Белов А. В. ПРОГРАММИРОВАНИЕ ARDUINO

Белов А. В.

ПРОГРАММИРОВАНИЕ ARDUINO

СОЗДАЕМ ПРАКТИЧЕСКИЕ УСТРОЙСТВА

Scan & DjVu Bookingolz

Готовые проекты «под ключ» своими руками!

Виртуальный диск с электронными версиями всех программ, инсталляционными пакетами, видеообзорами и справочной информацией

Белов А.В.

ПРОГРАММИРОВАНИЕ ARDUINO

СОЗДАЕМ ПРАКТИЧЕСКИЕ УСТРОЙСТВА

+ ВИРТУАЛЬНЫЙ ДИСК

Наука и Техника, Санкт-Петербург

ББК 32.812
УДК 621.314:621.311.6

Белов А.В.

Программирование ARDUINO. Создаем практические устройства + виртуальный диск. — СПб.: Наука и Техника, 2018. — 272 с., илл.

ISBN 978-5-94387-882-4

Книга посвящена созданию практических устройств с использованием модуля Ардуино. Этот модуль в настоящее время стал очень популярным. Он оказался настолько удачной разработкой и получил настолько широкое распространение в мире, что сегодня признан идеальной основой для изучения премудростей микроконтроллерной техники.

Для данной книги автор специально разработал ряд практических схем и устройств, на основе которых читатель постепенно, от простого к сложному, сможет научиться писать программы и разрабатывать свои устройства на основе модуля Ардуино.

Книга содержит подробное описание каждой включенной в нее программы. Вы узнаете как создается алгоритм, как разрабатывается схема и как пишется программа. Параллельно, на тех же примерах, идет изучение языка программирования Ардуино. Все функции, операторы и другие элементы этого языка подробно описываются перед тем, как они будут использованы в очередной конкретной программе.

Сотни тысяч плат Ардуино используются каждый день, стимулируя людей во всем мире создавать что-то новое и интересное. Книга предназначена для широкого круга радиолюбителей и для всех, кто изучает языки программирования и учится создавать электронные устройства.

Виртуальный диск содержит тексты всех программных примеров из книги, инсталляционный пакет среды разработки IDE, архивы используемых в книге программных библиотек, видеоролики, набор вспомогательной справочной информации и многое другое.

ISBN 978-5-94387-882-4

Автор и издательство не несут ответственности за возможный ущерб, причиненный в ходе использования материалов данной книги

Контактный телефон издательства
(812) 412-70-26

Официальный сайт: www.nit.com.ru

© Белов А.В.
© Наука и Техника (оригинал-макет)

ООО «Наука и Техника».
Лицензия № 000350 от 23 декабря 1999 года.
198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.
Подписано в печать . Формат 70×100 1/16.
Бумага офсетная. Печать офсетная. Объем 17 п. л.
Тираж 1300 экз. Заказ № 1596.

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, г. Ульяновск, ул. Гончарова, 14

СОДЕРЖАНИЕ

Глава 1. Прежде чем начать читать книгу **6**
Здравствуй, Ардуино! 6
Почему стал популярным модуль Ардуино? 7
Как будем осваивать язык Ардуино 8
Кто и зачем создал модуль Ардуино? 10
Глава 2. Ардуино — конструктор для домохозяек **10**
Как же удалось достичь такой популярности? 11
Первые варианты Ардуино 12
Знакомимся с модулем Arduino UNO 15
Полезные упрощения в модуле 17
Группа аналоговых входов 18
Команда аналогового вывода 19
Контакты питания «Power» 21
Поддерживаемые языки программирования 21
Схема распиновки модуля Arduino UNO 23
Глава 3 . Среда разработки IDE **24**
Для чего нужно специальное приложение «Среда разработки Arduino IDE»? 24
Команды и функции языка Ардуино 25
Внутренние библиотеки 27
Скачиваем программный пакет с сайта разработчика 29
Варианты установочных пакетов для Windows 29
Запуск программы 31
Основное окно среды разработки 31
Панель инструментов 33
Выбор номера COM порта в настройках программы 34
Выбор типа используемой платы Ардуино 35
Скетч: открытие, сохранение, загрузка 36
Организация обмена информацией между программой на Ардуино и компьютером 38
Глава 4. Простейшая программа «Hello, world!» **41**
Постановка задачи 41
Схема 42
Алгоритм 45
Первый вариант программы 47
Второй вариант программы 55

Глава 5. Переключаемый светодиод **64**
Постановка задачи 64
Схема 64
Алгоритм 65
Первый вариант программы 65
Второй вариант программы 69
Третий вариант программы 71
Глава 6 . Боремся с дребезгом контактов **74**
Постановка задачи 74
Схема 76
Антидребезг простыми средствами 76
Алгоритм 76
Программа 77
Применение внешней библиотеки Button 79
Метод проверки ожидания стабильного состояния сигнала .. 85
Метод фильтрации сигнала по среднему значению 85
Глава 7. Мигающий светодиод **87**
Постановка задачи 87
Схема 87
Алгоритм 88
Программа 88
Глава 8. Бегущие огни **91**
Постановка задачи 91
Схема 92
Алгоритм 92
Первый вариант программы 94
Второй вариант — используем один универсальный цикл.... 100
Глава 9. Альтернативные способы формирования задержки **104**
Постановка задачи 104
Схема 106
Алгоритм 106
Первый вариант программы 107
Второй вариант программы 110
Глава 10. Работа с прерываниями по таймеру **117**
Постановка задачи 117
Схема 120
Используем внешнюю библиотеку прерываний по таймеру.. 120
Алгоритм 122
Программа 122
Совместное использование таймера 0 126

Глава 11. Формирование звука **135**
Постановка задачи 135
Схема 137
Алгоритм 139
Программа 139
Глава 12. Ввод аналоговой информации **144**
Постановка задачи 144
Схема 145
Алгоритм 146
Программа 148
Глава 13. Вывод аналоговой информации **151**
Широтно-импульсная модуляция 151
Простейший способ аналогового вывода 154
Схема 154
Алгоритм 155
Программа 156
Более сложный пример аналоговой индикации 157
Схема 158
Алгоритм 158
Программа 160
Глава 14. Передача данных из Ардуино на компьютер **166**
Постановка задачи 166
Схема 167
Алгоритм 169
Программа 169
Глава 15. Передача данных с компьютера на Ардуино **178**
Постановка задачи 178
Схема 179
Алгоритм 180
Программа 181
Глава 16. Музыкальная шкатулка **185**
Постановка задачи 185
Схема 191
Алгоритм 191
Программа 192
Глава 17. Кодовый замок **206**
Постановка задачи 206
Схема 210
Алгоритм 212
Программа 214

Глава 18. Кодовый замок с музыкальным звонком 233
Постановка задачи 233
Схема 234
Алгоритм 235
Программа 236
Глава 19. Платы Arduino: особенности и возможности 247
Arduino Due 247
Arduino Leonardo 248
Arduino Yun 248
Arduino Micro 249
Arduino UNO 249
Arduino Ethernet 249
Arduino Duemilanove 250
Arduino Diecimila 251
Arduino Nano 251
Arduino Mega 252
Arduino Mega 2560 252
Arduino ADK 253
Arduino LilyPad 253
Arduino Fio 254
Arduino Mini 254
Arduino Pro 255
Arduino Pro Mini 255
USB Serial Light Адаптер 256
Глава 20. Arduino shields или платы расширения 257
Для чего нужны платы расширения? 257
Плата расширения Arduino WiFi 258
Плата расширения Xbee Shield 258
Плата расширения Arduino Motor 259
Плата расширения Ethernet Shield 259
Глава 21. Подводя итоги.... 260
Приложение 1 .Основные операторы языка Ардуино 261
Главные функции 261
Управляющие операторы 262
Операторы цифрового ввода/вывода 264
Операторы аналогового ввода/вывода 265
Операторы времени 266
Расширенный ввод/вывод 267
Работа с последовательным портом 269
Приложение 2. Типы данных в Arduino IDE 270
Список литературы 271
Список ссылок на ресурсы в интернет 271

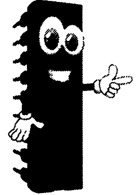
ГЛАВА 1

ПРЕЖДЕ ЧЕМ НАЧАТЬ ЧИТАТЬ КНИГУ

Здравствуй, Ардуино!

Уважаемый читатель! Ты держишь в руках книгу, которую можно использовать как **самостоятельный самоучитель** по разработке устройств на основе модулей Ардуино и программ для них. Но, в то же время, книга фактически является продолжением хорошо известной в России и странах ближнего зарубежья другой книги авторства Александра Белова, которая называется:

«Микроконтроллеры AVR: от азов программирования до создания практических устройств».



ПРИМЕЧАНИЕ.

Это позиция ***[1]*** *в списке литературы в конце данной книги.*

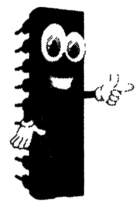
Те читатели, которые уже прочитали книгу **[1]**, знают, что она дает основы знаний по цифровой и микропроцессор-

ной технике, начиная от азов и заканчивая рядом примеров по разработке конкретных несложных микропроцессорных устройств.

Изучение материала в [1] происходит на примере микроконтроллеров AVR фирмы Atmel. При этом уроки программирования даются сразу на двух языках:

- языке Ассемблера;
- языке СИ.

Однако, после выхода книги [1] в мире получил широкое распространение микроконтроллерный модуль Ардуино. Разработка устройств на основе этого модуля и разработка программ для них — это новое, оригинальное, бурно развивающееся направление, ориентированное на начинающих и самодеятельных конструкторов. Это подвигло автора написать НОВУЮ книгу, которую можно считать как бы продолжением к известной книге [1].



ПРИМЕЧАНИЕ.

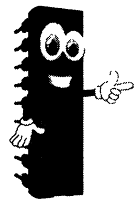
В новой книге на тех же примерах, на которых в [1] читатель постигал секреты конструирования и программирования классическими методами, автор решил рассказать, как это можно сделать средствами Ардуино.

В то же время, решено было построить НОВУЮ книгу так, чтобы читатель мог пользоваться ей как абсолютно самостоятельным самоучителем по Ардуино.

Почему стал популярным модуль Ардуино?

Модуль Ардуино — это небольшая плата, собранная на микроконтроллере AVR, которая служит основой для создания

любых электронных устройств управления, сигнализации и автоматики, которые могут применяться в самых разных областях электронной техники.



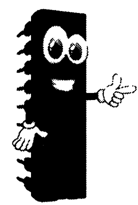
ПРИМЕЧАНИЕ.

Ардуино является отличной альтернативой для начинающих разработчиков благодаря тому, что в модуле использован ряд оригинальных решений упрощающих разработку как схем, так и программ.

Модуль не требует отдельного программатора и предоставляет разработчику простые средства обмена информацией с компьютером по последовательному USB каналу. Ардуино оказался настолько удачной разработкой и получил настолько широкое распространение во всем мире, что теперь именно он считается самой удачной основой для изучения азов микроконтроллерной техники.

Как будем осваивать язык Ардуино

Книга, которую вы держите сейчас в руках, содержит набор специально разработанных примеров, на основе которых читатель постепенно, от простого к сложному, учится разрабатывать схемы и создавать программы с использованием модуля Ардуино.



ПРИМЕЧАНИЕ.

*За основу взяты примеры из **[1]**, которые дополнены целым рядом дополнительных примеров, иллюстрирующих либо различные способы решения одной и той же задачи, плюс несколько примеров, решающих абсолютно новые задачи.*

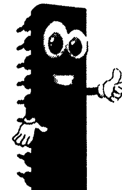
Каждый пример снабжен:

- подробным описанием решаемой проблемы;
- четко сформулированной постановкой задачи;
- описанием алгоритма;
- подробным описанием схемы и готовой программы.

Традиционно, в книге вы увидите очень подробное описание каждой программы.

Параллельно идет изучение языка программирования Ардуино. Все функции, операторы и другие элементы языка программирования подробно описываются перед тем, как они будут использованы в очередной конкретной программе.

Тем читателям, кто прежде, чем начать изучение данной книги, все-таки решил предварительно изучить книгу [1], следует знать, что вместо книги [1], с таким же успехом, можно использовать одно из предыдущих ее изданий. В списке литературы, приведенном в конце данной книги, вы найдете все эти издания. А на обратной стороне книги вы увидите обложку книги [1] (издание 2-е).



СОВЕТ.

Использовать можно позиции [2], [3], [4], [5], [6], [7] или [8] списка. Конечно, лучше всего использовать самое последнее издание, то есть книгу [1]. Но если у вас уже есть другая книга из приведенного выше списка, то она вполне подойдет в том случае, если вас интересует в конечном итоге работа с Ардуино. При этом можно даже опустить в ***[1] главу 6****.*

ГЛАВА 2

АРДУИНО – КОНСТРУКТОР ДЛЯ ДОМОХОЗЯЕК

Кто и зачем создал модуль Ардуино?

Модуль Ардуино придумали пятеро друзей из маленького итальянского городка Ивреа, стоящего на реке Дора Балтея. Городок знаменит своими королями-неудачниками. В 1002 году король по имени **АРДУИН** стал правителем страны, но через два года был свергнут королем Германии Генри II.

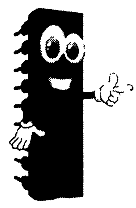
Сегодня есть бар ди Ре Ардуино, расположенный в исторической части этого городка. Назван бар в честь короля и стоит на том самом месте, где, по преданию, родился король. Бар является пивнушкой Массимо Банци (Massimo Banzi), итальянского соучредителя уникального проекта в сфере электроники, который был назван в честь этого места.

Появился модуль в 2005 году. Создателей Ардуино звали так:

- Дэвид Куар-тилльз (David Cuartielles);
- Джанлука Мартино (Gianluka Martino);
- Том Иго (Tom Igoe);
- Дэвид Мелис (David Mellis);
- Массимо Банци (Massimo Banzi).

Эта компания из пяти единомышленников разработала **учебное пособие** для студентов Института проектирования

взаимодействий города Иврея. При помощи этого модуля студенты изучали микроэлектронику.



ПРИМЕЧАНИЕ.

Название модуля *было придумано позже по имени упонянутого ранее бара «Ардуино», в котором любили собираться друзья и обсуждать там все свои идеи.*

Модуль получился настолько удачным, что моментально стал известен во всем мире! Ардуино — недорогая микроконтроллерная плата, которая позволяет даже новичку делать по-настоящему удивительные вещи.

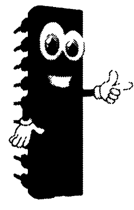
Как же удалось достичь такой популярности?

Прежде всего, разработчики Ардуино (первоначально он вообще не имел названия, но будем называть его так) решили упростить процесс прошивки программ в программную память микроконтроллера. Традиционно, для прошивки микроконтроллеров использовались отдельные устройства — **программаторы**.

Программатор подключается к компьютеру, а уже к программатору подключается микроконтроллер. Разработчики решили избавиться от такого вспомогательного устройства, как программатор. Решение напрашивалось само собой. Большинство выпускаемых микроконтроллеров к тому времени уже имело такую полезную функцию, как возможность **самопрограммирования**.

Для этого программная память микроконтроллера **разбивалась на две области** соответствующими настройками (точнее установкой его битов конфигурации FUSE-переключателями):

- часть памяти выделялась под область загрузчика;
- оставшаяся же память использовалась для размещения туда основной рабочей программы.



ЧТО ЕСТЬ ЧТО.

Область загрузчика *— это защищенная область программной памяти. Программный код, записанный в эту область, не может быть переписан* ***самопрограммированием****.*

Такая конфигурация программной памяти позволяла программисту поместить в область загрузчика программу, которая в нужный момент, используя любые внешние каналы связи микроконтроллера, будет:

- получать извне коды нового варианта рабочей программы;
- прошивать эти коды в основную часть программной памяти.

Подробнее об этом вы можете прочитать в **[1]**. **Программа-загрузчик** обычно разрабатывается программистом, и алгоритмы ее работы зависят от его квалификации и фантазии. В настоящее время технология самопрограммирования широко используется в самых разных электронных устройствах. Она позволяет заливать обновленные версии программ, используя самые разные источники данных. Например, Интернет, компьютер и т. п.

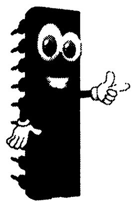
Первые варианты Ардуино

Самые первые варианты Ардуино представляли собой плату, на которую был установлен микроконтроллер (использовались микроконтроллеры AVR) и элементы самой необходимой внешней обвязки:

- цепи сброса;
- внешний кварцевый резонатор;

- стабилизатор напряжения;
- несколько светодиодов для индикации питания и сигналов канала связи.

С компьютером первые Ардуино связывались при помощи последовательного порта. В микроконтроллере для этого использовался встроенный канал UART, а в компьютере — COM порт.



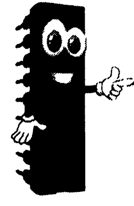
ПРИМЕЧАНИЕ.

*Оба этих стандарта основаны на едином протоколе **RS-232**. Но UART отличается от COM уровнями сигнала.*

Последовательный канал в микроконтроллере работает с сигналом, принимающим значения: **0 В** и **+5 В**.

В **стандарте COM** используются уровни **плюс 12 В** и **минус 12 В**. Поэтому на плате Ардуино присутствовала **схема преобразования уровней**.

На **рис. 2.1** изображен один из первых вариантов Ардуино, использующих COM порт для связи с компьютером.



ПРИМЕЧАНИЕ.

*Как известно, **COM порты** в настоящее время уже полностью вытеснены таким универсальным стандартом, как **USB**. Поэтому и Ардуино пришлось приспосабливаться.*

На платах Ардуино появились **микросхемы с адаптерами USB-COM**. В одном из вариантов маленький модуль с преобразователем USB-COM выполнялся как отдельный модуль, который после загрузки и отладки программы в основную плату снимался и мог быть использован для отладки и прошивки в других разработках.

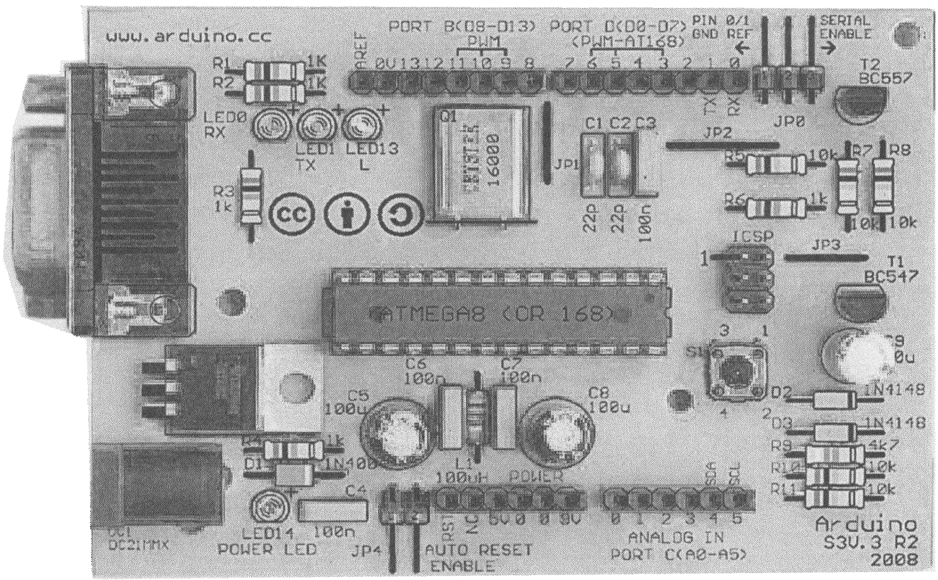


Рис. 2.1. Модуль Ардуино для COM порта

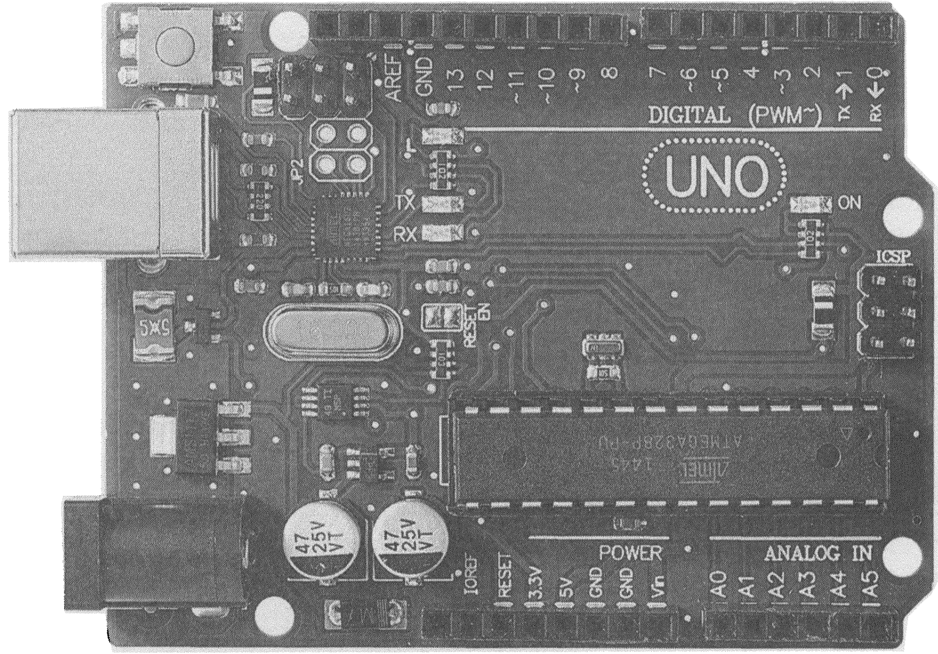


Рис. 2.2. Модуль Arduino UNO

В настоящее время все модели Ардуино снабжены USB портами. На сегодняшний день в мире разработано и производится большое количество самых различных моделей Ардуино. На **рис. 2.2** изображена плата одного из вариантов модуля, который носит название **«Arduino UNO»**. Это довольно удачная модель, которая получила довольно широкое распространение по всему миру, в том числе и в России. Ее мы и будем использовать в качестве примера в нашей книге.

Знакомимся с модулем Arduino UNO

Для того чтобы понять, какие **преимущества** дает начинающему разработчику электронных устройств архитектура модуля Ардуино, рассмотрим модель Arduino UNO подробнее. Существует несколько версий Arduino UNO. Они отличаются по:

- схеме;
- размещению некоторых деталей на плате;
- типу корпуса микросхем.

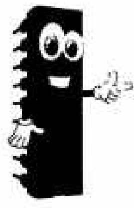
Но все они полностью совместимы между собой и взаимозаменяемы.

На **рис. 2.2** показан внешний вид одного из вариантов Arduino UNO. В основе этой модели модуля лежит **микроконтроллер ATmega328P**. При этом существует две разновидности:

- вариант на микросхеме в DIP корпусе (как раз его мы видим на **рис. 2.2**);
- вариант на микросхемах в SMD корпусе.

Разные модели также отличаются **типом преобразователя USB-COM**:

- в одних вариантах используется специализированная микросхема CH340G;
- в других же (как на **рис. 2.2**) в качестве преобразователя используется микроконтроллер Atmega16U2 с соответствующей программой, зашитой в ее программную память.



ПРИМЕЧАНИЕ.

Но какой бы вариант не использовался, определенные параметры в модуле всегда остаются неизменными. ***Во-первых****, это установочные и габаритные размеры модуля.* ***Во-вторых****, расположение и функциональное назначение всех его выводов.*

Выводы Arduino UNO выполнены в виде разъемов, выстроенных в две цепочки гнезд, расположенные по краям платы. К таким разъемам удобно подключать внешние элементы. При желании, например, светодиод можно просто вставить в нужные гнезда разъема.

На своих строго определенных местах установлены:

- разъем внешнего питания;
- разъем USB.

Чаще всего в модулях Ардуино (в том числе и в Arduino UNO) используются **аппаратные USB розетки**, такие же, какие используются в современных принтерах. Поэтому для подключения модуля Ардуино к компьютеру можно использовать USB кабель от принтера.

Кроме разъемов всегда находятся на стандартных местах следующие элементы:

- кнопка сброса (Reset);
- светодиоды, предназначенные для индикации.

На плате Arduino UNO имеется четыре светодиода. **Светодиод индикации питания** загорается всегда, когда на шине питания модуля появляется напряжение +5 В. Напряжение может поступать как от порта USB, так и от разъема внешнего питания.

Два светодиода подключены параллельно линиям внутреннего COM порта, то есть к линиям, идущим от преобразователя USB-COM к основному микроконтроллеру модуля:

- один из этих светодиодов индицирует процесс передачи информации от компьютера в модуль Ардуино;
- второй светодиод индицирует процесс передачи информации от микроконтроллера в компьютер.

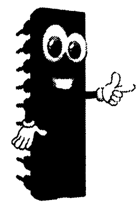
Scan & DjVu Bookingolz

Четвертый светодиод подключен к выводу PB5 порта PB основного микроконтроллера ATmega323 и предназначен для использования программистом по своему усмотрению.

Полезные упрощения в модуле

Как уже говорилось выше, главным **преимуществом** проекта Ардуино является его простота в понимании и использовании. Для того чтобы работа с Ардуино стала доступнее, авторами был придуман целый ряд упрощений.

И одно из них – **сквозная нумерация линий ввода-вывода**. Так, все цифровые контакты объединены в одну общую группу и пронумерованы, как контакт 0, контакт 1 и так далее до 13 (см. **рис. 2.2**).



ПРИМЕЧАНИЕ.

На самом деле к контактам 0...7 подключены разряды D0...D7 порта PD микроконтроллера, а к контактам 8...13 – разряды B0...B5 порта PB.

Однако на языке программирования, который используется для создания программ в модуле Ардуино, обращение к любому цифровому контакту **производится по его номеру**.

Например, для чтения уровня сигнала на цифровом контакте используется команда *digitalRead* (*НомерКонтакта*).

Например: *a = digitalRead(0);*
или *b = digitalRead(13);*

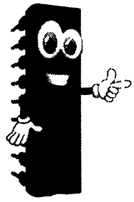
Вывод уровней на любой цифровой контакт тоже очень прост. Для перевода контакта в режим вывода информации используется команда:

pinMode (НомерКонтакта, OUTPUT);

Второй параметр этой функции имеет значение OUTPUT. Служебное слово OUTPUT настраивает контакт на вывод информации. Затем, для вывода на контакт нужного цифрового уровня используется команда:

digitalWrite (НомерКонтакта, Значение);

Параметр «*Значение*» может принимать величину:
- или HIGH (высокий логический уровень);
- или LOW (низкий логический уровень).

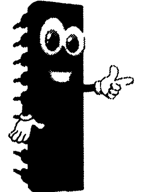


ПРИМЕЧАНИЕ.

Также можно задавать значение и в цифровом виде: 1 или 0.

Группа аналоговых входов

Кроме группы цифровых контактов («пинов») в Ардуино имеется группа так называемых **аналоговых входов**. Это шесть контактов, расположенных на противоположном «борту» модуля. В Ардуино аналоговые входы называются A0, A1, ... A5 (см. **рис. 2.2**).



ПРИМЕЧАНИЕ.

На самом деле аналоговыми эти входы можно назвать условно. В качестве аналоговых входов используются разряды PC0...PC5 порта PC микроконтроллера. То есть, это обычные разряды порта, которые могут служить и цифровыми входами, и цифровыми выходами.

Но альтернативной функцией именно этих шести выводов микроконтроллера ATmega328P является то, что они могут служить **входами внутреннего АЦП. Для упрощения эти входы объявлены аналоговыми**. И это в Ардуино основной режим их работы.

В языке программирования Ардуино для того, чтобы прочитать уровень аналогового сигнала с любого аналогового входа, нужно всего лишь дать одну простую команду:

analogRead();

Например, следующая строка программы прочитает уровень аналогового сигнала с входа A0:

c = analogRead (A0);

Очевидно, что любой аналоговый вход всегда можно использовать как цифровой вход или выход. Это позволяет архитектура микроконтроллера. В языке Ардуино учитывается и эта возможность. На этот случай выводы A0...A5 имеют альтернативную нумерацию. В командах цифрового ввода и цифрового вывода к ним можно обращаться как к контактам 14...19. Например, командой

d = digitalRead(14);

вы можете прочитать цифровой уровень (HIGH или LOW) с «аналогового входа» A0.

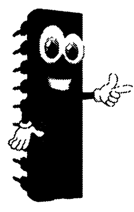
Команда аналогового вывода

Еще одной интересной находкой, упрощающей программирование на языке Ардуино, является **команда аналогового вывода**. Выглядит эта команда следующим образом:

analogWrite (НомерКонтакта, Значение);

Несмотря на то, что команда выводит некий уровень, называемый **аналоговым**, выводимый этой командой сигнал не

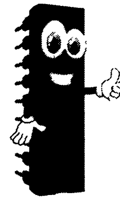
совсем аналоговый. И выводить «аналоговый сигнал» можно как раз только на цифровые выводы, и то не на все.



ПРИМЕЧАНИЕ.

*Дело в том, что «аналоговый» сигнал выводится при помощи **широтно-импульсной модуляции (ШИМ)**.*

На плате Ардуино контакты, при помощи которых можно производить «аналоговый» вывод информации, обозначены значком «~». Это тоже связано с архитектурой микроконтроллера ATmega328P.



ПРИМЕЧАНИЕ.

*Подобный «аналоговый» вывод возможен только на те разряды портов ввода/вывода, которые поддерживают ШИМ. Поэтому параметр **НомерКонтакта** в модуле Arduino UNO может принимать только следующие значения 3, 5, 6, 9, 10, 11.*

В более ранних версиях аналоговый выход был возможен только на контактах 9, 10, 11. Параметр «*Значение*» может лежать в диапазоне 0...255.

Как уже говорилось, команда *analogWrite()* вызывает на выходе «*НомерКонтакта*» появление прямоугольных импульсов. Частота импульсов приблизительно равна 490 герц. Длительность зависит от параметра «*Значение*».

Чем больше значение, тем больше длительность импульса.

Если «*Значение*» = 0, импульсы прекращаются (импульсы нулевой длины), и на выходе устанавливается логический ноль. Если «*Значение*» = 255, то ширина импульсов становится равной их периоду. Поэтому импульсов также не будет, и на выходе установится постоянное значение логической единицы.

