Leonardo, которая позволяет имитировать поведение периферийных устройств USB, таких как клавиатура и мышь. А также рассмотрим некоторые аспекты программирования интерфейса USB.

8 Похожее описание на русском языке: **http://arduino.ru/Reference/Serial**. — *Примеч. пер.*

9 Описание на русском языке: **http://arduino.ua/ru/prog/SoftwareSerial**. — *Примеч. пер.*

11. Программирование интерфейса USB

В этой главе рассматриваются разные аспекты использования порта USB на плате Arduino. В том числе возможность эмуляции клавиатуры и мыши, поддерживаемой платой Arduino Leonardo, а также подключения клавиатуры или мыши к соответственно оборудованной плате Arduino.

Эмуляция клавиатуры и мыши

Три модели Arduino — Due, Leonardo и Micro (основанная на модели Leonardo) — позволяют использовать их порт USB для эмуляции клавиатуры или мыши. Существуют также Arduino-совместимые платы, такие как LeoStick от компании Freetronics (рис. 11.1), поддерживающие аналогичную возможность.

Эта возможность широко используется, например, с музыкальными контроллерами, что позволяет плате Arduino взаимодействовать с музыкальными синтезаторами и управляющими программами, такими как Ableton Live. Благодаря ей можно, к примеру, на основе Arduino создавать новые музыкальные инструменты, управляющие музыкальным программным обеспечением с помощью датчиков поворота, прерываемых лучей света или педальной клавиатуры. К несерьезным применениям этой возможности можно отнести устройства, которые создают впечатление, что мышь живет

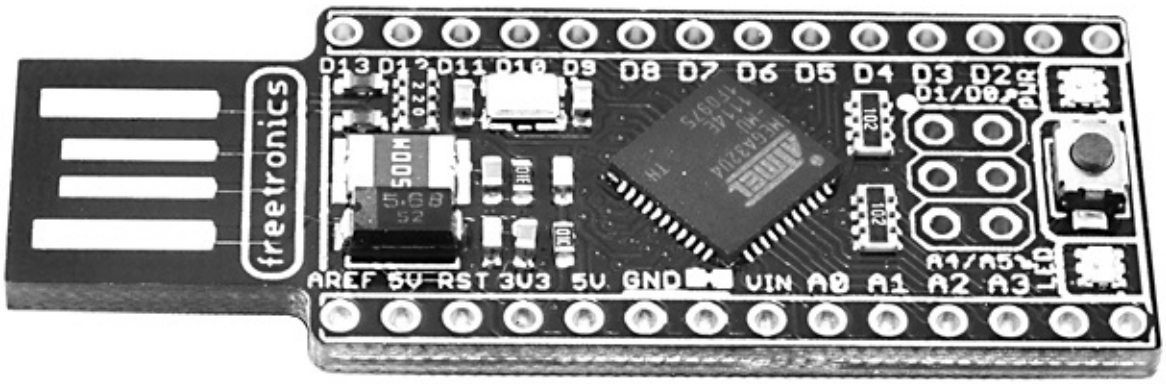


Рис. 11.1. Плата LeoStick

своей жизнью независимо от действий пользователя или клавиатура сама печатает случайные символы.

Модель Arduino Due имеет два порта USB. Эмуляция клавиатуры и мыши осуществляется через *локальный порт USB*, а программирование Arduino Due — через *порт USB для программирования* (рис. 11.2).

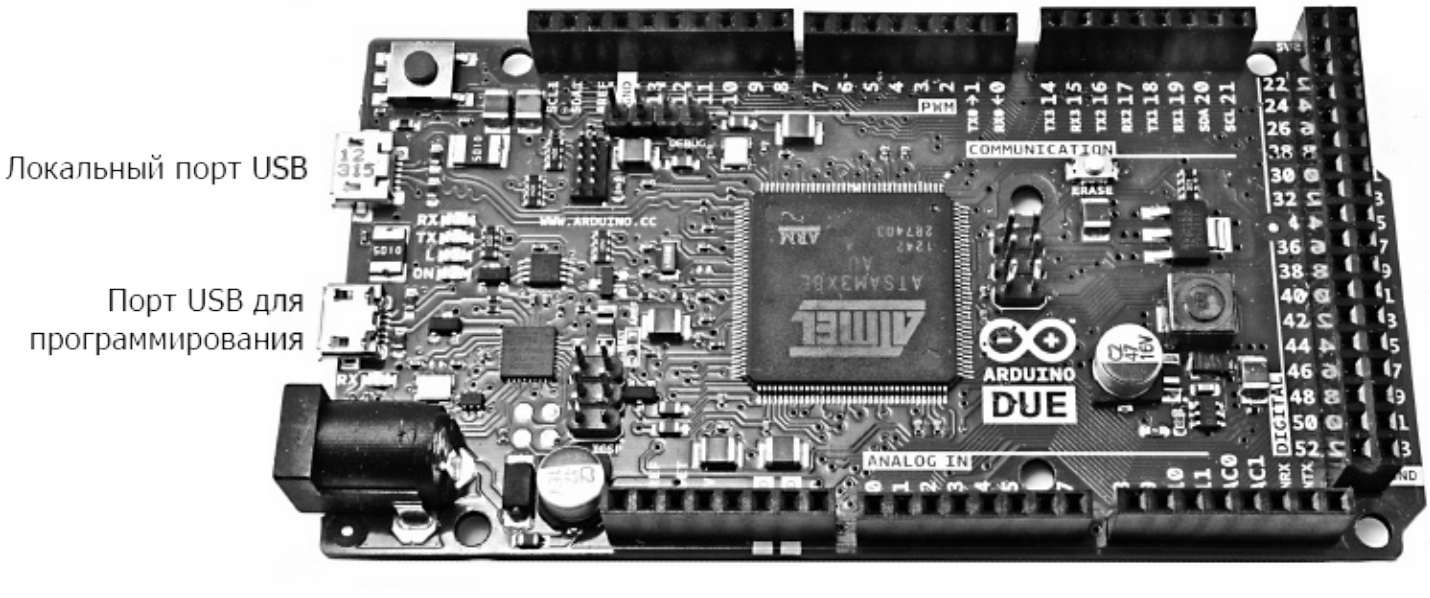


Рис. 11.2. Плата Arduino Due с двумя портами USB

Эмуляция клавиатуры

Функции эмуляции клавиатуры просты в использовании. Они входят в состав стандартной библиотеки языка, поэтому для доступа к ним не требуется подключать дополнительные библиотеки. Чтобы включить режим эмуляции клавиатуры, добавьте в функцию `setup` следующую команду:

```
Keyboard.begin();
```

Чтобы заставить Arduino печатать что-нибудь, можно воспользоваться командами `print` и `println`, и переданный им текст появится в позиции текстового курсора:

```
Keyboard.println("It was the best of times.");
```

Чтобы сымитировать нажатие клавиш-модификаторов, например ввести комбинацию **CTRL+C**, используйте команду `press`:

```
Keyboard.press(KEY_LEFT_CTRL);
Keyboard.press('x');
delay(100);
Keyboard.releasrAll();
```

Команда `press` имеет односимвольный параметр, в котором ей можно передавать любые обычные символы или предопределенные константы, такие как `KEY_LEFT_CTRL`. После вызова команды `press` плата будет имитировать удержание клавиши в нажатом состоянии, пока не будет вызвана команда `releaseAll`. Полный список специальных клавиш можно найти по адресу http://arduino.cc/en/Reference/KeyboardModifiers.

ПРИМЕЧАНИЕ

При использовании функций эмуляции клавиатуры и мыши можно столкнуться со сложностями при программировании платы, так как она будет пытаться вводить текст, пока вы пытаетесь запрограммировать ее. Чтобы преодолеть эту проблему, примените следующий трюк: нажмите и удерживайте нажатой кнопку сброса Reset и отпустите ее, только когда в строке состояния Arduino IDE появится сообщение «uploading» («загрузка»).

Пример эмуляции клавиатуры

Следующий пример автоматически вводит текст по вашему выбору (например, пароль) после каждого сброса платы Arduino:

```
// sketch_11_01_keyboard

char phrase[] = "secretpassword";

void setup()
{
  Keyboard.begin();
  delay(5000);
  Keyboard.println(phrase);
}

void loop()
{
}
```

Этот пример можно сделать эффективнее, если активировать ввод нажатием внешней клавиши: если вы пользуетесь компьютером Mac, операционная система будет думать, что в момент сброса платы к компьютеру подключается новая клавиатура, и выведет системный диалог, который нужно успеть закрыть до того, как плата напечатает текст.

Эмуляция мыши

Эмуляция мыши реализуется с применением того же шаблона, что и эмуляция клавиатуры. Вообще говоря, нет никаких причин, препятствующих эмуляции обоих устройств ввода в одном скетче.

Чтобы запустить эмуляцию, прежде всего следует выполнить команду

```
Mouse.begin();
```

Управление указателем мыши осуществляется командой `Mouse.move`. Она имеет три параметра: смещение по осям *x* и *y* и поворот колесика. Все три параметра измеряются в пикселах. Эти значения могут быть положительными (смещение указателя вправо или вниз) или отрицательными (смещение указателя влево или вверх). Смещения откладываются относительно текущей позиции указателя, а так как нет никакой возможности указать абсолютные координаты указателя, эта команда лишь имитирует поведение мыши, перемещающей указатель, но не управляет самим указателем.

Сымитировать щелчок мышью можно с помощью команды `click`. Без параметров эта команда имитирует щелчок левой кнопкой мыши. При желании ей можно передать аргумент `MOUSE_RIGHT` или `MOUSE_MIDDLE`.

Для управления длительностью щелчка можно использовать команды `Mouse.press` и `Mouse.release`. Команда `Mouse.press` принимает те же необязательные аргументы, что и команда `Mouse.click`. Эти две команды могут пригодиться, например, чтобы заставить свою «мышь» на основе Arduino выполнять щелчок при изменении состояния цифрового входа на плате. Кроме того, с их помощью можно сымитировать двойной или даже тройной щелчок.

Пример эмуляции мыши

Следующий пример перемещает указатель мыши по экрану в случайных направлениях. Чтобы прервать программу и восстановить управление компьютером, нажмите и удерживайте кнопку сброса **Reset** или просто отсоедините плату от компьютера.

```
// sketch_11_02_mouse

void setup()
{
  Mouse.begin();
}

void loop()
}
  int x=random(61)-30;
  int y=random(61)-30;
  Mouse.move(x, y);
  delay(50);
}
```

Программирование хоста USB

Модели Leonardo, Due и Micro имеют возможность действовать как клавиатура или мышь, но только Due и менее известная модель Arduino Mega ADK позволяют подключать клавиатуру или мышь и использовать их в качестве устройств ввода. Эта особенность называется *хостом USB* (USB Host). Непосредственная поддержка этой особенности реализована только в модели Due, тем не менее существуют платы сторонних производителей, которые можно подключать к Arduino Uno или Leonardo, чтобы получить собственный хост USB.

Более того, если у вас имеется беспроводная клавиатура или мышь (за исключением моделей, подключаемых через Bluetooth), они также должны работать, если включить адаптер в разъем порта USB на плате расширения. Таким способом можно организовать беспроводное удаленное управление платой Arduino.

Хост USB позволяет подключать не только клавиатуру и мышь, но и множество других периферийных устройств USB: игровые джойстики, камеры, Bluetooth-устройства и даже телефоны на платформе Android.

Плата хоста USB и библиотека

Плата хоста USB и сопутствующие библиотеки созданы довольно давно и в настоящее время поддерживают широкий диапазон периферийных устройств. Первая плата хоста USB была разработана в Circuits@home (**www.circuitsathome.com**). Нынче доступны совместимые платы USB, производимые компаниями Sparkfun, SainSmart и, возможно, другими. На рис. 11.3 изображена плата Sparkfun USB Host, подключенная к Arduino Uno. Обратите внимание на то, что на момент написания данных строк эти платы были несовместимы с Arduino Leonardo, а также с другими моделями, более экзотичными, чем Uno. Поэтому перед приобретением убедитесь в совместимости плат.

Рис. 11.3. Плата Sparkfun USB Host

Данная конкретная плата имеет область, удобную для макетирования, где можно спаять дополнительные компоненты. Альтернативой платам расширения может служить плата Freetronics USBDroid (рис. 11.4). На ней имеются два порта USB: порт микроUSB для программирования и полноразмерный разъем USB для подключения клавиатуры и подобных устройств.

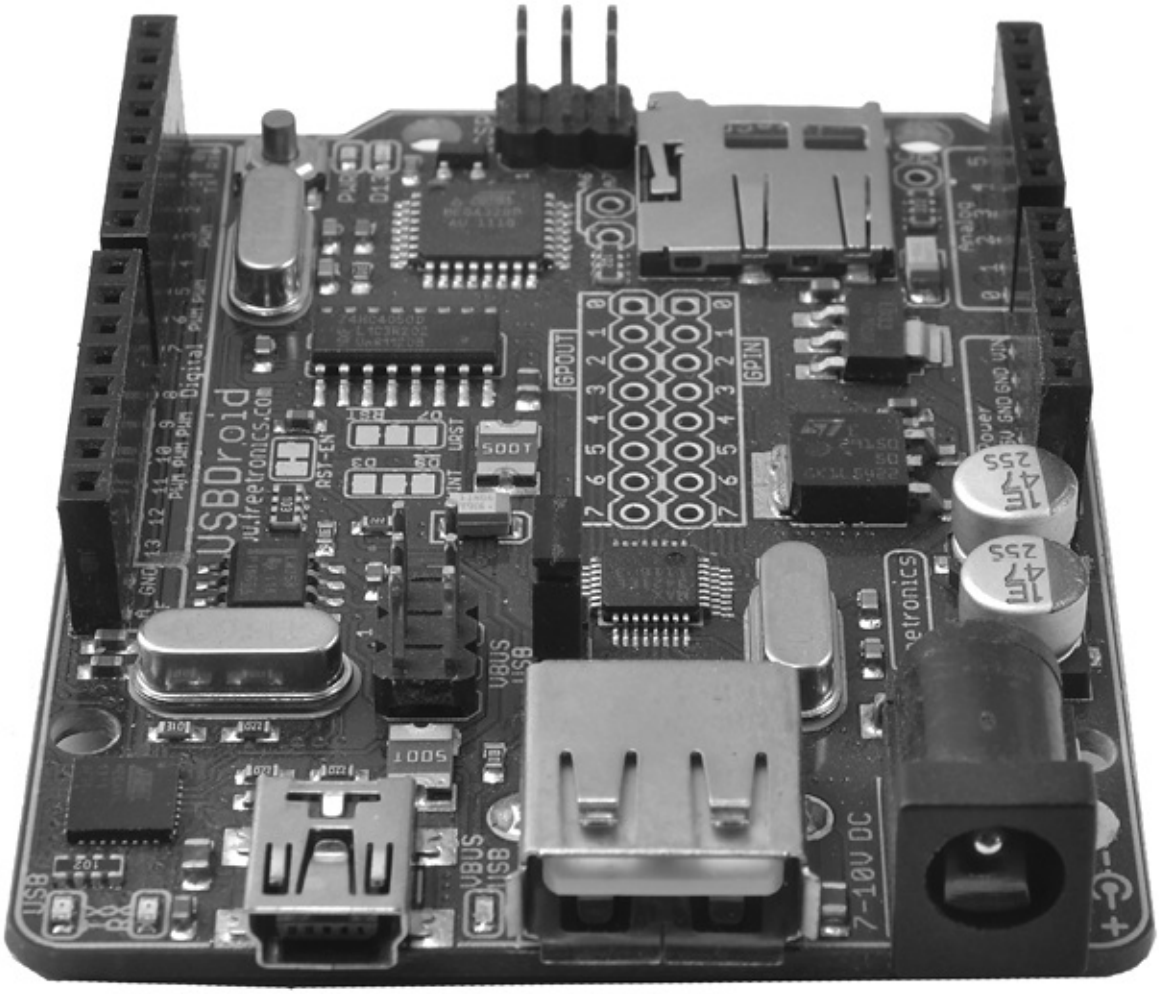


Рис. 11.4. Плата Freetronics USBDroid

При работе с USBDroid или неофициальными моделями плат хостов USB следует использовать оригинальную библиотеку USB_Host_Shield от Circuits@Home. При работе с официальной платой следует использовать библиотеку USB_Host_Shield_2, поддерживающую более широкий спектр устройств.

Программирование интерфейса USB с применением упомянутых библиотек — не самая простая задача. Библиотеки предоставляют крайне низкоуровневый доступ к шине USB. На веб-сайте автора (**www.simonmonk.org**) можно найти пример скетча `sketch_11_03_host_keyboard`, реализующего подключение клавиатуры с использованием платы хоста USB.

Этот скетч является адаптацией одного из примеров, сопровождающих библиотеку USB_Host_Shield. В нем, в отличие от оригинала, информация о нажатых клавишах выводится в монитор последовательного порта вместо жидкокристаллического дисплея.

Этот скетч (и пример, на котором он основан) может служить отличным шаблоном для создания собственных скетчей, так как оба они корректно обрабатывают нажатия всех клавиш. Если вас интересуют только цифровые клавиши и клавиши управления курсором, вы можете существенно упростить свой скетч.

Скетч слишком длинный, чтобы привести его здесь целиком, поэтому я покажу только наиболее важные фрагменты. Возможно, вам будет полезно загрузить этот

скетч и заглядывать в него, читая описание в книге.

Скетч импортирует три библиотеки:

```
#include <Spi.h>
#include <Max3421e.h>
#include <Usb.h>
```

Библиотека Spi.h необходима для взаимодействий с контроллером хоста USB. В роли контроллера используется микросхема **Max3421e**, поэтому следует импортировать одноименную библиотеку. И наконец, нужно подключить еще одну библиотеку (Usb.h), основанную на библиотеке Max3421e.h, которая скрывает некоторые сложности выполнения операций с контроллером.