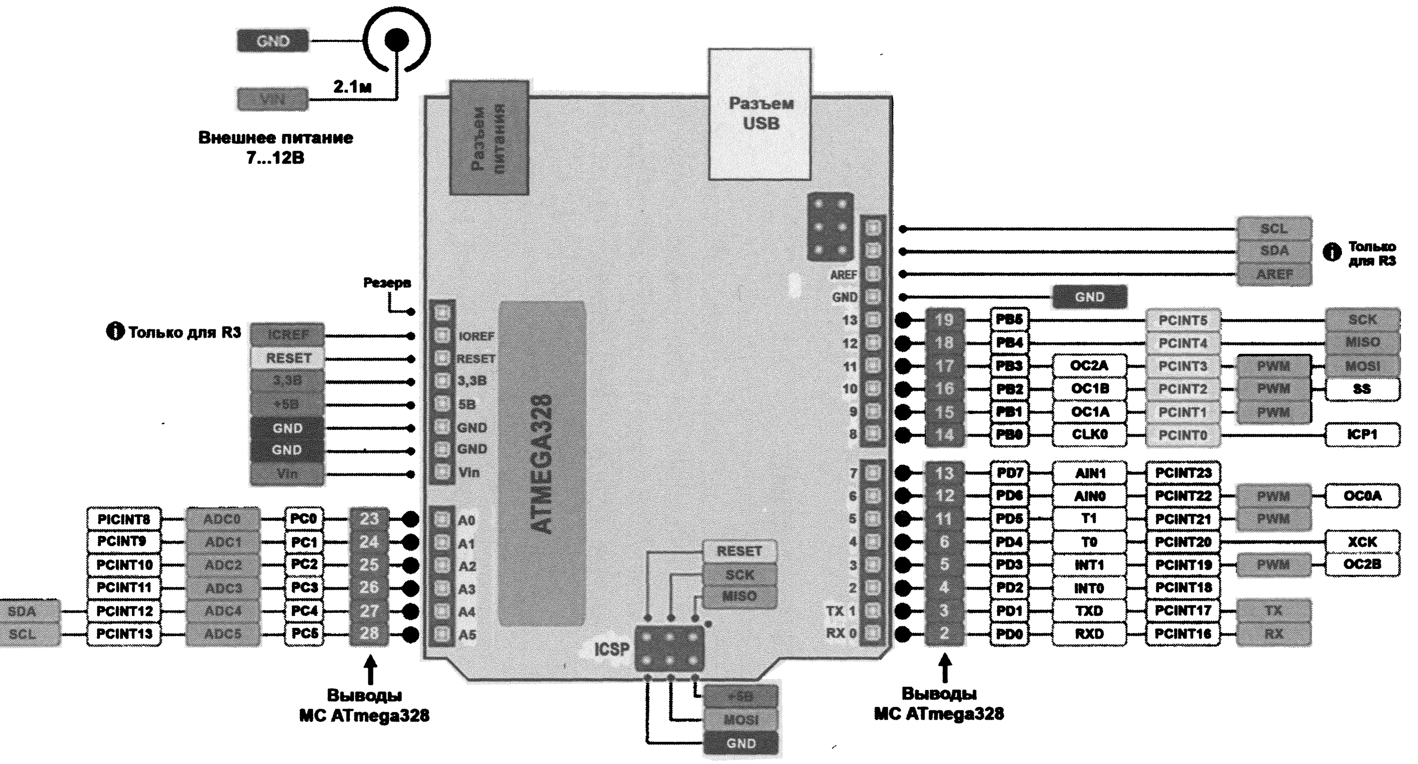


Рис. 2.3. Схема распиновки модуля Ардуино

Контакты питания «Power»

Кроме вышеописанных контактов, на плате Ардуино есть группа контактов под общим названием **«Power»** (**питание**). Сюда выведены: шина питания +5 В; опорное напряжение 3,3 В; контакт общего провода; сигнал сброса.

Так как большинство контактов модуля Ардуино напрямую подключены каждый к своему выводу микроконтроллера ATmega328P, для каждого из контактов модуля дублируются все альтернативные функции, характерные для соответствующего вывода микроконтроллера.

Поддерживаемые языки программирования

Разработчики предусмотрели возможность при необходимости использовать все возможности микроконтроллера. Для этого, кроме специализированного языка программирования Ардуино, модуль поддерживает и **классический язык С++**. Для тех, кто изучил язык СИ, описанный в книге **[1]**, не составит труда перейти на **язык С++ Ардуино**. Эти языки очень похожи.

На **рис. 2.3** изображена схема распиновки модуля Arduino UNO, где для каждого контакта указаны все поддерживаемые им функции. Это поможет вам разобраться с дополнительными функциями каждого контакта.

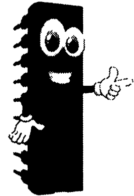
ГЛАВА 3

СРЕДА РАЗРАБОТКИ IDE

Для чего нужно специальное приложение «Среда разработки Arduino IDE»?

Специальное приложение для компьютера, которое называется «Среда разработки Arduino IDE», нужно для таких целей:

- составлять программы для Ардуино;
- сохранять программы на компьютере;
- транслировать и загружать программы в программную память модуля.

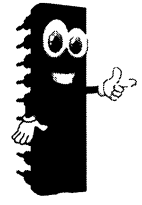


ПРИМЕЧАНИЕ.

*Это приложение полностью бесплатно, и его можно свободно скачать с сайта разработчика (**https://www.arduino.cc**). Существуют версии для всех основных операционных систем (Windows, Mac X, Linux).*

Среда разработки включает в себя **специализированный текстовый редактор**, при помощи которого вы можете писать

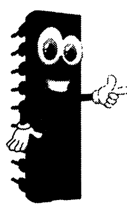
программы для Ардуино. Для написания программ используется **специальная версия языка программирования**, которую мы будем называть далее **«Язык Ардуино».**



ЧТО ЕСТЬ ЧТО.

Программа, написанная на этом языке в среде IDE, называется ***Скетч*** *и записывается на жесткий диск компьютера в виде файла с расширением* ***ino****.*

Программа может состоять из нескольких файлов *ino*, составляющих проект. Один из файлов проекта считается **главным**.

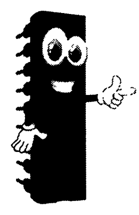


ВНИМАНИЕ!

Все файлы проекта (даже если в проекте один файл) должны размещаться в отдельной директории, имя которой должно совпадать с именем главного файла проекта.

Команды и функции языка Ардуино

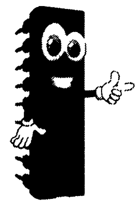
Мы уже познакомились с несколькими командами и функциями языка Ардуино. Простота обучения и использования Ардуино в значительной степени обусловлена эффективностью его языка программирования.



ПРИМЕЧАНИЕ.

Список основных команд и функций языка Ардуино приведен в ***Приложении 1****.*

Язык Ардуино является **клоном языка СИ**, и его синтаксис максимально приближен к синтаксису этого популярного языка программирования.



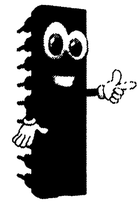
СОВЕТ.

В том случае, когда при программировании вам не хватает набора команд языка Ардуино, вы можете использовать команды языка С++.

При трансляции разработанной вами программы, созданной в среде разработки Ардуино, создается **машинный код,** который, кроме команд заложенных программистом, содержит **ряд системных процедур**.

Одна из таких процедур — **внутренние системные часы**. При запуске на Ардуино любой вашей программы параллельно запускаются эти часы. Они работают все время, пока работает ваша программа, и постоянно отсчитывают количество миллисекунд, прошедших с момента запуска программы.

Для этого используется **системный таймер 0** микроконтроллера и прерывание по переполнению этого таймера.



ПРИМЕЧАНИЕ.

*Подробнее мы опишем этот процесс в **главе 10**.*

Этот таймер используется, например, для работы функций *millis()* или *micros()*, возвращающих значение счетчика времени в миллисекундах и в микросекундах, соответственно.

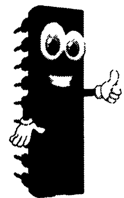
Кроме системного таймера, программа на Ардуино организует еще несколько **вспомогательных процессов**.

Например, **таймер 1** микроконтроллера используется для команд аналогового вывода на контакты 3, 5, 6, а **таймер 2** для аналогового вывода на контакты 9, 10, 11 и для формиро-

вания звуковых сигналов. Для этого таймеры переводятся в режим ШИМ.

Некоторые ресурсы микроконтроллера система программирования самостоятельно использует для своих системных нужд. Поэтому программист, создающий программы на языке Ардуино, уже не может распоряжаться ресурсами микроконтроллера так же свободно, как он может это делать в программах на классическом языке СИ. Зато такое решение облегчает программирование для начинающих программистов.

Язык Ардуино позволяет не утруждать себя подробным изучением всех сложных режимов работы таймеров, прерываний, выбором нужных режимов, определением того, какие коды и в какие системные регистры записать для того, чтобы правильно выбрать нужный режим. Язык Ардуино все это делает сам.



ПРИМЕЧАНИЕ.

*Главный **минус** подобного решения состоит в том, что это сужает гибкость готовой программы.*

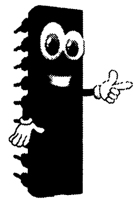
Допуская вольное использование ресурсов нужно всегда помнить, что тем самым вы мешаете языку Ардуино выполнять те либо иные команды.

Например, используя самостоятельно для своих нужд таймер 1, вы уже не сможете осуществлять аналоговый вывод для контактов 3, 5, 6.

Внутренние библиотеки

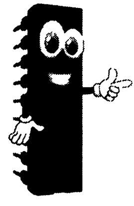
Набор стандартных команд Ардуино можно существенно расширить применением **внешних библиотек**. Пакет среды

разработки Ардуино имеет довольно большой набор внешних библиотек для самых различных задач.



ПРИМЕР.

Существует Библиотека для работы с внутренней долговременной памятью данных, или библиотека для работы с последовательным каналом CPI. Есть библиотека для работы с Ethernet, Wi-Fi, GSM, внешним SD модулем памяти и многое другое.



СОВЕТ.

Если нужная библиотека отсутствует в стандартном пакете среды разработки, вы можете использовать библиотеку сторонних разработчиков. Огромное количество таких библиотек можно свободно скачать из сети Интернет.

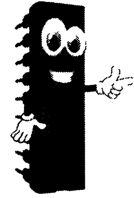
Например, существует библиотека, позволяющая использовать прерывания по таймеру. Набор внешних библиотек в проекте Ардуино просто огромен. Совместными усилиями разработчиков и энтузиастов проекта разработаны библиотеки на все случаи жизни.

После присоединения к вашей программе нужной вам библиотеки в вашем распоряжении появляется набор новых функций, которые дают вам новые нужные вам возможности. Вы можете самостоятельно разработать свою собственную библиотеку и применять ее в своих программах. Далее, по мере изучения разных примеров в нашей книге, мы будем часто использовать внешние библиотеки.

Скачиваем программный пакет с сайта разработчика

Для установки среды программирования Ардуино на ваш компьютер нужно сначала скачать программный пакет с сайта разработчика (**arduino.cc**). Точный адрес ссылки на страницу загрузки вы можете посмотреть в списке ссылок в конце книги. Сайт англоязычный. Его удобно просматривать при помощи браузера с переводчиком. Например, в Яндекс-браузере.

Находясь на сайте, выберите в главном меню пункт *Software* (Программное обеспечение). На открывшейся по этой ссылке странице вы увидите раздел «*Download the Arduino IDE*» (Скачать Arduino IDE).



ПРИМЕЧАНИЕ.

*В момент написания книги (январь 2018 года) номер последней стабильной версии среды разработки – **1.8.5**.*

Справа от описания последней версии в синем прямоугольнике вы увидите набор ссылок на разные варианты пакета для разных операционных систем.

Варианты установочных пакетов для Windows

Для Windows, например, на сайте имеется **три варианта установочных пакетов.**

Вариант №1. ***Windows Installer*** **— пакет с инсталлятором.** Представляет собой исполняемую программу (с расширением **.exe**). При запуске инсталлятора он сам автоматически устанавливает все компоненты на ваш компьютер. В том числе и драйвер USB интерфейса для платы Ардуино.

Вариант №2. ***Windows ZIP file for non admin install*** **— пакет в виде ZIP архива.** В архиве находится папка с именем

«arduino-x.x.x»,

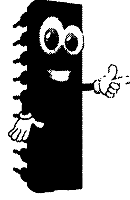
где x.x.x — это номер версии программного пакета.

Папка содержит все файлы, составляющие пакет среды разработки IDE. Для установки среды вам нужно извлечь папку со всем ее содержимым из ZIP-архива и поместить в любом удобном для вас месте на жестком диске вашего компьютера.

После этого вам вручную придется установить драйвер USB интерфейса. Для этого вы просто подключаете модуль Ардуино к USB порту компьютера. Компьютер обнаружит USB устройство и попытается установить драйвер самостоятельно. Ему это не удастся, и он выведет на дисплей стандартное окно, сообщающее, что драйвер установить не удалось.

В этом окне нужно:

- выбрать команду «Установить драйвер с известного устройства»;
- выбрать и указать файл драйвера и нажать кнопку «Установить».



ПРИМЕЧАНИЕ.

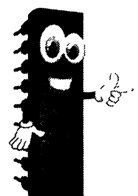
Драйвер вы найдете в папке «arduino-x.x.x» в подпапке «drivers». Там вы увидите несколько вариантов драйвера. Выберите тот, который подходит для вашей системы.

Программа установки «увидит» только тот вариант, который подходит для вашего случая.

Вариант №3. ***Windows app Get*** — версия для Windows 10. Эта версия также выполнена в виде ZIP-архива и устанавливается аналогично предыдущей.

Запуск программы

После установки драйверов запустите программу. Если вы использовали для установки ZIP-архив, то на рабочем столе у вас будет отсутствовать иконка для запуска программы. В этом случае вы должны знать, что запускать нужно файл *arduino.exe* в папке *arduino-x.x.x*.



СОВЕТ.

Для удобства дальнейшей работы вам рекомендуется самостоятельно ***создать иконку*** *для запуска этой программы на рабочем столе.*

Для этого проще всего сначала щелкнуть правой кнопкой мышки по файлу программы (*arduino.exe*) и выбрать пункт меню «Копировать». А затем щелкнуть правой кнопкой мыши в любом месте рабочего стола и выбрать пункт «Вставить ярлык».

Основное окно среды разработки

После запуска программы откроется **основное окно среды разработки** (см. **рис. 3.1**). Основное окно имеет два встроенных окна:

- строка меню;
- панель инструментов.

В центре основного окна программы находится **окно редактора** для написания текста программ. Текст программы пишется на белом фоне.

Редактор имеет подсветку синтаксиса. Поэтому разные элементы текста автоматически раскрашиваются разными

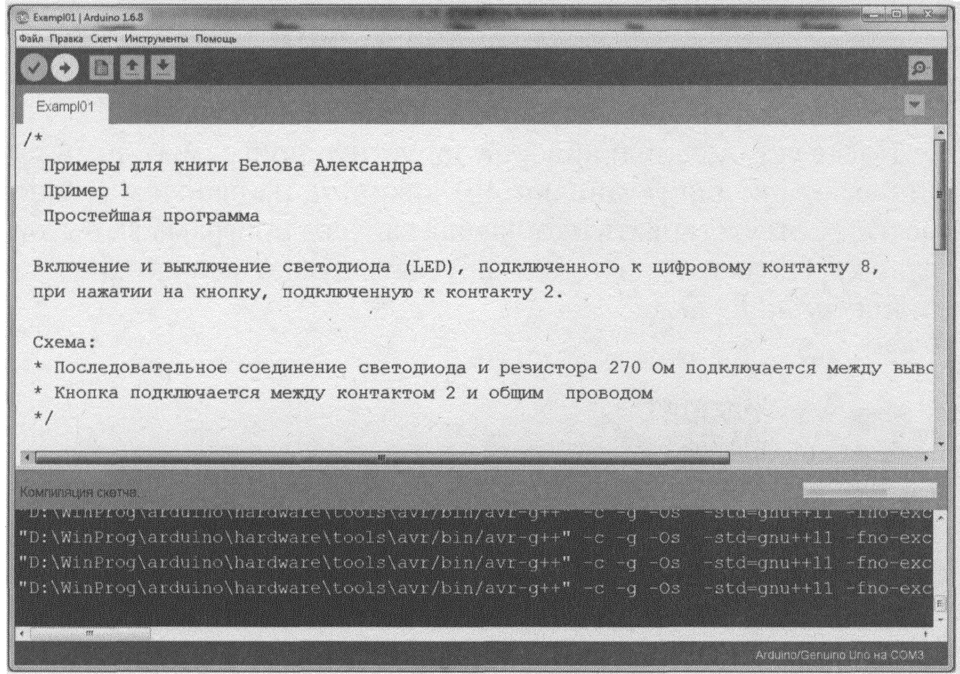


Рис. 3.1. *Основное окно среды разработки IDE*

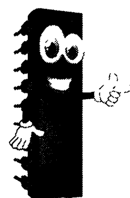
цветами. Например, выделяются элементы текста такими цветами:

- управляющие слова языка программирования — красным;
- зарезервированные константы — голубым;
- тексты примечаний — серым;
- основной текст программы — темно-синим.

Как уже говорилось, файл с текстом программы называется «**скетч**». Сразу под окном редактора программ находится другое узкое **окошко, предназначенное для вывода системных сообщений**.

Сюда выводятся сообщения при трансляции программы и записи ее в модуль Ардуино. Сюда же выводятся все сообщения об ошибках. Сообщения в этом окне выводятся белым цветом на черном фоне. Сообщения об ошибках выделяются красным.

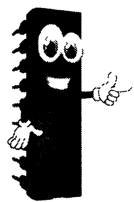
В верхней части окна системных сообщений можно видеть **поле заголовка**.



ПРИМЕЧАНИЕ.

*В обычном состоянии **поле заголовка** голубого цвета.*

В поле заголовка в процессе работы выводится сообщение о текущем режиме работы IDE (проверка, трансляция, загрузка и т. д.).



ВНИМАНИЕ.

*Если процесс трансляции программы закончится **ошибкой**, то поле заголовка становится желтым. В нем появляется сообщение о том, что произошла ошибка, а также появится кнопка, при помощи которой вы можете распечатать все содержимое из окна системных сообщений на принтере.*

Содержимое окна будут содержать как сообщения, обычно выводимые при трансляции, так и все сообщения об ошибках. Полный текст сообщений значительно больше, чем то, что вы видите в окне сообщений. Весь текст вы можете просмотреть также, пролистав это окно при помощи полосы прокрутки.

В самой верхней части основного окна среды разработки расположено **главное меню программы**. При помощи этого меню можно производить все операции управления средой IDE.

Панель инструментов

Сразу под строкой основного меню расположена **панель инструментов**. На этой панели расположено несколько иконок, которые дублируют самые важные функции основного меню. При наведении курсора мыши на любую из этих ико-

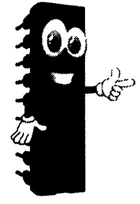
нок, справа сразу за последней иконкой появляется подсказка с пояснением функции, на которую указывает мышь. В **табл. 3.1** приведено назначение всех элементов панели управления.

Таблица 3.1. *Назначение элементов панели инструментов*

Иконка	Название	Команда меню	Описание
✔	Проверить	Скетч/Проверить	Проверка синтаксиса программы
➔	Загрузить	Скетч/Загрузить	Загрузка программы в память модуля Ардуино
	Новый	Файл/Новый	Создание нового файла программы (новый скетч)
	Открыть	Файл/Открыть	Открыть файл программы (скетч)
	Сохранить	Файл/Сохранить	Записать файл программы (скетч) на диск
	Монитор порта	Инструменты/ Монитор порта	Открыть окно монитора порта

Выбор номера COM порта в настройках программы

Прежде чем начать работу в среде разработки, нужно правильно выбрать номер COM порта в настройках программы.

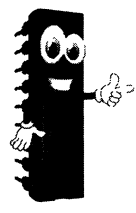


ПРИМЕЧАНИЕ.

Драйвер USB модуля Ардуино работает в режиме эмуляции COM порта. При установке драйвера он создает в списке устройств компьютера виртуальный COM порт. Номер создаваемого порта выбирается автоматически в процессе установки.

В зависимости от конфигурации системы номер может быть разный. **По умолчанию** среда разработки настроена на **порт COM1**.

Для того, чтобы узнать правильный номер порта, нужно открыть диспетчер устройств Windows (Компьютер/Свойства/Диспетчер устройств). Откроется окно диспетчера (см. **рис. 3.2**). В списке устройств откройте ветку «Порты COM и LPT». Среди COM портов должен появиться порт Ардуино (на **рис. 3.2** мы обвели эту запись серой линией).



ВНИМАНИЕ.

Учтите, что порт в этом списке виден ***только при подключенном к компьютеру*** *модуле Ардуино. При отключении Ардуино порт в окне диспетчера исчезает.*

Запомните номер COM порта. Затем зайдите в настройки среды разработки: меню «Инструменты/Порт». При выборе этого пункта меню вы увидите список всех имеющихся в вашем компьютере последовательных портов. Выберите **нужный порт из списка**.

Выбор типа используемой платы Ардуино

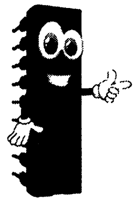
Кроме номера порта, вы должны выбрать **тип вашей платы Ардуино**. Для этого выберите пункт меню «Инструменты/Плата». Появится список всех поддерживаемых плат. Выберите из списка вашу плату. Например «Arduino Uno».

После того как вы выберите порт и плату, в самой нижней части окна среды разработки (в строке состояния основного окна) вы увидите надпись:

«Arduino/Genuino Uno на COM1».

Теперь все готово к работе.

Рис. 3.2. Порт Ардуино в окне диспетчера устройств

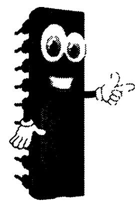


ЧТО ЕСТЬ ЧТО.

Genuino *– это проект, аналогичный Ардуино. Он разработан другим производителем, но эти два проекта совместимы на уровне среды разработки IDE.*

Скетч: открытие, сохранение, загрузка

При каждом последующем запуске среды разработки в окно редактора программ загружается последний редактируемый при предыдущем запуске программы **скетч**. Если среда разработки запускается впервые, в окне редактора программ создается новый пустой не именованный скетч.



ПРИМЕЧАНИЕ.

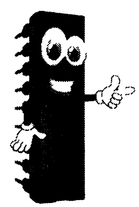
После написания или редактирования текста программы скетч можно записать на жесткий диск при помощи команды меню

«Файл/Сохранить» или «Файл/Сохранить как».

Прочитать уже записанный файл можно при помощи команды «Файл/Открыть».

Когда программа написана, ее можно оттранслировать и загрузить в память модуля Ардуино. Для этого нужно просто выбрать пункт меню «Скетч/Загрузка» или щелкнуть по соответствующей иконке на панели инструментов.

При выборе команды «Загрузка» скетч сначала записывается на жесткий диск, затем начнется процесс проверки и трансляции программы. По окончании транслирования среда IDE сразу же начинает передавать оттранслированную программу в модуль Ардуино посредством USB интерфейса.



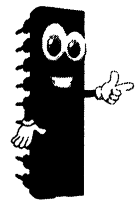
ПРИМЕЧАНИЕ.

При этом модуль Ардуино должен быть подключен к компьютеру и готов к работе.

Процесс трансляции идет достаточно продолжительное время. Если все пройдет нормально, то по окончании трансляции и загрузки в заголовке окна системных сообщений вы увидите надпись:

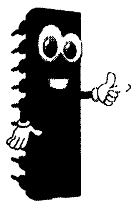
«Загрузка окончена».

А в самом окне увидите сообщение о результатах трансляции.



ВНИМАНИЕ!

При заливке программ из компьютера в модуль Ардуино используются цепи, связанные с цифровыми контактами 0 и 1. Поэтому в момент загрузки на эти контакты нельзя подавать никакие сигналы, нельзя замыкать их или как-то по-другому мешать прохождению сигналов.



СОВЕТ.

Лучше всего, по возможности, вообще не использовать цифровые контакты 0 и 1.

Сразу после загрузки программы в модуль Ардуино она автоматически начинает выполняться. При этом выполнение программы тут же прекращается, как только вы начнете новый процесс заливки программы из компьютера в модуль. Главное не забывать, что цифровые контакты 0 и 1 модуля не должны быть заблокированы.

Организация обмена информацией между программой на Ардуино и компьютером

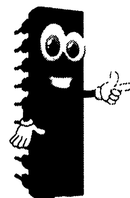
Удобной особенностью Ардуино является простой способ организации обмена информацией между программой на Ардуино и компьютером **с использованием USB интерфейса**.

Ряд несложных команд, описанных в **Приложении 1** (раздел «Работа с последовательным портом»), позволяют вашей программе передавать результаты своей работы в компьютер и принимать данные из компьютера. При этом среда разработки IDE имеет специальные средства для того, чтобы принимать и отображать информацию, поступающую по каналу

USB от модуля Ардуино, а также, наоборот, передавать данные в Ардуино. Эти средства называются «Монитор порта» и «Плоттер по последовательному соединению». Рассмотрим работу этих двух функций среды IDE по порядку.

Монитор порта. Отображает поступающие через USB порт данные в текстовом виде. Запускается монитор из меню «Инструменты/Монитор порта» или при помощи соответствующей иконки на панели инструментов. После запуска монитора открывается специальное окно.

В окне два поля. **Верхнее поле** — узкое в одну строку. В него вы можете ввести с клавиатуры данные для передачи в модуль Ардуино. В конце этого поля имеется кнопка «Отправить». При нажатии на эту кнопку все набранные данные отправляются по каналу USB.



ПРИМЕЧАНИЕ.

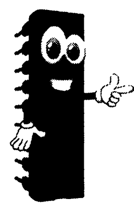
Разработанная вами программа, работающая в модуле Ардуино, должна принять эти данные и обработать так, как вы задумали.

Второе большое поле в окне монитора порта сразу после запуска монитора работает на **прием данных**. Все поступившие от модуля Ардуино данные сразу же отображаются на экране. Монитор порта предполагает, что передаваемые и принимаемые данные — это символьные строки.

Плоттер по последовательному соединению. Плоттер вызывается аналогично монитору при помощи команды меню:

«Инструменты/
Плоттер по последовательному соединению».

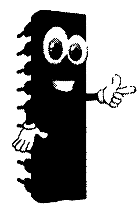
Окно плоттера похоже на окно монитора, но в нем имеется только одно большое поле. И это поле отображает поступающие из Ардуино по USB порту данные в графическом виде.



ПРИМЕЧАНИЕ.

Монитор порта и Плоттер по последовательному соединению предназначены лишь для отладки алгоритмов передачи и приема данных.

Для практического применения вам придется разработать **специализированные программы для компьютера**, которые будут работать в паре с программами, загруженными в модуль Ардуино. Это позволит, например, создавать на Ардуино систему сбора информации с разных датчиков (один из вариантов — пункт метеонаблюдения), и передавать эту информацию в компьютер. А компьютер сможет обрабатывать и систематизировать полученные данные.



ПРИМЕЧАНИЕ.

Правда, в среде разработок IDE вы сможете создать только ту часть программного пакета, которая будет работать в самом модуле Ардуино.

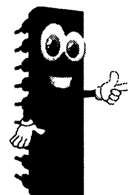
Приложение для компьютера вам придется создавать при помощи других систем программирования. Рассмотрение этого вопроса не входит в задачу данной книги.

ГЛАВА 4

ПРОСТЕЙШАЯ ПРОГРАММА «HELLO, WORLD!»

Постановка задачи

Итак, в предыдущих главах мы узнали общие принципы построения модулей Ардуино и среды разработки IDE.



ВНИМАНИЕ!

Дальнейшая наша цель — научиться программировать, изучить основные тонкости языка Ардуино, узнать приемы составления программ.

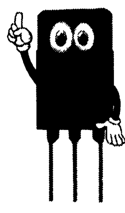
Лучший способ научиться программировать — изучать материал на конкретных примерах. Именно такой способ обучения языкам Ассемблера и СИ используется в книге [1]. Поэтому и в данной книге изучение материала мы будем производить на конкретных примерах.

За основу набора примеров, предложенных в этой книге, взяты примеры, используемые в [1]. Примеры специально придуманы так, чтобы изучать программирование **от простого к сложному**. Каждый новый пример раскрывает, какой-либо

новый аспект или прием программирования. Этот набор простейших алгоритмов хорошо зарекомендовал себя на практике.

И начнем мы с самого простого, элементарного примера. Задача, поставленная в этом примере, не имеет практического смысла. Но как первый шаг в мир программирования подходит идеально.

Первый урок по программированию для компьютера обычно начинается с программы, выводящей на экран надпись **«Hello, world!»**. Считайте, что наша первая задача — это «Hello, world!» для микроконтроллеров.



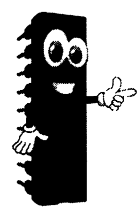
ЗАДАЧА.

Создать программу для модуля Ардуино, которая должна управлять свечением светодиода в зависимости от нажатия кнопки. Светодиод должен быть подключен к одному из цифровых контактов модуля, а кнопка — к другому цифровому контакту. При нажатии кнопки светодиод должен загораться, а при отпускании — гаснуть.

Схема

Один из возможных вариантов схемы изображен на ***рис. 4.1.***

Для подключения кнопки мы используем цифровой контакт 2, а для подключения светодиода — контакт 8.



ПРИМЕЧАНИЕ.

Выбор именно этих двух контактов не критичен. Для подобной задачи мы могли бы выбрать любые два других цифровых контакта. Не желательно лишь использовать контакты 0 и 1 по причине, о которой говорилось в предыдущей главе книги.

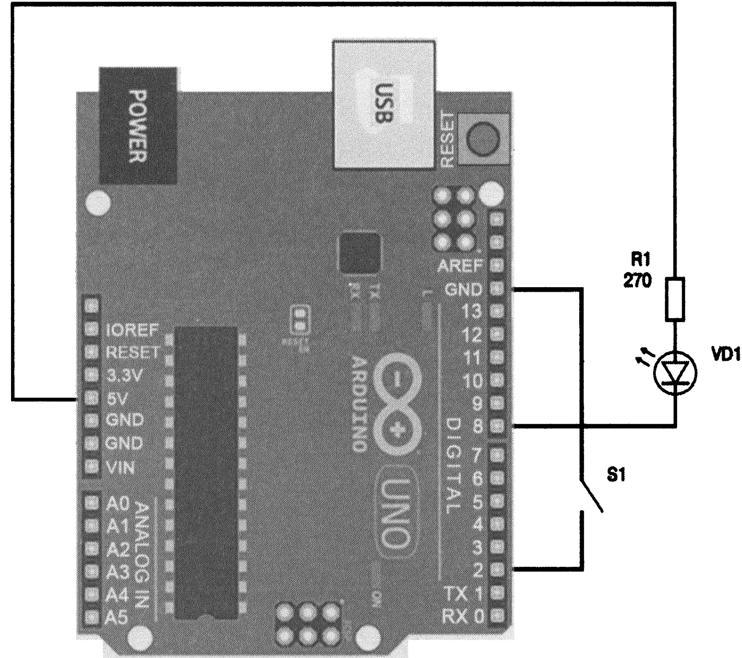


Рис. 4.1. Электрическая схема с кнопкой и светодиодом

Выбранная нами схема максимально приближена к схеме аналогичного примера из книги [1]. Для подключения кнопки выбрана схема с замыканием на общий провод и использованием внутреннего подтягивающего резистора. Такое включение кнопки требует минимума деталей и позволяет использовать специальные возможности входных цепей микроконтроллера.

Рис. 4.2 иллюстрирует принцип работы кнопки, подключенной по выбранной нами схеме. На **рис. 4.2** изображена часть схемы внутреннего устройства одного из разрядов порта ввода/вывода микроконтроллера AVR. Именно к такой линии подключен каждый цифровой (да и аналоговый) контакт модуля Ардуино.

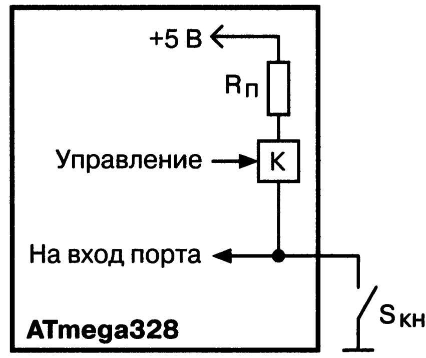
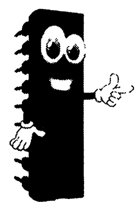


Рис. 4.2. Схема работы кнопки и подтягивающего резистора

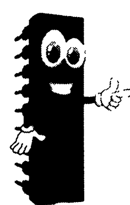
Схема, приведенная на **рис. 4.2**, представляет собой лишь часть общей схемы линии порта — ту часть, которая отвечает за ввод информации в порт. Схема содержит резистор Rп (подтягивающий резистор). При помощи программно управляемого ключа «К» резистор может подключаться между входом линии порта и шиной питания +5 В.



ВНИМАНИЕ!

В нашем случае ключ «К» должен быть постоянно включен.

В результате, если кнопка Sкн не нажата, то благодаря резистору Rн, на входную линию порта поступает сигнал логической единицы. Если контакт Sкн замкнуть, то вход порта будет соединен с общим проводом, а это эквивалентно подаче логического нуля. В результате этого микроконтроллер может легко считывать состояние кнопки. Если прочитанный сигнал равен единице — кнопка отпущена. Если нулю — кнопка нажата.



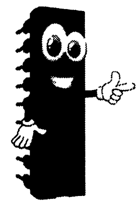
ВНИМАНИЕ!

При написании программы нужно обязательно предусмотреть команду, которая включит ключ «К» и подключит к схеме подтягивающий резистор. Если резистор не подключен, то при разомкнутых контактах кнопки Sкн на входе будут наводиться электромагнитные помехи, мешающие работе программы.

Отключают подтягивающий резистор в двух случаях:
- когда линия порта работает на вывод информации;
- когда на вход подается сигнал с выхода других микросхем с четкими уровнями нуля и единицы.

Свой подтягивающий резистор имеется на каждой линии каждого порта микроконтроллера, и все они включаются и выключаются независимо друг от друга.

Для подключения светодиода выбран **классический способ** (см. **рис. 4.1**). Светодиод подключается через токоограничивающий резистор между цифровым контактом модуля и шиной питания. Для того чтобы зажечь светодиод, подключенный по такой схеме, нужно установить на выходе (в данном случае на контакте 8) уровень логического нуля. В этом случае перепад потенциалов между нулем на выходе и напряжением питания составит 5 вольт. Светодиод загорится.



ПРИМЕЧАНИЕ.

Токоограничивающий резистор необходим для того, чтобы ток через светодиод не превысил максимально допустимого значения.

Описанную выше схему удобнее всего собрать с использованием универсального **макетного контактного модуля**, так как это показано на **рис. 4.3**.

Алгоритм

Программа должна непрерывно многократно считывать состояние кнопки, подключенной к контакту 2 модуля, и, в зависимости от считанного уровня, управлять сигналом на контакте 8 (выход светодиода).

Если кнопка нажата (на входе 2 логический ноль), то программа должна включить светодиод (подать логический ноль на выход 8).

Если кнопка отпущена (на входе единица) — выключить светодиод (подать на выход единицу). Фактически, программа должна многократно в цикле считывать сигнал с входа 2 и передавать его на выход 8.

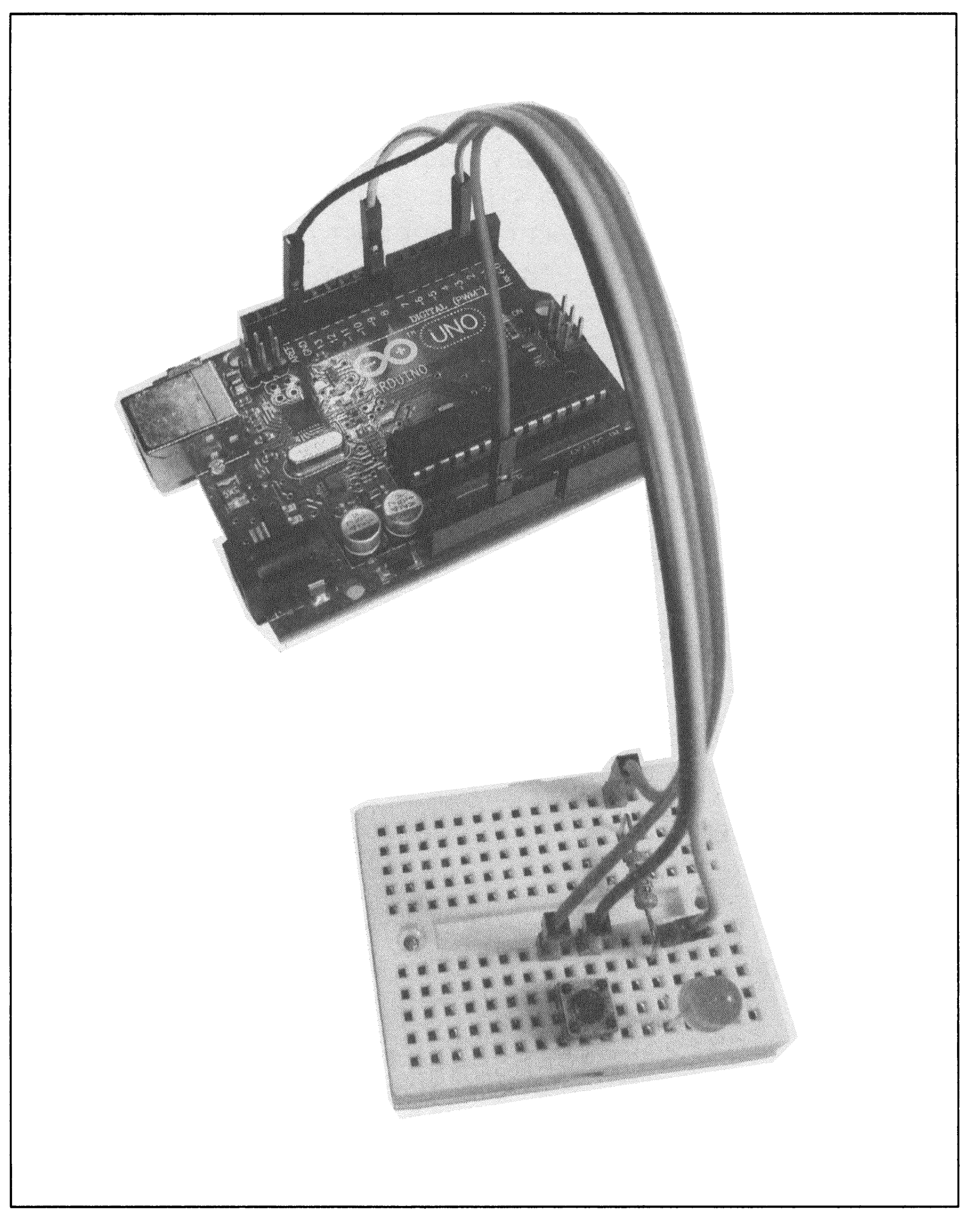
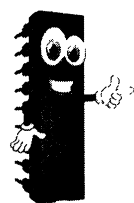


***Рис. 4.3.** Пример монтажа простейшей схемы на макетной плате*

Первый вариант программы

Для решения поставленной выше задачи предлагается два варианта программы. **Первый вариант** — это решение задачи «В лоб». Программа, приведенная в **листинге 4.1**, реализует сформулированный нами алгоритм буквально. Разберем подробнее синтаксис программы.



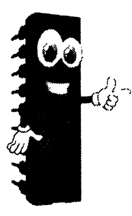
ПРИМЕЧАНИЕ.

Для удобства восприятия материала читателем мы пронумеровали строки программы. Так удобнее ссылаться на ту или иную строку в тексте описания. Хотя ***синтаксис языка Ардуино не предполагает нумерации строк****.*

Каждая команда программы отделяется от следующей символом **«точка с запятой»**. А на строки программа разбивается из соображений удобочитаемости.

Так что **программа** — это то, что в листинге находится справа от столбца с номерами строк. Именно этот текст вы должны набрать при создании скетча в окне редактора программ среды разработки.

Кроме команд и операторов, составляющих собственно программу, текст программы содержит еще и **комментарии**.



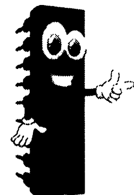
ПРИМЕЧАНИЕ.

Любой язык программирования обязательно имеет средства, которые позволяют вставлять в текст программы комментарии.

В языке Ардуино используется Си-подобный синтаксис. Поэтому **комментарии** в программу вставляются двумя способами.

Первый способ. Как комментарий воспринимается любая часть текста, ограниченная символами /* */. То есть все, что находится между парой символов /* и второй парой символов */ воспринимается программой как комментарий.

Второй способ позволяет добавлять комментарий в конце строки. Все, что находится после **двойного слеша** // до конца строки, считается комментарием. При трансляции программы комментарии игнорируются. Они предназначены для человека, который читает программу, для лучшего ее понимания.



ПРИМЕЧАНИЕ.

*В **листинге 4.1**, как и в других листингах этой книги, пронумерованы только те строки, которые содержат команды и операторы языка. Строки, содержащие только комментарии или строки, в которых располагается продолжение команд, занимающих несколько строк, не нумерованы.*

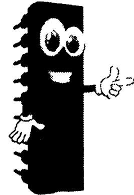
Любая программа, написанная на языке Ардуино, должна содержать две обязательные функции:

setup() и *loop()*.

Эти две функции обязательно должны присутствовать в каждой программе, даже если они не содержат никаких команд. Это структурные функции, определяющие два основных этапа работы любой программы для микроконтроллера.

Функция *setup()* выполняется один раз, сразу после запуска программы или после нажатия кнопки сброса. В эту функцию нужно включить команды начальной настройки режимов всех узлов и систем микроконтроллера, и определение начальных значений переменных.

Функция *loop()* — это **основной цикл программы**. Этот цикл начинает выполняться сразу после окончания функции начальных установок и выполняется многократно все время, пока работает микроконтроллер.



ПРИМЕЧАНИЕ.

Любая программа для микроконтроллера всегда работает по такой схеме. Сначала однократно выполняется модуль начальной инициализации, а затем начинается основной цикл программы, в котором обычно и реализуется весь основной алгоритм.

В языке Ардуино такая типовая организация программы заложена в структуру языка путем применения двух обязательных функций *setup()* и *loop()*. Кроме двух обязательных функций, программист может создавать любое количество собственных функций по своему усмотрению.

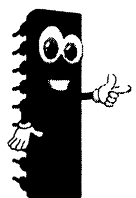
Любая функция в языке Ардуино объявляется следующим образом:

```
тип ИмяФункции(тип Параметр1, тип Прараметр2 ... )
{
Команда1;
Команда2;
.....
}
```

Имя функции выбирается произвольно. Имя должно подчиняться обычным **требованиям для синтаксиса имен**:

- только латинские буквы и цифры;
- начинается с буквы;
- не допускаются пробелы.

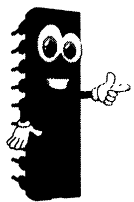
Перед именем функции необходимо указать **тип возвращаемого значения**. Тип может принимать значения *int*, *char* или другой допустимый в языке Ардуино.



ПРИМЕЧАНИЕ.

Полный список допустимых типов значений приведен в ***Приложении 2*** *в конце книги.*

В круглых скобках указывается список параметров, передаваемых в функцию при ее вызове. Для каждого параметра вы также должны указать его **тип**. Сразу за определением функции в фигурных скобках указываются **команды, составляющие тело функции**.

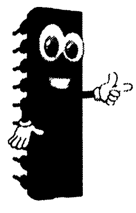


ПРИМЕЧАНИЕ.

Если ваша функция в результате своей работы должна возвращать некое значение, то тело функции должно содержать хотя бы один ***оператор return****, в параметре которого указывается это самое возвращаемое значение.*

Оператор *return*, в какой бы строке внутри функции он не находился, немедленно завершает выполнение функции. В языке Ардуино нет правила «сначала описание функции, затем ее использование». Вы можете помещать описание функции в любое место программы (но не внутрь другой функции).

Особый случай, когда функция не возвращает никакого значения.



ПРИМЕЧАНИЕ.

По-другому такие функции иногда называют ***процедурами****.*

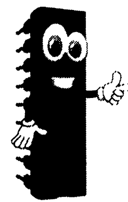
Если функция не возвращает никакого значения, то перед функцией вместо типа возвращаемого значения нужно поместить управляющее слово *void*. Функции *setup()* и *loop()* относятся к функциям никогда не возвращающим значения. Более подробно о том, как используются функции, мы рассмотрим далее, по мере разбора конкретных примеров.

Теперь, когда мы немного разобрались с некоторыми элементами синтаксиса, давайте разберем по порядку программу, приведенную в **листинге 4.1**.

В **строках 1 и 2** происходит определение констант. Использование констант добавляет программе наглядности, а также упрощает в будущем изменение значений этих самых констант. Например, в нашей программе определяются значения двух констант.

В **строке 1** определяется константа *buttonPin* (номер контакта для подключения кнопки).

В **строке 2** определяется константа *ledPin* (номер контакта для подключения светодиода). Этим константам присваиваются значения 2 и 8, соответственно. Далее в программе эти константы используются вместо номеров этих контактов.

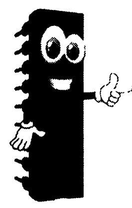


ПРИМЕЧАНИЕ.

*Вместо номера контакта для подключения кнопки мы будем писать **buttonPin**, а вместо номера контакта для подключения светодиода – **ledPin**. Согласитесь, что это нагляднее.*

А если вы захотите **поменять номер контакта**? Например, решите подключить светодиод к контакту номер 9. Вам достаточно будет просто поменять 8 на 9 в строке 2 программы. После этого везде, где в программе используется *ledPin*, программа будет обращаться к контакту номер 9.

Если программа большая и сложная, то в ней каждая константа может использоваться десятки раз.

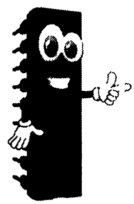


ПРИМЕЧАНИЕ.

Использование именованных констант гарантирует от ошибок при изменении параметров программы и упрощает изменение значений определяемых константами параметров.

Описание константы начинается с ключевого слова *const*. Далее указывается тип значения определяемой константы. В данном случае тип значения — *int* (intedger). Это означает — целое число.

В **строке 3** определяется **переменная**. Имя переменной *buttonState*.

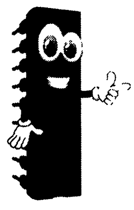


ЧТО ЕСТЬ ЧТО.

Переменная *– это некая величина, которая может изменяться в процессе работы программы. Каждой переменной (так же, как и константе) присваивается свое уникальное имя. Далее в программе обращение к переменной происходит по ее имени.*

Под переменную в оперативной памяти микроконтроллера выделяется одна или несколько ячеек. Программа может:

- присваивать переменной различные значения;
- читать содержимое переменной;
- использовать переменную в различных математических выражениях.



ПРИМЕЧАНИЕ.

*В нашем случае переменная **buttonState** используется для временного хранения значения состояние кнопки.*

В **строке 4** программы начинается описание функции *setup()*.

В теле функции всего две команды (**строки 5** и **6**).

В **строке 5** определяется режим работы контакта *ledPin* (вывода для подключения светодиода). Режим работы контакта определяется как OUTPUT (вывод информации). Определение режима любого цифрового контакта в языке Ардуино производится при помощи функции *pinMode()*. Эту функцию мы уже описывали в предыдущей главе книги.

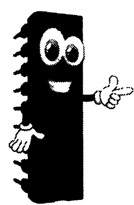
В **строке 6** при помощи той же функции определяется режим работы контакта *buttonPin* (контакт для подключения кнопки). Этот режим определяется как INPUT_PULLUP (ввод информации с включенным подтягивающим резистором).

В **строке 7** программы объявляется функция *loop()* (основной цикл).

Функция *loop()* также содержит всего две команды (**строки 8** и **9**).

В **строке 8** происходит чтение состояния кнопки (считывается значение сигнала на входе *buttonPin*). Для этого используется функция *digitalRead()*. Считанное значение присваивается переменной *buttonState*.

В **строке 9** это значение выводится на выход *ledPin* при помощи функции *digitalWrite()*. Функции *digitalRead()* и *digitalWrite()* используемые для чтения и вывода информации так же были описаны в предыдущей главе.



ПРИМЕЧАНИЕ.

Полный список функций вы всегда можете найти в ***Приложении 1****.*

В результате работы функции *loop()* программа все время считывает состояние кнопки и тут же выводит его на светодиод. Если кнопка не нажата, то считанное состояние будет равно 1. Единица на выходе светодиода приведет к тому, что он не будет гореть.

Листинг 4.1. Пример простой программы (вариант 1)

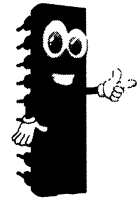
```
  /*
   Простейшая программа:
   Светодиод и кнопка
  */

  // Постоянные константы. В данной программе
  // константы определяют номера используемых контактов:
1 const int buttonPin = 2;   // номер контакта кнопки
2 const int ledPin = 8;        // номер контакта светодиода

  // Изменяемые переменные:
3 int buttonState = 0;   // статус кнопки (нажата/не нажата)

4 void setup() {
    // Инициализируем контакт светодиода (как выход):
5   pinMode(ledPin, OUTPUT);
    // Инициализируем контакт кнопки (как вход):
6   pinMode(buttonPin, INPUT_PULLUP);
  }

7 void loop() {
    // Читаем статус кнопки и помещаем
    // его значение в переменную:
8   buttonState = digitalRead(buttonPin);
    // Помещаем считанное значение на выход светодиода:
9   digitalWrite(ledPin, buttonState);
  }
```

Если при очередном считывании кнопка окажется нажатой, то ее состояние станет равно нулю. Ноль тут же передастся на выход светодиода и светодиод загорится.

Второй вариант программы

Недостаток программы, приведенной в **листинге 4.1**, состоит в том, что микроконтроллер полностью занят всего одной задачей — сканированием кнопки и передачей значения ее состояния на светодиод. В такую программу трудно встроить какие-нибудь другие задачи.

Но существует простой способ, как реализовать этот же алгоритм так, чтобы он выполнялся в фоновом режиме. То есть, так, чтобы основной цикл программы был не занят.



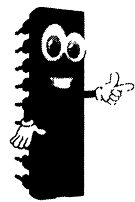
ПРИМЕЧАНИЕ.

Это дает возможность использовать основной цикл для решения любых других задач. И поможет нам создать такую программу механизм прерываний.

Механизм прерываний существует в любом современном микроконтроллере. Очень подробно все вопросы организации прерываний описаны в [1]. В нашем случае в данном конкретном примере мы будем использовать лишь один из видов прерываний, поддерживаемых микроконтроллером ATmega328. Это, так называемое, **внешнее прерывание**.

Прерывание работает при поступлении внешнего сигнала на специальный вход микроконтроллера. При поступлении сигнала на вход прерывания выполнение текущей программы приостанавливается, и контроллер переходит к выполнению закрепленной за этим прерыванием процедуры, которая называется **процедурой обработки прерывания**.

Микроконтроллер Atmega328, а, значит, и модуль Ардуино, имеет **два входа внешних прерываний**. Они совмещены с линиями PD2 и PD3 порта PD микроконтроллера и, соответственно, с цифровыми контактами 2 и 3 модуля Ардуино. В нашей схеме кнопка как раз подключена к входу 2.



ПРИМЕЧАНИЕ.

Поэтому мы без изменения схемы можем использовать прерывание по этому входу для обработки события: «нажатие кнопки».

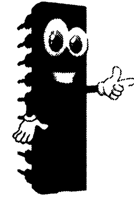
Теперь основной цикл программы не будет заниматься сканированием кнопки и переключением светодиода. Эту задачу мы возложим на процедуру обработки прерывания. А на основной цикл программы можно возложить любые другие придуманные вами задачи.

Новая программа будет работать так. В **момент нажатия кнопки**:

- возникает прерывание;
- выполнение основной программы приостанавливается;
- запускается процедура обработки прерывания.

Эта процедура однократно читает состояние кнопки с того же самого входа 2 и посылает считанное значение на выход светодиода. Затем процедура обработки прерывания завершается, и выполнение основной программы продолжается.

При отпускании кнопки снова будет вызвано то же самое прерывание. И снова процедура обработки прерывания передаст состояние кнопки на светодиод. Обработка прерывания происходит мгновенно и практически не отражается на скорости выполнения основной программы. Поэтому все операции с кнопкой и светодиодом будут выполняться как бы параллельным процессом.



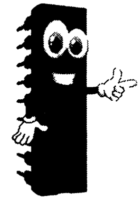
ПРИМЕР.

*Подобным образом выглядит **процесс управления указателем мыши** на экране компьютера. Какую бы сложную программу не выполнял компьютер, курсор мышки легко бегает по экрану, подчиняясь движениям самой мыши по столу. А ведь прорисовка курсора на экране – это тоже специальная небольшая программа, которая также работает по прерыванию от сигнала мыши.*

Программа, реализующая задачу управления светодиодом при помощи кнопки с использованием внешнего прерывания, приведена в **листинге 4.2**. Благодаря тому, что язык Ардуино имеет очень простые и удобные средства работы с этим видом прерываний, программа настолько же проста, как и программа без прерываний. Рассмотрим листинг программы подробнее.

Определения констант (**строки 1 и 2**) происходит так же, как и в предыдущем случае.

В **строке 3** также мы видим команду определения переменной *buttonState*. Однако теперь перед строкой определения добавлено управляющее слово: *volatile*.



ПРИМЕЧАНИЕ.

Это указание для компилятора, которое означает, что изменение значения данной переменной будет производиться в теле процедуры обработки прерывания, а, значит, не под контролем основной программы. То есть, в какой-то момент, когда выполнение основной программы будет приостановлено, значение переменной может измениться.

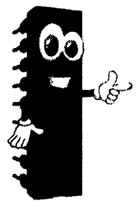
Программа должна учитывать такую возможность. Ведь для основной программы это будет выглядеть так, как будто значение неожиданно прямо в процессе вычислений вдруг изменилось.

Возможна ситуация, когда в основной программе переменная используется несколько раз в сложном математическом выражении. Такое выражение вычисляется не мгновенно, а **поэтапно**. И может получиться так, что первая часть выражения будет вычисляться с одним значением переменной, а вторая часть — уже с другим.

Если переменная имеет тип *volatile*, то при трансляции в машинные коды программа строится так, что не переходит к обработке прерывания до тех пор, пока очередное выражение

не будет вычислено до конца. В нашем простом случае такая предосторожность не обязательна, но мы для порядка все же выполним это правило.

Далее в теле функции *setup()* в **строках 5, 6** мы видим уже знакомые нам команды, устанавливающие режимы работы внешних контактов *ledPin* и *buttonPin*.

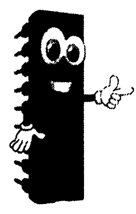


ПРИМЕЧАНИЕ.

Но кроме этого в новом варианте программы в тело функции setup() добавлены еще две, отсутствующие ранее команды.

В **строке 7** на выход светодиода подается высокий логический уровень, который принудительно гасит светодиод. Такая команда понадобилась здесь потому, что в новой программе оценка состояния кнопки и передача значения на выход светодиода происходит только в момент нажатия либо отпускания кнопки.

При запуске программы кнопка не нажата, а на любом выходе (в том числе и на *ledPin*) по умолчанию устанавливается логический ноль.



ВНИМАНИЕ.

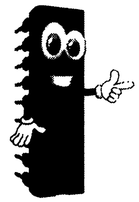
Если не предусмотреть принудительное гашение светодиода, то при включении питания светодиод загорится, и будет гореть, пока не нажмешь кнопку.

В первом варианте программы, когда опрос кнопки и вывод ее состояния на светодиод начинались сразу после включения питания, и продолжались непрерывно все время работы, светодиод принимал значение состояния кнопки моментально и гас, не успевая загореться.

Листинг 4.2. Простая программа вариант 2 (с использованием прерываний)

```
   /*
     Простейшая программа Светодиод и кнопка
     с использованием прерывания
   */

   // Определение констант:
1  const int buttonPin = 2;  // номер вывода с кнопкой
2  const int ledPin =  8;    // номер вывода со светодиодом

   // Определение переменных:
3  volatile int buttonState = 0;  // состояние кнопки

4  void setup() {
     // Установить режим работы контакта светодиода:
5    pinMode(ledPin, OUTPUT);
     // Установить режим работы контакта кнопки:
6    pinMode(buttonPin, INPUT_PULLUP);
7    digitalWrite(ledPin, HIGH);
     // Прикрепить прерывание к вектору ISR
8    attachInterrupt(digitalPinToInterrupt(buttonPin),
                                        pinISR, CHANGE);
   }

   // Основной цикл программы
9  void loop()
   {
10   // Здесь ничего нет!
   }

   // Обработчик прерывания
11 void pinISR()
   {
12   buttonState = digitalRead(buttonPin);
13   digitalWrite(ledPin, buttonState);
   }
```

Ну и последняя команда, добавленная в новой версии программы в тело функции *setup()*, — это **команда определения вектора прерывания**.



ПРИМЕЧАНИЕ.

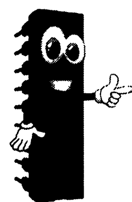
*Выражение «**определить вектор прерывания**» означает — закрепить за конкретным прерыванием процедуру его обработки.*

Сама процедура расположена ниже, в конце программы (**строки 11, 12, 13**). Оператор, определяющий вектор прерывания, в чистом виде выглядит так:

attachInterrupt(interrupt, function, mode)

Оператор имеет три параметра.

Параметр *interrupt* — номер прерывания. Наш Arduino UNO, как уже говорилось выше, имеет всего два внешних прерывания. Прерывания в Ардуино различаются по номеру. Нумеруются они по порядку. Разные модели Ардуино отличаются как количеством внешних прерываний, так и номерами контактов, от которых они работают. Но в любом Ардуино есть прерывание 0 и прерывание 1. Некоторые модели имеют большее число прерываний.



ПРИМЕР.

Arduino Mega2560 имеет шесть внешних прерываний. Они нумеруются цифрами от 0 до 5. Модуль Arduino UNO имеет следующие два внешних прерывания: прерывание 0, работающее от входа 2 и прерывание 1, работающее от входа 3.

Параметр *function* — имя функции обработки прерывания.

Параметр *mode* — режим работы прерывания. Параметр *mode* может принимать следующие **значения**:

LOW — вызывает прерывание, когда на входе прерывания установится низкий логический уровень (LOW);

CHANGE — прерывание вызывается при смене значения на входе, с LOW на HIGH или наоборот;

RISING — прерывание вызывается только при смене значения на входе с низкого логического уровня (LOW) на высокий (HIGH);

FALLING — прерывание вызывается только при смене значения на входе с высокого уровня (HIGH) на низкий (LOW).

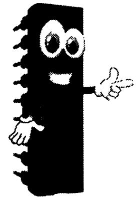
Если мы посмотрим на **строку 8** программы (**листинг 4.2**), то увидим, что вместо номера прерывания в функцию *attachInterrupt()* подставлена другая функция:

digitalPinToInterrupt(*buttonPin*),

в качестве параметра которой выступает номер контакта для подключения кнопки.

Как легко догадаться эта специальная функция возвращает номер прерывания по номеру цифрового входа. Конечно, мы могли бы просто поставить в этом месте номер прерывания (в данном случае — это 0). Но использование специальной функции делает программу более гибкой.

Теперь, если вы захотите подключить кнопку не на контакт 2, а на контакт 3, вам нужно будет просто изменить номер контакта в строке 1 программы.



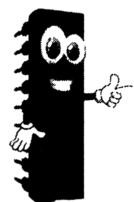
СОВЕТ.

В данной простейшей программе это не очень актуально, но в больших сложных программах, где прерывание вызывается много раз, такой прием очень облегчит изменение конфигурации системы.

В **строке 9** программы начинается **основной цикл программы.** В теле цикла нет ни одного оператора.

Вместо комментария в **строке 10** вы можете поместить в основной цикл любую сложную программу, которая будет выполнять нужные вам задачи совершенно независимо от задачи управления светодиодом при помощи кнопки.

Описание функции обработки прерывания начинается в **строке 11**.



ПРИМЕЧАНИЕ.

*Прежде чем приступить к разбору этой функции, опишем **общие принципы построения** подобных функций.*

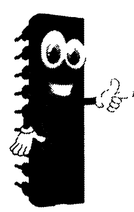
Любая функция, предназначенная для обработки прерывания, имеет следующие особенности:

1. функция не должна иметь никаких входных параметров и не должна возвращать никаких значений;
2. внутри функции обработки прерывания не работает оператор *delay()*, а значения возвращаемые *millis()* не изменяются;
3. возможна потеря данных, передаваемых по последовательному соединению в момент выполнения функции обработки прерывания;
4. как уже говорилось выше, переменные, изменяемые в этой функции, должным быть объявлены как *volatile*.

Такой тип функции имеет специальное название ***Interrupt Service Routine*** **(обработчик прерывания)**. Сокращенно **ISR**. Именно поэтому мы назвали нашу функцию **pinISR**.

В теле функции pinISR (**строки 12, 13**) мы видим те же две команды, которые в первом варианте программы были размещены в основном цикле:

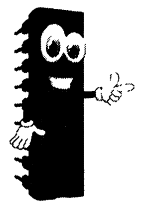
- первая команда (в **строке 12**) читает состояние кнопки;
- вторая команда (в **строке 13**) записывает его на выход светодиода.



ПРИМЕЧАНИЕ.

Оба приведенных выше варианта программы можно ***упростить****.*

Синтаксис языка СИ, который используется в Ардуино, позволяет отказаться от переменной *buttonState*.



ПРИМЕР.

Два оператора в строках 8 и 9 ***листинга 4.1****, так же как два аналогичных оператора в строках 12 и 13* ***листинга 4.2*** *можно заменить на одну строку, в которую поместить следующее выражение:*

digitalWrite(ledPin, digitalRead(buttonPin));

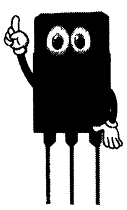
Результат работы этого выражения точно такой же, как и результат работы двух строк, использующих промежуточную переменную. Так как переменная в этом случае больше не нужна, в обоих вариантах программы можно удалить строку с ее определением. В обоих вариантах программы это ***строка 3****.*

ГЛАВА 5

ПЕРЕКЛЮЧАЕМЫЙ СВЕТОДИОД

Постановка задачи

В этом примере мы немного изменим программу, не изменяя схемы.



ЗАДАЧА.

Создать программу, переключающую состояние светодиода (горит / не горит) при каждом нажатии кнопки. Если светодиод не горит, то при кратковременном нажатии и отпускании кнопки он должен загореться. При очередном кратковременном нажатии и отпускании кнопки светодиод должен снова погаснуть.

Схема

Схема устройства такая же, как в предыдущем примере (см. ***рис. 4.1****).*

Алгоритм

Алгоритм явственно вытекает из поставленной задачи. Есть только одна **тонкость**. Наша программа должна определить факт кратковременного нажатия кнопки. То есть момент, когда кнопка будет нажата, а затем отпущена. Поэтому алгоритм будет звучать так.

1. Если кнопка не нажата, ждать нажатия кнопки.
2. Когда кнопка будет нажата, проверить состояние светодиода. Если светодиод горит, программа должна его потушить. Если светодиод потушен — нужно его зажечь.
3. Снова проверить состояние кнопки. Если кнопка еще нажата, ждать отпускания.
4. Когда кнопка будет отпущена, перейти к пункту 1 алгоритма.

Первый вариант программы

Простейшая программа, реализующая вышеописанный алгоритм, приведена в **листинге 5.1**. Так как схема у нас не поменялась, светодиод и кнопка подключены к тем же контактам, начало программы у нас полностью соответствует программам из **главы 4**.

В **строках 1** и **2** определяются константы.

В **строках 3, 4, 5** расположена функция *setup()*. Она, как и раньше, определяет режимы работы используемых в программе цифровых контактов *ledPin* и *buttonPin*.

Оставшуюся часть (**строки 6...12**) занимает основной цикл программы *loop()*. Именно в основном цикле реализуется нужный нам алгоритм. Разберем команды основного цикла подробнее.

В **строке 7** реализован локальный цикл ожидания нажатия кнопки. Для организации этого цикла используется оператор *while()*.

While — это один из стандартных операторов цикла языка СИ. Формат этого оператора следующий:

```
while (условие)
{
Тело цикла;
}
```

Английское слово while означает «пока». Операторы, входящие в тело цикла, выполняются многократно, *пока* условие в круглых скобках — истина. В нашей программе (см. **строку 7**) условием является

digitalRead (buttonPin)==HIGH.

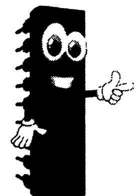
Двойное «равно» — это операция сравнения двух величин. Результат операции «истина», если сравниваемые величины равны. Очевидно, что цикл будет выполняться до тех пор, пока значение, считанное с входа *buttonPin*, будет равно HIGH. **Зарезервированное слово HIGH** означает высокий логический уровень (или просто 1). В теле цикла ожидания нет никаких команд. Цикл должен просто ждать, пока сигнал с кнопки станет равным LOW (то есть нулю).