За командами импортирования библиотек следуют определения констант, например:

```
#define BANG        (0x1E)
```

Это просто еще один способ определения констант в C. Данную константу можно было бы определить иначе:

```
cons tint BANG = 0x1E;
```

Далее создаются объекты типов MAX3421E и USB, и в функции setup вызывается функция powerOn объекта Max:

```
MAX3421E Max;
USB Usb;
```

В функции loop вызываются функции Task обоих объектов, Max и Usb. Они проверяют готовность интерфейса USB.

```
void loop() {
    Max.Task();
    Usb.Task();
      if( Usb.getUsbTaskState() == USB_STATE_CONFIGURING ) {   //
ждать завершения настройки
        kbd_init();
        Usb.setUsbTaskState( USB_STATE_RUNNING );
    }
       if( Usb.getUsbTaskState() == USB_STATE_RUNNING ) {   //
опросить клавиатуру
        kbd_poll();
    }
```

```
}
```

При первом запуске интерфейс USB переходит в состояние настройки USB_STATE_CONFIGURING, в котором находится, пока с помощью kbd_init не будет установлено соединение с клавиатурой. Эта функция использует структуру записи конечной точки (ep_record), куда помещаются части сообщения, необходимого для установки соединения с клавиатурой:

```
ep_record[ 0 ] = *( Usb.getDevTableEntry( 0,0 ));
ep_record[ 1 ].MaxPktSize = EP_MAXPKTSIZE;
ep_record[ 1 ].Interval  = EP_POLL;
ep_record[ 1 ].sndToggle = bmSNDTOG0;
ep_record[ 1 ].rcvToggle = bmRCVTOG0;
Usb.setDevTableEntry( 1, ep_record );
/* Настроить устройство */
rcode = Usb.setConf( KBD_ADDR, 0, 1 );
```

После инициализации, вероятнее всего, клавиатура перейдет в состояние нормального функционирования (USB_STATE_RUNNING), обнаружив которое скетч вызовет kbd_poll для проверки нажатия клавиши на клавиатуре.

Ключевая строка в kbd_poll

```
rcode = Usb.inTransfer( KBD_ADDR, KBD_EP, 8, buf );
```

читает скан-код клавиши, чтобы определить, была ли нажата какая-нибудь клавиша. Этот код не является кодом ASCII. Преобразование скан-кодов в коды ASCII осуществляется в функции HIDtoA. Эта функция — самая сложная в скетче, но вы можете просто копировать ее в свои скетчи, не вдаваясь в детали ее работы. Список скан-кодов и порядок их преобразования в коды ASCII можно найти по адресу **www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html**.

Одной из интересных особенностей протокола USB-устройств для взаимодействия с человеком (Human Interface Device, HID), используемого для работы с клавиатурой, является возможность управления светодиодными индикаторами **Scroll Lock**, **Caps Lock** и **Num Lock**. Функция kbd_poll включает и выключает эти индикаторы в ответ на нажатия клавиш **Scroll Lock**, **Caps Lock** и **Num Lock**, однако вы можете написать коротенький скетч, например sketch_11_04_host_scroll_lock, который просто мигает светодиодными индикаторами на клавиатуре.

Ключевая функция в этом скетче:

```
void toggleLEDs( void )
{
  if (leds == 0) {
```

```
      leds = 0b00000111;
    }
    else {
      leds = 0;
    }
    Usb.setReport( KBD_ADDR, 0, 1, KBD_IF, 0x02, 0, &leds );
  }
```

Три младших бита в байте — это три флага, управляющих состояниями индикаторов на клавиатуре.

Хост USB на плате Arduino Due

Плата Arduino Due способна действовать как встроенный хост USB. На момент написания этих строк данная возможность рассматривалась командой разработчиков Arduino как экспериментальная. Загляните в официальную документацию Arduino (**http://arduino.cc/en/Reference/USBHost**), чтобы определить текущее состояние дел в этой области и выяснить, произошли ли изменения в способе ее использования.

Модель Due не имеет полноразмерного разъема USB, к которому можно было бы подключить клавиатуру или мышь. Для подключения таких устройств требуется использовать кабель Micro USB OTG Host Cable (рис. 11.5). На этом снимке к плате Arduino Due подключен адаптер беспроводной клавиатуры, но точно так же можно подключить обычную клавиатуру USB.

Библиотеки поддержки USB в Arduino Due имеют более простой интерфейс, чем библиотека обслуживания хоста USB, и вместо скан-кодов

Рис. 11.5. Arduino Due с подключенным кабелем Micro USB OTG Host Cable и клавиатурой

возвращают коды ASCII нажатых клавиш. Следующий пример демонстрирует взаимодействие с клавиатурой. Он просто выводит в монитор последовательного порта символы, соответствующие нажатым клавишам:

```
// sketch_11_05_keyboard_due

#include <KeyboardController.h>

USBHost usb;
KeyboardController keyboard(usb);

void setup()
{
  Serial.begin(9600);
  Serial.println("Program started");
  delay(200);
}

void loop()
{
  usb.Task();
}

// Эта функция обрабатывает нажатия клавиш
void keyPressed()
{
  char key = keyboard.getKey();
  Serial.write(key);
}
```

Библиотека KeyboardController вызывает функцию `keyPressed` в скетче каждый раз, когда происходит нажатие какой-либо клавиши. Аналогично можно перехватывать отпускания клавиш с помощью функции `keyReleased`. Чтобы определить нажатую клавишу, необходимо вызвать следующие функции объекта `keyboard`:

- `getModifiers` — возвращает битовую маску с информацией об удерживаемых нажатыми клавишах-модификаторах (**Shift**, **Ctrl** и т.д.). Коды клавиш модификаторов можно найти по адресу http://arduino.cc/en/Reference/GetModifiers;
- `getKey` — возвращает код ASCII текущей нажатой клавиши;
- `getOemKey` — возвращает скан-код клавиши.

Взаимодействие с мышью осуществляется так же просто и с применением похожего шаблона. Следующий пример выводит буквы L, R, U или D в зависимости от направления перемещения указателя мыши — влево, вправо, вверх или вниз:

```
// sketch_11_06_mouse_due

#include <MouseController.h>

USBHost usb;
MouseController mouse(usb);

void setup()
{
  Serial.begin(9600);
  Serial.println("Program started");
  delay(200);
}

void loop()
{
  usb.Task();
}

// Эта функция обрабатывает перемещения мыши
void mouseMoved()
{
  int x = mouse.getXChange();
  int y = mouse.getYChange();
  if (x > 50) Serial.print("R");
  if (x < -50) Serial.print("L");
  if (y > 50) Serial.print("D");
  if (y < -50) Serial.print("U");
}
```

Помимо `mouseMoved` в скетч можно добавить следующие функции для обработки других событий от мыши:

- `mouseDragged` — это событие генерируется, когда происходит перемещение мыши с нажатой левой кнопкой;

- `mousePressed` — это событие генерируется, когда происходит нажатие кнопки мыши, и должно сопровождаться вызовом `mouse.getButton` с аргументом `LEFT_BUTTON`, `RIGHT_BUTTON` или `MIDDLE_BUTTON` для определения нажатой кнопки — возвращает `true`, если была нажата кнопка, соответствующая аргументу;

- `mouseReleased` — это событие является парным для `mousePressed` и генерируется, когда происходит отпускание кнопки.

В заключение

В этой главе мы познакомились с несколькими способами использования Arduino с устройствами USB.

В следующей главе посмотрим, как подключать Arduino к проводным и беспроводным сетям, познакомимся с некоторыми приемами программирования сетевых взаимодействий, а также попробуем использовать платы расширения Ethernet и WiFi.

12. Программирование сетевых взаимодействий

Область Интернета, которую часто называют *Интернетом вещей*, простирается далеко за рамки браузеров и веб-серверов и широко используется аппаратными устройствами, обладающими поддержкой сетевых взаимодействий. Принтеры, устройства бытовой автоматики и даже холодильники не только становятся все более интеллектуальными, но и поддерживают возможность подключения к Интернету. Платы Arduino занимают передовые позиции среди самодельных интернет-устройств, поддерживая возможность проводного или беспроводного подключения к Интернету с помощью плат расширения Etherner или WiFi. В этой главе мы посмотрим, как запрограммировать Arduino, чтобы задействовать имеющееся подключение к сети.

Сетевое оборудование

У вас на выбор есть несколько вариантов подключения Arduino к сети. Можно использовать плату расширения Ethernet с Arduino Uno, или приобрести модель Arduino со встроенным адаптером Ethernet, или раскошелиться и приобрести плату расширения WiFi для подключения к беспроводной сети.

Плата расширения Ethernet

Плата расширения Ethernet (рис. 12.1) не только дает возможность подключения к сети Ethernet, но и имеет слот для карты памяти microSD, которую можно использовать для хранения данных (см. раздел «Использование SD-карты» главы 6).

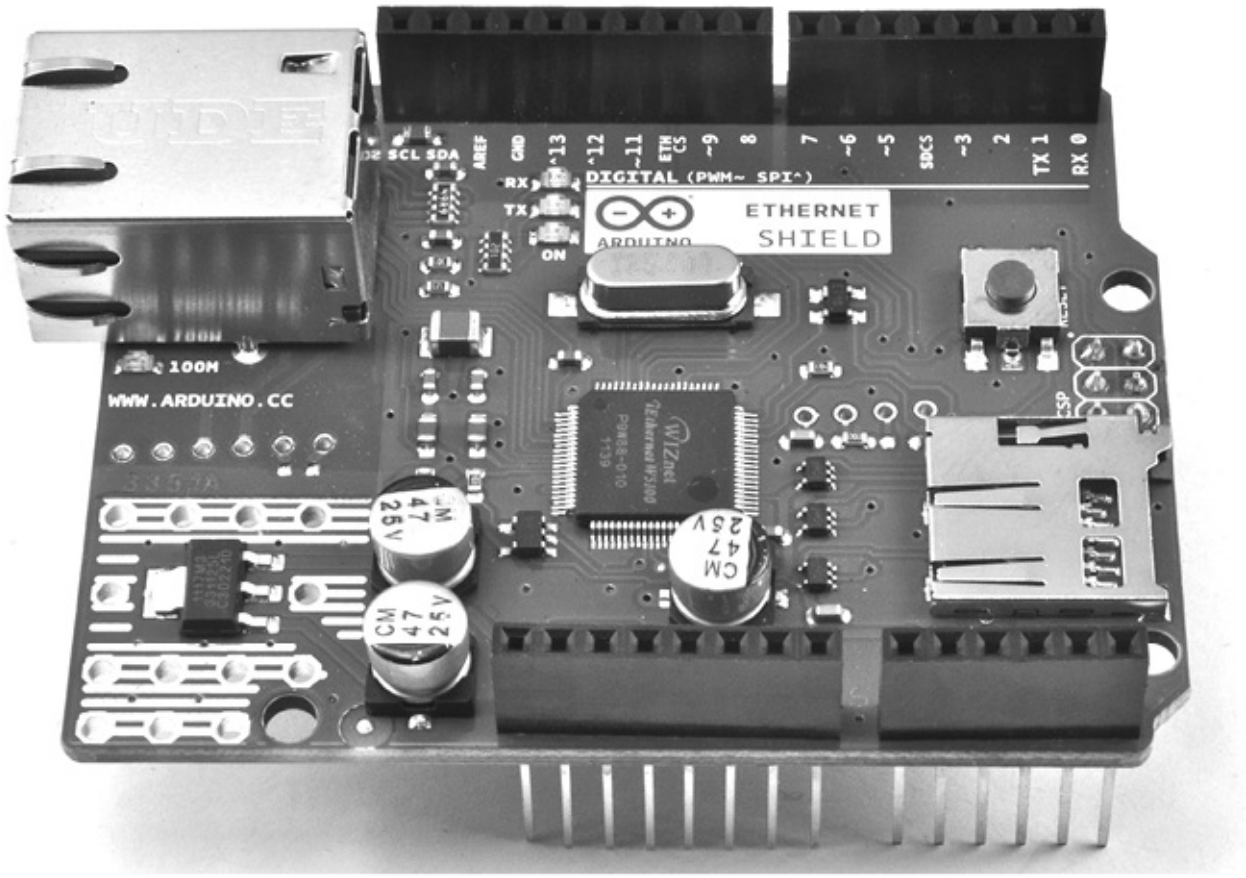


Рис. 12.1. Плата расширения Ethernet

На официальных платах используется микросхема W5100, можно найти более дешевые платы расширения Ethernet на наборе микросхем ENC28J60. Но эти более дешевые платы не совместимы с библиотекой Ethernet, и лучше избегать их, если только вы не ограничены во времени или в средствах.

Arduino Ethernet/EtherTen

Альтернативой использованию отдельной платы расширения является покупка Arduino со встроенным адаптером Ethernet. Официальной считается модель Arduino Ethernet, однако в продаже имеется очень неплохая и совместимая с Uno плата EtherTen, производимая компанией Freetronics (www.freetronics.com) (рис. 12.2).

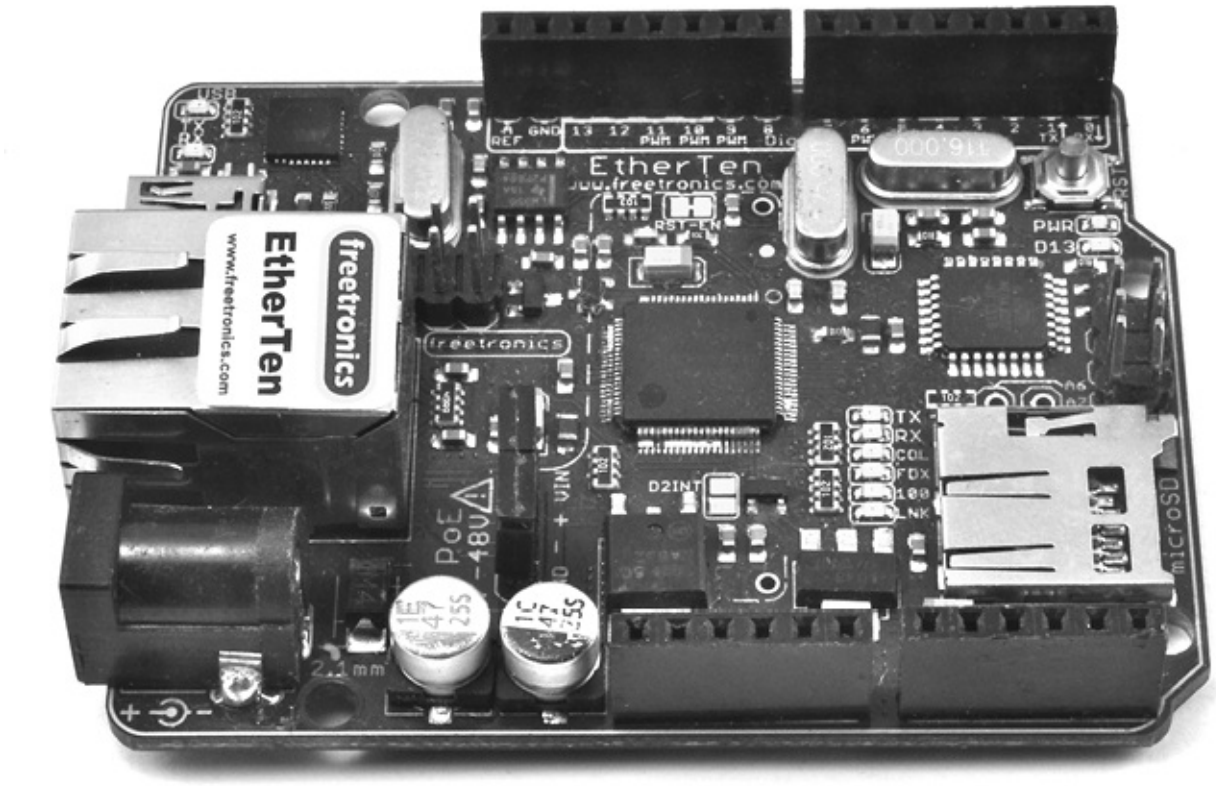


Рис. 12.2. Плата EtherTen

Комбинированные платы, содержащие все необходимое, наиболее предпочтительны для сетевых проектов на основе Arduino. Платы Arduino Ethernet поддерживают технологию питания по линиям Ethernet (Power over Ethernet, PoE) через отдельный инжектор PoE, что позволяет уменьшить количество проводов, идущих к плате Arduino, до единственного кабеля Ethernet. Платы EtherTen выпускаются уже настроенными на питание с использованием технологии PoE. Более полную информацию об использовании технологии PoE в платах EtherTen можно найти по адресу **www.doctormonk.com/2012/01/power-over-ethernet-poe.html**.

Arduino и WiFi

Главная проблема подключения к Интернету через Ethernet заключается в необходимости прокладки кабеля. Если вы хотите подключить Arduino к Интернету или сети без использования проводов, то вам потребуется плата расширения WiFi (рис. 12.3). Эти платы стоят довольно дорого, но есть более дешевые альтернативы сторонних производителей, такие как Sparkfun WiFly (**https://www.sparkfun.com/products/9954**).