Как только кнопка будет нажата, программа выйдет из цикла ожидания и перейдет к **процедуре переключения светодиода (строки 8, 9, 10, 11)**. Для переключения светодиода используется оператор *if*.

Формат оператора следующий:

```
if (условие) Команда1;
else Команда2;
```

Слово *if* с английского переводится как «если», а слово *else* — как «иначе».

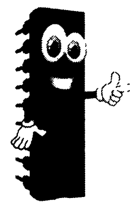


ПРИМЕЧАНИЕ.

Если условие в круглых скобках – истина, то выполняется Команда1, иначе – Команда2.

Вместо каждой команды можно поставить группу команд, заключив их в фигурные скобки.

Оператор *if* в **строке 8** программы используется для оценки состояния светодиода (горит / не горит). Для этого используется следующий прием: при помощи оператора *digitalRead()* мы считываем информацию с контакта *ledPin*.



ПРИМЕЧАНИЕ.

Напоминаю, что контакт ledPin у нас запрограммирован на вывод информации! Но это не отменяет возможность считывать информацию с этого контакта. Так устроены все информационные контакты микроконтроллера.

Оператор *if* проверяет значение сигнала, считанного с выхода на светодиод. Если считанное значение равно HIGH, то выполняется команда в **строке 9** программы, которая устанавливает на том же выходе светодиода уровень LOW.

В противном случае, выполняется **строка 11**, и на выходе светодиода устанавливается HIGH.

В **строке 12** расположен **цикл ожидания отпускания кнопки**. Это такой же пустой цикл, как и цикл ожидания нажатия в **строке 7** программы. Изменилось только условие. Цикл выполняется до тех пор, пока значение состояния кнопки, считанное с входа *buttonPin,* равно LOW. Когда кнопка будет отпущена, цикл ожидания отпускания заканчивается. Но продолжается основной цикл *loop()*. Управление передается на его начало, то есть к **строке 7**, и все повторяется заново.

Листинг 5.1. Первый вариант программы переключаемого светодиода

```
   /*
     Программа «Переключаемый светодиод»
   */

   // Постоянные константы.
1  const int buttonPin = 2;  // номер контакта кнопки
2  const int ledPin = 8; // номер контакта светодиода

3  void setup() {
     // инициализируем контакт светодиода на вывод:
4    pinMode(ledPin, OUTPUT);
     // инициализируем контакт кнопки на ввод:
5    pinMode(buttonPin, INPUT_PULLUP);
   }

6  void loop() {
     // Ожидание нажатия кнопки:
7    while (digitalRead(buttonPin)==HIGH) {};
     // Переключение светодиода:
8    if (digitalRead(ledPin)==HIGH)
9       digitalWrite(ledPin, LOW);
10   else
11     digitalWrite(ledPin, HIGH);
     // ожидание отпускания кнопки:
12   while (digitalRead(buttonPin)==LOW) {};
   }
```

Второй вариант программы

Этот вариант программы приведен для того, чтобы продемонстрировать, как можно написать программу, не используя операцию чтения уровня сигнала с выхода на светодиод.

Новый вариант программы приведен в **листинге 5.2**. Новая программа мало отличается от предыдущей. Рассмотрим **различия**. Так как в новом варианте программы мы отказываемся от считывания непосредственно с выхода *ledPin*, мы вводим переменную для хранения текущего состояния светодиода. Имя переменной *ledState*. Переменная будет хранить текущее значение состояния светодиода.

При каждом нажатии кнопки это значение будет изменяться на противоположное (с ноля на единицу, а с единицы на ноль), а затем выводить уже новое значение на контакт *ledPin*.

В **строке 3** программы не только создается эта переменная, но ей сразу присваивается начальное значение.

В теле функции *setup()* также появилась новая команда. Она уже на стадии начальных установок выводит значение переменной *ledState* на контакт светодиода *ledPin* (**строка 7** программы). Эта команда нужна для того, чтобы начальное состояние светодиода (выключен), к тому времени уже записанное в переменную *ledState*, установилось на выходе светодиода.

В цикле *loop()* новой программы также есть **существенные изменения**. Циклы ожидания нажатия и отпускания кнопки (**строки 9 и 12**) остались без изменений. А вот процедура изменения состояния светодиода изменилась полностью.

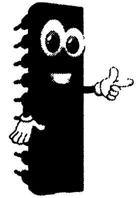
Сразу после обнаружения факта нажатия кнопки выполняется инверсия значения переменной *ledState*. Значение переменной инвертируется и присваивается ей же.

В языке СИ **оператор «!»** означает **инверсию**. Эта операция как раз и изменяет ноль на единицу, а единицу на ноль.

В **строке 11** это новое, уже инвертированное значение выводится на контакт светодиода.

Листинг 5.2. Второй вариант программы переключаемого светодиода

```
   /*
     Переключаемый светодиод (с использованием
     переменной для хранения состояния кнопки)
   */
   // Определение констант:
1  const int buttonPin = 2;  // номер контакта кнопки
2  const int ledPin = 8;  // номер контакта светодиода
   // Определение переменной:
3  int ledState = HIGH;    // состояние светодиода

4  void setup() {
     // Установка режимов контактов
5    pinMode(ledPin, OUTPUT);  // контакт светодиода
6    pinMode(buttonPin, INPUT_PULLUP); // контакт кнопки
7    digitalWrite(ledPin, ledState); // нач. сост. светодиода
   }

8  void loop() {
     // Ожидание нажатия кнопки
9    while (digitalRead(buttonPin)==HIGH) {};
     // Определение состояния светодиода
10   ledState = !ledState;  // Инвертирование
11   digitalWrite(ledPin, ledState);   // Вывод
     // Ожидание отпускания кнопки
12   while (digitalRead(buttonPin)==LOW) {};
   }
```



ПРИМЕЧАНИЕ.

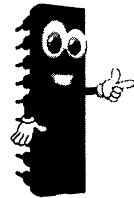
В остальном программа работает так же, как и в предыдущем варианте.

Третий вариант программы

В третьем варианте программы мы покажем, как можно задачу переключаемого светодиода решить **с использованием механизма внешнего прерывания**. Такая программа приведена в **листинге 5.3**.

Начало программы (**строки с 1 по 7**) полностью соответствует предыдущему варианту программы (**листинг 5.2**).

В **строке 8** мы видим уже знакомое нам выражение, определяющее вектор прерывания. Такое же прерывание мы использовали в **главе 4**.

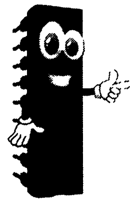


ПРИМЕЧАНИЕ.

В связи с использованием прерывания основной цикл, и в этой программе оставлен пустым.

Как мы уже знаем, туда вы можете поместить команды, которые будут выполнять любые другие нужные вам задачи.

В **строках 11, 12, 13** находится процедура обработки прерывания.



ПРИМЕЧАНИЕ.

*В теле процедуры имеются всего две команды. Это те же самые команды, которые в программе из **листинга 5.2**. были расположены в основном цикле.*

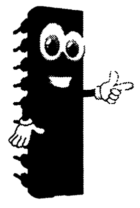
Алгоритм работы программы (листинг 5.3) очень простой. После выполнения команд начальных установок в теле функции *setup()*, программа переходит к основному циклу *loop()*.

Листинг 5.3. Программа переключаемого светодиода с использованием прерывания

```
   /*
     Переключаемый светодиод
     (с использованием механизма внешнего прерывания)
   */
   // Константы
1  const int buttonPin = 2;   // номер контакта кнопки
2  const int ledPin = 8;      // номер контакта светодиода
   // Переменная
3  volatile int ledState = HIGH;  // состояние светодиода

4  void setup() {
     // Установка режимов контактов
5    pinMode(ledPin, OUTPUT);   // контакт светодиода
6    pinMode(buttonPin, INPUT_PULLUP);   // конт. кнопки
7    digitalWrite(ledPin, ledState); // нач. состояние св-диода
     // Прикрепление прерывания к вектору ISR:
8    attachInterrupt(digitalPinToInterrupt(buttonPin),
                                          pinISR, FALLING);
   }

9  void loop()
   {
10   // Основной цикл остается свободным!
     // Тут можно размещать основную программу, на
     // фоне которой будет выполняться переключение
     // светодиода в фоновом режиме
   }

   // Обработчик прерывания
11 void pinISR()
   {
12   ledState = !ledState;
13   digitalWrite(ledPin, LEDstate);
   }
```

При нажатии кнопки активизируется прерывание. Выполнение основного цикла программы прерывается и вызывается процедура обработки прерывания.

Эта процедура инвертирует значение переменной *ledState* (**строка 12**) и выводит новое ее значение на выход *ledPin* (**строка 13**).



ПРИМЕЧАНИЕ.

Циклы ожидания нажатия и отпускания кнопки в данном случае не требуется. Факт нажатия кнопки определяется настройками внешнего прерывания.

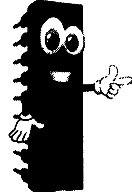
При инициализации вектора прерывания (в **строке 8**) был выбран режим FALLING. В этом режиме прерывание возникает только в момент смены сигнала на входе с высокого логического уровня на низкий. А значит только в момент отпускания кнопки.

ГЛАВА 6

БОРЕМСЯ С ДРЕБЕЗГОМ КОНТАКТОВ

Постановка задачи

Иногда, собрав схему и опробовав программу, приведенную в предыдущей **главе 5**, вы можете заметить, что программа переключения светодиода работает нестабильно. Выражается это в том, что при нажатии кнопки светодиод иногда может не переключиться. Это связано с таким явлением, как **дребезг контактов.**



ПРИМЕЧАНИЕ.

Дребезг *происходит в момент замыкания или размыкания любых механических контактов. Эффект дребезга состоит в том, что в момент замыкания надежное соединение не происходит моментально. Замыкание сопровождается целым рядом очень быстрых замыканий и размыканий, прежде чем установится надежный контакт. Такие же процессы сопровождают и в процесс размыкания контактов.*

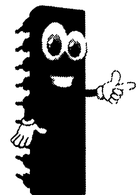
Дребезг контактов является проблемой и мешает работе электроники. Например, в программе переключения светодиода при нажатии кнопки из-за дребезга происходит не одно, а несколько переключений за одно нажатие. Это известная проблема, которая хорошо знакома разработчикам электронной аппаратуры.

В инженерной практике существует множество приемов борьбы с этой проблемой. Подробно о дребезге контактов рассказывается в [1].

Существуют **аппаратные** и **программные** методы борьбы с дребезгом. Аппаратные методы — это различные схемные ухищрения. Их мы сейчас рассматривать не будем. При желании подробнее смотрите в [1]. Мы рассмотрим несколько программных методов борьбы с дребезгом **путем усовершенствования программы.**

Самый простой метод борьбы с дребезгом — **введение программной задержки.** Как уже говорилось, дребезг контактов происходит некоторое время в момент после замыкания контактов либо некоторое время сразу после размыкания.

По этому методу для борьбы с дребезгом после обнаружения самого первого замыкания контакта программа должна выдержать небольшую паузу.



ПРИМЕЧАНИЕ.

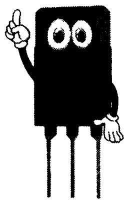
Длительность паузы должна быть такой, чтобы за время паузы закончился весь процесс дребезга. То есть программа должна пропустить все остальные замыкания и размыкания контактов, возникающие из-за дребезга.

Точно такую же задержку нужно сформировать сразу после первого же размыкания контактов. Более продвинутые **программные методы борьбы с дребезгом контактов:**

- метод статистической обработки сигналов, поступающих от кнопки;
- метод программного интегрирования цифрового сигнала;

- так называемый алгоритм ожидания момента стабильного состояния кнопки.

Сформулируем задачу.



ЗАДАЧА.

Доработать программу переключения светодиода, описанную в главе 5, изменив ее так, чтобы она обеспечивала эффект антидребезга.

Схема

*Схема у нас опять остается без изменений (смотри **рис. 4.1**).*

Антидребезг простыми средствами

Простейший способ обеспечить антидребезг — включить в нужные места программы команды, обеспечивающие задержку на время дребезга. Для формирования **задержки** в языке Ардуино имеется специальная команда *delay(Период)*. Команда формирует программную задержку. Длительность задержки определяется параметром «*Период*». Длительность задается в миллисекундах.

Алгоритм

Алгоритм в основном остается таким же, который был описан в **главе 5** для программы переключения светодиода.

Отличие состоит лишь в том, что после цикла ожидания нажатия кнопки, а также после цикла ожидания отпускания, мы введем **команду программной задержки**.

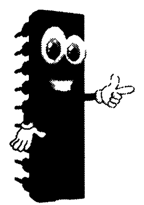
Программа

В **листинге 6.1** приведена доработанная программа, основу которой составляет программа, приведенная в **листинге 5.1**, в которую введены задержки антидребезга. Для этого в блок определений констант (**строка 3 листинга 6.1**) введена новая константа *DelayTime*.

Константа определяет время задержки антидребезга. Команды задержки введены в основной цикл программы:

- одна — в **строке 9** сразу после цикла ожидания нажатия кнопки;
- вторая — в **строке 12** сразу после цикла ожидания отпускания кнопки.

Кроме изменений, связанных с антидребезгом, в программе на **листинге 6.1** представлен еще один способ реализации процедуры переключения состояния светодиода.



ПРИМЕЧАНИЕ.

*Вся процедура реализуется в виде выражения, которое занимает всего одну строку программы (**строка 10**).*

Это сложное выражение, которое при помощи оператора *digitalWrite()* выводит на контакт светодиода *ledPin* инвертированное значение, считанное оператором *digitalRead()* с того же самого контакта *ledPin*. В результате программа обходится и без переменной для хранения состояния светодиода и без оператора *if() ... else*.

Листинг 6.1. Программа переключаемого светодиода с антидребезгом

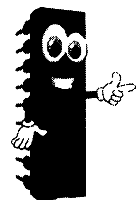
```
   /*
     Программа переключаемого светодиода
     с функцией антидребезга
   */

   // Определение констант.
1  const int buttonPin = 2;   // номер контакта кнопки
2  const int ledPin = 8;      // номер контакта светодиода
3  const int DelayTime = 20;  // Задержка антидребезга

4  void setup() {
     // Инициализация контактов:
5    pinMode(ledPin, OUTPUT);
6    pinMode(buttonPin, INPUT_PULLUP);
   }

7  void loop() {
     // Ожидание нажатия кнопки:
8    while (digitalRead(buttonPin)==HIGH) {};
     // Задержка после нажатия кнопки
9    delay (DelayTime);
     // Переключение светодиода:
10   digitalWrite(ledPin, !digitalRead(ledPin));
     // Ожидание отпускания кнопки:
11   while (digitalRead(buttonPin)==LOW) {};
     // Задержка после отпускания кнопки
12   delay (DelayTime);
   }
```

Применение внешней библиотеки Button

Простейший способ антидребезга, примененный в предыдущей версии программы (**листинг 6.1**), не всегда качественно отрабатывает все проблемы, связанные с вводом данных от внешних контактов. В сложных условиях эксплуатации, при изношенных контактах дребезг контактов может быть достаточно велик.

К тому же, кроме проблем, создаваемых дребезгом контактов, на проводах, идущих от кнопки к плате, могут наводиться **электромагнитные помехи**, особенно если их длина значительная. Помехи также мешают правильной работе. Для надежного избавления от всех видов помех требуются более сложные алгоритмы.

И вот тут очень удобно воспользоваться **внешней библиотекой функций** обслуживания кнопок и контактов.



ПРИМЕЧАНИЕ.

Стандартный набор библиотек пакета программ Ардуино не содержит такой библиотеки.

Но в Интернете вы можете найти множество вариантов подобных библиотек. Все они обычно доступны для свободного скачивания.

Для примера возьмем библиотеку, которая называется **«Button», автор Калинин Эдуард**. Скачать библиотеку можно на сайте «Мир книг по электронике» (**book.mirmk.ru**). Библиотека представляет собой папку с именем «Button», которая содержит три файла:

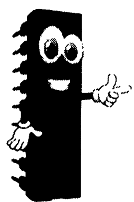
- собственно библиотека (Button.h);
- текст программ библиотеки на языке C++ (Button.cpp);
- список ключей (keywords.txt).

После скачивания эту папку нужно извлечь из архива и поместить в пакет программ Ардуино в папку «libraries».

Если вы поместили пакет программ среды разработки в корневом каталоге диска «C:», то путь к папке библиотеки Button должен стать следующим:

«C:\arduino\libraries\Button\».

Теперь эту библиотеку вы можете использовать в ваших программах.



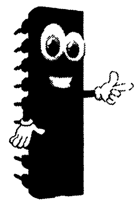
ПРИМЕЧАНИЕ.

Как это сделать в нашем конкретном случае, рассмотрим на примере программы, приведенной в ***листинге 6.2****.*

Для начала нужно присоединить библиотеку к нашей программе. Среда разработки Ардуино имеет встроенную функцию, облегчающую поиск нужной библиотеки. Выбранная библиотека автоматически включается в программу пользователя.

Для включения библиотеки в программу выберите пункт меню «Скетч/Подключить библиотеку». В открывшемся списке библиотек выберите «Button». Сразу после того, как вы щелкните по названию выбранной библиотеки мышкой, в текст программы в верхней ее части добавится оператор *#include*, а за ним в треугольных скобках — имя файла библиотеки.

В **листинге 6.2** этот оператор вы можете видеть в **строке 1**.



ПРИМЕЧАНИЕ.

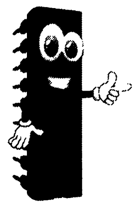
Наличие оператора ***#include*** *в тексте программы означает, что библиотека подключена.*

Оператор *#include* — это стандартный оператор, имеющийся практически в любом языке программирования. Этот оператор как бы вставляет библиотечные функции в текст вашей программы в том ее месте, где этот оператор находится.

Листинг 6.2. Программа переключения светодиода с использованием внешней библиотеки

```
   /*
     Программа переключаемого светодиода
     антидребезг с использованием внешней библиотеки
   */

1  #include <Button.h>

   // Определение констант:
2  const int buttonPin = 2;     // номер контакта кнопки
3  const int ledPin = 8;        // номер контакта светодиода
4  const int DelayTime = 20;    // Задержка антидребезга

   // Создание объекта «кнопка»
5  Button button1 (buttonPin, DelayTime);

6  void setup() {
     // Инициализация контактов модуля
7    pinMode(ledPin, OUTPUT);    // конт. светодиода
8    pinMode(buttonPin, INPUT_PULLUP);  // конт. кнопки
9    digitalWrite(ledPin, HIGH);
   }

10 void loop() {
     // Вызов метода ожидания стабильного состояния кнопки
11   button1.scanState();
12   if ( button1.flagClick == true ) {
13     button1.flagClick = false;  // сброс признака клика
       // Переключение светодиода:
14     digitalWrite(ledPin, !digitalRead(ledPin));
     }
   }
```

Вы можете использовать *#include* для подключения своих собственных программных процедур к своей основной программе. Запишите текст этих процедур в отдельный файл, а в основную программу впишите оператор *#include*. В качестве параметра укажите имя вашего присоединяемого файла.

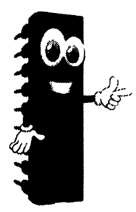


СОВЕТ.

Для подключения библиотеки совсем не обязательно использовать команду «Подключить библиотеку». Вы можете просто вписать нужную строку в текст вашей программы вручную.

Если вы скачаете с сайта «Мир книг по микроэлектронике» приведенный ниже пример программы в электронном виде, то там уже имеется нужная запись, а, значит, библиотека уже подключена.

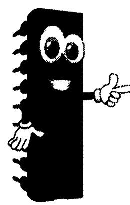
После присоединения к вашей программе библиотеки «Button» в распоряжении программиста появляется ряд новых функций, помогающих работать с кнопками и контактами. Эти функции реализуют алгоритмы, устраняющие как проблемы с дребезгом контактов, так и влияние других видов помех.



ПРИМЕЧАНИЕ.

В связи с тем, что библиотека «Button» написана на языке C++, все библиотечные функции работают в стиле объектного программирования.

Это значит, что сначала мы должны создать объект «Кнопка», при помощи которого мы и будем осуществлять доступ к нашей физической кнопке. Если бы у нас в программе было несколько кнопок, то для каждой из них мы должны были бы создать свой отдельный объект типа «Кнопка».



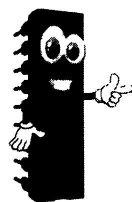
ПРИМЕЧАНИЕ.

О том, как создается объект и как с ним нужно работать, мы узнаем в процессе построчного разбора программы, приведенной в ***листинге 6.2****.*

Итак, начнем по порядку. В начале программы, в **строках 2, 3, 4** определяются все необходимые для работы программы константы. Эта часть программы полностью соответствует ее предыдущей версии (см. **листинг 6.1**). Оператор, создающий объект «Кнопка», расположен в **строке 5**. Формат оператора следующий:

Button ИмяКнопки (НомерКонтакта, ПериодОбработки);

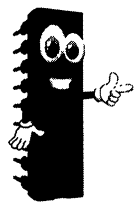
Имя кнопки программист может выбрать по своему усмотрению.



ПРИМЕЧАНИЕ.

Оно должно подчиняться стандартным требованиям к синтаксису имен.

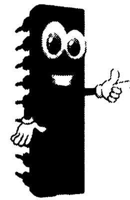
Параметр «*НомерКонтакта*», в нашем случае, соответствует переменной *buttonPin*. *ПериодОбработки* — имеет такой же смысл, как время задержки в предыдущей версии программы.



ПРИМЕЧАНИЕ.

Это время должно немного превышать продолжительность дребезга для ваших конкретных контактов. Однако оно не должно быть слишком большим. ***ВремяОбработки*** *должно быть значительно меньше интервала времени между двумя последовательными нажатиями кнопки. Иначе вы не сможете нажимать кнопку слишком быстро.*

В нашей программе в **строке 5** мы создаем объект «*Кнопка*» с именем *Button1*. В объектном программировании любой объект имеет свои «свойства» и «методы». При обращении к «методу» объекта он выполняет определенные действия.



ЧТО ЕСТЬ ЧТО.

*«**Метод**» – это аналог понятию «функция» в обычной программе. Понятие **«свойство»** аналогично понятию **«переменная»**. Значение «свойства» объекта можно прочитать. «Свойству» можно присвоить новое значение.*

Объект «Кнопка» имеет **два встроенных метода**:

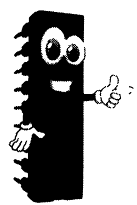
- метод проверки ожидания стабильного состояния сигнала;
- метод фильтрации сигнала по среднему значению.

Например, для объекта *Button1* методы будут выглядеть следующим образом:

Button1.scanState(); // метод проверки ожидания стабильного состояния сигнала

Button1.filterAvarage(); // метод фильтрации сигнала по среднему значению

В своей программе вы можете использовать любой, но только один из этих двух методов. Но ваша программа должна обеспечить многократный вызов выбранного вами метода.



ПРИМЕЧАНИЕ.

Для этого вы должны либо включить его в основной цикл программы (желательно в самое его начало), либо обеспечить вызов метода по прерыванию от таймера с максимально возможной частотой вызова.

Что же делает каждый их этих методов?

Метод проверки ожидания стабильного состояния сигнала

При каждом вызове производит опрос состояния закрепленной за ним кнопки и ведет от вызова к вызову статистическую обработку. По результатам этой обработки устанавливаются следующие свойства объекта:

Button.flagPress — примет значение *true* при стабильно нажатой кнопке;

Button.flagPress — примет значение *false* при стабильно отжатой кнопке;

Button.flagClick — примет значение *true* при кратковременном нажатии и отпускании кнопки.

Все свойства устанавливаются только в момент достижения стабильного состояния контактов кнопки.

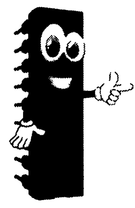
Метод фильтрации сигнала по среднему значению

Отличается от предыдущего метода лишь алгоритмом обработки входного сигнала. В процессе работы устанавливает следующие свойства:

Button.flagPress — примет значение *true* когда на входе стабильно установится сигнал низкого уровня

Button.flagPress — примет значение *false* когда на входе стабильно установится сигнал высокого уровня

Button.flagClick — примет значение *true* при изменении сигнала на входе с высокого уровня на низкий.



ПРИМЕЧАНИЕ.

Подробное описание алгоритма работы и текста программ библиотеки Button вы можете найти в Интернете по адресу: http://mypractic.ru/urok-8-cifrovaya-filtraciya-signalov-v-programmax-dlya-arduino.html

Вернемся к нашей программе и посмотрим, как **методы и свойства объекта «Кнопка»** используются для решения нашей задачи. Описывать функцию *setup()* мы не будем, так как размещенные в теле этой функции команды уже много раз описывались в предыдущих программах.

Нововведения мы увидим в функции *loop()*. Первая команда функции *loop()* (**строка 11**) вызывает метод *scanState()* объекта *Button1*.

Метод производит заложенные в нем статистические действия, и управление переходит к **строке 12**. В этой строке находится оператор *if*, который оценивает значение свойства *flagClick* объекта *Button1*.

Строки 13 и 14 выполняются только в том случае, если значение свойства *flagClick* равно *true*. То есть, если обнаружен факт кратковременного нажатия и отпускания кнопки.

В **строке 13** программа сбрасывает значение *flagClick* для того, чтобы избежать повторной обработки события «Нажатие кнопки».

В **строке 14** мы видим уже знакомое нам выражение, переключающее светодиод на выходе *ledPin*.

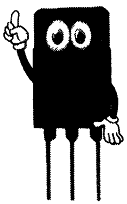
В результате работы описанной выше программы при каждом нажатии кнопки светодиод переключится в противоположное состояние. При этом факт нажатия будет определен путем статистической обработки исключающей влияния дребезга контактов и электромагнитных помех.

ГЛАВА 7

МИГАЮЩИЙ СВЕТОДИОД

Постановка задачи

Программу переключаемого светодиода, описанную в предыдущей главе, легко превратить в программу мигающего светодиода.



ЗАДАЧА.

Создать программу, которая при нажатии кнопки будет вызывать мигание светодиода, а при отпускании кнопки мигание должно прекратиться, и светодиод должен погаснуть.

Схема

*Мы снова используем схему, которая использовалась во всех предыдущих примерах (см. **рис. 4.1**).*

Алгоритм

1. Считать состояние кнопки.
2. Оценить состояние кнопки. Если кнопка нажата, перейти к процедуре мигания. Если кнопка не нажата, принудительно погасить светодиод.
3. В процедуре мигания один раз выполнить следующие действия:
   3.1. зажечь светодиод;
   3.2. сформировать задержку заданной длительности;
   3.3. погасить светодиод;
   3.4. сформировать заданную задержку.
4. Перейти к пункту 1 алгоритма.

В результате работы такого алгоритма перед каждым циклом загорания и потухания светодиода происходит оценка состояния кнопки. Это позволяет прекратить мигание сразу, после того, как кнопка будет отпущена.

Программа

Вариант программы, реализующий описанный выше алгоритм, приведен **в листинге 7.1**.

Начальная часть программы (**строки 1...6**) пояснений, я думаю, не требуют. Они уже не раз встречались в наших предыдущих программах.

В **строке 7** начинается основной цикл программы. Команды основного цикла последовательно выполняют все пункты приведенного выше алгоритма.

В **строке 8** считывается и оценивается значение на входе *buttonPin*. Если это значение равно LOW (кнопка нажата), выполняются операции мигания светодиода (**строки 9...12**).

В противном случае выполняется оператор, непосредственно следующий за оператором *else* (**строка 13**). Он записывает на выход светодиода (*ledPin*) значение высокого логического уровня (тушит светодиод).

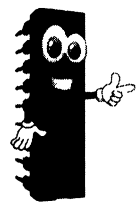
Листинг 7.1. Программа мигающего светодиода

```
   /*
     Программа «Мигающий светодиод»
   */

   // Определение констант
1  const int buttonPin = 2;     // номер контакта кнопки
2  const int ledPin = 8;        // номер контакта
   светодиода
3  const int DelayBlink = 200;  // Задержка мигания

4  void setup() {
     // Инициализиация контактов модуля:
5    pinMode(ledPin, OUTPUT);            // конт. светодиод
6    pinMode(buttonPin, INPUT_PULLUP);   // конт. кнопки
   }

7  void loop() {
     // Если кнопка нажата:
8    if (digitalRead(buttonPin)==LOW) {
9      digitalWrite(ledPin, LOW);  // Зажигаем светодиод
10     delay (DelayBlink);         // Задержка
11     digitalWrite(ledPin, HIGH); // Тушим светодиод
12     delay (DelayBlink);         // Задержка
     }
13   else digitalWrite(ledPin, HIGH); // Тушим светодиод
   }
```

Процедура мигания также очень проста. В **строке 9** на выход светодиода *ledPin* записывается низкий логический уровень (светодиод зажигается).

В **строке 10** формируется задержка. Для этого используется команда *delay()*.

В качестве параметра процедуры используется константа *DelayBlink*, заданная в начале программы (**строка 3**).

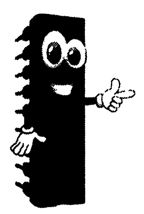


ПРИМЕЧАНИЕ.

Константа DelayBlink определяет время свечения светодиода, а, значит, половину периода мерцания.

В **строке 11** находится команда, которая тушит светодиод.

В **строке 12** мы видим еще одну команду задержки. Она определяет длительность нахождения светодиода в погашенном состоянии. Эта длительность также равна *DelayBlink*. Если учесть, что значение константы *DelayBlink* равно 200, то период мигания нашего светодиода будет равен 400 миллисекундам, то есть 0,4 секунды.



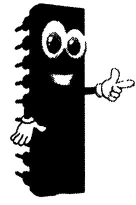
ПРИМЕЧАНИЕ.

Временем выполнения остальных операций (чтения состояния кнопки, оценки этого состояния и вывода значений на светодиод) можно пренебречь.

Поэтому частота мигания светодиода будет равна:

1/0,4 = 2,5 Гц (два с половиной герца).

Главным **недостатком** программы, приведенной в **листинге 7.1**, можно считать то, что при своей работе она полностью забирает все ресурсы микроконтроллера. Эта проблема легко решается применением **механизма прерываний**.



ПРИМЕЧАНИЕ.

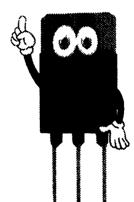
*Прерывания по таймеру мы рассмотрим в **главе 10**. А в следующей главе, мы добавим к нашей схеме еще несколько дополнительных светодиодов, вспомним новогодний праздник и заставим «бежать огни».*

ГЛАВА 8

БЕГУЩИЕ ОГНИ

Постановка задачи

В настоящее время тема бегущих огней потеряла свою актуальность. В продаже имеется такое огромное количество всяческих гирлянд, у которых вся схема управления запрятана прямо в вилке или в малюсенькой коробочке, висящей на проводе. Все уже забыли те времена, когда созданием и сборкой различных схем бегущих огней в канун нового года занимались тысячи радиолюбителей нашей страны. Но мы в данной главе все-таки используем эту задачу в качестве примера.



ЗАДАЧА.

Создать схему и программу, реализующую эффект «бегущих огней». Программа «бегущих огней» должна управлять цепочкой из восьми светодиодов (в дальнейшем вместо каждого светодиода можно будет подключить целую гирлянду). В каждом цикле «пробегания» огня светодиоды должны зажигаться по очереди, начиная с первого и заканчивая последним. В каждый момент времени должен гореть один из восьми свето-

диодов. Разрабатываемое устройство также должно иметь кнопку управления. Назначение кнопки – реверс направления движения огней. При отжатой кнопке огни должны бежать слева направо, а при нажатой кнопке – справа налево.