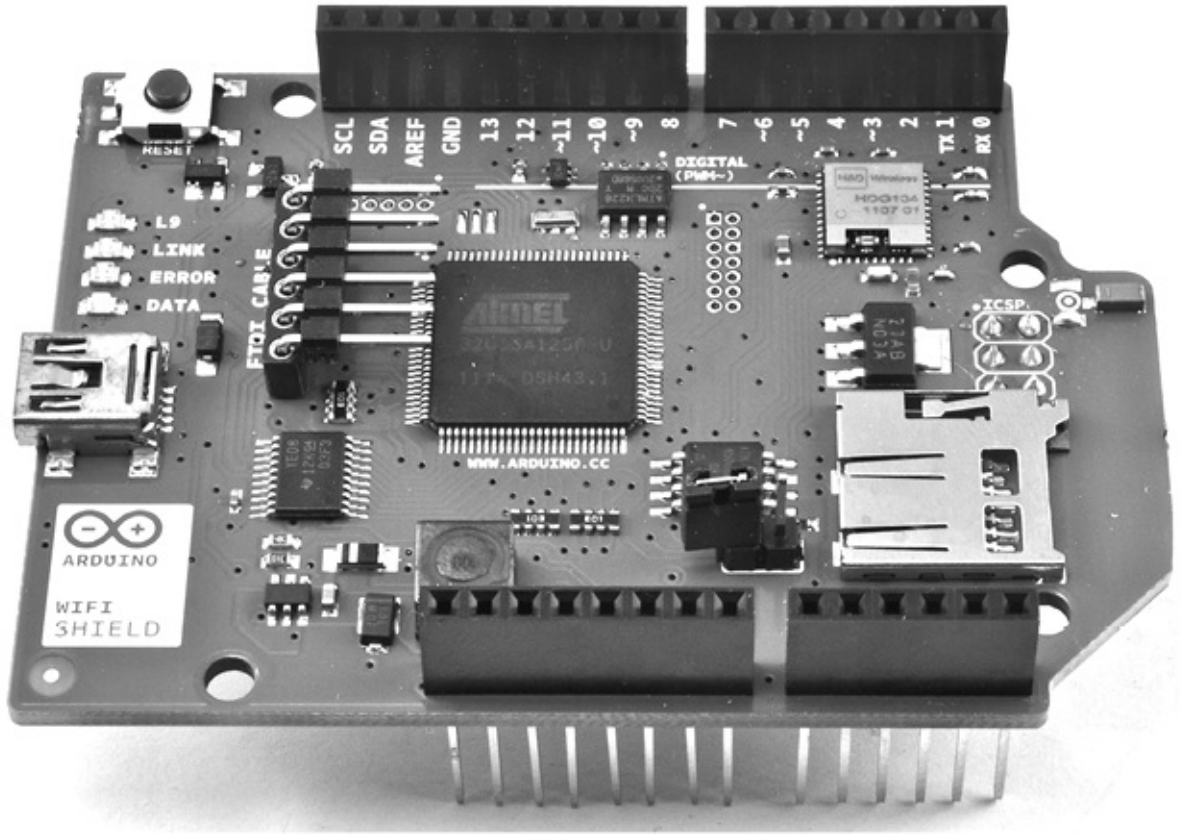


Рис. 12.3. Плата Arduino WiFi

Библиотека Ethernet

Библиотека Ethernet претерпела существенные изменения с момента выпуска в 2011 году версии Arduino 1.0. Она не только позволяет плате Arduino с адаптером Ethernet действовать в роли веб-сервера или веб-клиента (возможность посылать запросы, подобно браузерам), но и реализует дополнительные возможности, такие как поддержка протокола динамической конфигурации сетевого узла (Dynamic Host Configuration Protocol, DHCP), автоматически присваивающего плате IP-адрес.

ПРИМЕЧАНИЕ

Превосходное описание библиотеки Ethernet можно найти в официальной документации Arduino: http://arduino.cc/en/reference/ethernet[10].

Создание соединения

На первом этапе, прежде чем приступить к взаимодействиям по сети, необходимо установить соединение с сетью. Эту задачу решает функция `Ethernet.begin()`. Она позволяет вручную указать все параметры соединения с использованием следующего синтаксиса:

```
Ethernet.begin(mac, ip, dns, gateway, subnet)
```

Рассмотрим каждый из этих параметров:

- `Mac` — MAC-адрес сетевой карты (я расскажу о нем чуть позже);
- `Ip` — IP-адрес платы (можно выбрать любой допустимый для вашей сети);
- `Dns` — IP-адрес сервера доменных имен (Domain Name Server, DNS);
- `Gateway` — IP-адрес шлюза для выхода в Интернет (ваш домашний концентратор);
- `Subnet` — маска подсети.

Этот синтаксис кажется немного пугающим тем, кто не имеет опыта настройки параметров подключения к сети вручную. К счастью, все параметры, кроме `mac`, являются необязательными, и в 90% случаев вам придется указывать только параметры `mac` и `ip` или, весьма вероятно, только `mac`. Все остальные параметры будут настроены автоматически.

MAC-адрес, или адрес доступа к среде (Media Access Control), — это уникальный идентификатор сетевого интерфейса. Иными словами, это адрес платы расширения Ethernet или чего-то другого, предоставляющего сетевой интерфейс в распоряжение Arduino. Этот адрес должен быть уникальным только для вашей сети. Его обычно можно найти на наклейке с обратной стороны платы Ethernet или WiFi (рис. 12.4) или на упаковке. Если вы пользуетесь старой платой, не имеющей MAC-адреса, то можете просто создать свой адрес. Но не используйте в своей сети один и тот же адрес дважды.

Можно создать соединение с сетью с применением DHCP и получить динамический IP-адрес, как показано далее:

```
#include <SPI.h>
#include <Ethernet.h>

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
void setup()
{
  Ethernet.begin(mac);
}
```

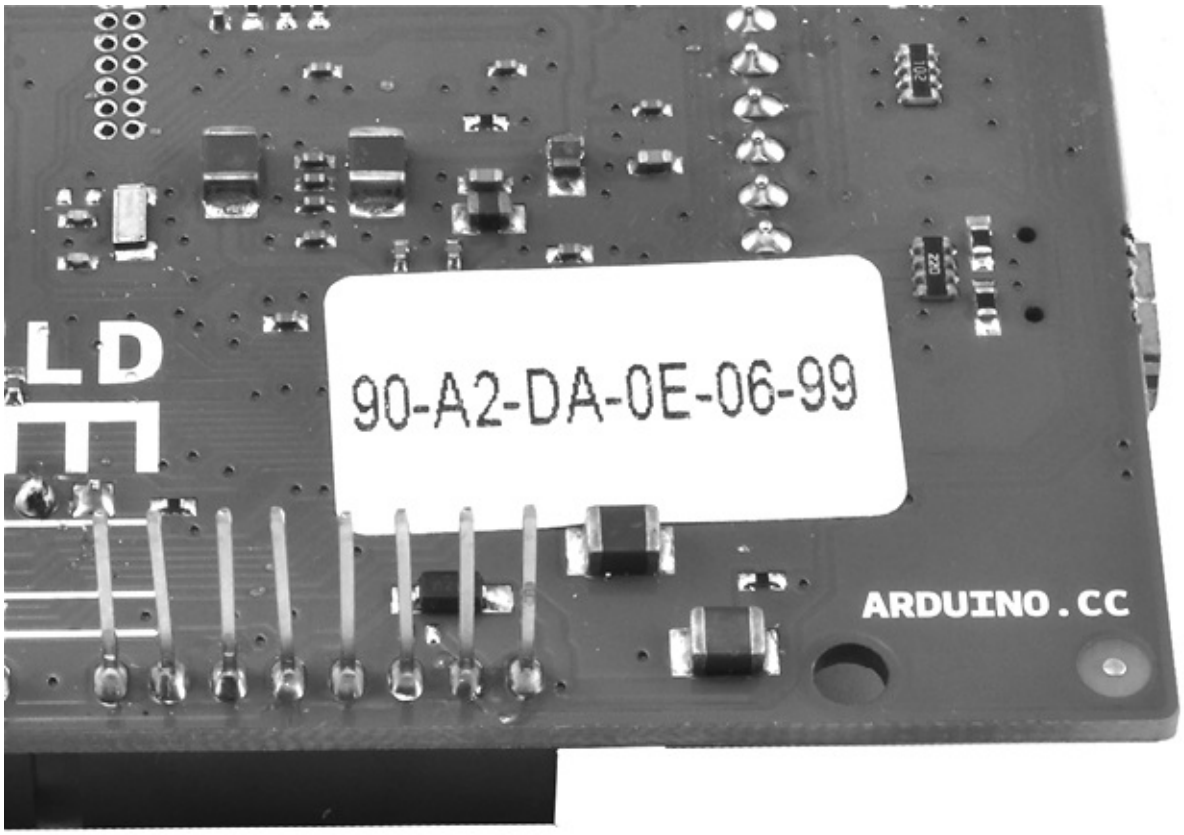


Рис. 12.4. Наклейка с MAC-адресом на плате WiFi

Если потребуется присвоить плате фиксированный IP-адрес, что желательно, когда плата Arduino действует в роли веб-сервера, используйте примерно такой код:

```
#include <SPI.h>
#include <Ethernet.h>

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 10, 0, 1, 200 };
void setup()
{
  Ethernet.begin(mac, ip);
}
```

IP-адрес в параметре `ip` должен быть допустимым для вашей сети. Если вызвать функцию `Ethernet.begin` без параметра с IP-адресом, она попытается получить его с использованием DHCP и вернет 1, если соединение было установлено и динамический IP-адрес успешно получен, в противном случае вернет 0. Можно написать тестовый скетч, который будет устанавливать соединение и вызывать функцию `localIP` для получения IP-адреса, присвоенного Arduino. Следующий пример выполняет такую проверку и выводит сообщение с результатами в монитор последовательного порта. Это полноценный скетч, который вы можете опробовать самостоятельно. Но не забудьте заменить в коде MAC-адрес на указанный на вашей плате:

```
// sketch_12_01_dhcp

#include <SPI.h>
#include <Ethernet.h>

byte mac[] = { 0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x02 };

void setup()
{
  Serial.begin(9600);
  while (!Serial){}; // для совместимости с Leonardo

  if (Ethernet.begin(mac))
  {
    Serial.println(Ethernet.localIP());
  }
  else
  {
    Serial.println("Could not connect to network");
  }
}

void loop()
{
}
```

Настройка веб-сервера

Проект «Физический веб-сервер», описанный далее в этой главе, иллюстрирует организацию скетча, реализующего веб-сервер. А в этом разделе мы рассмотрим функции, помогающие реализовать веб-сервер.

Б ольшая часть функций, необходимых для реализации веб-сервера, содержится в классе `EthernetServer`. Для запуска веб-сервера после установки соединения с сетью требуется пройти еще два этапа. Во-первых, нужно создать новый объект сервера, указав номер порта, который должен использоваться для приема входящих запросов. Это объявление находится в скетче перед функцией `setup`:

```
EthernetServer server = EthernetServer(80);
```

Обычно для приема запросов веб-серверы используют порт 80. То есть если вы

настроили свой веб-сервер на обслуживание порта 80, вам не придется добавлять этот номер в адреса URL, чтобы связаться с сервером.

Во-вторых, чтобы фактически запустить сервер, в функции `setup` нужно выполнить следующую команду:

```
server.begin();
```

Эта функция запустит сервер, который будет ждать, пока кто-то не запросит страницу, которую обслуживает данный сервер. Фактическое обслуживание осуществляется в функции `loop` с применением функции `available`. Эта функция возвращает `null` (если нет запросов для обслуживания) или объект `EthernetClient`. Данный объект, как это ни странно, используется также для отправки исходящих запросов из Arduino к внешним веб-серверам. В данном случае `EthernetClient` представляет соединение между веб-сервером и браузером клиента.

Получив этот объект, можно прочитать входящий запрос с помощью `read` и вернуть HTML-ответ с помощью функций `write`, `print` и `println`. Закончив отправку HTML-ответа клиенту, нужно завершить сеанс вызовом функции `stop` объекта клиента. Я расскажу, как это сделать, в разделе «Физический веб-сервер» далее в этой главе.

Выполнение запросов

Плата Arduino может действовать не только как веб-сервер, но и как веб-браузер, посылая запросы удаленным веб-серверам, которые могут находиться в вашей сети или в Интернете.

Чтобы выполнить запрос, сначала нужно установить соединение с сетью, как в случае с веб-сервером, описанном в предыдущем разделе, но вместо объекта `EthernetServer` создать объект `EthernetClient`:

```
EthernetClient client;
```

Больше с объектом клиента ничего не требуется делать до отправки веб-запроса. Чтобы отправить веб-запрос, следует выполнить следующие действия:

```
if (client.connect("api.openweathermap.org", 80))
{
        client.println("GET     /data/2.5/weather?q=Manchester,uk
HTTP/1.0");
  client.println();
  while (client.connected())
  {
    while (client.available())
```

```
    {
      Serial.write(client.read());
    }
  }
  client.stop();
}
```

Функция `connect` вернет `true`, если соединение с веб-сервером было успешно установлено. Две команды `client.println` посылают веб-серверу запрос на получение желаемой страницы. Затем два вложенных цикла `while` читают данные, пока клиент остается подключенным к веб-серверу и продолжают поступать данные.

Может показаться заманчивым объединить два цикла `while` в один с условием `client.available() && client.connected()`, но такое объединение — далеко не то же самое, что два отдельных цикла, так как данные могут поступать от веб-сервера фрагментами из-за низкой скорости сети или по другим причинам. Внешний цикл поддерживает соединение открытым, а внутренний извлекает данные.

Это блокирующее решение (скетч не сможет производить никаких других действий, пока выполнение запроса не завершится). Если для вашего проекта такое решение неприемлемо, включите во внутренний цикл `while` код, выполняющий проверку других условий.

Примеры использования Ethernet

Далее демонстрируются два примера практического использования библиотеки Ethernet. Вместе они охватывают большинство вариантов сетевых взаимодействий, которые могут вам пригодиться при создании своих проектов на Arduino.

Физический веб-сервер

Первый пример иллюстрирует наиболее частый случай использования сетевых возможностей Arduino. Здесь плата выступает в роли веб-сервера. Подключившись к веб-серверу Arduino, посетители смогут не только читать аналоговые входы, но и изменять уровни на цифровых выходах, щелкая на кнопках в веб-странице (рис. 12.5).

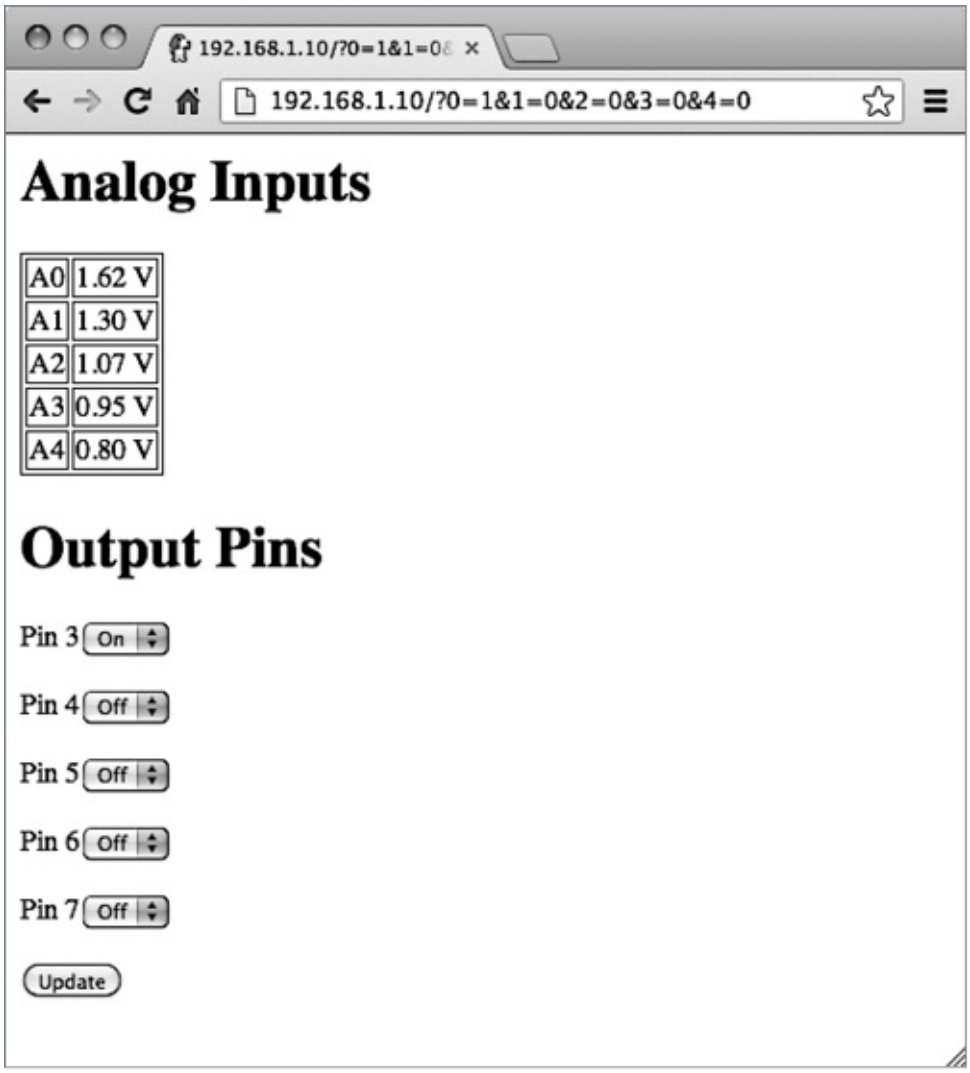


Рис. 12.5. Интерфейс физического веб-сервера

Этот пример демонстрирует отличный способ связать плату Arduino со смартфоном или планшетным компьютером, так как для отправки запросов Arduino достаточно самого простого браузера. Скетч, реализующий этот пример (sketch_12_02_server), содержит 172 строки кода, поэтому он не будет приводиться здесь целиком, но я рекомендую открыть его в Arduino IDE и заглядывать в него в процессе чтения моих пояснений.

Первая часть скетча содержит код, стандартный для любого скетча, реализующего сетевые взаимодействия. Здесь выполняется импортирование библиотек и определение объектов `EthernetServer` и `EthernetClient`.

Далее определяются переменные, служащие разным целям:

```
const int numPins = 5;
int pins[] = {3, 4, 5, 6, 7};
int pinState[] = {0, 0, 0, 0, 0};
char line1[100];
char buffer[100];
```

Константа `numPins` определяет размер массивов `pins` и `pinState`. Массив

pinState предназначен для хранения состояний цифровых выходов, HIGH или LOW. Функция setup настраивает все контакты, перечисленные в массиве pins, на работу в режиме цифровых выходов. Она также устанавливает соединение с сетью, как было показано в примерах ранее. Наконец, массивы символов line1 и buffer предназначены для хранения первой и последующих строк HTTP-запроса соответственно.

Далее приводится функция loop:

```
void loop()
{
  client = server.available();
  if (client)
  {
    if (client.connected())
    {
      readHeader();
      if (! pageNameIs("/"))
      {
        client.stop();
        return;
      }
      client.println(F("HTTP/1.1 200 OK"));
      client.println(F("Content-Type: text/html"));
      client.println();

      sendBody();
      client.stop();
    }
  }
}
```

Функция проверяет наличие любых запросов от браузеров, ожидающих обработки. Если был получен запрос и соединение с клиентом еще не разорвано, вызывается функция readHeader. Эту функцию вы найдете ближе к концу скетча. Функция readHeader читает содержимое заголовка запроса в буфер (line1) и пропускает остальные строки запроса. Это необходимо, чтобы получить имя страницы, запрошенной браузером, и любые дополнительные параметры запроса, если они имеются.

Обратите внимание: из-за большого объема текста, который посылается скетчем в монитор последовательного порта и сеть, я использовал функцию F, сохраняющую

массивы символов во флеш-памяти (см. главу 6).

После чтения заголовка вызывается функция pageNameIs (также находится ближе к концу скетча), чтобы проверить совпадение имени запрошенной страницы с именем корневой страницы (/). Если была запрошена не корневая страница, такой запрос игнорируется. Это важно, потому что многие браузеры посылают веб-серверу дополнительные запросы с целью получить значок для веб-сайта. Эти запросы не следует путать с другими запросами к серверу.

Теперь нужно сгенерировать ответ с заголовком и некоторой разметкой HTML, которую смог бы отобразить браузер. Функция sendHeader генерирует ответ «OK», чтобы показать, что запрос браузера признан допустимым. Функция sendBody, представленная далее, организована намного сложнее:

```
void sendBody()
{
  client.println(F("<html><body>"));
  sendAnalogReadings();
  client.println(F("<h1>Output Pins</h1>"));
  client.println(F("<form method='GET'>"));
  setValuesFromParams();
  setPinStates();
  sendHTMLforPins();
  client.println(F("<input type='submit' value='Update'/>"));
  client.println(F("</form>"));
  client.println(F("</body></html>"));
}
```

Она выводит простой макет HTML-страницы, опираясь на множество вспомогательных функций, которые были созданы, чтобы разбить код на более управляемые фрагменты. Первая из них — sendAnalogReadings:

```
void sendAnalogReadings()
{
  client.println(F("<h1>Analog Inputs</h1>"));
  client.println(F("<table border='1'>"));
  for (int i = 0; i < 5; i++)
  {
    int reading = analogRead(i);
    client.print(F("<tr><td>A")); client.print(i);
     client.print(F("</td><td>")); client.print((float) reading /
205.0);
```

```
    client.println(F(" V</td></tr>"));
  }
  client.println("</table>");
}
```

Она выполняет обход всех аналоговых входов, читает их значения и выводит HTML-таблицу с прочитанными значениями в вольтах.

Возможно, вы обратили внимание на то, что `sendBody` вызывает также функции `setValuesFromParams` и `setPinStates`. Первая записывает в массив `pinStates` состояния `HIGH` или `LOW` цифровых выходов, извлекая их из параметров запроса с помощью функции `valueOfParam`:

```
int valueOfParam(char param)
{
  for (int i = 0; i < strlen(line1); i++)
  {
    if (line1[i] == param && line1[i+1] == '=')
    {
      return (line1[i+2] - '0');
    }
  }
  return 0;
}
```

Функция `valueOfParam` ожидает получения параметра запроса в виде единственной цифры. Как выглядят эти параметры, можно увидеть, если запустить пример, открыть страницу в браузере и щелкнуть на кнопке **Update** (Обновить). Адрес URL в адресной строке браузера изменится, и в нем появятся параметры, как показано далее:

```
192.168.1.10/?0=1&1=0&2=0&3=0&4=0
```

Список параметров начинается после символа `?`. Параметры имеют вид *X*=*Y* и отделяются друг от друга символом `&`. Слева от знака `=` находится имя параметра (в данном случае цифры от 0 до 4), а справа — значения (в данном примере `1` означает «включено», а `0` — «выключено»). Для простоты параметры в этом примере могут иметь только односимвольные значения. Функция `setPinStates` устанавливает состояние цифровых выходов в соответствии со значениями элементов массива `pinStates`.

А теперь вернемся к функции `sendBody`. Вслед за таблицей со значениями аналоговых входов нужно послать разметку HTML с коллекцией раскрывающихся

списков, соответствующих цифровым выходам. В каждом списке нужно выбрать пункт **On** (Включено) или **Off** (Выключено) в зависимости от текущего состояния цифрового выхода. Для этого нужно добавить текст «selected» в значение, соответствующее состоянию данного выхода в массиве `pinStates`.

Код разметки HTML для цифровых выходов заключается в форму, чтобы посетитель мог изменить значения в форме и, щелкнув на кнопке **Update** (Обновить), сгенерировать новый запрос к этой странице с соответствующими параметрами для установки цифровых выходов. А теперь посмотрим, как выглядит разметка HTML-страницы:

```
<html><body>
<hq>Analog Inputs</h1>
<table border='1'>
<tr><td>A0</td><td>0.58 V</td></tr>
<tr><td>A1</td><td>0.63 V</td></tr>
<tr><td>A2</td><td>0.60 V</td></tr>
<tr><td>A3</td><td>0.65 V</td></tr>
<tr><td>A4</td><td>0.60 V</td></tr>
</table>
<h1>Output Pins</h1>

<form method='GET'>
<p>Pin 3<select name='0'>
<option value='0'>Off</option>
<option value='1' selected>On</option>
</select></p>
<p>Pin 4<select name='1'>
<option value='0' selected >Off</option>
<option value='1'>On</option>
</select></p>
<p>Pin 5<select name='2'>
<option value='0' selected >Off</option>
<option value='1'>On</option>
</select></p>
<p>Pin 6<select name='3'>
<option value='0' selected >Off</option>
<option value='1'>On</option>
</select></p>
<p>Pin 7<select name='4'>
```

```
<option value='0' selected >Off</option>
<option value='1'>On</option>
</select></p>
<input type='submit'> value='Update'/>
</form>
</body></html>
```

Увидеть этот код можно, воспользовавшись функцией **View Source** (Исходный код страницы) в браузере.

Использование веб-службы JSON

Для иллюстрации возможности отправки веб-запросов из платы Arduino внешним веб-сайтам я воспользуюсь веб-службой, возвращающей данные о погоде в определенном географическом пункте. Плата будет выводить краткое описание погоды в монитор последовательного порта (рис. 12.6). Описываемый скетч посылает запрос один раз в момент запуска, но его нетрудно изменить, чтобы он запрашивал погоду каждый час и выводил результаты на двухстрочный жидкокристаллический дисплей.

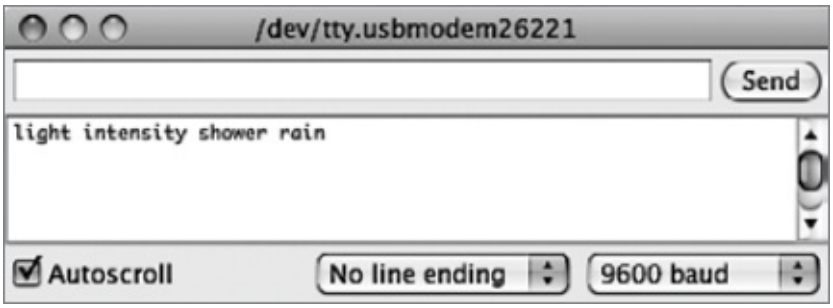


Рис. 12.6. Получение информации о погоде от веб-службы

Скетч для этого примера получился очень коротким, всего 45 строк кода (sketch_12_03_web_request). Наибольший интерес для нас представляет функция `hitWebPage`:

```
void hitWebPage()
{
  if (client.connect("api.openweathermap.org", 80))
  {
          client.println("GET   /data/2.5/weather?q=Manchester,uk
HTTP/1.0");
    client.println();
    while (client.connected())
    {
      if (client.available())
```

```
      {
        client.findUntil("description\":\"", "\0");
        String description = client.readStringUntil('\"');
        Serial.println(description);
      }
    }
    client.stop();
  }
}
```

Прежде всего необходимо подключить клиента к порту 80 сервера. Если соединение благополучно установлено, серверу посылается заголовок запроса:

```
client.println("GET /data/2.5/weather?q=Manchester,uk HTTP/1.0");
```

Дополнительная команда `println` нужна, чтобы отметить конец заголовка запроса и побудить сервер прислать ответ.

Далее в цикле `while` инструкция `if` проверяет получение данных от сервера, пока соединение с ним не закрыто. Непосредственное чтение данных из потока помогает избежать необходимости сохранять все данные в памяти. Данные поступают в формате JSON:

```
{"coord":{"lon":-2.23743,"lat":53.480949},
"sys":{"country":"GB","sunrise":1371094771,
"sunset":1371155927},"weather":[{"id":520, "main":"Rain",
"description":"light intensity shower rain", "icon": "09d"}]
"humidity":87,"temp_min":283.15,"temp_max":285.93},
"wind":{"speed:5.1,"deg":270},"rain":{"1h":0.83},
"clouds":{"all":40},"dt":1371135000,"id":3643123,
"name":"Manchester","cod":200}
```

Функция `hitWebPage` с помощью функций `findUntil` и `readStringUntil` извлекает фрагмент текста, следующий за словом «description», с двоеточием и двойной кавычкой до следующей двойной кавычки.

Функция `findUntil` просто игнорирует все, пока не встретит указанную строку. Затем функция `readStringUntil` читает текст из потока, пока не встретит двойную кавычку.

Библиотека WiFi

Библиотека WiFi, как можно было ожидать, очень похожа на библиотеку Ethernet. Если в скетче заменить `Ethernet` на `WiFi`, `EthernetServer` на `WiFiServer` и

`EthernetClient` на `WiFiClient`, остальной код останется почти неизменным.

Создание соединения

Главное отличие библиотеки WiFi от Ethernet заключается в подключении к сети. Прежде всего нужно импортировать библиотеку WiFi:

```
#include <SPI.h>
#include <WiFi.h>
```

Чтобы установить соединение с сетью, следует вызвать команду `WiFi.begin` и передать ей имя беспроводной сети и пароль:

```
WiFi.begin("MY-NETWORK-NAME", "mypassword");
```

Пример в разделе «Пример использования WiFi» иллюстрирует еще одно отличие, о котором вы должны знать.

Особые функции в библиотеке WiFi

Библиотека WiFi включает несколько специальных функций. Они перечислены в табл. 12.1.

Полное описание библиотеки WiFi можно найти по адресу **http://arduino.cc/en/Reference/WiFi**[11].

Таблица 12.1. Специальные функции в библиотеке WiFi

Функция	Описание
WiFi.config	Позволяет установить статические IP-адреса платы, сервера имен (DNS) и шлюза
WiFi.SSID	Возвращает строку идентификатора беспроводной сети SSID
WiFi.RSSI	Возвращает значение мощности сигнала типа **long**
WiFi.encriptionType	Возвращает числовой код, соответствующий методу шифрования
WiFi.scanNetworks	Возвращает количество найденных сетей, но никакой дополнительной информации о них не возвращается
WiFi.macAddress	Помещает MAC-адрес адаптера WiFi в шестибайтный массив, переданный как параметр

Пример использования WiFi

Для этого примера я изменил скетч sketch_12_02_server, адаптировав его для работы с платой расширения WiFi. Полный исходный код можно найти в скетче sketch_12_04_server_wifi. Я не буду повторять пример целиком, только отмечу отличия

от оригинальной версии.

Чтобы подключиться к беспроводной точке доступа, нужно указать имя беспроводной сети и пароль:

```
char ssid[] = "My network name";  // имя сети (SSID)
char pass[] = "mypassword";       // пароль для доступа к сети
```

Также нужно изменить имена классов сервера и клиента, `EthernetServer` и `EthernetClient`, на `WiFiServer` и `WiFiClient`:

```
WiFiServer server(80);
WiFiClient client;
```

При определении объекта сервера все так же требуется указать номер порта 80.

Следующее ракзличие между двумя платами заключается в инициализации подключения. В данном случае должна использоваться команда

```
WiFi.begin(ssid, pass);
```

Прочий код остался почти без изменений. В функции `loop` вы увидите команду `delay(1)` перед остановкой клиента, она дает клиенту время завершить чтение до того, как соединение будет закрыто. В версии на основе библиотеки Ethernet такая задержка не нужна. Обратите также внимание на то, что кое-где я объединил несколько вызовов `client.print` в один, чтобы каждый вызов выводил более длинные строки. Этот прием увеличивает скорость взаимодействий, потому что плата WiFi крайне неэффективно обрабатывает короткие строки. Но не забывайте, что каждый отдельный вызов `client.print` или `client.println` не может обрабатывать строки длиннее 90 байт — они просто не будут отправлены.

Версия программы на основе библиотеки WiFi работает заметно медленнее версии на основе библиотеки Ethernet и тратит на загрузку около 45 секунд. Плата расширения WiFi поддерживает возможность изменения прошивки, и если в будущем разработчики Arduino повысят эффективность работы платы WiFi, вам определенно стоит подумать об обновлении прошивки в своей плате расширения. Инструкции по прошивке платы WiFi можно найти на странице **http://arduino.cc/en/Main/ArduinoWiFiShield**.

В заключение

В этой главе мы познакомились с разными способами подключения платы Arduino к сети и выполнения некоторых сетевых операций с использованием плат расширения Ethernet и WiFi. Здесь вы также узнали, как использовать Arduino в качестве веб-сервера и веб-клиента.

В следующей главе вы познакомитесь с цифровой обработкой сигналов (Digital

Signal Processing, DSP) с помощью Arduino.

10 Перевод на русский язык: **http://arduino.ua/ru/prog/Ethernet**. — *Примеч. пер.*

11 Перевод на русский язык: **http://arduino.ua/ru/prog/WiFi**. — *Примеч. пер.*

13. Цифровая обработка сигналов

Плата Arduino способна выполнять простую обработку сигналов. В этой главе обсуждаются разные способы такой обработки, от фильтрации сигнала, поступающего на аналоговый вход, с применением программного обеспечения вместо внешних электронных устройств до вычисления относительной величины различных частотных сигналов с применением быстрого преобразования Фурье.

Введение в цифровую обработку сигналов

Принимая информацию с датчика, вы фактически измеряете сигнал. Сигналы принято отображать в виде линии (обычно извилистой), идущей слева направо с течением времени. Именно так отображаются электрические сигналы на экране осциллографа. Ось *y* отражает *амплитуду* сигнала (его силу), а ось *x* — время. На рис. 13.1 изображен сигнал, соответствующий воспроизведению музыкального фрагмента длительностью 1/4 секунды, который был захвачен с помощью осциллографа.

Вы можете заметить некоторую цикличность сигнала. Скорость повторения циклов называют *частотой*. Она измеряется в герцах (Гц). Сигнал с частотой 1 Гц повторяет цикл со скоростью 1 раз в секунду, с частотой 10 Гц — 10 раз в секунду. Взгляните на сигнал в левой половине рис. 13.1 — один цикл сигнала длится примерно 0,6 квадрата координатной

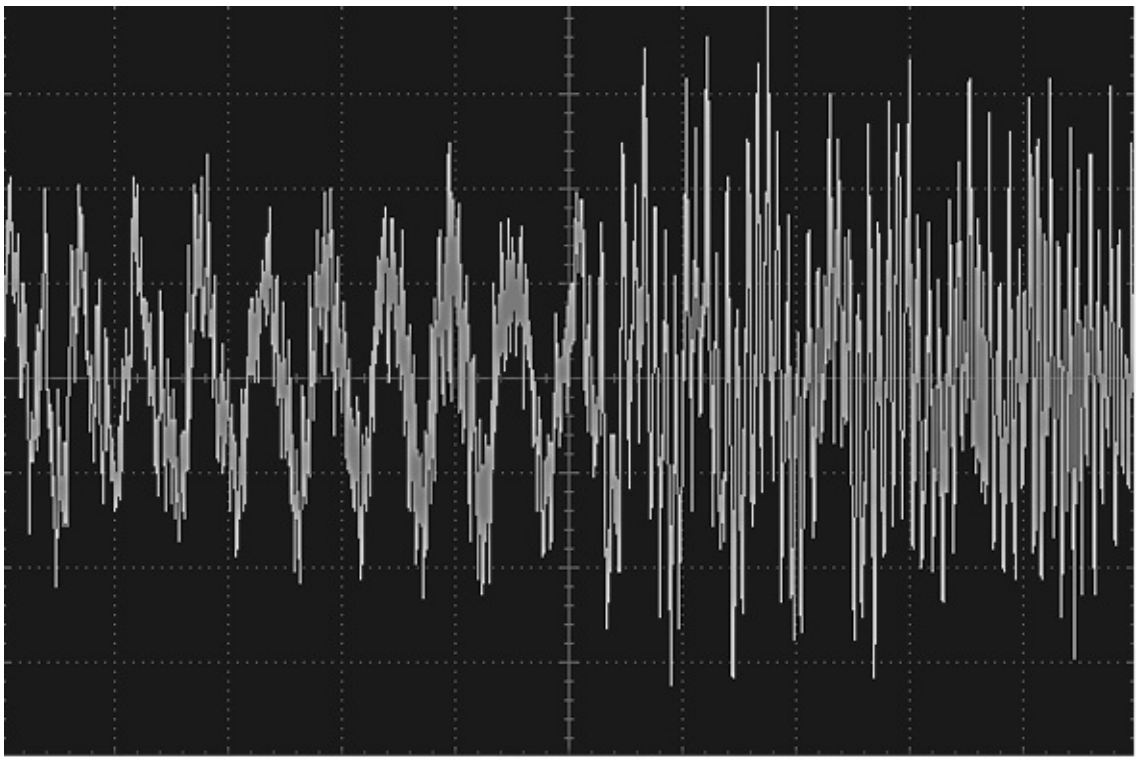


Рис. 13.1. Сигнал от источника музыки

сетки. Так как при настройке осциллографа размер стороны квадрата по оси *x* был

выбран равным 25 мс, частота этой части сигнала составляет 1/(0,6 × 0,025) = 67 Гц. Если увеличить масштаб, выбрав более короткий промежуток времени, можно будет увидеть множество других частотных составляющих звука, подмешивающихся к основному сигналу. Если сигнал не является чистой синусоидой (как показано далее на рис. 13.5), он всегда будет включать в себя целый спектр гармоник.