Схема

*Один из вариантов возможной схемы бегущих огней приведен на **рис. 8.1**.*

Собрать такую схему можно на **универсальном монтажном контактном поле** (см. **рис. 8.2**). Такое поле позволяет создавать несложные устройства без использования пайки. Подобные монтажные контактные модули, соединительные кабели с разъемами и другие комплектующие, используемые в нашем примере, можно купить в магазине радиодеталей, на радиорынке или в Интернете.

Алгоритм

1. Проверить состояние кнопки.
2. Если кнопка отпущена, запустить процесс однократного пробегания огня слева направо (один раз по очереди загораются все восемь светодиодов).
3. Если кнопка отпущена, запустить процесс пробегания огня справа налево.
4. По окончанию процесса однократного пробегания, продолжить выполнение алгоритма сначала. То есть, перейти к пункту 1 данного алгоритма.

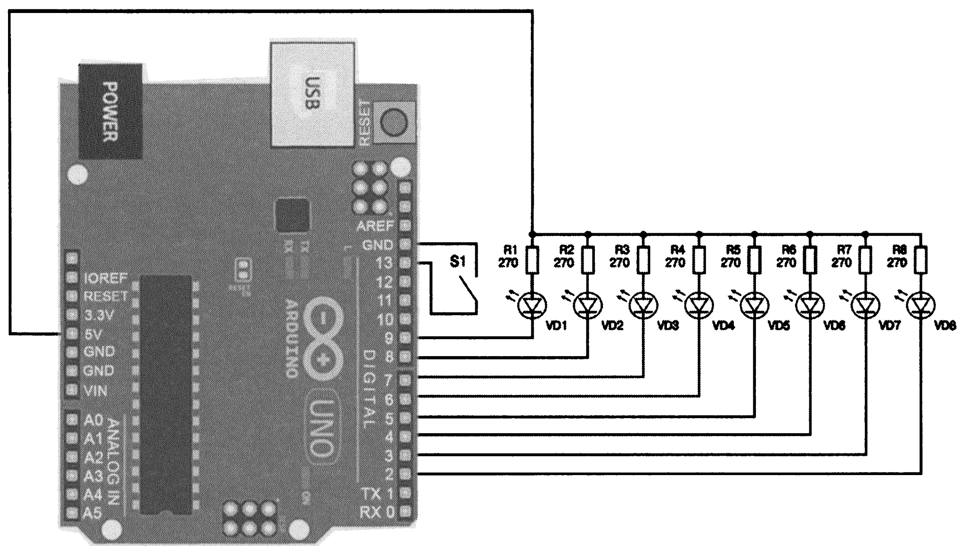
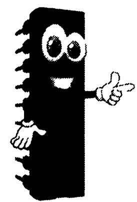


Рис. 8.1. *Схема Бегущих огней*

Рис. 8.2. *Внешний вид «бегущих огней» на монтажной плате*



ПРИМЕЧАНИЕ.

Особенность данного алгоритма в том, что проверка состояния кнопки происходит один раз за один цикл пробегания (8 шагов). Поэтому, если нажать кнопку в тот момент, когда огонь еще не «добежал» до конца, он сначала закончит свой «бег», а уже потом «побежит» в обратную сторону.

Первый вариант программы, реализующей описанный выше алгоритм, приведен в **листинге 8.1**.

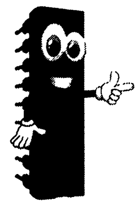
Первый вариант программы

Начинается программа, как обычно, с определения констант и переменных. В данной программе мы определяем значение всего двух констант.

В **строке 1** определяется номер контакта для подключения кнопки.

В **строке 2** задается величина задержки одного шага «бега» огней.

В **строках 3...6** расположена функция начальных установок *setup()*. В теле функции происходит установка режимов работы всех используемых в программе информационных контактов модуля Ардуино.



ПРИМЕЧАНИЕ.

*Для задания режимов работы группы контактов, предназначенных для подключения светодиодов, организован **программный цикл**.*

Для организации цикла в данном случае мы будем использовать оператор *for*. Этот оператор применяется и в других местах данного программного примера (см. **листинг 8.1**). **Например**, в **строках 4, 10, 15**.

Формат оператора *for* следующий:

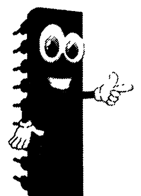
for (НачЗнач, Условие, Инкремент);
{
Команды тела цикла;
}

Оператор имеет три параметра.

Параметр «*НачЗнач*» — это команда, присваивающая начальное значение переменной, которая называется **«счетчик цикла»**. Значение счетчика цикла изменяется в процессе его работы. В **строке 4** программы в качестве счетчика цикла используется переменная *thisPin*.

Параметр «*Условие*» — это логическое выражение. Цикл выполняется до тех пор, пока результат этого выражения — **TRUE (Истина)**. Обычно в этом выражении сравнивается значение счетчика цикла и его предельное значение. Но допустимо и любое другое логическое выражение.

Параметр «*Инкремент*» — представляет собой выражение, которое увеличивает значение счетчика цикла при каждом проходе (итерации). Чаще всего за один проход значение счетчика цикла увеличивается на единицу.



ПРИМЕЧАНИЕ.

Но никаких ограничений тут нет. Увеличивать счетчик цикла, при необходимости, можно на любую величину, а также Инкремент можно менять на Декремент. То есть с каждой итерацией не увеличивать, а уменьшать значение счетчика.

Здесь уместно сказать, что в языке СИ именно для Инкрементации и Декрементации различных величин часто используются два оператора, характерных именно для этого языка программирования:

- двойной плюс «++»;
- двойной минус «– –».

Каждый из этих операторов используется двумя разными способами. **Например**, для переменной X применение этих операторов будет выглядеть так:

X++ — увеличение значения переменной на единицу после ее использования;

++X — увеличение значения переменной на единицу перед ее использованием;

X– – — уменьшение значения переменной на единицу после ее использования;

– –X — уменьшение значения переменной на единицу перед ее использованием.

В **табл. 8.1** работа операторов инкрементации и декрементации описана на примерах. Значение, которое мы решили присвоить переменной X перед началом операции, взято просто для примера. Под операцией мы понимаем процесс вычисления Выражения.

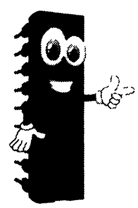
Логика работы операторов инкременации и декременации *Таблица 8.1*

Выражение	Значение X до операции	Значение Y после операции	Значение X после операции
Y = X++	5	5	6
Y = ++X	5	6	6
Y = X– –	5	5	4
Y = – –X	5	4	4

Но вернемся к описанию программы.

В **строке 4** программы организован цикл, который перебирает цифровые контакты модуля Ардуино со второго по девя-

тый, и настраивает режим работы каждого из этих контактов. Эти контакты используются для подключения светодиодов.



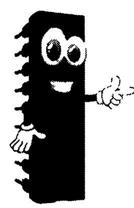
ПРИМЕЧАНИЕ.

Как уже говорилось, в качестве ***счетчика цикла*** *используется переменная thisPin.*

Первый параметр цикла присваивает счетчику значение 2. В качестве условия используется выражение *thisPin<10*. В третьем параметре значение счетчика цикла увеличивается на единицу. Такие параметры обеспечивают перебор значений переменной *thisPin* от 2 до 9.

В теле цикла всего два оператора:

- первый (**строка 5**) настраивает контакт с номером *thisPin* на вывод информации;
- второй (**строка 6**) устанавливает на выводе номер *thisPin* высокий логический уровень.



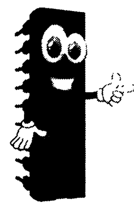
ВЫВОД.

Таким образом, цикл настраивает по очереди режим работы всей группы контактов, используемых для подключения светодиодов, и тушит каждый светодиод, подавая на выход логическую единицу.

Основной цикл программы *loop()* начинается с команды, считывающей и оценивающей состояние кнопки (**строка 9**). Если состояние кнопки равно HIGH (кнопка отпущена) выполняется цикл однократного «пробегания огня» в прямом направлении (**строки 10, 11, 12 и 13**). В противном случае выполняется цикл «пробегания огня» в обратном направлении (**строки 15, 16, 17 и 18**).

Листинг 8.1. Программа «Бегущие огни»

```
   /*
    Простая программа «Бегущие огни»
   */

1  const int buttonPin = 13;    // номер контакта кнопки
2  const int LoopTime = 200;    // задержка шага бег. огня

3  void setup() {
     // Цикл инициализации контактов светодиодов:
4    for (int thisPin = 2; thisPin < 10; thisPin++) {
5      pinMode(thisPin, OUTPUT);
6      digitalWrite(thisPin, HIGH);
     }
     // Инициализация контакта кнопки:
7    pinMode(buttonPin, INPUT);
   }

8  void loop() {
9    if (digitalRead(buttonPin)==HIGH) {
       // Цикл одного пробега в прямом направлении:
10     for (int thisPin = 2; thisPin < 10; thisPin++) {
         // Включение текущего светодиода:
11       digitalWrite(thisPin, LOW);
12       delay(LoopTime);  // Задержка
         // Выключение текущего светодиода:
13       digitalWrite(thisPin, HIGH);
       }
     }
14   else {
       // Цикл одного пробега в обратном направлении:
15     for (int thisPin = 9; thisPin > 1; thisPin--) {
16       digitalWrite(thisPin, LOW);
17       delay(LoopTime);  // Задержка
18       digitalWrite(thisPin, HIGH);
       }
     }
   }
```



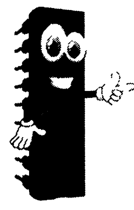
ПРИМЕЧАНИЕ.

Оба цикла (прямой и обратный) практически одинаковы. ***Разница*** *только в том, что переменная счетчика цикла в одном случае с каждой итерацией цикла увеличивается на единицу, а во втором случае – уменьшается.*

Обратимся сначала к **прямому циклу**, который начинается в **строке 10** программы. Все параметры этого цикла аналогичны уже рассмотренному нами циклу, который находится в **строке 4**. Поэтому цикл в **строке 10** тоже перебирает все значения счетчика цикла *thisPin* от 2 до 9.

Для каждого значения *thisPin* программа выполняет три последовательных операции:

- в **строке 11** программа зажигает текущий светодиод, подключенный к выходу *thisPin*;
- в **строке 12** формирует задержку длительностью *LoopTime*;
- в **строке 13** затем тушит текущий светодиод.



ПРИМЕЧАНИЕ.

В результате за восемь итераций (шагов) цикла загораются по очереди все 8 светодиодов.

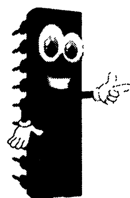
Цикл обратного «пробегания» расположен в **строках 15, 16, 17 и 18**. Обратный цикл работает так же, как и прямой цикл, но перебирает контакты светодиодов в обратной последовательности (с девятого по второй).

По этой причине обратный цикл от прямого отличается лишь значениями своих параметров. Теперь в качестве начального значения счетчику цикла *thisPin* присваивается значение 9. В качестве условия продолжения цикла используется выражение

thisPin>1.

А вместо команды инкрементации используется команда **декрементации** (*thisPin--*). В результате мы получим эффект обратного перебора значений. Набор команд в теле обратного цикла ничем не отличается от таких же команд цикла прямого.

Второй вариант – используем один универсальный цикл



ПРИМЕЧАНИЕ.

*Если вы посмотрите на текст предыдущей программы (**листинг 8.1**), вы легко заметите, что цикл прямого и цикл обратного «пробега» отличаются только параметрами. Это дает возможность использовать вместо двух циклов **один универсальный цикл**.*

В **листинге 8.2** показано, как это сделать. Программа, приведенная в **листинге 8.2**, представляет собой доработанную программу из **листинга 8.1**. Поэтому опишем только изменения.

Для начала в программу мы введем три переменных (**строки 3, 4 и 5**). В этих переменных будут храниться и в нужный момент изменяться параметры цикла, именно:

- значение начала цикла;
- значение конца цикла;
- значение шага (+1 или –1).

Функция *setup()* не имеет никаких изменений. А вот в теле функции *loop()* все становится немного по-другому.

Оператор *if* в **строке 12** считывает и оценивает состояние клавиатуры. В зависимости от этого состояния он присваивает трем нашим новым переменным соответствующие значения.

Если кнопка нажата, то переменным присваиваются значения параметров цикла для пробега вперед (**строка 13**).

Если кнопка отпущена, то переменным присваиваются параметры цикла «пробега назад» (**строка 15**).

Листинг 8.2. Программа «Бегущие огни» с общим универсальным циклом

```
   /*
     Программа «Бегущие огни»
     с общим универсальным циклом
   */

1  const int buttonPin = 13;   // Номер контакта кнопки
2  const int LoopTime = 200;   // Задержка шага бег. огня

3  int loopStart = 2;     // Начало цикла бега
4  int loopStop = 10;     // Конец цикла бега
5  int loopStep = 1;      // Шаг цикла бега

6  void setup() {
     // Цикл инициализации контактов светодиодов:
7    for (int thisPin = 2; thisPin < 10; thisPin++) {
8      pinMode(thisPin, OUTPUT);
9      digitalWrite(thisPin, HIGH);
     }
     // Инициализация контакта кнопки:
10   pinMode(buttonPin, INPUT);
   }

11 void loop() {
12   if (digitalRead(buttonPin)==HIGH)
13      { loopStart = 2; loopStop = 10; loopStep = 1; }
14   else
15      { loopStart = 9; loopStop = 1; loopStep = -1; }
     // Цикл одного пробега бегущего огня:
16   for (int thisPin = loopStart; thisPin != loopStop;
   thisPin+=loopStep) {
       // Включение текущего светодиода:
17     digitalWrite(thisPin, LOW);
18     delay(LoopTime);  // Задержка
       // Выключение текущего светодиода:
19     digitalWrite(thisPin, HIGH);
     }
   }
```

Сам универсальный цикл пробега находится в **строках 16...19**.

Строки 17, 18 и 19 составляют **тело цикла** и полностью соответствуют командам, которые мы видели в теле цикла в предыдущем примере. Ну, а параметры цикла определяются следующим образом.

В качестве начального значения для переменной *thisPin* присваивается значение переменной *loopStart*.

В качестве условия работы цикла используется логическое выражение *thisPin != loopStop*.

Оператор «!=» — в языке СИ представляет собой логический оператор, выполняющий проверку условия «не равно». Представленное выражение возвращает значение TRUE («Истина») в том случае, если значения переменных *thisPin* и *loopStop* не равны между собой. То есть, цикл будет длиться до тех пор, пока значение счетчика цикла *thisPin* будет не равно его предельному значению, хранящемуся в переменной *loopStop*. Ну, а третий параметр нашего универсального цикла выглядит следующим образом:

thisPin += loopStep;

Составной оператор «+=» означает: присвоить переменной в левой части выражения значение суммы значений из левой и правой части. То есть, вышеприведенное выражение полностью эквивалентно выражению *thisPin = thisPin + loopStep;*

Если кнопка не была нажата, значение переменной *loopStep* равно единице, и при каждой итерации значение *thisPin* увеличивается на единицу.

Если кнопка нажата, значение *loopStep* равно минус один. А это значит, что при каждой итерации счетчик цикла *thisPin* будет уменьшаться на единицу.

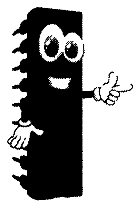
Это приведет к тому, что при нажатой и при отпущенной кнопке цикл будет:

- в первом случае — прямым;
- во втором случае — обратным.

И в том, и в другом случае цикл будет начинаться со значения *loopStart* и заканчиваться значением *loopStop*. Но при

прямом и при обратном направлении перебора значения, присвоенные переменным *loopStart* и *loopStop*, будут разные (см. **строки 13 и 15**).

Для того чтобы превратить схему, у которой лишь 8 светодиодов на выходе, в настоящие бегущие огни нужно подключить на выходы Ардуино вместо каждого из светодиодов силовой ключ. К каждому ключу нужно подключить гирлянду лампочек или светодиодов.



ПРИМЕЧАНИЕ.

Светодиоды в каждой гирлянде могут быть включены как последовательно, так и параллельно.

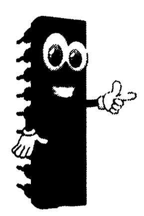
Нужно только рассчитать питающее напряжение. При последовательном соединении придется подавать достаточно большое напряжение. Оно будет вычисляться по формуле:

$$U_{герл} = U_{раб} \times N_{св}$$

где $U_{герл}$ — напряжение на гирлянде; $U_{раб}$ — рабочее напряжение одной лампочки (светодиода); $N_{св}$ — количество светодиодов.

Все восемь гирлянд нужно сложить таким образом, чтобы светодиоды от разных гирлянд перемежались по порядку. Сначала светодиод первой гирлянды, затем светодиод второй, затем третьей и т. д. до светодиода восьмой. После этого все должно повторяться: светодиод первой гирлянды, второй и т. д.

Мы не будем приводить подробную схему полной гирлянды, так как это не является целью данной книги. Схема и программа «бегущих огней» является просто хорошим учебным примером в конструировании и составлении программ.



СОВЕТ.

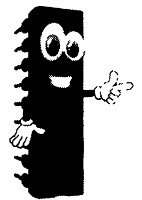
При желании, схему силового ключа можно легко найти в Интернете.

ГЛАВА 9

АЛЬТЕРНАТИВНЫЕ СПОСОБЫ ФОРМИРОВАНИЯ ЗАДЕРЖКИ

Постановка задачи

Во всех предыдущих программах, приведенных в этой книге, в которых нужно было сформировать задержку по времени, мы это делали при помощи функции *delay()*.



ПРИМЕЧАНИЕ.

*Однако у этой функции есть большой **недостаток**: в процессе своей работы она полностью занимает все ресурсы микроконтроллера.*

Если программа вызывает команду *delay()*, то до тех пор, пока эта команда не закончит свою работу, никакая другая команда выполниться не может. То же самое касается и еще одной функции, которая также используется для формирования программной задержки в языке Ардуино.

Функция **называется *delayMicroseconds()*** и от функции *delay()* отличается тем, что формирует задержку, которая задается в микросекундах.

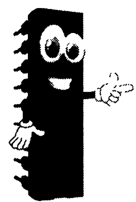
Специально для тех случаев, когда в момент формирования задержки необходимо выполнять еще какие-либо действия, в языке Ардуино имеется другой механизм формирования интервалов времени.

В основе этого механизма лежат встроенные **системные часы**. Их, как мы знаем, транслятор автоматически создает при трансляции вашей программы. Системные часы постоянно отсчитывают время, прошедшее с момента запуска программы. Для чтения текущего времени системных часов в наборе команд языка Ардуино имеются две специальные функции:

millis() — возвращает количество миллисекунд

micros() — возвращает количество микросекунд

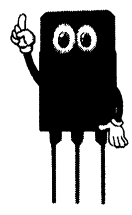
Используются эти функции следующим образом. Перед тем, как сформировать задержку, программа должна при помощи одной из вышеприведенных функций считать текущее значение системного таймера, и сохранить считанное значение в одной из переменных.



ПРИМЕЧАНИЕ.

*Момент времени, когда мы запомнили значение системных часов, и будет считаться началом **периода формирования задержки**.*

Затем программа может выполнять любые другие необходимые действия, но периодически она должна проверять текущее время. Для этого она должна периодически считывать значение системного таймера и вычислять разность между текущим и ранее сохраненным значениями времени. Момент, когда эта разница превысит заданное время задержки, считается **моментом ее окончания**.



ЗАДАЧА.

Доработать одну из программ бегущих огней, из главы 8, исключив из нее операторы delay(), и используя для формирования задержки системные часы и функцию mills(). Для начала возьмем за основу программу, приведенную в листинге 8.1.

Схема

*Разумеется, схема бегущих огней останется прежней (см. **рис. 8.1**).*

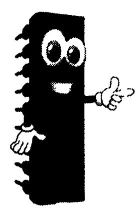
Алгоритм

1. Проверить состояние кнопки.

2. Если кнопка отпущена, запустить процесс однократного пробегания огня слева направо (один раз по очереди загораются все восемь светодиодов).

3. Если кнопка отпущена, запустить процесс пробегания огня справа налево.

4. По окончанию процесса однократного пробегания, продолжить выполнение алгоритма сначала. То есть, перейти к пункту 1 данного алгоритма.



ПРИМЕЧАНИЕ.

По большому счету, алгоритм программы не изменяется. Меняется лишь способ формирования задержки.

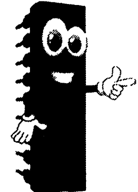
Подробнее о том, как это делается, рассмотрим в процессе детального разбора каждой из двух версий представленных ниже программ.

Первый вариант программы

Самый простой способ решения вышеописанной задачи — заменить команду *delay()* другой, специально созданной программой задержки, которая, в свою очередь, будет использовать системный таймер и функцию *mills()*.

Программа, реализующая данный способ формирования задержки, приведена в **листинге 9.1**. Если сравнить эту программу с оригиналом (см. **листинг 8.1**), то изменения в новой версии программы минимальны.

В программу добавлена новая переменная *lastTime* (**строка 3** программы). В этой переменной мы будем хранить значение системного таймера на момент начала периода формирования задержки.



ПРИМЕЧАНИЕ.

*Обратите внимание на тип этой переменной. Она имеет тип: **unsigned long**. Переменная, которая должна хранить значение системного счетчика времени, должна иметь **именно такой тип**, так как размерность системного счетчика времени – четыре байта. Такую же размерность должна иметь и переменная!*

Следующее изменение новой программы по сравнению со старой — своя собственная специально созданная процедура задержки. Процедуру мы **назвали *ndelay()*** и поместили в конце программы в **строках 20, 21 и 22**.

Листинг 9.1. Бегущие огни с альтернативной процедурой задержки

```
   /*
     Бегущие огни. Без использования функции delay()
   */

1  const int buttonPin = 13;       // Номер контакта кнопки
2  const int LoopTime = 200;   // Задержка шага бег. огня
3  unsigned long lastTime;      // Переменная текущ. время

4  void setup() {
     // Цикл инициализации контактов светодиодов:
5    for (int thisPin = 2; thisPin < 10; thisPin++) {
6      pinMode(thisPin, OUTPUT);
7      digitalWrite(thisPin, HIGH);
     }
     // Инициализация контакта кнопки:
8    pinMode(buttonPin, INPUT);
   }

9  void loop() {
10   if (digitalRead(buttonPin)==HIGH) {
       // Цикл одного пробега в прямом направлении:
11     for (int thisPin = 2; thisPin < 10; thisPin++) {
         // Включение текущего светодиода:
12       digitalWrite(thisPin, LOW);
13       ndelay(LoopTime);  // Задержка
         // Выключение текущего светодиода:
14       digitalWrite(thisPin, HIGH);
       }
     }
15   else {
       // Цикл одного пробега в прямом направлении:
16     for (int thisPin = 9; thisPin > 1; thisPin--) {
17       digitalWrite(thisPin, LOW);
18       ndelay(LoopTime);  // Задержка
```

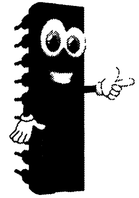
```
19        digitalWrite(thisPin, HIGH);
        }
      }
    }

20  void ndelay (int delayTime) {
21    lastTime = millis();
22    while (millis() - lastTime < delayTime) {
23      // тут можно вставить любые команды
        // .............
      }
    }
```

Процедура имеет один входной параметр *delayTime* (см. **строку 20**). При помощи этого параметра в процедуру передается длительность задержки.

Функция *ndelay()* работает следующим образом.

В **строке 21** считывается текущее значение системного таймера. Считанное значение помещается в переменную *lastTime*.

Далее в **строке 22** начинается **цикл ожидания**. Это уже знакомый нам цикл *while* (цикл пока...). Условием выполнения цикла является выражение, которое читает значение системного таймера и вычисляет время, прошедшее с момента начала задержки. То есть, вычисляет разность *millis() – lastTime*.



ПРИМЕЧАНИЕ.

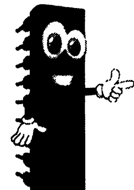
Пока эта разность не превышает значение переменной delayTime, цикл продолжается. В противном случае цикл завершается. В результате завершается и сама функция ndelay().

Функция *ndelay()* используется везде в программе там, где раньше использовалась функция *delay()*. Мы можем видеть вызов нашей новой функции **в строках 13 и 18** программы.

Применение функции *ndelay()* дает нам возможность в момент формирования задержки выполнять еще какие-либо дополнительные действия. Команды, выполняющие эти действия, мы должны разместить вместо комментариев в строках, обозначенных в листинге номером **23**.

Второй вариант программы

Как легко заметить из описания предыдущей версии программы (**листинг 9.1**), она действительно позволяет выполнять дополнительные действия в момент формирования задержки, но для очень узкого спектра применений.



ПРИМЕЧАНИЕ.

Тот факт, что добавлять дополнительные команды придется не в основной цикл, а в отдельную процедуру, очень сужает возможности этих дополнительных операций. Однако, искусство программирования позволяет скомпоновать программу таким образом, чтобы место, куда можно будет вставить дополнительные команды, оказалось в самом центре основного цикла программы.

Для решения этой задачи мы возьмем в качестве исходной другую версию программы из **главы 8**. А именно, программу, приведенную в **листинге 8.2**. Этот вариант программы **использует единый универсальный цикл для прямого и обратного «бега» «огней»**.

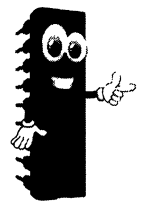
Это облегчает нам поставленную выше задачу. В **листинге 9.2** представлен вариант программы, в которой в один общий цикл объединены все входящие в нее циклы.

Это, **во-первых**, универсальный реверсивный цикл «пробегания огня». **Во-вторых**, цикл формирования задержки. И все это заключено в главный цикл программы *loop()*.

Начальная часть программы (**строки 1...11**) полностью соответствует программе-оригиналу. Лишь в конце функции *setup()* добавлены две строки, в которых определяются начальные значения двух важных переменных.

В **строке 12** указателю текущего выхода *thisPin* присваивается начальное значение. А именно, номер самого первого из выходов на светодиод (*loopStart*). В данном случае первым является выход, с которого будет начинаться «бег вперед».

Затем в **строке 13** обнуляется переменная *lastTime*.

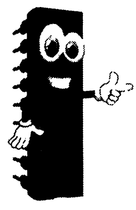


ПРИМЕЧАНИЕ.

Как говорилось выше, переменная lastTime предназначена для хранения текущего значения системного таймера.

Это значение считывается и помещается в переменную в момент начала формирования периода задержки. Нулевое же значение присваивается этой переменной в том случае, когда программа еще не начала формирование задержки. В результате переменная *lastTime* используется в этой программе двояко:

- для хранения мгновенного значения системного времени;
- и еще в одном качестве — в качестве флага.



ЧТО ЕСТЬ ЧТО.

***Флагом** в программировании называют переменную или ячейку памяти, которая в процессе работы программы принимает два значения (флаг установлен, флаг сброшен) и служит для регулирования хода выполнения программы.*

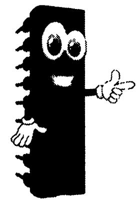
Где-то в одном месте программы, исходя из каких-либо условий, устанавливается или сбрасывается флаг. А затем в

других местах, в зависимости от значения флага, меняется ход выполнения программы.

Под значением «Флаг установлен» и «Флаг сброшен» можно понимать любые допустимые значения переменной. В нашем случае будет считаться, что флаг сброшен, если значение переменной *lastTime* равно нулю.

Когда *lastTime* больше нуля, то мы будем считать, что флаг установлен. То есть, значение времени, которое будет записано в переменную в процессе работы и будет означать, что «флаг установлен».

Значение системного времени в процессе работы программы постоянно меняется.



ПРИМЕЧАНИЕ.

Приблизительно через 50 дней непрерывной работы системных часов, счетчик системных часов переполнится, и в течение одной миллисекунды его значение будет равно нулю.

Но вероятность того, что именно это значение будет записано в переменную *lastTime,* ничтожно мала. Поэтому использование этой переменной в качестве флага вполне оправдано.

В данном конкретном случае наш флаг, будучи сброшенным (равным нулю), индицирует тот факт, что формирование задержки еще не началось. Если же переменная *lastTime* содержит значение текущего времени (отличное от нуля), это означает, что флаг установлен и программа находится в режиме формирования задержки.

Итак, в программе существует один универсальный цикл. И этим циклом служит основной цикл программы *loop().* В начале и в конце основного цикла *loop()* находятся два программных модуля, каждый из которых выполняется только при определенных условиях (далее мы узнаем, при каких).

Начальный модуль цикла располагается в **строках 15, 16 и 17**. А конечный модуль расположен в **строках 19...26**. Условия, при которых выполняется каждый из модулей, определены соот-

ветствующими операторами *if* (в **строке 15** и в **строке 19**, соответственно).

Посредине между этими двумя модулями располагается область, куда программист может включить небольшую программу, выполняющую дополнительные действия.

Рассмотрим работу основного цикла *loop()* программы подробнее.

В **строке 15** происходит оценка значения переменной *lastTime*.

Остальные команды начального блока (**строки 16 и 17**) выполняются только в том случае, если *lastTime* равно нулю, то есть в том случае, если формирование задержки еще не началось. Если условие выполнено, то сначала в **строке 16** соответствующая команда включает текущий светодиод (подавая LOW на выход *thisPin*).

А в **строке 17** считывается и запоминается текущее значение системного времени. После этого значение *lastTime* уже не равно нулю, а, значит, программа переходит в **режим формирования задержки**.

С этого момента начальный блок команд (**строки 16 и 17**) не выполняется до момента окончания задержки. При этом зажженный в **строке 16** светодиод на все время задержки продолжает гореть.

За окончание периода задержки отвечает конечный блок основного цикла (**строки 19...26**). В **строке 19** размещено выражение, при помощи которого программа читает текущее значение системного времени, вычисляет время, прошедшее с начала периода ожидания и сравнивает с заданным в переменной *LoopTime* контрольным значением времени задержки.

Если время окончания периода задержки еще не наступило, то остальные команды этого блока (**строки 20...26**) не выполняются, и управление передается на начало основного цикла (т. е. к **строке 15** программы).

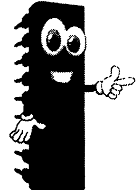
Не выполняются также и команды начального блока (**строки 16 и 17**), так как *lastTime* не равно нулю.

Листинг 9.2. Бегущие огни с одним общим циклом

```
   /*
     Бегущие огни без использования функции delay()
     с одним универсальным общим циклом
   */
   // Определение констант:
1  const int buttonPin = 13;  // номер контакта кнопки
2  const int LoopTime = 200;  // задержка шага бегущ. огня

   // Определение переменных
3  int loopStart = 2;        // начало цикла бега
4  int loopStop = 10;        // конец цикла бега
5  int loopStep = 1;         // шаг цикла бега
6  int thisPin;              // текущий контакт
   unsigned long lastTime;   // текущее время

7  void setup() {
     // Цикл инициализации контактов светодиодов:
8    for (int thisPin = 2; thisPin < 10; thisPin++) {
9      pinMode(thisPin, OUTPUT);
10     digitalWrite(thisPin, HIGH);
     }
     // Инициализация контакта кнопки:
11   pinMode(buttonPin, INPUT);
12   thisPin = loopStart;
13   lastTime = 0;
   }

14 void loop() {
15   if (lastTime == 0) {
16     digitalWrite(thisPin, LOW);
17     lastTime = millis();
     }

18   // Тут можно выполнять другие действия
     // ...........
     // ...........

19   if (millis() - lastTime > LoopTime) {
20     digitalWrite(thisPin, HIGH);
21     thisPin += loopStep;
22     if (thisPin == loopStop) {
```

23	`if (digitalRead(buttonPin)==HIGH) {` `loopStart = 2; loopStop = 10; loopStep = 1; }`
24	`else { loopStart = 9; loopStop = 1; loopStep = -1;` `}`
25	`thisPin = loopStart;` `}`

26	`lastTime = 0;`
	`}`
	`}`



ПРИМЕЧАНИЕ.