Сигнал, изображенный на рис. 13.1, можно попытаться захватить с использованием одного из аналоговых входов на плате Arduino. Этот прием называется *оцифровкой*, потому что в ходе его реализации аналоговый сигнал переводится в цифровую форму. Чтобы получить довольно точную копию оригинального сигнала, замеры нужно выполнять очень быстро.

Суть цифровой обработки сигналов (Digital Signal Processing, DSP) заключается в том, чтобы оцифровать сигнал с помощью аналого-цифрового преобразователя (АЦП), выполнить некоторые манипуляции с ним и затем сгенерировать выходной аналоговый сигнал с помощью цифроаналогового преобразователя (ЦАП). Самое современное звуковое оборудование, MP3-плееры и сотовые телефоны используют цифровую обработку сигналов для частотной коррекции, управляя относительной мощностью высоких и низких частот при воспроизведении музыкальных произведений. Однако иногда не требуется выводить измененную версию входного сигнала, когда цифровая обработка нужна, только чтобы убрать из сигнала нежелательные помехи и тем самым получить от датчика более точное значение.

В общем случае платы Arduino — не самые лучшие устройства для цифровой обработки сигналов. Они не способны захватывать аналоговый ввод с высокой скоростью, а их аналоговые выходы ограничены возможностями технологии широтно-импульсной модуляции (ШИМ). Исключение составляет модель Arduino Due, которая имеет несколько АЦП, быстрый процессор и два истинных ЦАП. То есть модель Due обладает достаточными аппаратными возможностями для того, чтобы ее можно было использовать для оцифровки звукового стереосигнала и выполнения каких-то манипуляций с ним.

Усреднение замеров

При чтении сигнала с датчика часто обнаруживается, что лучших результатов можно добиться, выполняя усреднение по нескольким замерам. Одно из возможных решений этой задачи — использование циклического буфера (рис. 13.2).

При использовании циклического буфера каждый новый замер сохраняется в ячейке буфера с текущим индексом, после чего индекс увеличивается на единицу. Когда будет заполнена последняя ячейка, значение индекса обнулится, и новые замеры начнут записываться поверх старых. Следуя

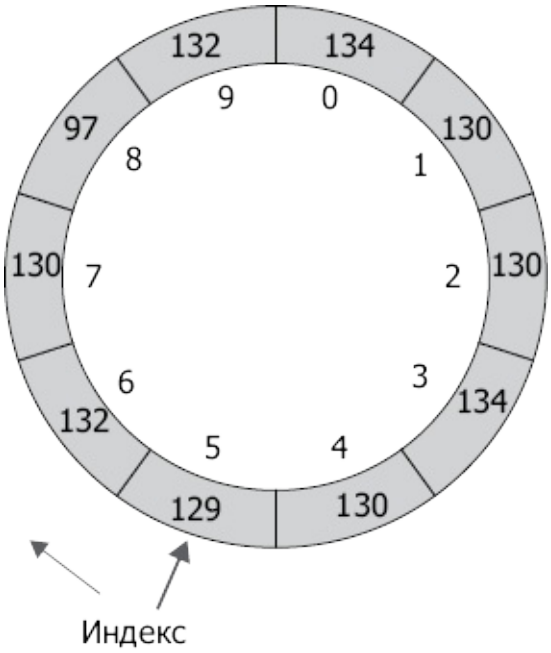


Рис. 13.2. Циклический буфер

этой методике, вы всегда будете иметь *N* последних замеров, где *N* — это размер буфера.

Следующий фрагмент реализует циклический буфер:

```
// sketch_13_01_averaging

const int samplePin = A1;

const int bufferSize = 10;
int buffer[bufferSize];
int index;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  int reading = analogRead(samplePin);
  addReading(reading);
  Serial.println(average());
  delay(1000);
}
```

```
void addReading(int reading)
{
  buffer[index] = reading;
  index++;
  if (index >= bufferSize) index = 0;
}

int average()
{
  long sum = 0;
  for (int i = 0; i < bufferSize; i++)
  {
    sum += buffer[i];
  }
  return (int)(sum / bufferSize);
}
```

Это решение дает неверный усредненный результат, пока буфер не заполнится. На практике это не должно быть большой проблемой, так как не составит труда заполнить буфер замерами перед тем, как начать запрашивать усредненные замеры.

Обратите внимание на то, что переменная `sum` для хранения суммы в функции `average` объявлена с типом `long`. Это особенно важно, если используется емкий буфер и есть вероятность, что сумма превысит максимальное положительное значение `int`, которое немногим больше 32 000. Отметьте также, что она безопасно может возвращать результат усреднения в виде значения `int`, потому что среднее значение будет находиться в диапазоне значений отдельных замеров.

Введение в фильтрацию

Как говорилось в разделе «Введение в цифровую обработку сигналов», любой сигнал обычно состоит из целого спектра гармоник. Иногда бывает желательно исключить некоторые из этих гармоник, и тогда следует использовать прием фильтрации.

Наиболее распространенный способ фильтрации в Arduino — *низкочастотная фильтрация*. Представьте, что у вас имеется датчик освещенности и вы пытаетесь определить общий уровень освещенности и поминутную динамику ее изменения, например, чтобы определить момент, когда станет достаточно темно, чтобы включить освещение. Но вам нужно устранить высокочастотные изменения освещенности, вызванные такими событиями, как быстрое перемещение вблизи датчика объектов, заслоняющих свет, или засветка датчика искусственными источниками света, которые в действительности мерцают с частотой напряжения питания (50 Гц, если вы живете в

России). Если вас интересует только медленно изменяющаяся часть сигнала, то вам нужен низкочастотный фильтр. И наоборот, если требуется откликаться на скоротечные события и игнорировать более протяженные тенденции, используйте *высокочастотный фильтр*.

Вернемся к проблеме искажений, вызываемых частотой переменного тока в электросети. Если, к примеру, вам нужно исключить только паразитную гармонику с частотой 50 Гц и оставить гармоники с частотами выше и ниже этого значения, тогда простое отсечение низкочастотных гармоник не даст желаемого результата. Для решения этой задачи следует использовать *полосовой фильтр*, который удалит только гармонику с частотой 50 Гц или, что более вероятно, все гармоники с частотами от 49 до 51 Гц.

Простой низкочастотный фильтр

Часто в циклическом буфере нет никакой необходимости, если требуется всего лишь сгладить сигнал. Такое сглаживание можно рассматривать как низкочастотную фильтрацию, которая отсекает высокочастотные составляющие сигнала и оставляет только общую динамику. Подобного рода фильтрация часто используется при работе, например, с датчиками поворота, чувствительными к высокочастотным изменениям, которые могут не интересовать вас, или когда вам достаточно знать, на какой угол повернуто устройство.

Простой и эффективный способ решения этой задачи заключается в сохранении некоторого скользящего среднего по нескольким замерам. Скользящее среднее вычисляется как пропорция между текущим скользящим средним значением и значением нового замера:

Сглаженное значениеn = (Коэффициент x Сглаженное значениеn–1) + ((1 – Коэффициент) x Замерn).

Коэффициент — это константа между 0 и 1. Чем выше значение коэффициента, тем сильнее эффект сглаживания.

Такое определение выглядит сложнее, чем есть на самом деле, поэтому взгляните на следующий код, чтобы убедиться, насколько этот прием прост в реализации:

```
// sketch_13_02_simple_smoothing
const int samplePin = A1;
const float alpha = 0.9;

void setup()
{
  Serial.begin(9600);
```

```
}

void loop()
{
  static float smoothedValue = 0.0;
  float newReading = (float)analogRead(samplePin);
  smoothedValue = (alpha * smoothedValue) +
    ((1 – alpha) * newReading);
  Serial.print(newReading); Serial.print(",");
  Serial.println(smoothedValue);
  delay(1000);
}
```

Скопировав результаты сглаживания из окна монитора последовательного порта и вставив их в электронную таблицу, можно построить график, чтобы увидеть, насколько хорошо выполняется сглаживание. На рис. 13.3 показан результат работы предыдущего скетча в плате, к аналоговому входу **A1** которой подключен некоторый источник переменного сигнала.

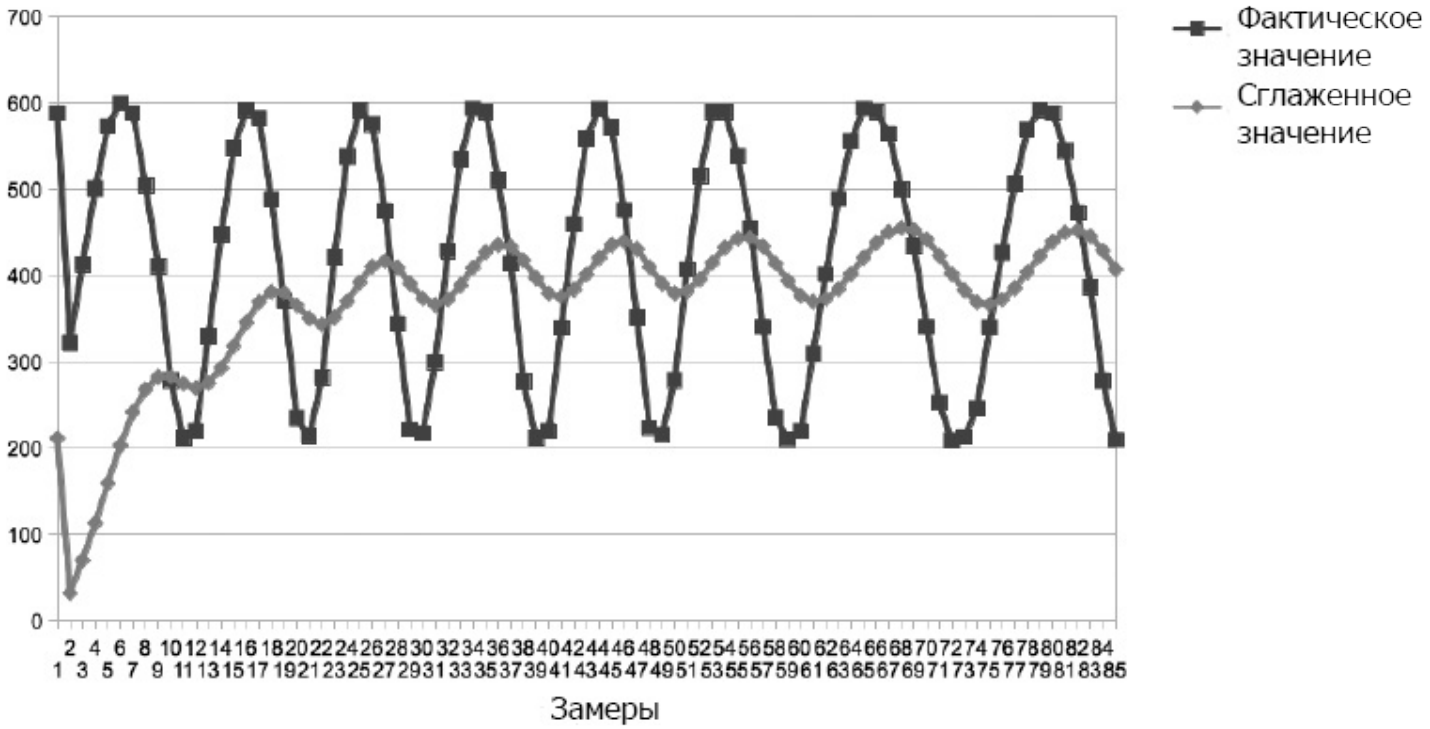


Рис. 13.3. График изменения сглаженных значений

Как видите, для выхода на нормальный уровень сглаживания требуется некоторое время. Если увеличить значение *коэффициента*, например, до 0,95, сглаживание получится еще более сильным. Построение графиков на основе данных, скопированных из окна монитора последовательного порта, — отличный способ проверить, насколько результат сглаживания соответствует вашим потребностям.

На рис. 13.4 изображена схема подключения источника сигнала звуковой частоты к контакту **A0** на плате Arduino и приемника выходного ШИМ-сигнала (10 кГц), генерируемого платой. В качестве генератора сигнала я использовал приложение на смартфоне и подключил выход для наушников на телефоне к плате Arduino.

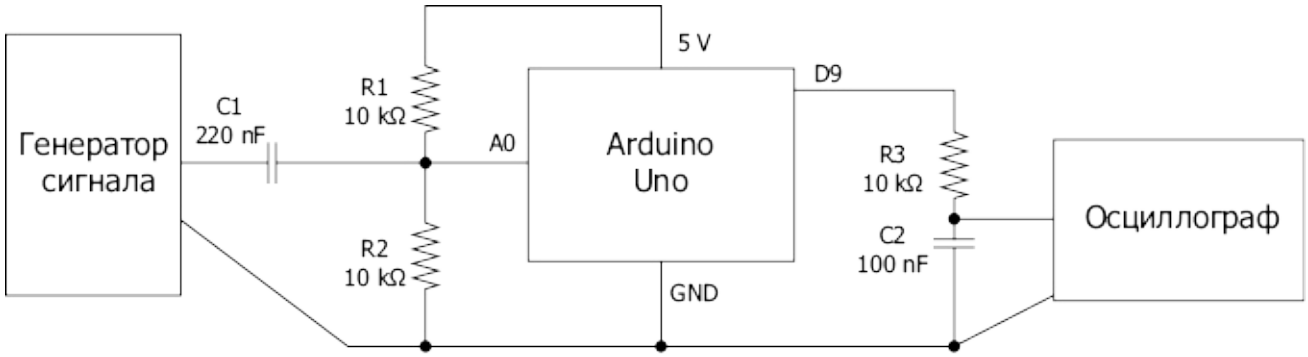


Рис. 13.4. Использование Arduino Uno для цифровой обработки сигнала

ВНИМАНИЕ

Предупреждаю, что подобное использование телефона может повлечь за собой аннулирование гарантийных обязательств производителя и выход телефона из строя.

Входной сигнал с генератора смещается с помощью *C*1, *R*1 и *R*2 так, чтобы он колебался относительно средней точки 2,5 В и АЦП мог читать любые уровни сигнала. Если убрать эти элементы, сигнал будет колебаться относительно 0 В.

На выходе я добавил простой *RC*-фильтр на элементах *R*3 и *C*2, чтобы устранить несущую частоту ШИМ. Несущая частота ШИМ, равная 10 кГц, к сожалению, слишком близка к частоте основного сигнала, чтобы ее можно было подавить полностью.

Выходной сигнал можно не только наблюдать на осциллографе, но и прослушивать, если подключить выход к усилителю, но, если вы решите подключить усилитель, убедитесь, что вход усилителя связан по переменному току.

Следующий скетч использует библиотеку TimerOne, чтобы сгенерировать ШИМ-сигнал и выполнять замеры с частотой 10 кГц:

```
// sketch_13_03_null_filter_uno

#include <TimerOne.h>

const int analogInPin = A0;
const int analogOutPin = 9;

void setup()
```

```
{
  Timer1.attachInterrupt(sample);
  Timer1.pwm(analogOutPin, 0, 100);
}

void loop()
{
}

void sample()
{
  int raw = analogRead(analogInPin);
  Timer1.setPwmDuty(analogOutPin, raw);
}
```

На рис. 13.5 изображен сигнал, подаваемый на вход Arduino (верхний график), и выходной сигнал, генерируемый платой Arduino (нижний график). Оба сигнала имеют частоту 1 кГц. Выходной сигнал имеет в целом неплохую форму до частоты 2–3 кГц, но на более высоких частотах начинает приобретать треугольную форму, что объясняется малым числом замеров, приходящихся на один цикл. На осциллограмме можно наблюдать остатки несущей гармоники, искажающие выходной сигнал, но в целом он имеет совсем неплохую форму. Этого вполне достаточно для обработки сигналов с речевой частотой.

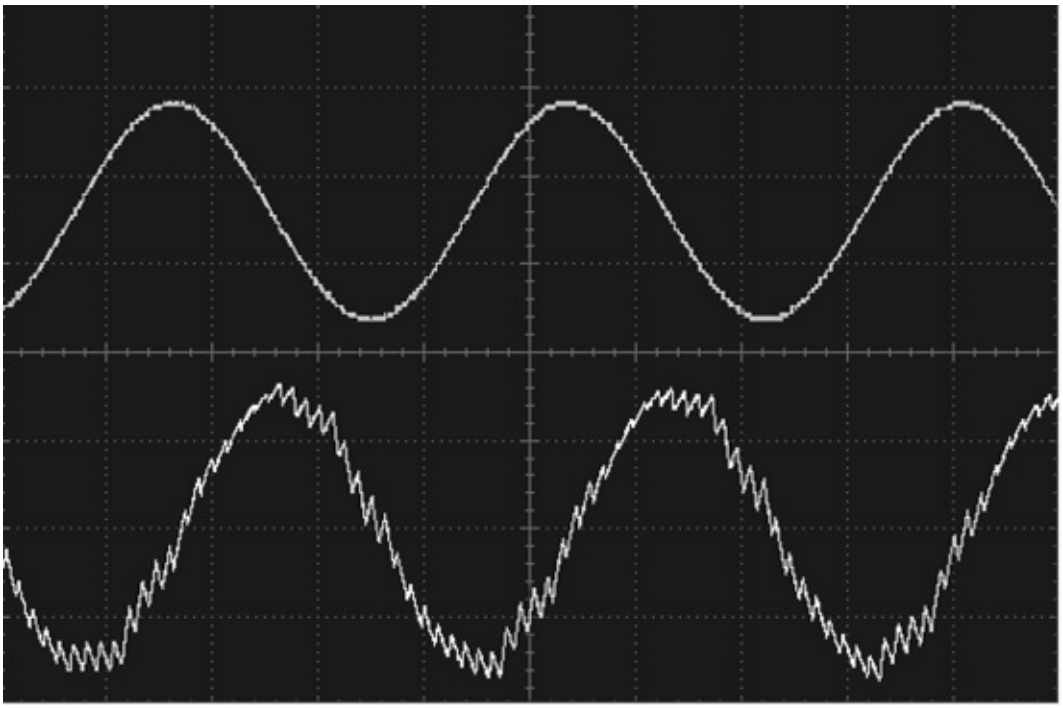


Рис. 13.5. Воспроизведение сигнала с частотой 1 кГц платой Arduino Uno

Цифровая обработка сигналов в Arduino Due

Теперь проведем тот же эксперимент с платой Arduino Due, способной производить замеры с более высокой частотой. Код для модели Uno из предыдущего раздела нельзя использовать с платой Due, так как она имеет иную архитектуру, не позволяющую использовать библиотеку TimerOne.