Поэтому весь основной цикл какое-то время, пока программа находится в режиме формирования задержки, крутится вхолостую. Выполняются лишь дополнительные команды, если вы их включили вместо комментариев в ***строке 18****.*

Длительность периода задержки в нашей программе выбрана равной 200 миллисекундам (см. **строку 2**). Этого времени хватит, чтобы дополнительные команды выполнили достаточно сложный круг задач.

Так продолжается до тех пор, пока системное время достигнет такого значения, когда условие оператора *if* в **строке 19** станет равно TRUE (конец задержки). В этом случае выполняется последняя группа операторов, расположенная в **строках 20...26**.

Сначала оператор в **строке 20** гасит текущий светодиод. Затем в **строке 21** производится Инкремент (или Декремент в режиме реверса) значения переменной *thisPin*. То есть изменяется номер текущего выхода на светодиод.

В **строке 22** проверяется, не достигла ли переменная *thisPin* своего предельного значения. Если достигла, это означает, что «огонь» добежал до конца линейки светодиодов и «бег» нужно начинать сначала. Для этого нужно присвоить переменной *thisPin* ее начальное значение.

Но сначала в **строках 23 и 24** считывается и проверяется состояние кнопки. Это делается для того, чтобы при изменении состояния кнопки можно было бы изменить направление «бега» огней.

По результатам оценки состояния кнопки переменным, *loopStart*, *loopStop* и *loopStep* присваиваются значения, соответствующие выбранному направлению «бега» огней:

- если кнопка отпущена, то переменным присваиваются значения для «бега вперед»;
- если кнопка нажата, то значения для «бега назад».

После того, как параметры бега определены, в **строке 25** переменной *thisPin* присваивается начальное значение уже с учетом выбранного направления.

В завершении, в **строке 26** обнуляется переменная *lastTime*. Нулевое значение переменной *lastTime* укажет программе, что формирование периода задержки закончено, и можно начинать новую задержку.

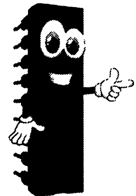
На этом завершающий блок команд цикла и сам основной цикл заканчиваются, и управление снова передается в начало (то есть к **строке 15**). Теперь *lastTime* равно нулю, поэтому начальный блок команд основного цикла на этот раз выполняется. В результате зажигается уже новый светодиод, начинается новый период задержки, и все повторяется сначала.

ГЛАВА 10

РАБОТА С ПРЕРЫВАНИЯМИ ПО ТАЙМЕРУ

Постановка задачи

В предыдущих двух главах мы рассмотрели два способа формирования задержки в программе бегущих огней. Способ, описанный в **главе 9**, в котором для формирования задержки использовались системные часы и функция *mills(),* позволил нам параллельно с задачей бегущих огней дополнительно выполнять вспомогательные задачи.



ПРИМЕЧАНИЕ.

*Однако **самый эффективный способ формирования временных интервалов** – использование механизма прерываний. При этом используется новый для нас вид прерывания – прерывание по таймеру.*

В **главе 4** мы использовали другой вид прерываний — **внешнее прерывание** (прерывание, вызываемое при поступлении внешнего сигнала). Для вызова прерывания по таймеру используется один из системных таймеров микроконтроллера.

Микроконтроллер **ATmega328 имеет на борту три таймера**:

- таймер 0 (8-ми разрядный);
- таймер 1 (16-ти разрядный);
- таймер 2 (так же 8-ми разрядный).

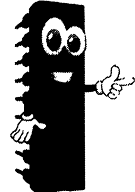
Каждый из таймеров имеет множество режимов работы, которые могут выбираться программным путем. Для формирования интервалов времени обычно используют **режим подсчета импульсов от внутреннего тактового генератора**.

С каждым таймером связано несколько видов прерываний. Чаще всего для формирования временных интервалов используется так называемое **прерывание по переполнению**.

Как работает прерывание по переполнению? Счетчик таймера подсчитывает тактовые импульсы. Каждый такой импульс увеличивает значение счетного регистра таймера на единицу. Когда оно достигнет максимального значения, возникает переполнение таймера. Таймер обнуляется, и счет продолжается с нуля.

В момент переполнения вырабатывается **сигнал прерывания**. При вызове прерывания, как мы знаем, приостанавливается выполнение основной программы, и микроконтроллер переходит к выполнению процедуры обработки прерывания.

Частота импульсов на входе таймера выбирается таким образом, чтобы переполнение происходило с нужным нам периодом.

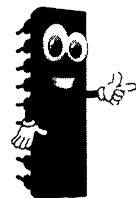


ПРИМЕР.

Таймер 0 *модуля Ардуино, как мы знаем, используется как* ***системные часы****. Частота тактовых импульсов, поступающая на его вход, выбрана так, что переполнение таймера происходит один раз за одну миллисекунду.*

Процедура обработки этого прерывания просто добавляет единицу к значению, хранящемуся в составном 32-х разрядном

регистре системного времени, составленном из четырех ячеек, зарезервированных в ОЗУ микроконтроллера.



ПРИМЕЧАНИЕ.

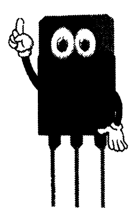
Используя прерывание по таймеру, легко создать программу, в которой некие действия, происходящие в строго фиксированные моменты времени, будут выполняться исключительно в процедуре обработки прерывания. А основной цикл программы останется свободным и может быть использован для решения любых дополнительных задач.

При использовании прерываний по таймеру в модуле Ардуино возникают определенные коллизии. Как уже говорилось выше, все таймеры модуля уже используются системой. Напомним:

- **таймер 0** используется в качестве системных часов, отвечает за ШИМ на цифровых выходах 5 и 6, а также обслуживает функции *millis()*, *micros()*, *delay()*;
- **таймер 1**, формирует ШИМ на цифровых выходах 9 и 10;
- **таймер 2** формирует ШИМ на выводах 3 и 11 и используется при формировании звукового сигнала при помощи команды *tone()*.

Если мы будем использовать один из таймеров для организации прерывания, то закрепленными за ним функциями нам, возможно, придется пожертвовать.

Далее, в этой главе мы опишем два способа организации прерывания для работы программы «бегущие огни».



ЗАДАЧА.

Создать программу «Бегущие огни», в которой для создания временных интервалов будет использоваться прерывание по таймеру.

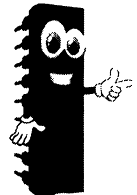
Схема

Схема и на этот раз остается прежней. Она приведена на ***рис. 8.1.***

Используем внешнюю библиотеку прерываний по таймеру

Для того, чтобы в своих программах на Ардуино вы могли использовать прерывание по таймеру, проще всего применить соответствующую внешнюю библиотеку. В стандартном пакете такая библиотека отсутствует. Поэтому нужно использовать библиотеку стороннего разработчика.

В нашем первом примере мы будем использовать **библиотеку *MsTimer2*.**



ПРИМЕЧАНИЕ.

Описание этой библиотеки приведено на ***сайте «Уроки программирования Ардуино» (mypractic.ru)****. Скачать zip-архив, содержащий все файлы библиотечного пакета можно на сайте «Мир книг по микроэлектронике»* ***(book.mirmk.ru)****.*

Подключить библиотеку *MsTimer2* к нашей программе так же просто, как библиотеку *Button*, которую мы учились подключать в **главе 6**. Библиотека *MsTimer2* — это специализированная библиотека для работы с самым последним таймером модуля Ардуино: таймером 2.

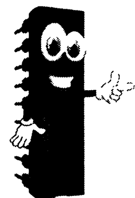
После присоединения библиотеки к программе, в распоряжении программиста появляются три новые функции. Все три функции являются методами класса *MsTimer*, поэтому

для обращения к каждой из функций используется синтаксис вызова метода языка С++. Ниже приводится описание всех этих трех функций.

MsTimer2::set(TimeInt, fnameInt);

Функция инициализации прерывания. Параметр *TimeInt* — это период вызова прерывания по таймеру. Период задается в миллисекундах. Переменная *TimeInt* должна иметь тип *unsigned long*.

Параметр *fnameInt* — это имя функции обработки прерывания. Функцию вы должны создать самостоятельно. Эта функция должна удовлетворять требованиям, предъявляемым к ISR функциям (Interrupt Service Routine).



ПРИМЕЧАНИЕ.

*Подробно эти требования описаны в **главе 4**.*

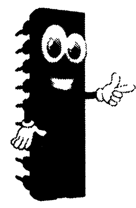
MsTimer2::start()

Запускает процесс периодического вызова прерывания по таймеру, определенный оператором *MsTimer2::set()*. Прерывание вызывается многократно с периодом, который был задан параметром *TimeInt*.

MsTimer2::stop()

Останавливает процесс вызова прерывания по Таймеру.

Исходя из вышеописанного набора функций, мы можем составить алгоритм нашей будущей программы.



ПРИМЕЧАНИЕ.

Определимся сразу: весь алгоритм «бегущих огней» должен выполняться исключительно в процедуре обработки прерывания. Основной цикл программы должен остаться абсолютно свободным.

Для определенности при описании алгоритма введем **два вспомогательных понятия**:

- «текущий зажженный светодиод»;
- «предыдущий зажженный светодиод».

Алгоритм

1. Организовать прерывание по таймеру. Установить период вызова прерывания равным времени одного шага «бегущего огня» (в предыдущих программах эта величина определялась временем задержки).
2. После вызова процедуры обработки прерывания выполнить следующие действия:
   2.1. потушить предыдущий зажженный светодиод. Если он и так не горит, то команда лишь подтверждает это его состояние;
   2.2. считать состояние кнопки и определить направление «бега» в зависимости от того, нажата она или отпущена;
   2.3. перейти к следующему по порядку светодиоду. Если кнопка отпущена, то к следующему в прямом направлении, если нажата — к следующему в обратном направлении;
   2.4. зажечь новый текущий светодиод.
3. Выйти из процедуры обработки прерывания и продолжить выполнение основной программы вплоть до очередного вызова прерывания.

Программа

Вариант программы «Бегущих огней», работающий по прерыванию от таймера и использующий библиотеку *MsTimer2*, приведен в **листинге 10.1**. Рассмотрим программу по порядку.

Листинг 10.1. Использование внешней библиотеки прерывания по таймеру

```
   /*
     Пример 7.  (вариант 1)
     Бегущие огни с использованием прерываний по таймеру
   */

1  #include <MsTimer2.h>

   // Определение констант:
2  const int buttonPin = 13;    // номер контакта кнопки
3  const int LoopTime = 200;    // задержки шага бегущ. огня
4  const int loopStart = 2;     // начало цикла бега
5  const int loopStop  = 9;     // конец цикла бега

   // Определение переменных:
6  volatile int thisPin;     // текущий разряд светодиодов

7  void setup() {
8    // Цикл инициализации контактов светодиодов:
9    for (int thisPin = loopStart; thisPin < = loopStop;
   thisPin++) {
10     pinMode(thisPin, OUTPUT);
11     digitalWrite(thisPin, HIGH);
     }
     // Инициализация контакта кнопки:
12   pinMode(buttonPin, INPUT);
13   thisPin = loopStart;     // уст. нач. значения указателя
     // Инициализация таймера и привязка к прерыванию
14   MsTimer2::set(LoopTime, timerInterupt); // уст. периода
15   MsTimer2::start();      // разрешаем прерывание
   }

16 void loop() {
```

```
17 |   // Основной цикл остается свободным
   |   // Сюда можно поместить основную программу
   |   // на ее фоне будет работать программа бегущ. огней
   | }
   |
   | // Обработчик прерывания
18 | void  timerInterupt() {
19 |    digitalWrite(thisPin, HIGH);   // гасим тек. св.диод
20 |    if (digitalRead(buttonPin)==HIGH) {
   |       // Один шаг бегущ. огня в прямом направлении:
21 |       thisPin++;
22 |       if (thisPin > loopStop) thisPin = loopStart;
   |    }
23 |    else {
   |       // Один шаг бегущ. огня в обратн. направлении:
24 |       thisPin--;
25 |       if (thisPin < loopStart) thisPin = loopStop;
   |    }
26 |    digitalWrite(thisPin, LOW);  // зажечь следующ. св.диод
   | }
```

В **строке 1** к тексту программы присоединяется библиотека *MsTimer2*.

В **строках 2...5** определяются все необходимые константы программы. Константы *buttonPin* и *LoopTime* нам хорошо знакомы. Так как в данной программе значения начала и конца цикла «пробега огней» (*loopStart* и *loopStop*) не изменяются, они оформлены в виде констант, а не переменных.

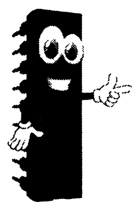
В **строке 6** определяется переменная *thisPin*, которая указывает на текущий обрабатываемый выход светодиода. Изменение значения этой переменной будет производиться в процедуре обработки прерывания, поэтому она объявляется как *volatile*.

Строки 8...12 функции *setup()* нам хорошо знакомы из предыдущих программ. Они настраивают все входы и выходы модуля.

В **строке 13** присваивается начальное значение указателю текущего светодиода *thisPin*. В последних строках функции *setup()* мы видим две команды, связанные с настройкой прерывания.

В **строке 14** определяются режимы прерывания. Период вызова прерывания устанавливается равным *LoopTime*. В качестве процедуры обработки прерывания указывается процедура с именем *timerInterupt*.

В **строке 15** дается старт процессу вызова прерывания.



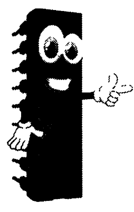
ПРИМЕЧАНИЕ.

Как видите, основной цикл нашей программы остается абсолютно свободным. Вместо строк, помеченных номером 17, можно вставить практически любую программу, выполняющую любые дополнительные функции.

Как мы и обещали, весь алгоритм «бегущих огней» выполняется в теле процедуры обработки прерывания.

Процедура расположена в конце программы и начинается в **строке 18**.

Сначала, в **строке 19** программа гасит текущий светодиод. Этот светодиод обычно горит еще с предыдущего вызова процедуры обработки прерывания. Между двумя вызовами прерывания всегда горит один из светодиодов.



ПРИМЕЧАНИЕ.

Период вызова прерываний определяет длительность горения очередного светодиода (то есть длительность одного шага «бега» «огней»).

Далее программа должна определить, какой светодиод должен загореться следующим. Определяется это так: в **строке 20** программа читает и оценивает состояние кнопки.

Если **кнопка отпущена**, перемещаем «огонь» на шаг в прямом направлении (**строки 21 и 22**).

Если **кнопка нажата**, делаем один шаг назад (**строки 24 и 25**).

Шаг вперед и шаг назад очень похожи. Для продвижения в прямом направлении сначала в **строке 21** производится инкремент указателя текущего светодиода *thisPin*.

В **строке 22** производится оценка нового значения указателя. Если это значение превысит верхний предел, то указателю присваивается значение нижнего предела («движение» «огня» начинается сначала). При обратном шаге производятся обратные действия.

В **строке 24** производится декремент указателя светодиода.

В **строке 25** проверяется, не оказался ли указатель ниже нижнего предела. Если оказался, то ему присваивается значение верхнего предела. В результате номер нового текущего светодиода будет определен.

Затем в **строке 26** новый текущий светодиод зажигается. И на этом процедура обработки прерывания заканчивается.

Микроконтроллер возвращается к выполнению основной программы, а текущий светодиод продолжает гореть до следующего вызова прерывания.

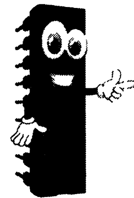
Совместное использование таймера 0

На сайте «Роботоша» (robotosha.ru) я нашел очень остроумное решение, как организовать прерывание по таймеру для своих целей и при этом не нарушить ни одной из стандартных функций языка Ардуино, связанных с использованием таймеров. Фокус состоит в том, что все настройки всех трех таймеров,

устанавливаемые транслятором среды разработки, остаются в неприкосновенности.

Но при этом активизируется еще один вид прерываний, который язык Ардуино не использует. Этот вид прерываний, который предлагается использовать называется **«прерывание по совпадению»**.

Прерывание по совпадению также связано с таймером. Например, таймер 0 поддерживает два прерывания по совпадению. Для этого в составе таймера имеется специальный регистр, который называется регистр совпадения. Программным способом в этот регистр можно записать некое число — порог совпадения.

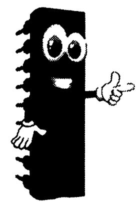


ПРИМЕЧАНИЕ.

Прерывание по совпадению возникает в тот момент, когда содержимое основного счетного регистра таймера окажется равным содержимому регистра совпадения. Меняя содержимое регистра совпадения, можно менять момент возникновения прерывания.

Идея, предложенная на сайте «Роботоша», состоит в том, чтобы двояко использовать таймер 0. Как мы уже не однократно говорили, таймер 0 в любой программе, написанной на языке Ардуино, работает в режиме генерации миллисекундных импульсов. За одну миллисекунду таймер успевает просчитать от 0 до своего максимального значения — 255 и затем вызвать прерывание по переполнению.

При помощи этого прерывания происходит подсчет системного времени. А вот прерывание по совпадению система Ардуино не использует. Это прерывание мы и будем использовать для наших нужд, не меняя никаких других параметров таймера.



ПРИМЕЧАНИЕ.

Если в регистр совпадения поместить число, примерно равное половине максимального значения счетного регистра таймера, и активизировать прерывание по совпадению, то такое прерывание будет вызываться также 1 раз каждую миллисекунду, но со сдвигом во времени от основного прерывания.

Сдвиг по времени будет очень полезен для того, чтобы процедуры обработки прерываний не мешали друг другу. У таймера 0 микроконтроллера ATmega323 имеется два регистра совпадения. Имя первого из них OCR0A. Его мы и будем использовать для вызова нашего прерывания.

Но как же быть с **периодом вызова прерывания**? Ведь нам нужно, чтобы процедура обработки прерывания в программе «бегущих огней» вызывалась один раз за 200 миллисекунд? Да очень просто!

В нашей новой программе прерывание будет вызываться каждую миллисекунду, но выполнять нужные нам действия только один раз за 200 вызовов. Как это сделать, сейчас увидим.

Для этого разберем подробно текст нового программного примера, приведенный в **листинге 10.2**.

За основу программы в **листинге 10.2** взят первый вариант программы, приведенный в **листинге 10.1**. В новом варианте реализован описанный выше способ формирования прерывания по таймеру. Нам пришлось сделать совсем небольшие изменения.

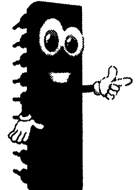
Разберем по порядку все эти изменения. Для начала из программы исключен оператор *#include*, так как нам теперь не нужна внешняя библиотека. Полностью изменены команды инициализации прерывания.

Новые команды инициализации вы можете видеть в **строках 13 и 14 листинга 10.2**.

Листинг 10.2. Использование прерывания по таймеру 0

```
   /*
     Бегущие огни. использование прерываний по таймеру 0
     (минимум вмешательства в настройки Ардуино)
   */

   // Определение констант:
1  const int buttonPin = 13;   // номер контакта кнопки
   // Опреление переменных:
2  int loopStart = 2;          // начало цикла бега
3  int loopStop  = 9;         // конец цикла бега
4  int LoopTime  = 200;  // время задержки шага бегущ. огня
5  volatile int TimeCore  = 0;    // Счетчик таймера
6  volatile int thisPin; // Текущий разряд светодиодов

7  void setup() {
     // Цикл инициализации контактов светодиодов:
8    for (int thisPin = 2; thisPin < 10; thisPin++) {
9      pinMode(thisPin, OUTPUT);
10     digitalWrite(thisPin, HIGH);
     }

     // Инициализация контакта кнопки:
11   pinMode(buttonPin, INPUT);
12   thisPin = loopStart;   // начальн. знач. указателю
   св.диода
     // Инициализация таймера (режим совпадения)
13   OCR0A = 0xA0;      // значение регистра совпадения
14   TIMSK0 |= (1 << OCIE0A);  // разрешение прерывания
   }

15 void loop() {
16   // Основной цикл остается свободным
     // Сюда вы можете поместить основную программу
     // на ее фоне будет выполняться программа бегущ. огней
   }
```

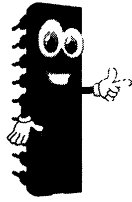
```
   | // Обработчик прерывания
17 | SIGNAL(TIMER0_COMPA_vect) {
18 |   TimeCore++;
19 |   if (TimeCore>LoopTime) {
20 |     TimeCore=0;
21 |     digitalWrite(thisPin, HIGH); // погасить текущ. св.диод
22 |     if (digitalRead(buttonPin)==HIGH) {
   |       // Один шаг бегущ. огня в прямом направлении:
23 |       thisPin++;
24 |       if (thisPin > loopStop) thisPin = loopStart;
25 |       digitalWrite(thisPin, LOW); // зажечь следующ. св.диод
   |     }
26 |     else {
   |       // Один шаг бегущ. огня в обратном направлении:
27 |       thisPin--;
28 |       if (thisPin < loopStart) thisPin = loopStop;
29 |       digitalWrite(thisPin, LOW);
   |                                 // Зажечь следующий светодиод
   |     }
   |   }
   | }
```



ПРИМЕЧАНИЕ.

Так как для инициализации прерывания мы используем нестандартный прием, приходится программировать на уровне регистров.

Так, в **строке 13** записывается значение в регистр совпадения OCR0A. Как уже говорилось, это значение находится где-то посредине между минимальным и максимальным значениями таймера.



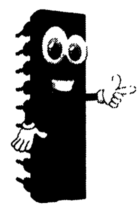
ПРИМЕЧАНИЕ.

__Автор идеи__ предложил величину __0xAF__ (записано в шестнадцатеричном формате). В десятичном формате 0xAF соответствует 175. __Я выбрал величину поближе к середине диапазона: 0xA0 (то есть 160)__.

При выборе значения, записываемого в регистр совпадения, разумнее всего учитывать соотношение времени выполнения обеих процедур обработки прерывания (по переполнению и по совпадению).

В **строке 14** определяется значение разряд OCIE0A в регистре маски TIMSK0. Регистр маски TIMSK0 микроконтроллера управляет включением и выключением всех видов поддерживаемых им прерываний по таймеру 0. Для каждого таймера есть свой регистр маски прерываний. Каждый разряд регистра включает или выключает (маскирует) свой вид прерывания.

Если какой-либо разряд равен нулю, соответствующее ему прерывание не разрешено. Чтобы разрешить прерывание, нужно установить соответствующий разряд в единицу. Каждый разряд регистра имеет свое собственное имя.



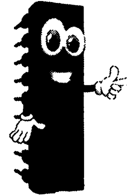
ЧТО ЕСТЬ ЧТО.

Разряд, который отвечает за прерывание по совпадению, __называется OCIE0A__. Выражение в строке 14 программы устанавливает в единицу разряд с номером OCIE0A.

Если вы желаете досконально изучить синтаксис логических выражений языка СИ, рекомендую обратиться к книге [1].

Но если говорить кратко, то выражение: TIMSK0 |= (1<<OCIE0A) выполняет два действия. Составной оператор «|=» выполняет операцию побитового ИЛИ между значением переменной TIMSK0 и результатом выражения в скобках (1<<OCIE0A), а результат этой операции присваивается переменной TIMSK0.

Операция побитового ИЛИ используется во многих языках программирования для установки в единицу одного или нескольких двоичных разрядов числа, оставляя при этом все остальные двоичные разряды без изменений.

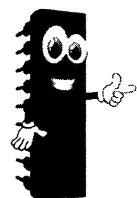


ЧТО ЕСТЬ ЧТО.

Значение в правой части от оператора «|=» ***называется маской****. Маска должна иметь единицы во всех двоичных разрядах, которые в исходном числе (в данном случае – в TIMSK0) необходимо установить в единицу. А те разряды, которые изменять в исходном числе не нужно, в маске должны быть равны нулю.*

Смысл выражения в скобках — получить маску с единицей в разряде номер OCIE0A. Для получения такой маски берется число 1 (в двоичном представлении это число, у которого во всех старших разрядах нули и единица в младшем разряде — 00000001). Двоичные разряды этого числа сдвигаются побитно влево. Число сдвигов равно OCIE0A.

Следующее изменение программы, приведенной в **листинге 10.2** по сравнению с программой из **листинга 10.1**, — это имя функции обработки прерывания.



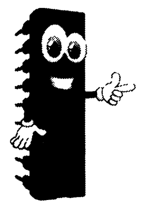
ПРИМЕЧАНИЕ.

Как вы могли заметить, при инициализации прерывания мы не определяли имени этой функции.

По законам программирования на уровне регистров контроллера у такой процедуры имеется **фиксированное системное имя**. Все функции обработки прерываний всегда имеют имя SIGNAL(), а параметр функции — это системное имя выбранного нами прерывания. В нашем случае имя прерывания TIMER0_COMPA_vect.

Достаточно создать в своей программе функцию с таким именем и таким параметром, и вектор прерывания окажется определен. В **листинге 10.2** описание этой функции начинается в **строке 17**. Тело функции расположено в **строках 18...29**.

В **строке 18** производится инкрементация значения переменной *TimeCore*. Эта переменная используется в качестве счетчика миллисекунд.



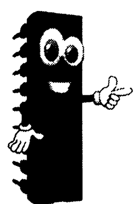
ПРИМЕЧАНИЕ.

Как уже говорилось, процедура SIGNAL() будет вызываться каждую миллисекунду, и каждый раз значение счетчика будет увеличиваться на единицу.

В **строке 19** оператор *if* сравнивает значение счетчика *TimeCore* с заданным в начале программы периодом шага «бегущих огней» — *LoopTime*, которое как раз и равно 200.

Операторы в фигурных скобках (**строки 20...29**) выполняются только тогда, когда значение счетчика миллисекунд достигнет значения *LoopTime*. Пока значение не достигнуто, никаких действий не выполняется, и процедура обработки прерывания заканчивается без последствий.

Когда значение счетчика миллисекунд *TimeCore* достигнет нужного значения (окажется равным *LoopTime*), первой же командой (**строка 20**) счетчик *TimeCore* обнуляется и выполняются команды, реализующие один шаг «бега огней» (**строки 21...29**).



ПРИМЕЧАНИЕ.

Эти команды полностью аналогичны таким же командам из ***листинга 10.1****.*

После этого процедура обработки прерываний также завершается. Следующий раз при вызове процедуры обработки прерывания подсчет миллисекунд начнется сначала. Таким образом, основная часть процедуры обработки прерывания будет выполняться один раз за время, заданное в переменной *LoopTime*, точно так же как это происходило в первом варианте программы (**листинг 10.1**).

ГЛАВА 11

ФОРМИРОВАНИЕ ЗВУКА

Постановка задачи

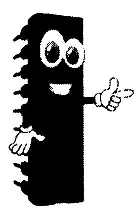
В языке Ардуино извлечение звука — достаточно несложная задача. Для этого используется простая и удобная функция *tone()*. Формат этой функции следующий:

tone(Контакт, ВысотаТона, Длительность);

Параметр *«Контакт»* — это номер контакта модуля Ардуино, на котором будет генерироваться звуковой сигнал.

Параметр *«ВысотаТона»* — это частота генерируемого сигнала в герцах. Тип значения этого параметра — *unsigned int*.

Параметр *«Длительность»* — не обязательный. Если он есть, то определяет длительность генерируемого звукового сигнала в миллисекундах. Тип значения — *unsigned long*.

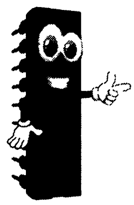


ПРИМЕЧАНИЕ.

Если параметр «Длительность» отсутствует, то звуковой сигнал генерируется непрерывно до тех пор, пока не придет команда noTone().

По **команде** ***tone()*** на выбранном контакте вырабатывается цифровой сигнал, представляющий собой меандр (т. е. сигнал у которого длительность импульса равна половине периода).

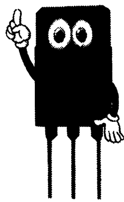
Для формирования сигнала используется **таймер 2** микроконтроллера. В каждый момент времени может генерироваться только один звуковой сигнал заданной частоты только на одном из контактов.



ПРИМЕЧАНИЕ.

Если сигнал уже генерируется на каком-либо контакте, то повторное использование функции tone() для этого контакта просто приведет к изменению частоты тона (если конечно параметр «ВысотаТона» новой команды изменился).

В то же время вызов функции *tone()* для любого другого контакта не будет иметь никакого эффекта. Сначала нужно выключить звук командой *noTone()*, и только после этого можно применить *tone()* для нового контакта.



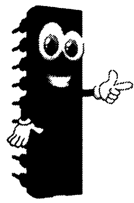
ЗАДАЧА.

Разработать схему и программу устройства на основе модуля Ардуино, позволяющего издавать различные звуки. При нажатии на одну из десяти кнопок, подключенных к этому устройству на выходе, к которому подключен звукоизлучатель, должен генерироваться звуковой сигнал. Для разных кнопок звук должен иметь разную высоту тона. Когда все кнопки отпущены, звук должен прекратиться.

Схема

Вариант схемы для реализации выше описанной задачи приведен на ***рис. 11.1****.*

На этой схеме впервые задействованы цифровые контакты 0 и 1 модуля Ардуино.



ПРИМЕЧАНИЕ.

Во всех предыдущих схемах мы не использовали цифровые контакты 0 и 1, чтобы не мешать процессу обмена информацией по последовательному каналу между модулем и компьютером. Но с увеличением количества контактов, необходимых для решения поставленных задач, нам все равно придеться когда-то их использовать.

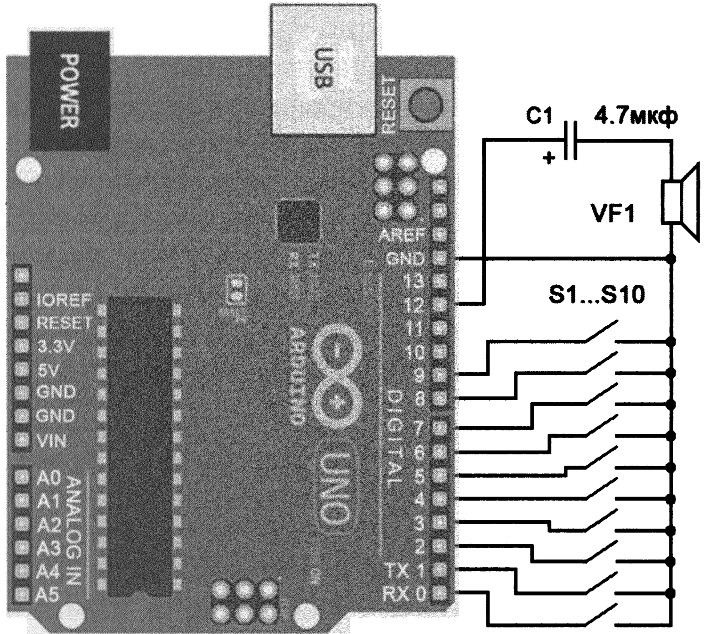
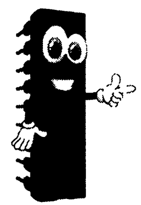


Рис. 11.1. *Схема устройства генерации звуковых сигналов*

Рис. 11.2. *Внешний вид устройства генерации звуковых сигналов*

Использование этих контактов для подключения кнопок — одно из самых безопасных решений. Если кнопка не нажата, то она никак не влияет на подключенные к ней цепи.



ВНИМАНИЕ.

Главное помнить *— в момент обмена данными с компьютером ни в коем случае нельзя нажимать на кнопки, подключенные к контактам 0 и 1.*

В качестве звукоизлучателя в схеме используется любой маломощный динамик. Он прекрасно работает, если его подключить через конденсатор емкостью 3...5 микрофарад. **Вариант готовой конструкции**, собранной на универсальной монтажной плате, приведен на **рис. 11.2**.