Аналоговые входы в модели Due способны принимать сигнал с напряжением до 3,3 В, поэтому сопротивление *R*1 следует подключить к контакту питания **3.3V**, а *не* **5V**. Так как Due имеет истинный аналоговый выход, можно убрать низкочастотный *RC*-фильтр на элементах *R*3 и *C*2 и подключить осциллограф непосредственно к контакту **DAC0**. На рис. 13.6 изображена схема подключения Due.

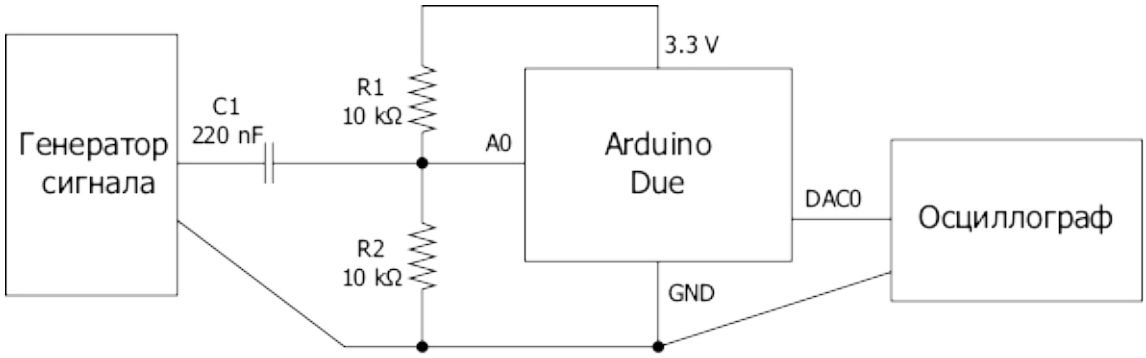


Рис. 13.6. Использование Arduino Due для цифровой обработки сигнала

Следующий скетч выполняет замеры с частотой 100 кГц!

```
// sketch_13_04_null_filter_due

const long samplePeriod = 10L; // микросекунды

const int analogInPin = A0;
const int analogOutPin = DAC0;

void setup()
{
  // http://www.djerickson.com/arduino/
  REG_ADC_MR = (REG_ADC_MR & 0xFFF0FFFF) | 0x00020000;
  analogWriteResolution(8);
  analogReadResolution(8);
}

void loop()
{
  static long lastSampleTime = 0;
```

```
  long timeNow = micros();
  if (timeNow > lastSampleTime + samplePeriod)
  {
    int raw = analogRead(analogInPin);
    analogWrite(analogOutPin, raw);
    lastSampleTime = timeNow;
  }
}
```

В отличие от других моделей, Arduino Due позволяет изменять разрешение АЦП и ЦАП. Для простоты и скорости оба настраиваются на разрешение 8 бит.

Следующая строка увеличивает скорость работы АЦП на плате Due, управляя значениями в регистрах. Посетите страницу, указанную в исходном коде, где можно найти подробное описание этого трюка.

```
REG_ADC_MR = (REG_ADC_MR & 0xFFF0FFFF) | 0x00020000;
```

Для управления частотой замеров скетч использует функцию `micros`. То есть замеры выполняются только по прошествии довольно большого числа микросекунд.

На рис. 13.7 показано, как эта схема воспроизводит входной сигнал с частотой 5 кГц. Как видите, в выходном сигнале имеются ступеньки, образованные 20 замерами на цикл, следующими с частотой 100 кГц.

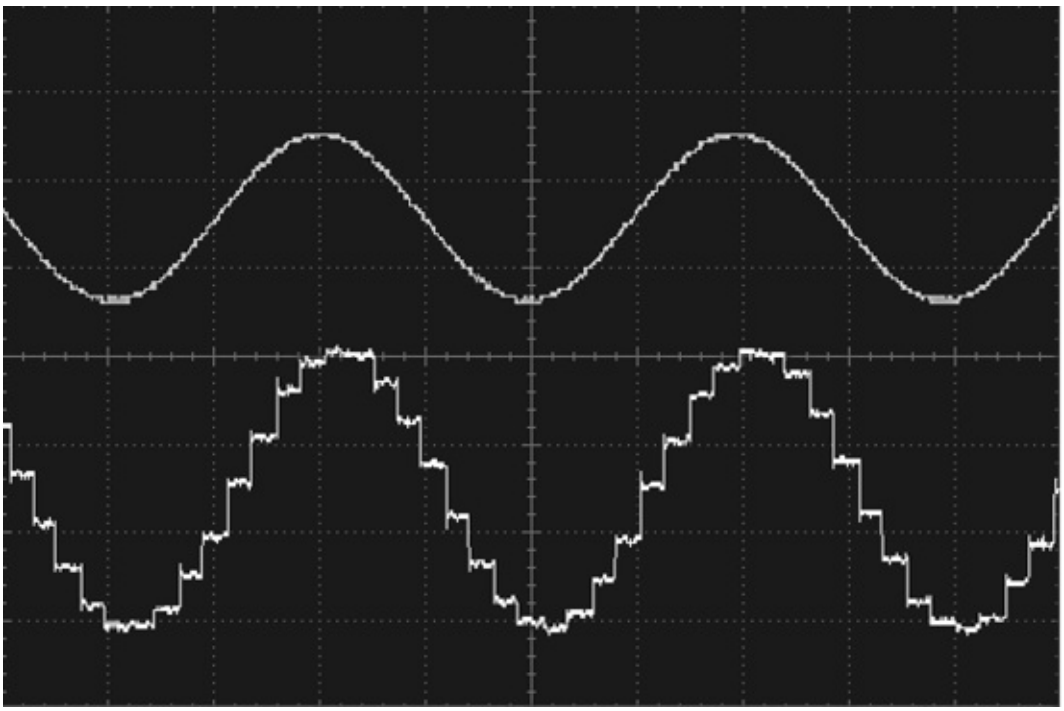


Рис. 13.7. Воспроизведение сигнала с частотой 5 кГц платой Arduino Due

Генератор реализаций фильтров

Если потребуется организовать более сложную фильтрацию, обратитесь к онлайн-генератору кода, с помощью которого вы сможете спроектировать фильтр и скопировать строки сгенерированного кода в свой скетч. Найти генератор можно по адресу **http://www.schwietering.com/jayduino/filtuino/.**

Альтернативой ему является изучение сложнейших математических приемов!

На рис. 13.8 показано, как выглядит интерфейс генератора фильтров. В нижней половине экрана находится сгенерированный код, и далее я кратко расскажу, как включить его в скетч.

В вашем распоряжении имеется масса параметров для настройки будущего фильтра. На рис. 13.8 демонстрируется проект полосового фильтра, целью применения которого является уменьшение амплитуды сигнала на частотах от 1 до 1,5 кГц. Начнем с первого ряда параметров в верхней части: **Butterworth**, **band stop** и **1st order**. **Butterworth** (фильтр Баттерворта) — это конструкция фильтра, название которого соответствует оригиналу из электроники (**https://ru.wikipedia.org/wiki/Фильтр_Баттерворта**). Фильтр Баттерворта хорошо подходит для разных целей и считается хорошим выбором по умолчанию.

Я также выбрал параметр **1st order** (первого порядка). Большее значение этого параметра приведет к увеличению числа хранимых предшествующих замеров и крутизны затухания амплитудно-частотной характеристики (АЧХ) на частотах полосы подавления. Для данного примера вполне подойдет значение **1st order**. Увеличение порядка потребует выполнения дополнительных вычислений и, возможно, уменьшения частоты следования замеров, чтобы плата Arduino успевала делать это.

Затем идут несколько неактивных полей ввода, имеющих отношение к фильтрам других конструкций, а еще ниже — параметр **samplerate** (частота замеров). Этот параметр определяет частоту, с которой будет производиться отбор данных, а также частоту, с которой сгенерированный код будет вызываться для фильтрации сигнала.

Далее я определил верхний и нижний пороги полосы подавления. В эти поля можно вводить частоту в герцах или ноту MIDI.

Раздел **more** (дополнительно) включает пару дополнительных параметров и даже содержит подсказки, как лучше их настроить. В разделе **output**

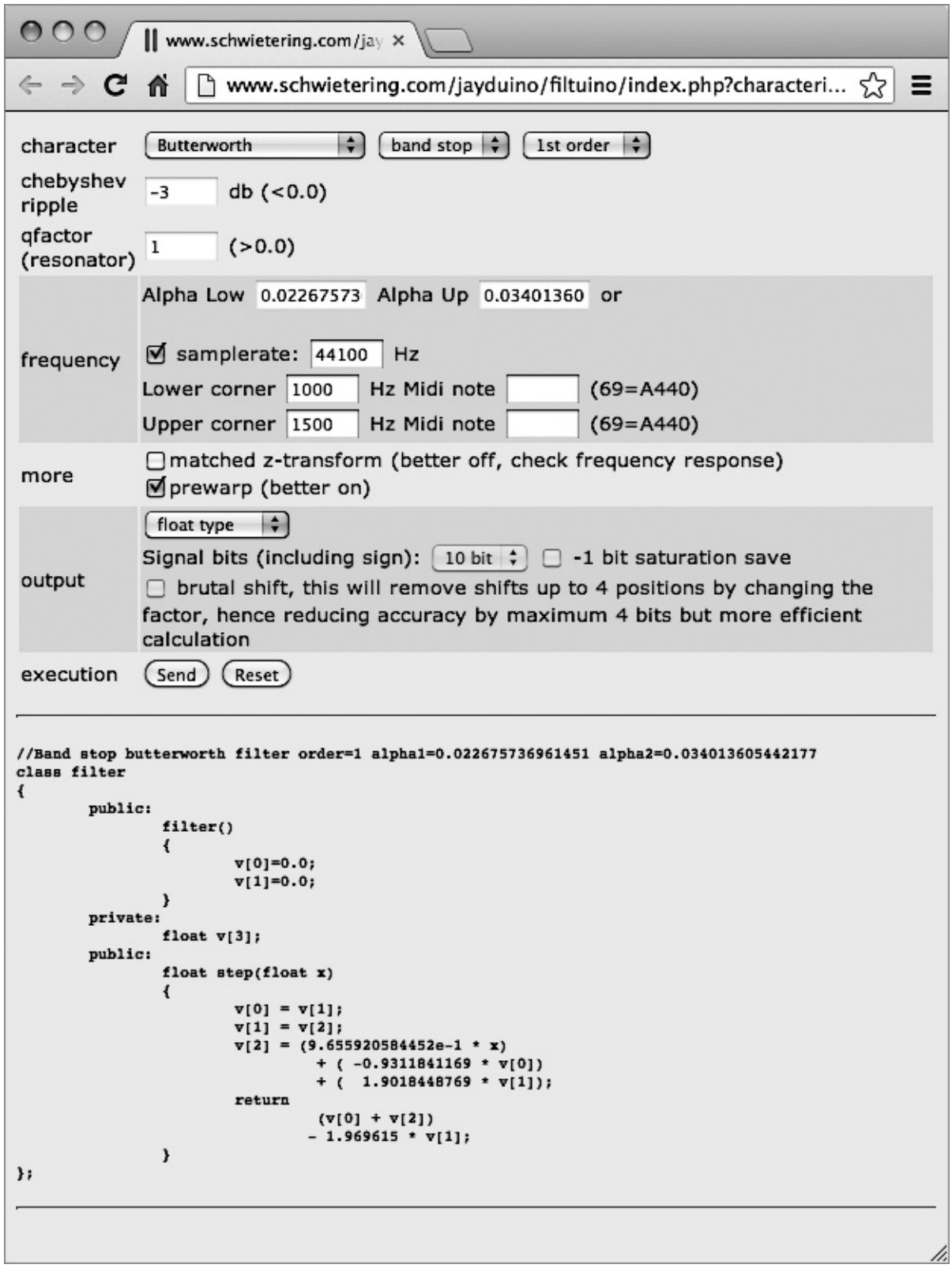


Рис. 13.8. Генератор реализаций фильтров для Arduino

(выходной сигнал) можно выбрать тип массива значений, который будет использоваться для фильтрации. Я выбрал тип **float type** (вещественный). В заключение щелкнул на кнопке **Send** (Отправить), чтобы сгенерировать код.

Для проверки можно взять за основу пример скетча null filter, который использовался в эксперименте с платой Due. Полный скетч можно найти в пакете примеров под именем sketch_13_05_band_stop_due.

Сначала скопируйте сгенерированный код в буфер обмена и вставьте его в базовый

пример null filter сразу после определения констант. Добавьте в код комментарий с адресом URL страницы генератора, чтобы при необходимости можно было вернуться и изменить настройки фильтра: URL хранит выбранные вами значения параметров. Сгенерированный фильтр оформлен в виде класса. Мы еще встретимся с классами в главе 15. А пока просто считайте его черным ящиком, выполняющим фильтрацию.

После вставленного кода добавьте следующую строку:

```
filter f;
```

Теперь нужно изменить функцию `loop`, чтобы вместо простого вывода входного сигнала плата Arduino выводила отфильтрованные значения:

```
void loop()
{
  static long lastSampleTime = 0;
  long timeNow = micros();
  if (timeNow > lastSampleTime + samplePeriod)
  {
    int raw = analogRead(analogInPin);

    float filtered = f.step(raw);

    analogWrite(analogOutPin, (int)filtered);
    lastSampleTime = timeNow;
  }
}
```

Чтобы получить отфильтрованное значение, достаточно просто вызвать функцию `f.step` и передать ей значение, прочитанное с аналогового входа. Возвращаемое значение этой функции и есть отфильтрованное значение, которое нужно привести к типу `int` перед записью в ЦАП.

Заглянув в функцию `step`, можно увидеть, что реализация фильтра хранит три последних и один текущий замер. Далее производятся некоторые манипуляции со значениями, а затем они масштабируются коэффициентами, чтобы получилось возвращаемое значение. Разве математика не прекрасна?

На рис. 13.9 показан результат фильтрации. С помощью генератора воспроизводились сигналы разной частоты, амплитуда выходного сигнала (измерялась с помощью осциллографа) записывалась в электронную таблицу, и затем по массиву полученных данных строился график.

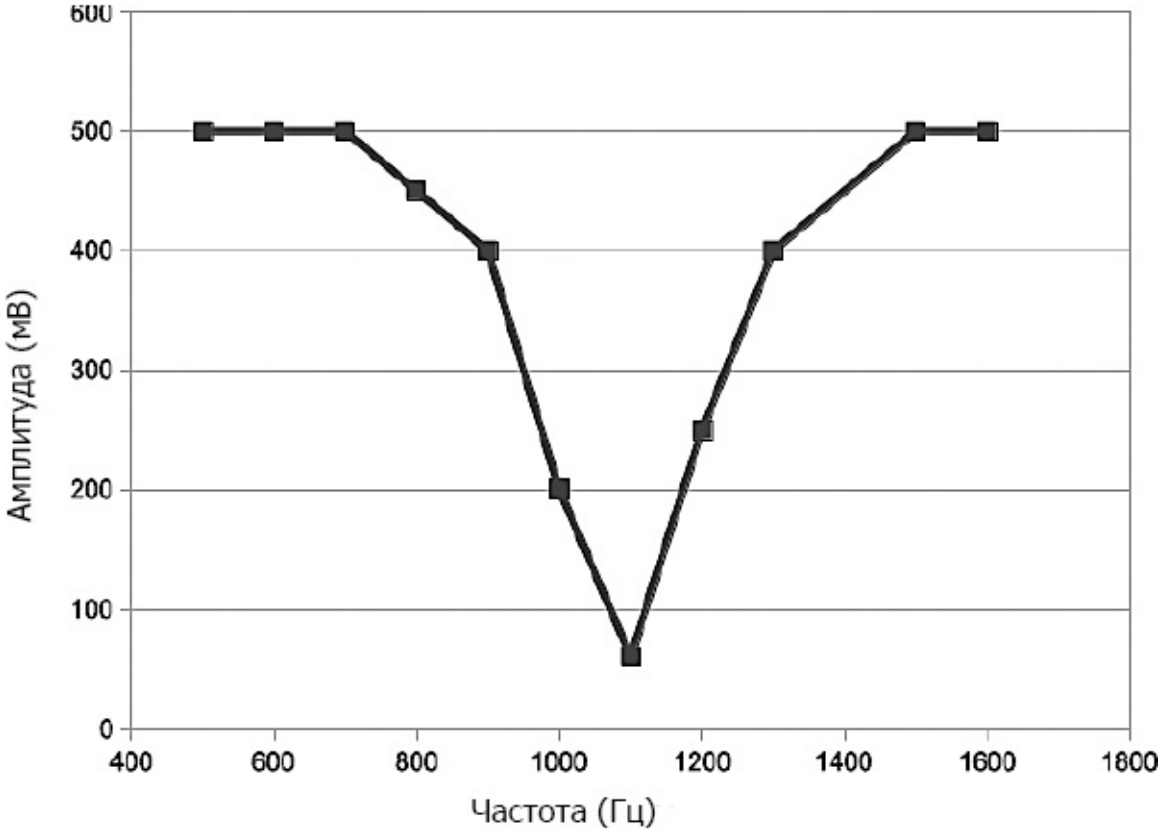


Рис. 13.9. АЧХ полосового фильтра на основе Arduino

Преобразование Фурье

Преобразование Фурье — удобный инструмент анализа частотных характеристик сигнала. Как уже говорилось во введении к этой главе, сигналы часто формируются путем наложения разного количества синусоид с разной частотой. Возможно, вам приходилось видеть дисплеи анализаторов спектра на музыкальном оборудовании или средства визуализации в программных проигрывателях MP3. Они имеют вид столбиковой диаграммы. Высота каждого столбика соответствует мощности соответствующей полосы частот, при этом низкочастотные басовые ноты отображаются слева, а высокочастотные — справа.

На рис. 13.10 показано, как один и тот же сигнал может быть представлен в виде одной линии (называется *временной областью*) и как множество значений мощности сигнала на разных частотах (называется *частотной областью*).