Алгоритм

1. Организовать цикл опроса кнопок. В цикле кнопки должны опрашиваться по возрастанию, начиная с кнопки номер 0 и заканчивая кнопкой номер 9.
2. При обнаружении в процессе опроса первой же нажатой кнопки программа должна прекратить опрос и запустить процесс генерации звука на выходе, куда подключен звукоизлучатель. Высота тона должна быть взята из таблицы. Для каждой кнопки своя нота. При этом перебор кнопок должен быть начат сначала (с кнопки номер 0).
3. Если в результате сканирования всех десяти кнопок ни одна из них не оказалась нажатой, программа должна принудительно прекратить генерацию звука.
4. В любом случае (нажата кнопка или нет) после завершения операций, описанных в пунктах 2 или 3, управление передается к пункту 1 данного алгоритма.

В результате выполнения описанного выше алгоритма программа будет постоянно опрашивать все кнопки, при обнаружении нажатия начнется генерация звука, а опрос кнопок начнется сначала. Если в процессе очередного перебора программа обнаружит нажатие той же кнопки, какая была нажата при предыдущем переборе, то новая команда вызова звукового сигнала только подтвердит прежний режим генерации звука.

Если номер нажатой кнопки окажется другим, то тон звукового сигнала изменится. Если же после перебора всех кнопок окажется, что ни одна кнопка не нажата, то генерация звука прекратится.

Программа

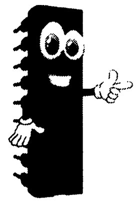
Вариант программы простейшего звукового устройства, реализующий описанный выше алгоритм, приведен в

листинге 11.1. Далее мы будем детально изучать используемые в программе приемы программирования, рассматривая ее построчно.

В **строке 1** определяется константа *SoundPin* — номер контакта, к которому подключается звукоизлучатель.

В **строке 2** определяется переменная *ButtonPin* (номер текущей кнопки). При помощи этой переменной программа будет осуществлять перебор кнопок.

Интерес для нас представляет **строка 3** программы. В этой строке описывается массив.



ЧТО ЕСТЬ ЧТО.

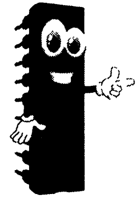
***Массив** – это набор однотипных переменных или констант используемых как единый набор данных. Каждый массив имеет свое имя.*

В данном случае имя массива *tabFreq* (массив хранит таблицу частот). В квадратных скобках указывается размер массива. В нашем случае размер массива равен 10. Это значит, что массив может хранить десять разных значений.

Все значения нашего конкретного массива присваиваются тут же, в **строке 3**. Значения перечисляются в фигурных скобках, следующих сразу за определением массива после символа «=» (равно). Значения в списке разделяются запятой.

Обращение в программе к любому значению из массива происходит по его номеру в формате

ИмяМассива[НомерЭлементаМассива].



ПРИМЕЧАНИЕ.

*Элементы массива **нумеруются**, начиная с элемента номер ноль. Это значит, что массив размерностью 10 состоит из элементов с номерами от 0 до 9.*

Листинг 11.1. Программа «Десять кнопок — десять нот»

```
   /*
     Сигнализатор звука «десять нот»
   */

   // Определение констант:
1  const int SoundPin = 12; // номер контакта звукоизлучателя

   // Определение переменных:
2  int ButtonPin;  // указатель номера контакта текущ. кнопки

   // Определение массива констант (частоты нот)
3  const int tabFreq[10] = { 1046, 987, 932, 880, 831, 784,
                             740, 698, 659, 622 };

4  void setup() {
     // инициализируем контакт звука:
5    pinMode(SoundPin, OUTPUT);
     // инициализируем контакты кнопок:
6    for (ButtonPin=0; ButtonPin<=9; ButtonPin++) {
7      pinMode(ButtonPin, INPUT_PULLUP);
     }
   }

8  void loop() {
9    boolean btFree = true; // сброс флага «кнопки отпущены»
     // Сканирование кнопок и воспроизведение звука:
10   for (ButtonPin=0; ButtonPin<=9; ButtonPin++) {
11     if (digitalRead(ButtonPin)==LOW) {
12       tone (SoundPin, tabFreq[ButtonPin]);
13       btFree = false;
14       break;
       }
     }
15   if (btFree) noTone(SoundPin);
   }
```

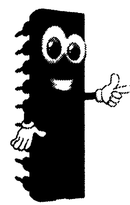
После блока команд определения констант и переменных начинается функция начальных установок *setup()*. В теле функции *setup()* производится настройка режимов всех контактов модуля.

В **строке 5** контакт звукоизлучателя переводится в режим OUTPUT.

В **строках 6 и 7** мы видим цикл, который перебирает все входы от кнопок с 0 по 9 и каждый из них переводит в режим INPUT_PULLUP (режим входа с включенным подтягивающим резистором).

Далее в программе начинается ее **основной цикл *loop()***. В теле функции *loop()* реализуется основной алгоритм программы.

В **строке 9** определяется локальная переменная *btFree*.



ЧТО ЕСТЬ ЧТО.

Локальная переменная *– это переменная определенная внутри какой-либо функции. Действие локальной переменной распространяется только на ту функцию, в которой она была определена. Вне этой функции локальная переменная не существует.*

Переменная *btFree* имеет тип *boolean*, то есть принимает значения *true* или *false*. В данной программе эта переменная служит флагом «Все кнопки отпущены».

В **строке 9** сразу после определения флаг устанавливается в *true*.

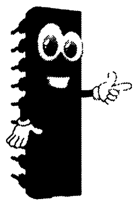
Далее в **строке 10** организуется цикл, который перебирает по очереди входы кнопок и проверяет, не нажата ли одна из них. Как только программа обнаружит нажатую кнопку, флаг сбрасывается в *false* (что сигнализирует в дальнейшем для программы, что не все кнопки отпущены). Если же в конце цикла перебора значение переменной *btFree* все еще равно *true*, то это значит, что ни одна кнопка не нажата, и нужно выключать звук.

В качестве счетчика цикла перебора кнопок используется переменная *ButtonPin* (см. **строку 10**). Ее начальное значение

равно нулю, а цикл выполняется пока *ButtonPin* меньше или равен девяти.

В **строке 11** считывается и оценивается состояние текущей кнопки с номером *ButtonPin*. Если значение состояния кнопки равно LOW (кнопка нажата), то выполняются **строки 12, 13 и 14**.

В **строке 12** вызывается функция генерации звука на выходе *SoundPin*.



ЧТО ЕСТЬ ЧТО.

> ***Частота звука*** *— это значение элемента массива tabFreq с номером, равным значению переменной ButtonPin.*

В **строке 13** сбрасывается переменная *btFree* («Все кнопки отпущены»), потому что обнаружилась нажатая кнопка.

В **строке 14** мы видим оператор *break*. Оператор *break* досрочно завершает текущий цикл опроса, и оставшиеся кнопки не опрашиваются. Это нужно для того, что бы выполнялось правило — программа реагирует только на самую первую из нажатых кнопок.

Строка 15 программы выполняется уже после цикла перебора кнопок. Это оператор *if*, который проверяет значение флага *btFree*. Если флаг установлен (его значение равно *true*), то выполняется оператор *noTone()*, и звук прекращается. Так как переменная *btFree* имеет логический тип, то составлять из нее логическое выражение не нужно. Ее значение уже само является условием для оператора *if* (имеет значение *true* или *false*).

На этом заканчивается тело основного цикла программы, и управление передается на его начало. Многократно сканируя все кнопки, программа реагирует на их нажатие, издавая различные звуки.

ГЛАВА 12

ВВОД АНАЛОГОВОЙ ИНФОРМАЦИИ

Постановка задачи

Конструкция модуля Ардуино, а также синтаксис его языка программирования, разработаны таким образом, чтобы максимально облегчить работу с **аналоговым сигналом**.

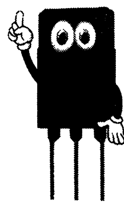
Для ввода аналогового сигнала?

Как уже говорилось выше, используются специальные аналоговые входы. Если говорить о модуле Arduino UNO, то он имеет шесть таких входов.

Для чтения уровня сигнала на любом из этих входов в языке Ардуино имеется специальный оператор *analogRead*(*НомерВхода*). Параметр *НомерВхода* может принимать следующие значения: A0, A1, A2, A3, A4, A5.

Функция *analogRead()* возвращает уровень сигнала на выбранном входе в относительных единицах. Допустимый диапазон напряжений на любом аналоговом входе от 0 до +5 вольт. При изменении напряжения на входе в этих пределах возвращаемое значение изменяется от 0 до 1024. То есть, если напряжение на входе равно нулю, функция возвращает 0. Если же на входе +5 вольт, то функция возвращает 1024.

Как видите, сама задача чтения аналоговых данных проста до примитивности. Поэтому рассмотрим пример, в котором эта задача будет дополнена задачей создания светодиодного индикатора уровня аналогового сигнала.



ЗАДАЧА.

Создать на основе модуля Ардуино устройство, которое будет считывать напряжение с движка переменного резистора и наглядно отображать считанное напряжение при помощи линейного индикатора, выполненного в виде цепочки светодиодов. Светодиоды должны загораться в виде светящегося столбика. Чем больше напряжение на входе модуля, тем больше длина столбика.

Схема

Схема устройства, позволяющая реализовать поставленную выше задачу, приведена на ***рис. 12.1.***

Переменный резистор R1 включен по схеме потенциометра между цепью питания (+5 В) и общим проводом. Напряжение с движка потенциометра поступает на вход А0 модуля. Светодиоды индикатора подключаются к контактам 2...9 точно

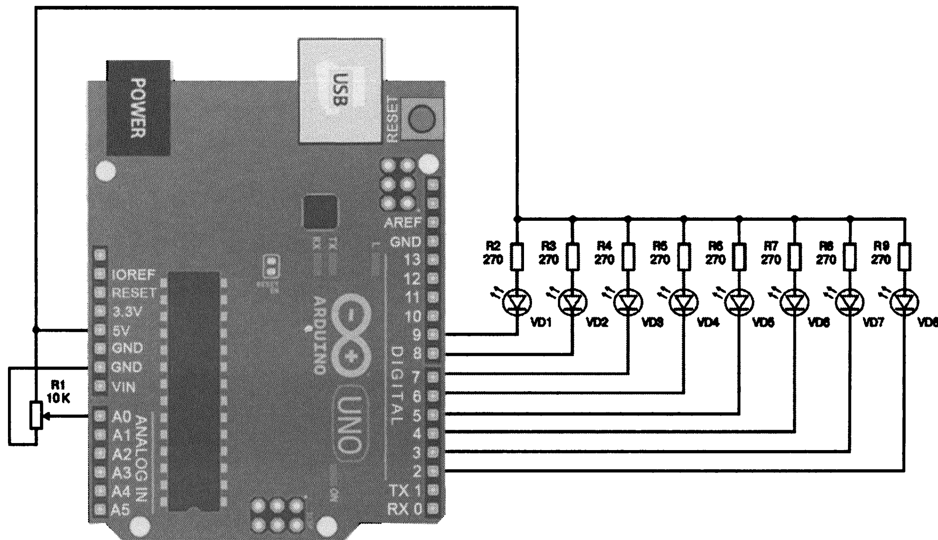


Рис. 12.1. *Схема устройства чтения и индикации аналогового сигнала*

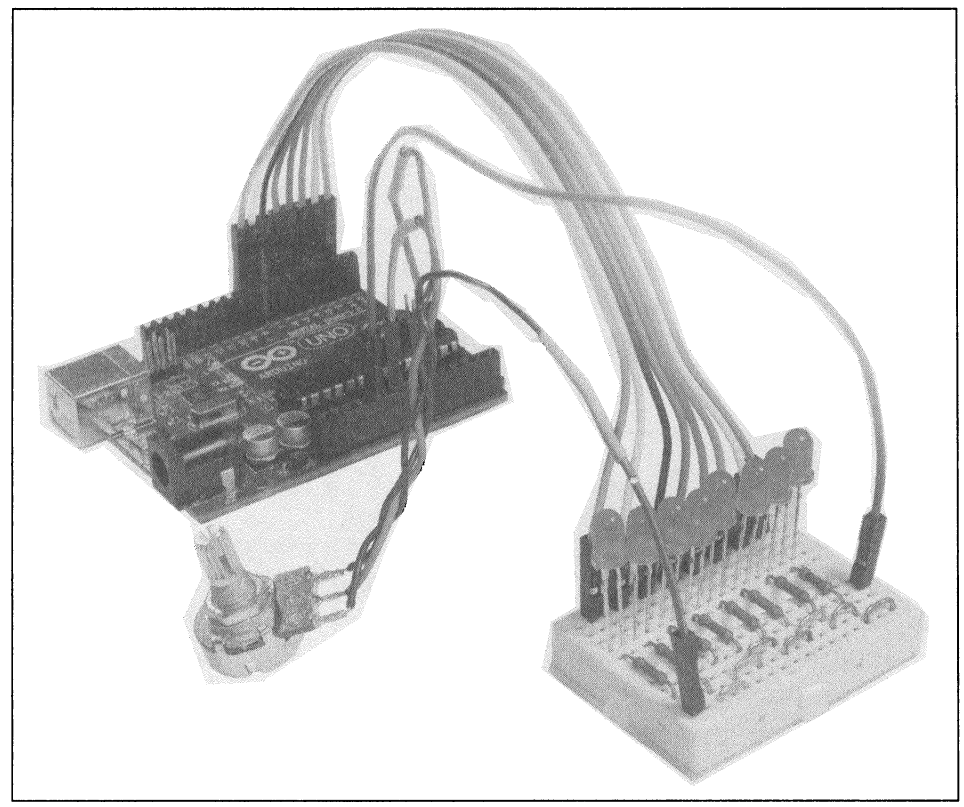


Рис. 12.2. Внешний вид устройства чтения и индикации аналогового сигнала

так же, как в схеме бегущих огней. Внешний вид уже собранного устройства показан на **рис. 12.2**.

Алгоритм

Программа должна представлять бесконечный цикл, в котором будет выполняться следующая последовательность действий.

1. Прочитать напряжение с аналогового входа. Далее мы будем оперировать понятием «считанное значение».

2. Подготовить программу к выполнению цикла вывода считанного значения на индикатор. Далее в цикле будет использоваться понятие «плавающего порога».
3. Присвоить плавающему порогу начальное значение равное нулю.
4. В цикле по очереди перебирать все выходы на светодиоды.
5. С каждым шагом перебора увеличивать значение плавающего порога на значение, которое мы назовем «шаг порога». Таким образом, с каждым шагом «плавающий порог» будет расти. Шаг порога должен быть выбран таким образом, чтобы за 8 шагов цикла порог изменился бы от 0 до 1024 (что перекроет весь диапазон считанного значения сигнала).
6. На каждом шаге сравнить считанное значение со значением плавающего порога.
7. Если считанное значение больше плавающего порога, текущий светодиод зажечь.
8. В противном случае светодиод потушить.
9. Продолжить перебор до тех пор, пока не будут перебраны все восемь светодиодов.
10. После перебора всех восьми светодиодов перейти к пункту 1 данного алгоритма.

В результате выполнения описанного выше алгоритма плавающий порог за восемь шагов перебора светодиодов изменится от нуля до максимума. Вначале перебора порог будет ниже считанного значения, и светодиоды на каждом шаге будут зажигаться. В какой-то момент плавающий порог превысит считанное значение.

С этого момента на каждом шаге перебора программа будет тушить текущий светодиод. Чем выше считанное с аналогового входа напряжение, тем больше светодиодов загорится. Напряжение на аналоговом входе вы можете менять, поворачивая вал переменного резистора. При этом длина столбика светящихся светодиодов будет меняться от минимума, когда ни один светодиод не горит, и до максимума, когда горят все светодиоды.

Программа

Программа, реализующая описанный выше алгоритм, приведена в **листинге 12.1**. Начинается программа с определения всех необходимых констант.

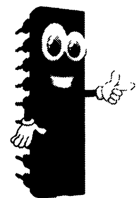
В **строке 1** определяется имя аналогового входа, к которому подключен движок потенциометра.

В **строке 2** мы определяем величину шага изменения плавающего порога. Так как за восемь шагов порог должен измениться от 0 до 1024, то шаг порога вычисляется так:

1024/8=128.

В **строках 3 и 4** определяются константы максимального и минимального номера контакта в группе контактов для подключения светодиодов. Эти константы используются для организации цикла перебора светодиодов.

В теле функции *setup()* находится цикл, определяющий режимы работы всех выводов для подключения светодиодов.



ПРИМЕЧАНИЕ.

Все контакты настраиваются на вывод информации, и на каждом выходе устанавливается высокий логический уровень. То есть, все светодиоды в начале работы программы оказываются выключенными.

Основной цикл программы (функция *loop()*) выполняет периодическое считывание сигнала с аналогового входа и вывод считанного значения на светодиодный индикатор.

Для начала в **строке 11** обнуляется значение плавающего порога *iVol*.

В **строке 12** программа считывает уровень входного сигнала с аналогового входа *analogPin* и присваивает считанное значение переменной *AnalogVol*.

Листинг 12.1. Программа ввода и индикации аналогового сигнала

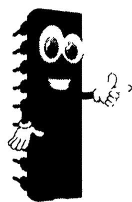
```
   /*
    Ввод и индикация аналогового сигнала
   */

   // Определение констант:
1  const int analogPin = A0;   // имя аналогового входа
2  const int StepVol = 128;    // шаг разрешения индикатора
3  const int loopStart = 2;    // начало цикла бега
4  const int loopStop  = 9;    // конец цикла бега
   // Определения переменных:
5  int thisPin;   // текущий разряд светодиодов

6  void setup() {
     // Цикл инициализации контактов светодиодов:
7    for (thisPin = loopStart; thisPin <= loopStop;
   thisPin++) {
8      pinMode(thisPin, OUTPUT);
9      digitalWrite(thisPin, HIGH);
     }
   }

10 void loop() {
11   int iVol = 0;
12   int AnalogVol = analogRead(analogPin);
13   for (thisPin = loopStart; thisPin <= loopStop;
   thisPin++) {
14     if (iVol<AnalogVol) digitalWrite(thisPin,LOW);
15     else digitalWrite(thisPin,HIGH);
16     iVol += StepVol;
     }
   }
```

В **строках 13...16** находится цикл вывода полученного значения на светодиодный индикатор.

В **строке 13** этот цикл объявляется. В качестве счетчика цикла используется переменная *thisPin*. В процессе работы

цикла значение этой переменной изменяется от *loopStart* до *loopStop*, перебирая, таким образом, все контакты, к которым подключены светодиоды.

В теле цикла происходит следующее. В **строке 14** идет сравнение считанного аналогового значения (переменная *AnalogVol*) и текущего значения плавающего порога (переменная *iVol*).



ПРИМЕЧАНИЕ.

*Если значение **iVol** окажется меньше чем значение **AnalogVol**, то на текущий светодиод (контакт **thisPin**) подается низкий логический уровень (LOW) и соответствующий светодиод загорается.*

В противном случае выполняется оператор, следующий за управляющим словом *else* (**строка 15**). Этот оператор подает на контакт *thisPin* высокий логический уровень (HIGH), и соответствующий светодиод гаснет.

В **строке 16** значение плавающего порога *iVol* увеличивается на величину шага порога *StepVol*. Теперь при следующей итерации цикла считанное значение будет уже сравниваться с новым значением плавающего порога.

Итак, **в этой главе** мы научились вводить значение аналогового сигнала. Дополнительно мы узнали, как простыми программными средствами можно индицировать уровень аналогового сигнала при помощи команд цифрового вывода.

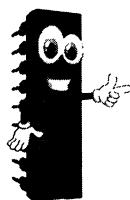
В **следующей главе** мы узнаем, как в модуле Ардуино реализован удобный способ аналогового вывода.

ГЛАВА 13

ВЫВОД АНАЛОГОВОЙ ИНФОРМАЦИИ

Широтно-импульсная модуляция

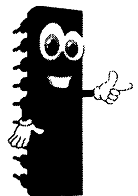
Язык программирования Ардуино имеет команду вывода аналогового сигнала *analogWrite()*, очень похожую по синтаксису на команду цифрового вывода *digitalWrite()*. Однако, как уже говорилось, при аналоговом выводе на выход поступает сигнал, представляющий из себя прямоугольные импульсы логических уровней (сам импульс имеет высокий логический уровень, в паузе — логический ноль). Частота этих импульсов примерно равна 490 Гц.



ЧТО ЕСТЬ ЧТО.

Аналоговая составляющая заложена в ширине импульсов, а вернее в соотношении импульса и паузы. Такой метод, как отмечалось ранее, называется ***«Широтно-импульсной модуляцией»*** *или ШИМ. В английской транскрипции это звучит так: «Pulse-width modulation» (PWM).*

Функция *analogWrite()* имеет всего один параметр — число в диапазоне от 0 до 255. Чем больше значение параметра, тем шире импульс сигнала ШИМ. При значении, равном 255, импульсы сливаются в постоянный единичный логический уровень. При уменьшении значения входного параметра ширина импульсов уменьшается. Когда параметр окажется равным нулю, импульсы прекращаются, и на выходе устанавливается постоянный уровень логического нуля.

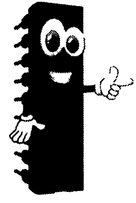


ПРИМЕЧАНИЕ.

Такой способ формирования аналогового уровня при определенных условиях вполне может заменить ***обычный способ формирования аналогового напряжения****. Например, любые инерционные приборы (лампочки, моторчики, светодиоды, нагревательные элементы), будучи подключенными через силовой ключ к такому выходу, прекрасно работают от сигнала с ШИМ. Даже лучше, чем с выходом, на котором плавно меняется аналоговое напряжение.*

Дело в том, что ШИМ работает в ключевом режиме, и силовой ключ, через который на нагрузку подается основная мощность, в таком случае имеет максимальный КПД. Это происходит потому, что в процессе работы силовой ключ:

- либо закрыт, и ток через него равен нулю;
- либо полностью открыт, и падение напряжения на ключе минимально.



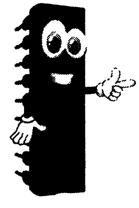
ПРИМЕЧАНИЕ.

В обоих случаях рассеиваемая на ключе мощность незначительна. А это значит, что такая схема не потребует громоздкого радиатора, а может и вообще может обойтись без какого либо охлаждения. При этом, чем шире импульсы, тем сильнее

светится лампочка, больше греется нагреватель, быстрее крутится моторчик (это должен быть мотор постоянного тока).

Если же подключаемая нагрузка не предназначена для питания импульсным сигналом, то импульсный сигнал с выхода ШИМ можно пропустить через интегрирующий фильтр, который **превратит импульсный сигнал в аналоговый**. Уровень такого сигнала тоже будет зависеть от ширины импульсов.

Для вывода сигнала в режиме ШИМ используются аппаратные возможности основного микроконтроллера модуля Ардуино. В связи с этим при использовании команды *analogWrite()* имеется ряд ограничений.



ПРИМЕЧАНИЕ.

Следует отметить, *что аналоговый вывод происходит не на аналоговые, а на цифровые контакты микроконтроллера. Причем не на все, а только на некоторые.*

На плате Ардуино контакты, которые поддерживают вывод сигнала в режиме ШИМ, обозначены символом «~».

Перед использованием контакта для аналогового вывода не требуется настраивать режим его работы, как это было необходимо сделать прежде, чем производить ввод или вывод цифровых сигналов при помощи тех же самых контактов.

Команда *analogWrite()* сама автоматически настроит контакт в режим ШИМ, и на контакте появятся импульсы заданной длительности. Эти импульсы будут присутствовать на выходе до тех пор, пока:

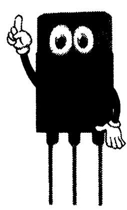
- не поступит другая команда *analogWrite()*;
- на тот же контакт не будет произведен вывод цифрового сигнала при помощи команды *diginalWrite()*.

В модуле Arduino UNO аналоговый вывод возможен на **контакты 3, 5, 6, 9, 10 и 11.**

Простейший способ аналогового вывода

Самый простой способ использования аналогового вывода — создать программу, аналогичную простейшей программе цифрового ввода и вывода, приведенной в **главе 4** нашей книги (см. **листинг 4.1**).

Достаточно просто заменить команды *digitalRead()* и *digitalWrite()* в той первой программе на команды *analogRead()* и *analogWrite()*. А так же следует выбрать другие входной и выходной контакты. Новая программа будет считывать уровень напряжения с аналогового входа и передавать его на аналоговый выход.



ЗАДАЧА.

Создать схему и программу устройства, которое будет считывать напряжение с переменного резистора, включенного по схеме потенциометра, и передавать считанный уровень командой аналогового вывода на светодиод. При нулевом напряжении на входе светодиод должен полностью погаснуть. При максимальном напряжении на входе — светодиод должен гореть в полную силу.

Схема

Схема устройства приведена на ***рис. 13.1****.*

В качестве аналогового входа используется вход A0, а в качестве выхода — цифровой контакт 9.

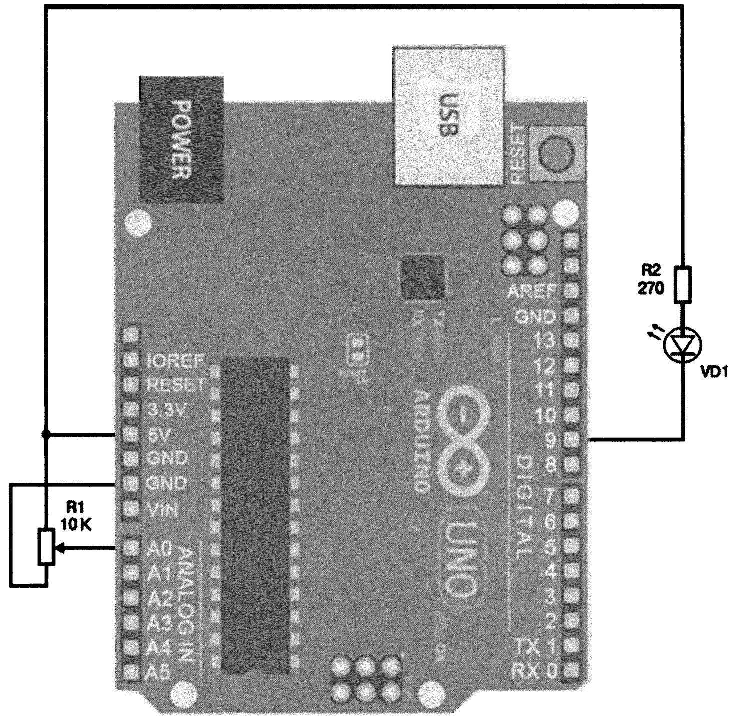


Рис. 13.1. Схема простейшего устройства аналогового ввода-вывода

Алгоритм

В основном цикле программы многократно выполнять следующие действия:

1. Прочитать аналоговый уровень с входа.
2. Привести значение считанного аналогового уровня в диапазоне команды аналогового вывода, учитывая, что диапазон аналогового ввода 0...1024, диапазон аналогового вывода 0...255.
3. Вывести приведенный уровень аналогового сигнала на аналоговый выход.
4. Перейти к пункту 1 данного алгоритма.

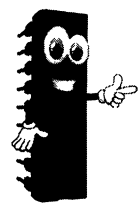
Листинг 13.1. Простейшая программа ввода и вывода аналогового сигнала

```
  /*
    Простейшая программа аналогового ввода и вывода
  */

1 const int analogPin = A0;  // номер аналогового входа
2 const int ledPin = 9;      // номер контакта светодиода

3 int analogVol;            // текущий аналоговый уровень

4 void setup() {}

5 void loop() {
    // Читаем значение сигнала с аналогового входа:
6   analogVol = analogRead(analogPin);
    // Выводим аналоговое значение на выход светодиода
7   analogWrite(ledPin, analogVol/4);
  }
```

Программа

Простейшая программа аналогового ввода-вывода приведена в **листинге 13.1**. Как видите, эта программа даже проще своего цифрового аналога.

В **строках 1...3** уже знакомым вам образом определяются две константы и одна глобальная переменная. Константы содержат номера контактов аналогового входа (*analogPin*) и аналогового выхода (*ledPin*).



ПРИМЕЧАНИЕ.

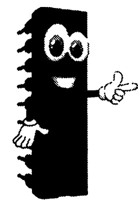
Переменная analogVol предназначена для временного хранения считанного аналогового значения. Так как режимы аналоговых входа и выхода настраивать не нужно, функция setup() ***(строка 3)*** *не содержит никаких команд.*

В **строке 4** программа читает уровень сигнала на входе *analogPin* и помещает его значение в переменную *analogVol*.

В **строке 7** считанное значение делится на 4 (приведение диапазонов) и результат от деления записывается на выход *ledPin*.

Более сложный пример аналоговой индикации

В качестве более сложного примера использования аналогового выхода предлагаю взять за основу схему из предыдущего раздела (см. **рис. 13.1**) и усовершенствовать ее работу, применив плавное зажигание и потухание светодиодов.



ПРИМЕЧАНИЕ.

Правда, для этого нам придется сократить количество светодиодов до шести и подключить их только к тем выходам, которые работают с ШИМ.

ЗАДАЧА.

Создать схему и программу устройства, считывающего напряжение с движка переменного резистора, включенного по схеме потенциометра и выводящего считанное аналоговое значение на светодиодный индикатор, состоящий из 6 светодиодов. Индикатор должен выводить уровень сигнала в виде светящегося столбика разной длины. Чем больше напряжение на входе, тем больше длина светящегося столбика светодиодов. Если напряжение на входе равно нулю, все светодиоды должны потухнуть. При максимальном напряжении на входе все светодиоды должны гореть. Отображая все изменения длины светящегося столбика, светодиоды должны загораться и потухать, плавно изменяя свою яркость, используя для этого аналоговый вывод.

Схема

Схема устройства, реализующая поставленную задачу, изображена на ***рис. 13.2.***

Алгоритм

Многократно повторять основной цикл, в котором программа должна выполнять два основных действия:

1. считать уровень входного сигнала на аналоговом входе;
2. выполнить цикл вывода считанного уровня на индикатор, состоящий из шести светодиодов.

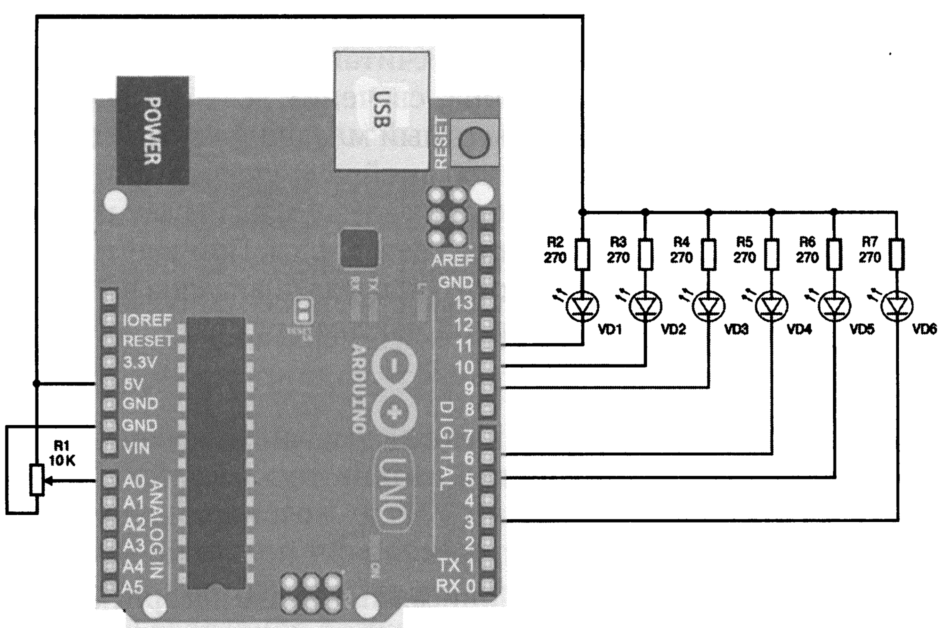


Рис. 13.2. *Схема усложненного устройства аналогового ввода-вывода*

В свою очередь, алгоритм работы светодиодного индикатора должен выглядеть следующим образом.