Алгоритм расчета частотной области из данных временной области сигнала называется *быстрым преобразованием Фурье* (БПФ). В вычислениях преобразования используются комплексные числа, и его реализация — задача не для слабых духом, если только вы не увлекаетесь математикой.

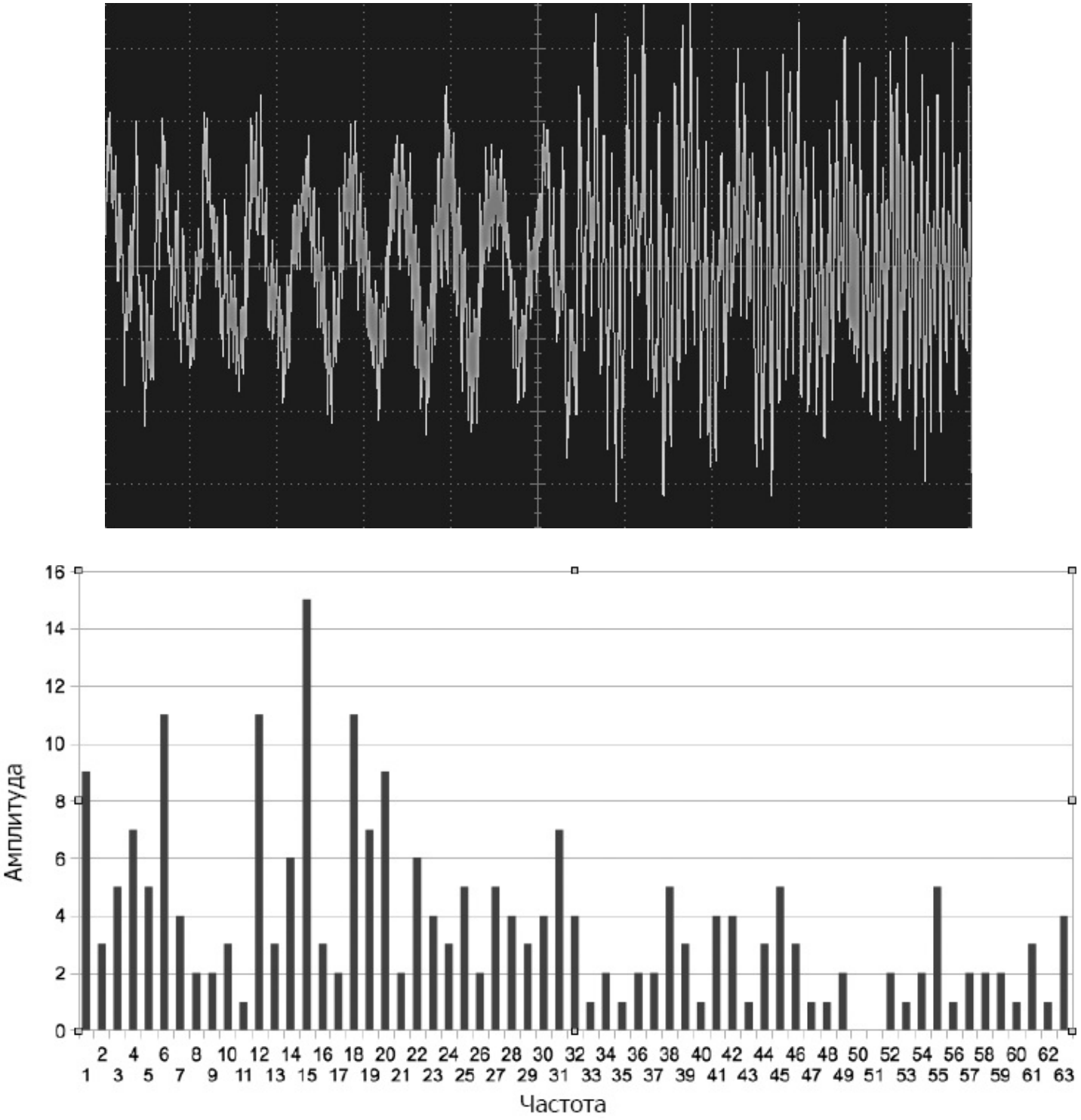


Рис. 13.10. Сигнал во временной и частотной областях

К счастью для нас, умные люди часто готовы поделиться своим кодом. Вы можете загрузить из Интернета функцию, выполняющую быстрое преобразование Фурье. Пример кода, который использовал я, не организован в библиотеку — он распространяется в виде двух файлов: заголовочного файла и файла реализации (с расширением **.h** и **.cpp** соответственно). Чтобы воспользоваться им, просто сохраните оба файла в папку со скетчем. Они находятся в пакете примеров, сопровождающем книгу, поэтому их не придется загружать из Интернета. Впервые код появился в статье на форуме Arduino (http://forum.arduino.cc/index.php/topic,38153.0.html). Эти же два файла можно найти в составе других примеров на следующих веб-сайтах: https://code.google.com/p/arduino-integer-fft/ и https://github.com/slytown/arduino-spectrum-

analyzer/.

Следующие два примера иллюстрируют код, действующий в плате Arduino Uno и выполняющий замеры аудиосигнала.

Пример анализатора спектра

Этот пример плата Arduino Uno использует для получения текстового отображения частотного спектра. Исходный код можно найти в скетче sketch_13_06_FFT_Spectrum. Скетч слишком длинный, чтобы воспроизводить его здесь целиком, поэтому я буду демонстрировать лишь некоторые его фрагменты. Откройте скетч в Arduino IDE, чтобы заглядывать в него в ходе обсуждения.

Алгоритм БПФ использует два массива типа `char`. Этот тип выбран вместо типа `byte` по той простой причине, что в Arduino C тип `byte` представляет однобайтовые целые числа без знака, тогда как сигнал, подлежащий преобразованию, как предполагается, будет колебаться относительно значения 0. После применения алгоритма БПФ массив `data` будет содержать мощность каждой частотной составляющей в заданном диапазоне. Диапазон частот зависит от частоты выполнения замеров. Данный скетч выполняется с максимальной скоростью, на которую только способна плата Uno, и обеспечивает анализ полосы частот с верхней границей около 15 кГц, что для каждого из 63 равномерно распределенных частотных интервалов дает ширину 240 Гц.

Чтобы максимально быстро выполнить аналоговое преобразование и получить приличную частоту замеров, был использован трюк увеличения аналогово-цифрового преобразования, обсуждавшийся в главе 4. Он заключен в следующих двух строках в функции `setup`:

```
ADCSRA &= ~PS_128;  // сбросить масштаб 128
ADCSRA |= PS_16;    // добавить масштаб 16 (1 МГц)
```

Функция `loop` содержит совсем немного кода:

```
void loop()
{
  sampleWindowFull();
  fix_fft(data, im, 7, 0);
  updateData();

  showSpectrum();
}
```

Функция `sampleWindowFull` заполняет временное окно 128 замерами данных. Я

расскажу о ней чуть позже. Затем к данным применяется алгоритм БПФ. Параметр 7 — это логарифм по основанию 2 от числа замеров. Это значение всегда будет равно 7. Параметр 0 — это признак инверсии, который также всегда будет равен 0, что означает `false`. После применения алгоритма БПФ производится обновление значений в массивах. В заключение вызывается функция `showSpectrum`, отображающая частотную информацию.

Функция `sampleWindowFull` читает значение аналогового входа 128 раз и предполагает, что сигнал колеблется относительно средней точки 2,5 В, поэтому она вычитает 512 из прочитанного значения, в результате чего может получиться положительное или отрицательное значение. Затем оно масштабируется константой `GAIN`, чтобы немного усилить слабые сигналы. Далее 10-битный замер делением на 4 преобразуется в 8-битное значение, чтобы можно было уместить его в массив типа `char`. Массив `im` хранит мнимую часть сигнала, установленную в 0. Это внутренняя особенность алгоритма; желающие больше узнать об этом могут обратиться к статье **https://ru.wikipedia.org/wiki/Быстрое_преобразование_Фурье**.

```
void sampleWindowFull()
{
  for (int i = 0; i < 128; i++)
  {
            int   val   =   (analogRead(analogPin)   –   512)   *
GAIN;
    data[i] = val / 4;
    im[i] = 0;
  }
}
```

Функция `updateData` вычисляет амплитуду в каждом частотном интервале. Сила сигнала вычисляется как длина гипотенузы прямоугольного треугольника, двумя другими сторонами которого являются действительная и мнимая части сигнала (практическое применение теоремы Пифагора!):

```
void updateData()
{
  for (int i = 0; i < 64; i++)
  {
    data[i] = sqrt(data[i] * data[i] + im[i] * im[i]);
  }
}
```

Результаты выводятся в монитор последовательного порта в одну строку через

запятую. Первое значение игнорируется, потому что содержит постоянную составляющую сигнала и обычно не представляет интереса.

Массив `data` можно было бы использовать, например, для управления высотой столбиков диаграммы на жидкокристаллическом дисплее. Подключить источник сигнала (например, аудиовыход MP3-плеера) можно с помощью той же схемы, обеспечивающей колебание сигнала относительно средней точки 2,5 В, что была показана ранее, на рис. 13.4.

Пример измерения частоты

В этом, втором примере плата Arduino Uno используется для вывода оценки частоты сигнала в монитор последовательного порта (sketch_13_07_FFT_Freq). Большая часть кода в этом скетче повторяет код из предыдущего примера. Главное отличие в том, что после обработки массива `data` определяется индекс элемента с наибольшим значением и используется для оценки частоты. Затем функция `loop` выводит это значение в монитор последовательного порта.

В заключение

Цифровая обработка сигналов — сложная тема, ей посвящено множество отдельных книг. Из-за ее сложности я коснулся только наиболее полезных приемов, которые можно попробовать применить при использовании платы Arduino.

В следующей главе мы обратимся к проблеме, возникающей при желании одновременно решать несколько задач в Arduino. С этой проблемой часто сталкиваются те, кто имеет опыт программирования в больших системах, где несколько потоков выполнения, действующих одновременно, считаются нормой.

14. Многозадачность с единственным процессом

Программисты, пришедшие в мир Arduino из мира больших систем, часто отмечают отсутствие поддержки многозадачности в Arduino как существенное упущение. В этой главе я попробую исправить его и покажу, как преодолеть ограничения однопоточной модели встроенных систем.

Переход из мира программирования больших систем

Плата Arduino привлекла множество энтузиастов (в том числе и меня), которые работают в индустрии программного обеспечения не один год, имеют опыт работы в составе коллективов из десятков человек, объединяющих усилия для создания больших программных продуктов, и привыкли решать все возникающие проблемы. Для нас возможность без продолжительного проектирования написать несколько строк кода и практически немедленно получить какое-нибудь интересное проявление в физическом мире является отличным противоядием от привычек, прививаемых в мире большого программного обеспечения.

Однако накопленный опыт часто заставляет нас искать в Arduino то, чем мы привыкли пользоваться в повседневной работе. При переходе из мира больших систем в миниатюрный мир Arduino первое, что сразу бросается в глаза, — это простота разработки программ для Arduino. Приступать к созданию большой системы без использования приемов разработки через тестирование, системы управления версиями и процедуры гибкой разработки было бы слишком опрометчиво. В то же время большой проект для Arduino может состоять всего из 200 строк кода, написанных одним человеком. Если этот человек — опытный программист, он просто будет хранить все тонкости в памяти, не нуждаясь в привлечении инструментов, обычных в большой разработке.

Поэтому прекращайте беспокоиться об управлении версиями, шаблонах проектирования, создании модульных тестов и поддержке рефакторинга в среде разработки и просто почувствуйте радость от простоты Arduino.

Почему вам не нужны потоки выполнения

Если вам так много лет, что вы застали времена, когда большое распространение имели домашние компьютеры, программируемые на Бейсике, вы должны помнить, что компьютеры «в каждый момент времени делают что-то одно». Если игре, написанной на Бейсике, требовалось одновременно перемещать несколько спрайтов, вам приходилось прибегать к уловке с общим циклом, в котором каждый спрайт перемещался на небольшое расстояние.

Этот образ мышления отлично подходит для программирования Arduino. Вместо создания множества потоков выполнения, каждый из которых отвечает за перемещение единственного спрайта, достаточно единственного потока, перемещающего спрайты по очереди небольшими шагами, не блокируя ничто другое.

Компьютеры, кроме тех, что имеют многоядерные процессоры, действительно в каждый момент времени могут делать что-то одно. Операционная система переключает внимание процессора между процессами, действующими в компьютере. В Arduino, где нет операционной системы и потребность в решении нескольких задач одновременно возникает довольно редко, многозадачность можно реализовать самостоятельно.

Функции setup и loop

В каждом скетче требуется реализовать две функции, `setup` и `loop`, и такой подход выбран не случайно. Фактически функция `loop` вызывается снова и снова, и именно по этой причине ее работа не должна блокироваться. Код в функции `loop` должен действовать виртуозно, чтобы она выполнялась моментально и тут же запускалась вновь.

Оценка, затем действие

Большинство проектов для Arduino предназначено для управления чем-то. Поэтому функция `loop` часто:

- проверяет нажатие кнопки или превышение данных с некоторого датчика порогового значения;
- выполняет соответствующее действие.

Простым примером может служить реализация мигания светодиода в результате нажатия кнопки.

Следующий пример иллюстрирует это. Однако, как будет показано далее, иногда необходимость ждать, пока выполняется код, управляющий миганием светодиода, бывает совершенно неприемлемой:

```
// sketch_14_01_flashing_1

const int ledPin = 13;
const int switchPin = 5;
const int period = 1000;
```

```
boolean flashing = false;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(switchPin, INPUT_PULLUP);
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {
    flashing = ! flashing;
  }
  if (flashing)
  {
    digitalWrite(ledPin, HIGH);
    delay(period);
    digitalWrite(ledPin, LOW);
    delay(period);
  }
}
```

Проблема данной реализации в том, что она проверяет нажатие кнопки только после того, как завершится цикл включения/выключения светодиода. Если кнопка будет нажата во время этого цикла, факт нажатия зафиксирован не будет. Это может быть не важно для нормальной работы скетча, но если важно фиксировать каждое нажатие кнопки, следует полностью исключить любые задержки в функции `loop`. Фактически после перехода в режим мигания Arduino будет тратить основное время на задержки и только малую часть времени — на проверку состояния кнопки.

Пример в следующем разделе решает эту проблему.

Пауза без приостановки

Предыдущий скетч можно переписать без использования функции `delay`:

```
// sketch_14_02_flashing_2

const int ledPin = 13;
const int switchPin = 5;
```

```
const int period = 1000;

boolean flashing = false;
long lastChangeTime = 0;
int ledState = LOW;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(switchPin, INPUT_PULLUP);
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {
    flashing = ! flashing;
    // и выключить светодиод
    if (! flashing)
    {
      digitalWrite(ledPin, LOW);
    }
  }
  long now = millis();
  if (flashing && now > lastChangeTime + period)
  {
    ledState = ! ledState;
    digitalWrite(ledPin, ledState);
    lastChangeTime = now;
  }
}
```

В этом скетче я добавил две новые переменные, `lastChangeTime` и `ledState`. Переменная `lastChangeTime` хранит время последнего перехода светодиода между состояниями «включено» и «выключено», а переменная `ledState` хранит текущее состояние светодиода — «включено» или «выключено», чтобы знать, в каком состоянии он пребывает, когда потребуется переключить его.

Теперь функция `loop` не выполняет задержек. В первой части `loop` проверяется нажатие кнопки, и, если кнопка нажата, переключается режим мигания.

Дополнительная инструкция `if`, следующая далее, просто выключает светодиод, если нажатие кнопки вызвало выключение режима мигания. В противном случае светодиод мог бы остаться включенным:

```
if (! flashing)
{
  digitalWrite(ledPin, LOW);
}
```

Во второй части функция `loop` читает текущее значение счетчика миллисекунд вызовом `millis()` и сравнивает со значением `lastChangeTime`, увеличенным на значение `period`. То есть код внутри этой инструкции `if` выполняется, только если с момента последнего переключения светодиода прошло более `period` миллисекунд.

Затем значение переменной `ledState` изменяется на противоположное, и на цифровом выходе устанавливается соответствующий уровень напряжения. Потом значение `now` копируется в `lastChangeTime`, чтобы можно было определить, когда наступит момент следующего переключения.

Библиотека Timer

Решение, представленное в разделе «Пауза без приостановки», было обобщено и реализовано в виде библиотеки, позволяющей планировать выполнение повторяющихся операций с использованием функции `millis`. Несмотря на свое название, библиотека не использует аппаратные таймеры и потому прекрасно работает в большинстве моделей Arduino.

Получить библиотеку можно по адресу **http://playground.arduino.cc//Code/Timer**.

Применение библиотеки может существенно упростить код, как показано далее:

```
// sketch_14_03_flashing_3

#include <Timer.h>

const int ledPin = 13;
const int switchPin = 5;
const int period = 1000;

boolean flashing = false;
int ledState = LOW;
Timer t;
```

```
void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(switchPin, INPUT_PULLUP);
  t.every(period, flashIfRequired);
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {
    flashing = ! flashing;
    if (! flashing)
    {
      digitalWrite(ledPin, LOW);
    }
  }
  t.update();
}
void flashIfRequired()
{
  if (flashing)
  {
    ledState = ! ledState;
    digitalWrite(ledPin, ledState);
  }
}
```

Чтобы задействовать возможности библиотеки, необходимо определить объект таймера (в данном скетче он получил имя `t`) и в функции `setup` указать функцию для вызова через установленные периоды:

```
t.every(period, flashIfRequired);
```

Затем нужно добавить в функцию `loop` следующую строку:

```
t.update();
```

В каждом вызове функция `update` проверит значение `millis`, определит необходимость выполнения повторяющихся действий и, если пришло время для этого, вызовет указанную функцию (в данном случае `flashIfRequired`).

Библиотека Timer имеет также множество других вспомогательных функций. Более подробную информацию о ней можно найти, если пройти по ссылке, приведенной в начале раздела.

В заключение

В этой главе вы узнали, как организовать решение сразу нескольких задач без использования механизма потоков выполнения. Для этого достаточно рассмотреть ограничения, накладываемые микроконтроллером, под другим углом.

В заключительной главе вы узнаете, как поделиться своими разработками с сообществом Arduino, создавая и публикуя библиотеки для Arduino.

15. Создание библиотек

Рано или поздно вы создадите нечто замечательное, что, по вашему мнению, могли бы использовать другие. Это будет самый подходящий момент оформить свой код в виде библиотеки и выпустить ее в свет. Эта глава покажет вам, как это сделать.

Когда создавать библиотеки

Библиотеки создаются не только разработчиками Arduino — любой пользователь Arduino может создать свою библиотеку. Если библиотека окажется по-настоящему полезной, в адрес ее создателя потечет поток благодарностей. Никто не занимается продажей библиотек, так как это противоречит ценностям сообщества Arduino. Библиотеки должны публиковаться с открытым исходным кодом и рассматриваться как помощь своим собратьям по увлечению Arduino.

Часто наиболее полезными оказываются библиотеки, дающие доступ к некоторому аппаратному обеспечению. Они нередко упрощают процесс использования аппаратных средств и некоторых сложных протоколов. Нет причин, почему более одного человека должны преодолевать тернистый путь, выясняя, как работает некоторая аппаратура, и, если вы опубликуете свою библиотеку, благодаря Интернету люди смогут отыскать ее.

СОВЕТ

Прикладной программный интерфейс (Application Programmer Interface, API) — это комплект функций, которые пользователь библиотеки будет включать в свой скетч. Проектируя API, всегда задавайте себе вопрос: «Что действительно волнует пользователя?» Низкоуровневые детали реализации должны быть максимально скрыты. В примере библиотеки, который будет представлен в разделе «Пример библиотеки (радиоприемник TEA5767)», мы продолжим обсуждение этой проблемы.

Использование классов и методов

Создатели скетчей обычно полагают, что пишут код на языке C, и используют довольно ограниченный круг возможностей, в действительности скетчи пишутся на языке C++, объектно-ориентированном расширении языка C. В этом языке используется понятие *классов* объектов, объединяющих информацию об объектах (их данные) и функции, которые применяются к этим данным. Эти функции похожи на обычные функции, но, когда они ассоциируются с конкретным классом, их называют *методами*. Кроме того, методы могут объявляться общедоступными, и тогда они доступны всем, кто пользуется ими, или приватными, и тогда они доступны только из

методов внутри класса.

Я говорю об этом, потому что создание расширений — один из немногих видов деятельности в мире Arduino, где использование классов считается нормой. Класс — отличный способ заключить все необходимое в некое подобие модуля. Кроме того, различия между закрытыми и общедоступными членами помогают спроектировать API так, чтобы заставить разработчиков скетчей думать о том, как взаимодействовать с библиотекой (общедоступные члены), а не о том, как она работает (приватные члены).

В дальнейшем в процессе изучения примеров вы увидите, как пользоваться классами.

Пример библиотеки (радиоприемник TEA5767)

Чтобы показать, как писать библиотеки для Arduino, я возьму программный код, который использовался в главе 7 для работы с УКВ-приемником TEA5767, и превращу его в библиотеку. Скетч очень прост и в основном занимается настройкой библиотеки, тем не менее он служит хорошим примером.

Далее перечислены основные этапы процесса.

1. Определение API.
2. Создание заголовочного файла.
3. Создание файла реализации.
4. Создание файла с ключевыми словами.
5. Создание нескольких примеров.

С точки зрения файлов и папок библиотека включает папку с именем, совпадающим с именем библиотечного класса. В данном случае я выбрал имя TEA5767Radio для библиотеки и класса. Внутри этой папки должны находиться два файла: **TEA5767Radio.h** и **TEA5767Radio.cpp**.

При желании можно также создать файл **keywords.txt** и папку **examples**, содержащую примеры скетчей, использующих библиотеку. Структура папок для этого примера показана на рис. 15.1.

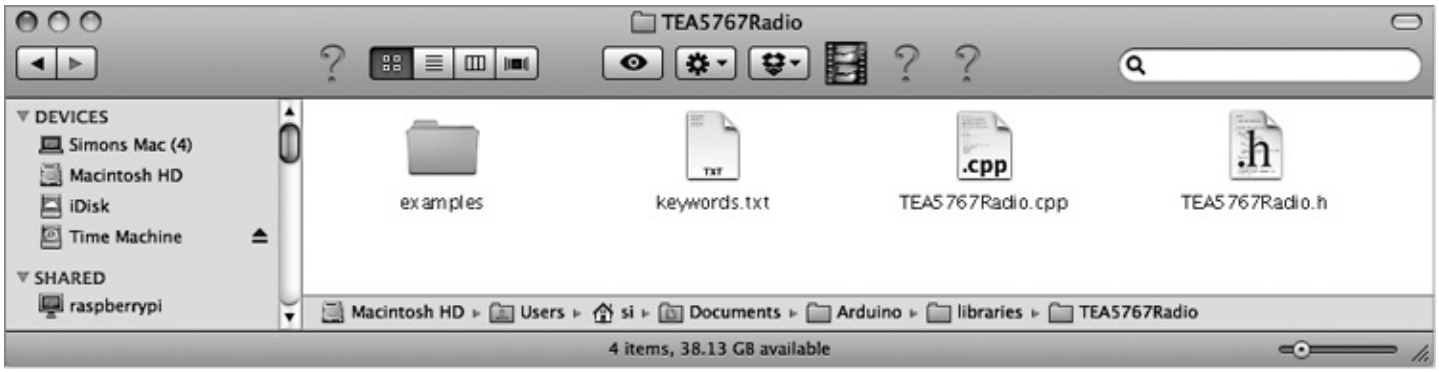


Рис. 15.1. Структура папок примера проекта

Библиотеку проще использовать, если она находится непосредственно в папке **libraries** в Arduino IDE, куда устанавливаются другие сторонние библиотеки. Файлы можно править прямо в этой папке. Среда разработки Arduino IDE заметит существование библиотеки только после перезапуска, но после этого любые изменения в содержимом файлов будут учитываться во время сборки проекта автоматически.

Оригинальный скетч, на котором основана библиотека, находится в файле **sketch_07_01_I2C_TEA5767**, а законченную библиотеку можно загрузить по адресу **http://playground.arduino.cc//Main/TEA5767Radio**.

Определение API

На первом этапе определяется интерфейс, которым будут пользоваться другие.

Если вам доводилось пользоваться разными библиотеками, вы наверняка заметили, что обычно они соответствуют двум шаблонам. Самый простой иллюстрирует библиотека Narcoleptic. Чтобы задействовать эту библиотеку, достаточно подключить ее и затем обращаться к ее методам, предваряя их имена именем библиотеки Narcoleptic, как в следующем фрагменте:

```
#include <Narcoleptic.h>
// и потом где-то в скетче
Narcoleptic.delay(500);
```

Этот же шаблон использует библиотека Serial. Если библиотека поддерживает только один объект, то применение этого шаблона оправданно. Но если есть вероятность, что в скетче потребуется использовать несколько объектов, то необходим другой подход. Поскольку есть шанс, что к плате Arduino будет подключено более одного радиоприемника, данный пример попадает во вторую категорию.

В таких ситуациях применяется шаблон, сходный с тем, что используется библиотекой SoftwareSerial. Чтобы на одной плате организовать несколько последовательных портов, обслуживаемых программно, скетч создает именованные экземпляры библиотеки SoftwareSerial, используя следующий синтаксис:

```
#include <SoftwareSerial.h>
SoftwareSerial mySerial(10, 11); // RX, TX
```

Когда возникнет потребность использовать данный конкретный последовательный порт (связанный с контактами 10 и 11), достаточно создать именованный объект для его обслуживания — в данном случае `mySerial` — и затем выполнить в него запись, как показано далее:

```
mySerial.begin(9600);
mySerial.println("Hello World");
```

Не заботясь о конкретной реализации, определим, как должен выглядеть код, использующий библиотеку.

После импортирования библиотеки было бы желательно иметь возможность создать именованный радиоприемник и определить, какой адрес I2C он будет обслуживать. Для простоты неплохо было бы иметь на выбор два варианта: первый, с использованием адреса по умолчанию 0x60, и второй, позволяющий определять нестандартный адрес:

```
#include <TEA5767Radio.h>
TEA5767Radio radio = TEA5767Radio();
// или TEA5767Radio radio = TEA5767Radio(0x60);
```

Поскольку библиотека обслуживает УКВ-радиоприемник, она должна уметь настраивать частоту, то есть пользователь должен иметь возможность написать такой код:

```
radio.setFrequency(93.0);
```

Числовой параметр здесь представляет частоту в мегагерцах. Автор скетча хотел бы задавать частоту именно в таком виде, а не в малопонятном формате `int`, который передается аппаратному модулю. Библиотека должна скрывать технические детали и самостоятельно выполнять необходимые преобразования.

Это все, что касается проектирования данной библиотеки. Теперь приступим к созданию кода.

Заголовочный файл

Обычно программный код библиотеки хранится в двух файлах. Один из них называется заголовочным и имеет расширение **.h**. Этот файл будет указываться в скетчах в инструкции `#include`. Заголовочный файл не содержит выполняемого программного кода, он лишь определяет имена классов и методов в библиотеке. Далее приводится заголовочный файл примера библиотеки:

```
#include <Wire.h>

#ifndef TEA5767Radio_h
#define TEA5767Radio_h

class TEA5767Radio
{
private:
int _address;
public:
  TEA5767Radio();
  TEA5767Radio(int address);
  void setFrequency(float frequency);
};

#endif
```

Команда `#ifndef` предотвращает многократное импортирование библиотеки и представляет стандартный прием для заголовочных файлов.

Далее следует определение класса, который имеет приватный (`private`) раздел с единственной переменной `_address`. Эта переменная хранит адрес I2C устройства.

Общедоступный раздел (`public`) включает две функции для создания объекта-радиоприемника, одна позволяет указать адрес устройства, а другая не позволяет и использует адрес по умолчанию. В общедоступном разделе имеется также функция `setFrequncy`.

Файл реализации

Код, фактически реализующий функции, находится в файле **TEA5767Radio.cpp**:

```
#include <Arduino.h>
#include <TEA5767Radio.h>

TEA5767Radio::TEA5767Radio(int address)
{
  _address = address;
}

TEA5767Radio::TEA5767Radio()
{
```

```
  _address = 0x60;
}

void TEA5767Radio::setFrequency(float frequency)
{
        unsigned int frequencyB = 4 * (frequency *
          1000000 + 225000) / 32768;
        byte frequencyH = frequencyB >> 8;
        byte frequencyL = frequencyB & 0XFF;
        Wire.beginTransmission(_address);
        Wire.write(frequencyH);
        Wire.write(frequencyL);
        Wire.write(0xB0);
        Wire.write(0x10);
        Wire.write(0x00);
        Wire.endTransmission();
        delay(100);
}
```

Создают новый объект радиоприемника два метода. Оба просто записывают в переменную `_address` адрес I2C устройства, либо переданный в параметре `address`, либо адрес по умолчанию 0x60. Метод `setFrequency` содержит почти весь код из оригинального скетча (sketch_07_01_I2C_TEA5767), кроме следующей строки, использующей значение переменной `_address`, чтобы соединиться с устройством I2C:

```
Wire.beginTransmission(_address);
```

Файл с ключевыми словами

Папка с библиотекой может также содержать файл с именем **keywords.txt**. Этот файл не является обязательным, библиотека будет работать и без него. Но этот файл позволяет среде разработки Arduino IDE выделять цветом любые ключевые слова библиотеки. В нашей библиотеке только два таких слова: ее имя `TEA5767Radio` и имя метода `setFrequency`. Файл с ключевыми словами может сдержать комментарии в строках, начинающихся с символа #. Далее приводится содержимое этого файла:

```
#######################################
# Карта подсветки синтаксиса для TEA5767Radio
#######################################
#######################################
# Типы данных (KEYWORD1)
```

```
########################################
TEA5767Radio KEYWORD1
########################################
# Методы и функции (KEYWORD2)
########################################
setFrequency KEYWORD2
```

Ключевые слова должны определяться как KEYWORD1 или KEYWORD2, даже при том что версия 1.4 среды разработки Arduino IDE выделяет ключевые слова обоих типов оранжевым цветом.

Папка с примерами

Если внутри папки библиотеки создать папку с именем **examples**, все скетчи в этой папке автоматически будут регистрироваться средой разработки Arduino IDE во время запуска, и вы сможете получить доступ к ним через меню **Examples** (Примеры) в подменю с именем библиотеки. Примеры скетчей могут быть самыми обычными скетчами, только хранящимися в папке библиотеки. Далее приводится пример использования этой библиотеки:

```
#include <Wire.h>
#include <TEA5767Radio.h>

TEA5767Radio radio = TEA5767Radio();

void setup()
{
  Wire.begin();
  radio.setFrequency(93.0); // выберите свою частоту
}

void loop()
{
}
```

Тестирование библиотеки

Чтобы протестировать библиотеку, достаточно запустить пример скетча, использующего библиотеку. Если вам не повезло (или вы были недостаточно внимательны) и первая попытка скомпилировать библиотеку завершилась неудачей,

прочитайте сообщения об ошибках в информационной области в нижней части окна Arduino IDE.

Выпуск библиотеки

Созданную библиотеку можно передать сообществу. Чтобы другие наверняка могли найти ее, создайте запись на общедоступной вики-странице http://playground.arduino.cc//Main/LibraryList. Библиотеку можно распространять в виде zip-архива, но некоторые предпочитают использовать репозитории на сайтах GitHub, Google Code или других и размещать на вики-странице ссылку на сайт хостинга.

Если вы пожелаете выгрузить свою библиотеку на сайт Arduino Playground, выполните следующие шаги.

1. Протестируйте библиотеку, чтобы убедиться в ее безупречной работе.

2. Создайте zip-архив с папкой библиотеки и дайте ему имя, совпадающее с именем класса, но с расширением **.zip**.

3. Зарегистрируйтесь на сайте www.arduino.cc.

4. Добавьте запись на вики-странице Arduino Playground http://playground.arduino.cc//Main/LibraryList, описывающую библиотеку и порядок ее использования. Посмотрите, как оформлены записи для других библиотек, и скопируйте соответствующие фрагменты вики-разметки. Создайте ссылку, используя, например, текст `[[TEA5767Radio]]`, чтобы создать заполнитель для новой страницы, который появится в списке библиотек в сопровождении знака ?. Щелкните на ссылке, чтобы создать новую страницу и открыть ее в вики-редакторе. Скопируйте вики-код из записи для другой библиотеки (например, TEA5767Radio) и приведите его в соответствие со своей библиотекой.

5. Чтобы выгрузить zip-архив с библиотекой, нужно включить в вики-разметку вкладку, такую как **Attach:TEA5767Radio.zip.** Сохранив страницу, щелкните на ссылке **download** (загрузить) и укажите имя zip-архива для выгрузки на вики-сервер.

В заключение

Создание библиотек — весьма похвальное занятие. Но прежде чем приступать к созданию какой-либо библиотеки, всегда проверяйте, не была ли создана подобная библиотека кем-то другим.

Книги по своей природе не в состоянии охватить все, что хотел бы узнать читатель. Но я надеюсь, что эта книга помогла вам разобраться в некоторых вопросах программирования Arduino.

Заходите ко мне в Twitter, где я зарегистрирован как **@simonmonk2**. Кроме того, подробную информацию об этой и других моих книгах можно найти на моем веб-сайте **www.simonmonk.org**.

Приложение. Компоненты

Эта книга посвящена в основном программированию, но в ней упоминаются и электронные компоненты, пусть их не так уж много. В этом приложении перечисляются компоненты, использовавшиеся в примерах, и некоторые возможные поставщики, у которых можно их приобрести.

Платы Arduino

Популярность Arduino достигла таких высот, что распространенные модели, такие как Uno и Leonardo, можно приобрести практически везде. Менее распространенные модели выпускают компании Adafruit и SparkFun в США, а также CPC в Великобритании. Их веб-сайты перечислены в разделе «Поставщики» в конце приложения.

Платы расширения

Компании Adafruit и SparkFun продают широкий спектр плат расширений для Arduino, как официальных, так и своих собственных. Интересные и недорогие платы расширения и клоны плат Arduino производит компания Seeed Studio.

Далее перечислены платы расширения, использованные в примерах этой книги. Коды продукта приводятся в круглых скобках после названия компании-поставщика.

Плата расширения	Глава	Поставщик
USB Host	11	SparkFun (DEV-09947)
Ethernet	12	Большинство поставщиков
WiFi	12	Большинство поставщиков

Компоненты и модули

Далее перечислены конкретные компоненты и модули, использованные в примерах для этой книги. Коды продукта приводятся в круглых скобках после названия компании-поставщика.

Модуль	Глава	Поставщик
Модуль УКВ-радиоприемника TEA5767 FM	7	eBay
Светодиодная матрица LED Backpack Matrix	7	Adafruit (902)
Модуль часов реального времени DS1307	7	Adafruit (264)
Датчик температуры DS18B20	8	Adafruit (374), SparkFun (SEN-00245)
Восьмиканальный АЦП MCP3008	9	Adafruit (856)

Модуль Venus GPS	10	SparkFun (GPS-11058)

Поставщики

Электронные компоненты и детали, имеющие отношение к Arduino, распространяет множество поставщиков. Далее перечислены лишь некоторые из них.

Поставщик	Адрес URL	Примечания
Adafruit	*www.adafruit.com*	Продукты Adafruit продают также многие местные поставщики по всему миру
SparkFun	*www.sparkfun.com*	Продукты SparkFun продают также многие местные поставщики по всему миру
Seeed Studio	*www.seeedstudio.com*	Необычные и недорогие модули и клоны плат Arduino
Mouser Electronics	*www.mouser.com*	Предлагает широкий диапазон любых электронных компонентов
RadioShack	*www.radioshack.com*	Продает наиболее известные компоненты для Arduino
Digi-Key	*www.digikey.com*	Предлагает широкий диапазон любых электронных компонентов
CPC	*cpc.farnell.com*	Предлагает широкий диапазон любых электронных компонентов в Великобритании
Maplins	*www.maplin.co.uk*	Продает наиболее известные компоненты для Arduino и имеет сеть магазинов в Великобритании