1. Для вывода считанного уровня аналогового сигнала должен производиться перебор всех шести светодиодов индикатора, по порядку с первого по шестой.
2. Для каждого светодиода определяется яркость его свечения. Для этого используется специальное вспомогательное значение — **плавающий порог**. В процессе перебора светодиодов плавающий порог должен изменяться от нуля до максимального значения.
3. На каждом шаге перебора светодиодов считанное значение сравнивается со значением плавающего порога. Разность между этими двумя значениями корректируется (масштабируется, смещается по уровню) и выводится на текущий светодиод командой аналогового вывода. Корректировка уровня, выводимого на светодиоды должна обеспечивать такую работу всего индикатора, чтобы с увеличением входного напряжения от нуля до максимума плавно должны

зажигаться сначала младший светодиод, затем по очереди более старшие. Когда входное считанное напряжение окажется в середине диапазона, свечение всех светодиодов должно быть таково, чтобы самый младший светодиод светился максимально. При дальнейшем увеличении входного напряжения по очереди к своему максимуму должны приближаться более старшие светодиоды. При достижении входного напряжения верхнего предела диапазона все шесть светодиодов должны гореть по максимуму.

Программа

Программа, реализующая описанный выше алгоритм, приведена в **листинге 13.2**. Несмотря на более сложный алгоритм работы, программа также получилась достаточно простая.

В **строке 1** определяется константа, означающая имя используемого в данной программе аналогового входа.

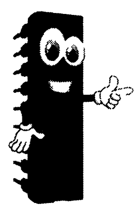
В **строке 2** определяется шаг изменения плавающего порога. Значение шага определяется по формуле 255/6. То есть, диапазон входного значения команды *analogWrite()* делится на количество шагов перебора светодиодов. Результат деления мы округляем до целого, для того, чтобы не усложнять задачу процессору работой с десятичными дробями

В **строке 3** определяется массив констант, представляющий собой таблицу номеров контактов, поддерживающих режим аналогового вывода. Именно к этим контактам подключены наши светодиоды. Таблица нам понадобится для того, чтобы по номеру светодиода находить номер контакта, к которому он подключен.

В **строке 4** определяется переменная, в которой будет храниться номер текущего светодиода. Как и в первой версии программы, функция *setup()* остается пустой.

В **строках 6...9** расположен основной цикл программы. Первая же команда в теле основного цикла (**строка 7**) читает

уровень входного аналогового сигнала с входа *analogPin*. Считанное значение делится на два и помещается в переменную *AnalogVol*.



ПРИМЕЧАНИЕ.

Уменьшение считанного сигнала вдвое делается для того, что бы привести его диапазон к требуемому диапазону для вывода на светодиоды.

Вы спросите: почему делим на два? Ведь, как мы говорили выше, диапазон считанного аналогового сигнала 0...1024, а диапазон выходного — 0...255. Диапазоны отличаются в 4 раза! Двойной диапазон нам нужен потому, что при повышении напряжения от нуля до половины максимального значения, младший светодиод должен плавно изменить яркость от полностью потушенного до полностью горящего. При этом самый старший светодиод должен только-только загореться. Еще полдиапазона нам понадобится, чтобы старший светодиод плавно загорелся до максимума.

Итак, в **строке 8** начинается цикл перебора светодиодов. Переменная *thisLed* меняется от 0 до 5, перебирая все 6 светодиодов.

В **строке 9** находится оператор аналогового вывода. Он выводит на контакт текущего светодиода с номером *thisLed* значение сигнала, которое определяется при помощи специально разработанной функции приведения уровня *rangeVol()*. Номер контакта, куда подключен текущий светодиод, определяется при помощи массива *AnalogPin[]*. В качестве указателя массива подставляется номер текущего светодиода *thisLed*.

Аргументом функции *rangeVol()* выступает выражение:

*AnalogVol+StepVol*thisLed*.

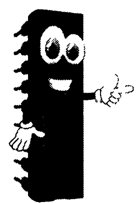
Листинг 13.2. Индикация столбиком светодиодов с плавным гашением

```
   /*
    Ввод и вывод аналогового сигнала Плавная индикация
   */

1  const int analogPin = A0;  // номер аналогового входа
2  const int StepVol = 42;    // шаг разрешения индикатора
3  const int AnalogPin[] = {3,5,6,9,10,11}; // аналог. выходы

4  int thisLed; // текущий разряд светодиодов

5  void setup() {}

6  void loop() {
7    int AnalogVol = analogRead(analogPin)/2;
8    for (thisLed = 0; thisLed < 6; thisLed++) {
9      analogWrite(AnalogPin[thisLed],
                     rangeVol(AnalogVol+StepVol*thisLed));
     }
   }

10 int rangeVol (int InpVol) {
11   int OutpVol = InpVol - 256;
12   if (OutpVol>255) OutpVol= 255;
13   if (OutpVol<0) OutpVol= 0;
14   return 255 - OutpVol;
   }
```

В этом выражении к уровню считанного с аналогового входа сигнала добавляется текущий уровень плавающего порога. В свою очередь, **уровень плавающего порога** вычисляется путем перемножения номера текущего светодиода на величину шага плавающего порога. Таким образом, на каждом шаге плавающий порог будет выше, а, значит, каждый следующий светодиод будет гореть сильнее.

Программа приведения уровней расположена в **строках 10...14**. Она выполняет следующие три важных преобразования.

Преобразование 1. *Смещение полученного входного значения вниз* для того, чтобы при нулевом уровне *AnalogVol* ни один светодиод не горел. Так как для последнего светодиода уровень плавающего порога будет равен 42×6=252, то при нулевом значении *AnalogVol* входное значение функции и будет равно 252.



ПРИМЕЧАНИЕ.

Значит, именно на эту величину нужно сместить вниз входной сигнал.

Данное действие происходит в **строке 11** программы. Только вместо 252 там используется число 256. Почему? А вспомните, как мы получили величину шага плавающего порога 42. Мы округлили до целого (отбросили дробную часть). Чтобы точнее попасть в центр нужного нам диапазона, мы должны добавить несколько единиц к величине смещения. Полученное значение помещается в переменную *OutpVol*.

Преобразование 2. *Ограничение диапазона сверху и снизу.* Все предыдущие преобразования уровней приведут к тому, что значение переменной *OutpVol* будет изменяться в диапазоне, выходящем за пределы стандартного (от 0 до 255). Поэтому сначала (в **строке 12**) производится ограничение этой величины сверху.

Оператор *if* проверяет уровень значения *OutpVol*. Если *OutpVol* выше 255, ему присваивается значение 255. Точно также в **строке 13** ограничивается уровень снизу. Если содержимое переменной *OutpVol* после предыдущих преобразований окажется меньше нуля, то ему присваивается значение ноль.

Преобразование 3. *Инвертирование выходного значения.* Наши светодиоды подключены таким образом, что загораются при уровне ноль на выходе, а гаснут при единичном уровне.

Наш же сигнал тем выше, чем выше сигнал на входе. Это значит, что при повышении сигнала на входе интенсивность свечения светодиода будет не увеличиваться, а уменьшаться.

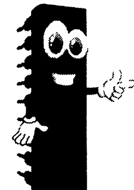
Операция инвертирования исправляет этот недостаток. Инвертирующее выражение включено в команду завершения функции.

В **строке 14** вы можете видеть команду *return* (возврат). Этот оператор завершает функцию и возвращает результат ее работы. Возвращаемое значение указывается как параметр команды *return*. В данном случае оно вычисляется при помощи выражения *255 - OutpVol*. Такое выражение переворачивает диапазон значений. При изменении значения *OutpVol* от 0 до 255, возвращаемое значение будет изменяться от 255 до нуля.

В результате работы данной программы при изменении напряжения на входе устройства, на светодиодном индикаторе отображается светящийся столбик переменной длины. Чем больше напряжение, тем длиннее столбик. Причем светодиоды, составляющие этот столбик, будут плавно загораться с каждым шагом.

При уменьшении напряжения на входе длина светящегося столбика будет уменьшаться. При этом с каждым шагом светодиоды будут тухнуть по очереди, но также плавно.

В **табл. 13.1** приведены значения, подаваемые на светодиоды при помощи команды аналогового вывода при разных значениях входного сигнала.



ПРИМЕЧАНИЕ.

*Для наглядности в **табл. 13.1** указаны неинвертированные значения сигнала (то есть вместо 255 – OutpVol указаны значения OutpVol). По сути, цифры в таблице отражают интенсивность свечения светодиодов: 0 — светодиод потушен, 255 — полностью зажжен. Не забывайте, что на самом деле в программе цифры обратные (вместо 0 подается 255, а вместо 255 подается 0).*

Значения, подаваемые на светодиоды при разных входных уровнях *Таблица 13.1*

Номер светодиода	Порог	Значение сигнала на входе										
		0	100	200	300	400	500	600	700	800	900	1000
Светодиод 1	0	0	0	0	0	0	0	44	94	144	194	244
Светодиод 2	42	0	0	0	0	0	36	86	136	186	236	255
Светодиод 3	84	0	0	0	0	28	78	128	178	228	255	255
Светодиод 4	126	0	0	0	20	70	120	170	220	255	255	255
Светодиод 5	168	0	0	12	62	112	162	212	255	255	255	255
Светодиод 6	210	0	4	54	104	154	204	254	255	255	255	255

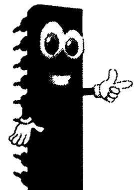
ГЛАВА 14

ПЕРЕДАЧА ДАННЫХ ИЗ АРДУИНО НА КОМПЬЮТЕР

Постановка задачи

Еще одна функция, которая легко выполняется при программировании для Ардуино по сравнению с программированием на других языках для микроконтроллеров, — это организация канала передачи данных из модуля Ардуино в компьютер по последовательному каналу.

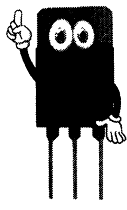
Программист легко может воспользоваться тем самым каналом, через который происходит загрузка оттранслированной программы в модуль.



ПРИМЕЧАНИЕ.

В книге [1] такой вопрос не рассматривался. Ведь задача передачи данных между компьютером и схемой на микроконтроллере предполагает, что на стороне компьютера должна работать специальная программа. Она должна принимать и хотя бы отображать то, что передаст микроконтроллер. А программы на компьютере – это отдельная большая тема, которую в рамках книги о микроконтроллерах поднимать нецелесообразно.

Среда разработчика Ардуино уже имеет свои встроенные средства, способные как принять данные от модуля Ардуино, так и передать информацию на модуль. Поэтому мы можем смело поставить следующую простую и достаточно интересную задачу.

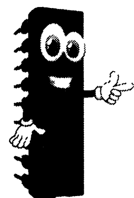


ЗАДАЧА.

Разработать схему и программу устройства, которое должно иметь десять кнопок для ввода цифр (помеченные как «1», «2», и т. д. до «9»), плюс еще одну кнопку «Ввод». При нажатии цифровых кнопок в последовательный канал должны передаваться коды соответствующих цифр, а при нажатии на кнопку «Ввод» на компьютер должна передаваться команда перевода строки. Передаваемые цифры мы будем наблюдать при помощи инструмента «Монитор порта», входящего в состав среды разработки Ардуино.

Схема

Вариант схемы, которую мы будем использовать для реализации вышеописанной задачи, приведен на ***рис. 14.1.***



ПРИМЕЧАНИЕ.

В связи с тем, что наша ***программа использует последовательный канал****, мы не стали использовать контакты 0 и 1 для подключения кнопок. Кнопки ввода цифр подключены к контактам 2...11. Кнопка «Ввод» подключена к контакту 12. Такая схема подключения клавиатуры позволяет программе перебирать все 11 кнопок одним циклом перебора.*

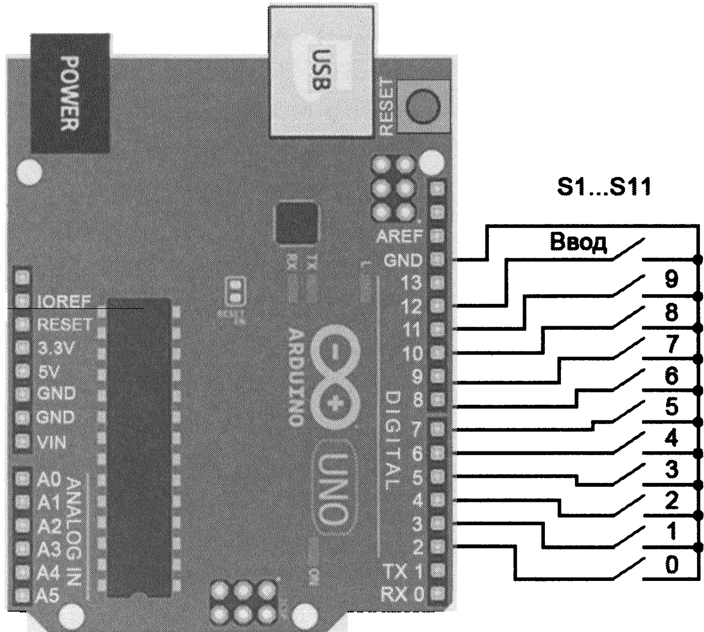


Рис. 14.1. Схема устройства для передачи цифр в последовательный порт

Перед тем, как приступить к формулированию алгоритма, следует отметить, что последовательный USB канал модуля Ардуино работает в режиме эмуляции COM порта. Как уже говорилось выше, получать и просматривать переданные из Ардуино данные мы собираемся при помощи Монитора порта, встроенного в среду разработки Ардуино. Монитор порта, как и любая программа «монитор», работает исключительно с кодами символов в ASCII кодировке.

Монитор отображает символы, коды которых поступают по последовательному каналу. Поэтому при нажатии цифровых кнопок наш модуль Ардуино должен передавать в последовательный канал ASCII коды символов цифр (см. **табл. 14.1**).

А при нажатии кнопки «Ввод» в последовательный порт должны передаться два символа:

- символ возврата каретки (код 13);
- символ перевода строки (код 10).

ASCII коды символов цифр *Таблица 14.1*

Символ	«0»	«1»	«2»	«3»	«4»	«5»	«6»	«7»	«8»	«9»	Возврат каретки	Перевод строки
Код	48	49	50	51	32	53	54	55	56	57	13	10

Алгоритм

1. Начать бесконечный цикл перебора всех 11 кнопок.
2. При обнаружении первой же нажатой кнопки перейти к процессу вывода кода, соответствующего этой кнопке через последовательный канал в компьютер. Для этого:
   2.1. выполнить задержку антидребезга контактов;
   2.2. выполнить цикл ожидания отпускания кнопки (осуществлять процесс передачи данных на компьютер лучше всего в момент отпускания кнопки);
   2.3. сформировать небольшую защитную задержку на время дребезга при отпускании кнопки, и затем передать код в последовательный порт;
      2.3.1. Если была нажата одна из цифровых кнопок, передать код кнопки.
      2.3.2. Если нажатой оказалась кнопка «Ввод», передать коды перевода строки и возврата каретки.
   2.4. сформировать защитную задержку на время передачи информации по последовательному порту;
   2.5. продолжить цикл перебора кнопок сначала, передав управление на пункт 1 данного алгоритма.

Программа

Вариант программы, реализующий описанный выше алгоритм, приведен в **листинге 14.1**. Рассмотрим программу по порядку.

Листинг 14.1. Программа «Цифры на Монитор»

```
   /*
     Передача данных на компьютер по USB каналу
   */

   // Константы
1  const int nPinMin = 2;  // Первый из контактов кнопок
2  const int nPinMax = 12; // Последний из контактов кнопок

   // Переменные:
3  byte NumberPin;  // Номер контакта текущей кнопки

4  void setup() {
     // инициализируем контакты кнопок на ввод:
5    for (NumberPin=nPinMin; NumberPin<=nPinMax;

   NumberPin++) {
6      pinMode(NumberPin, INPUT_PULLUP);
     }
7    Serial.begin(9600);  // Инициализация послед. канала
   }

8  void loop() {
     // Сканирование кнопок и воспроизведение звука:
9    for (NumberPin=nPinMin; NumberPin<=nPinMax;
   NumberPin++) {
10     if (digitalRead(NumberPin)==LOW) {
11       delay(20);
12       while (digitalRead(NumberPin)==LOW) {};
13       delay(20);
14       if (NumberPin!=12) Serial.print(NumberPin-2,DEC);
15       else  Serial.println(' ');
16       delay(20);
17       break;
       } ————————————————————
     }
   }
```

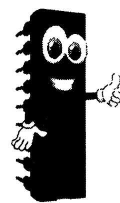
В **строках 1 и 2** происходит определение двух констант:

- номера первого контакта из группы контактов для подключения кнопок;
- номера последнего контакта из группы контактов для подключения кнопок.

В **строке 3** определяется переменная *NumberPin*, в которой мы будем хранить указатель на текущий контакт при переборе кнопок.

В функции *setup()* в **строках 5 и 6** расположен цикл, задающий режимы работы для всех контактов кнопок.

Новая для нас команда расположена **в строке 7**. Эта команда инициализирует последовательный канал. Все команды работы с последовательным каналом являются производными от набора команд *Serial*. Команда *Serial.begin*(*скорость*) подготавливает последовательный порт для его использования в вашей программе. В момент инициализации задается скорость, с которой будет работать последовательный порт.



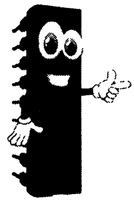
ПРИМЕЧАНИЕ.

Скорость порта в компьютере и в модуле Ардуино должны быть ***одинаковы****. Вы должны зайти в настройки порта в компьютере и узнать, какая скорость порта там установлена.*

В программе на Ардуино нужно установить такое же значение скорости. Скорость может принимать одно значение из стандартного набора:

110;
150;
300;
600;
1200;
2400;
4800;
9600;

14400;
19200;
38400;
57600;
115200.

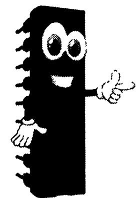


ПРИМЕЧАНИЕ.

*Но чаще всего в современных компьютерах по умолчанию порт настроен на **9600**.*

После *setup()* начинается основной цикл программы *loop()*. В теле функции *loop()* в **строке 9** инициализируется цикл перебора кнопок. Фактически, цикл занимает все тело функции *loop()*. Цикл перебирает контакты со 2 по 12, к которым подключены кнопки с «0» по «9» и кнопка «Ввод».

Первая команда в теле цикла (**строка 10**) — это чтение и оценка состояния кнопки, подключенной к текущему входу.



ПРИМЕЧАНИЕ.

***Строки 11...17** выполняются только в том случае, если кнопка окажется нажатой.*

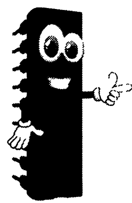
В **строке 11** вызывается задержка антидребезга.

В **строке 12** расположен цикл ожидания отпускания кнопки.

В **строке 13** — первая защитная задержка.

В **строке 14** производится оценка текущего номера контакта. Номер контакта проверяется на условие «не равно 12». К контакту номер 12 подключена кнопка «Ввод».

Если условие выполнено (нажатой оказалась не кнопка «Ввод»), то в последовательный порт передается код кнопки (команда *Serial.print(NumberPin-2,DEC)* в **строке 14**). В противном случае в последовательный порт передается пустое значение с командой перевода строки и возврата каретки (**строка 15**).



ПРИМЕЧАНИЕ.

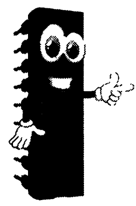
*Как видно из программы, **передача данных** в последовательный порт производится при помощи двух разных команд.*

Это аналоги команды *print()* и *println()*, которые имеются в любой версии языка СИ. Формат команд следующий:

Serial.print(Значение,Формат);

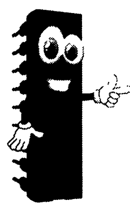
Serial.println(Значение,Формат);

Параметр «*Значение*» может быть любого из стандартных типов (см. **Приложение 2**), но данные любого типа преобразовываются и передаются в виде текста.



ПРИМЕР.

Число 32.43 передается как строка «32.43». В качестве значения также можно передавать строку символов, например, «Hello word».



ВНИМАНИЕ!

***Недостаток** транслятора Ардуино в том, что строки, написаные кириллицей, передаются неправильно. Переданные по последовательному каналу **строки, записанные русскими буквами**, при отображении в мониторе будут выглядеть как **набор иероглифов**. При необходимости, вы все же можете передать кириллический текст, но вам придется самостоятельно правильно определять и передавать в порт код каждой буквы.*

Параметр «*Формат*» в функциях *Serial.print()* и *Serial.println()* не обязательный. Он предназначен для изменения способа отображения данных. Допустимые значения, BIN (дво-

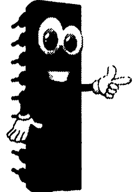
ичный), OCT (восьмеричный), DEC (десятеричный), HEX (шестнадцатеричный).

Для вещественных (дробных) чисел параметр «*Формат*» задает количество знаков после запятой. **Пример**:

Serial.print(78, BIN) выводит «1001110»
Serial.print(78, OCT) выводит «116»
Serial.print(78, DEC) выводит «78»
Serial.print(78, HEX) выводит «4E»
Serial.print(1.23456, 0) выводит «1»
Serial.print(1.23456, 2) выводит «1.23»
Serial.print(1.23456, 4) выводит «1.2346»

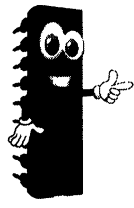
В **строке 14** программы команда *Serial.print()* передает в последовательный порт номер нажатой кнопки в виде десятичного числа. Номер нажатой кнопки вычисляется из номера текущего контакта путем уменьшения его на 2. Если номер текущего контакта равен 12, то выполняется команда *Serial.println()* в **строке 15**.

Команда *Serial.println()* аналогична команде *Serial.print()*, но отличается тем, что печать содержимого параметра «*Значение*» завершает передачей символов возврата каретки и перевода строки.



ПРИМЕЧАНИЕ.

*Так как в нашем случае ничего, кроме возврата каретки, передавать не нужно, параметр **«Значение»** содержит один символ пробела.*

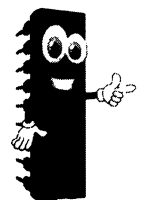


ВНИМАНИЕ!

*Функция **Serial.println()** не позволяет печатать пустую строку. Поэтому параметр **«Значение»** должен содержать хотя бы один любой символ! В нашем случае это пробел.*

После передачи данных в последовательный порт наша программа переходит к **строке 16**. В этой строке находится **команда формирования защитной задержки**.

Далее, в **строке 17** мы видим команду *break*. По этой команде программа досрочно выходит из цикла перебора кнопок. То есть, текущий цикл перебора кнопок прерывается. Однако функция *loop()* продолжает выполняться. Поэтому весь процесс повторяется. То есть управление передается на **строку 9** программы и перебор кнопок начинается сначала.



ПРИМЕЧАНИЕ.

В результате работы программы при каждом обнаружении нажатой кнопки ее код передается в последовательный порт. Повторной передачи кода уже нажатой кнопки удается избежать благодаря циклу ожидания отпускания кнопки. Пока кнопка нажата, сканирование кнопок приостановлено.

Для того **чтобы увидеть поступающие из модуля Ардуино данные на компьютере**, нужно, находясь в среде разработки Ардуино, запустить Монитор порта. Это можно сделать:

- либо при помощи меню: Инструменты/Монитор порта;
- либо при помощи иконки [иконка] в панели инструментов (в правом верхнем углу окна программы).

В открывшемся окне монитора при нажатии на цифровые кнопки, подключенные к модулю Ардуино, в поле приема данных будут появляться цифры, соответствующие нажатым кнопкам. Цифры выстроятся в число.

После нажатия на кнопку «Ввод» произойдет перевод строки, и следующие цифры будут выводиться уже в новой строке. Как это выглядит на экране, показано на **рис. 14.2**.

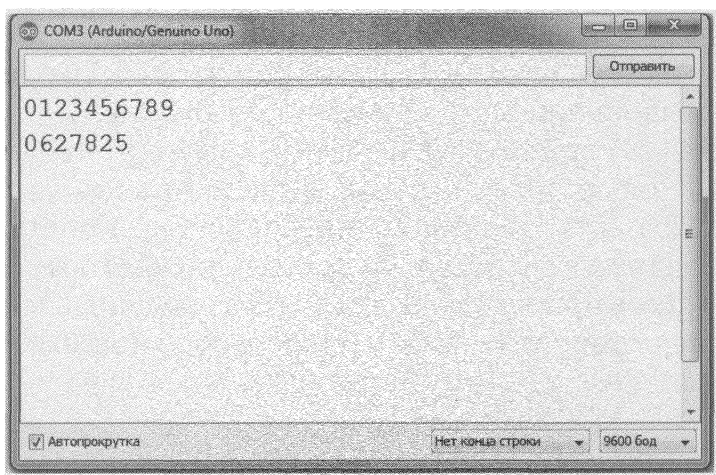
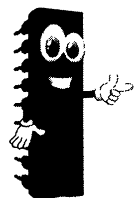


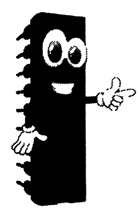
Рис. 14.2. Окно Монитора порта с результатами работы программы



ПРИМЕЧАНИЕ.

В заключение, приведем несколько полезных замечаний по поводу использования контактов 0 и 1.

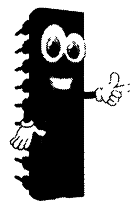
Возможна ситуация, когда для решаемой вами задачи может **потребоваться больше, чем 11 кнопок**. С некоторыми ограничениями и оговорками вы все же сможете использовать два этих контакта для подключения кнопок.



ПРИМЕЧАНИЕ.

Контакт номер 0 *в нашем случае использовать проще всего. Этот контакт связан с цепью, которая используется для передачи данных от компьютера в модуль Ардуино.*

Конфликт может возникать, только если случайно нажать на кнопку в момент загрузки программы в модуль. При работе программы данные от компьютера в модуль не передаются, поэтому кнопка будет работать без проблем.



ПРИМЕЧАНИЕ.

> ***Контакт номер 1*** *можно использовать только в крайнем случае. В нашей программе это возможно.*

Передача данных в компьютер у нас происходит после отпускания кнопки. В программе даже предусмотрена **защитная задержка**. То есть, в момент передачи данных контакты кнопки гарантированно будут не замкнуты. Данные будут переданы без искажений.

Но само нажатие кнопки воспринимается последовательным каналом как передача пустого значения (символа пробела). Если в нашей программе для подключения одной из кнопок использовать контакт 1, то при нажатии на эту кнопку в окне монитора порта перед символом номера кнопки будет выводиться лишний пробел.

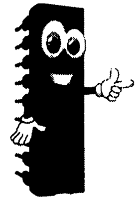
В некоторых программах, для некоторых задач это вполне допустимо. Тем более, что такая ситуация может быть легко обработана программно.

ГЛАВА 15

ПЕРЕДАЧА ДАННЫХ С КОМПЬЮТЕРА НА АРДУИНО

Постановка задачи

В предыдущей главе мы передавали информацию от модуля Ардуино на компьютер. В этой главе мы будем решать обратную задачу.



ПРИМЕЧАНИЕ.

Процесс приема данных из последовательного канала отличается от процесса передачи. Последовательный канал микроконтроллера сразу после его инициализации переходит в режим ожидания данных от компьютера.

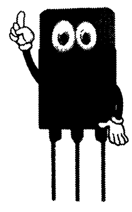
Как только придет байт данных, электроника последовательного канала записывает этот байт в специальный буфер обмена. Все процессы по приему байтов в буфер обмена выполняются в фоновом режиме с использованием прерывания, параллельно и не зависимо от основной программы. И

программисту ничего, кроме инициализации порта, для этого делать не нужно. Буфер обмена может хранить до 64 байт. Для чтения и использования данных из буфера в языке Ардуино имеются две функции:

Serial.available() — возвращает количество принятых байтов в буфере.

Serial.read() — читает один байт из буфера. Возвращает –1 (минус один), если байты закончились.

Попробуем создать программу, которая будет получать команды от компьютера по последнему порту и выполнять в зависимости от принятых данных те либо иные действия. Проще всего управлять генерацией звука. И так, ставим задачу!



ЗАДАЧА.

Создать программу, которая должна получать из компьютера сигналы в виде кодов нот (от 0 до 9). Получив код ноты, программа должна сформировать звуковой сигнал с высотой тона, соответствующего номеру ноты. При этом длительность каждого звука должна иметь некую фиксированную длину.

Схема

Исходя из условий поставленной задачи, наша схема должна состоять всего лишь из одного звукоизлучателя, подключенного к одному из цифровых контактов модуля Ардуино. Подключим его точно так же, как в схеме музыкальной шкатулки. В результате схема нашего устройства будет выглядеть так, как показано на ***рис. 15.1****.*

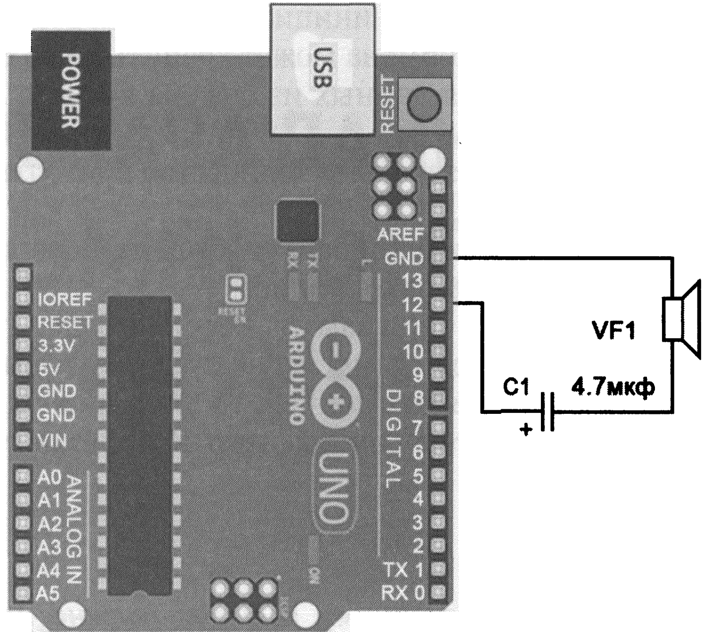


Рис. 15.1. Схема устройства, управления звуком с компьютера

Алгоритм

1. Проверить, есть ли данные в буфере данных.
2. Если буфер не пуст, извлечь очередной байт данных из буфера.
3. Используя принятый код, вычислить номер ноты.
4. Воспроизвести звуковой сигнал с высотой тона, взятой из таблицы частот, используя номер ноты. При вызове звука задать фиксированную длительность.
5. Одновременно с формированием звука сформировать задержку с длительностью в полтора раза большей, чем длительность ноты (чтобы между соседними звуками была пауза).
6. Продолжить выполнения алгоритма сначала, перейдя к строке 1 данного алгоритма.

Программа

Ниже приведена программа, реализующая вышеописанный алгоритм (см. **листинг 15.1**). Как видите, это одна из самых простых программ, приведенных в данной книге.

В **строке 1** программы определяется константа, указывающая на номер контакта звукоизлучателя.

В **строке 2** описан массив, содержащий **таблицу частот десяти основных нот**.

Далее, в **строке 3** начинается описание основного цикла *setup()*. В теле основного цикла всего две команды.

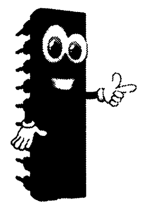
Команда в **строке 4** устанавливает режим работы контакта звукоизлучателя.

В **строке 5** расположена уже знакомая нам команда инициализации последовательного канала.

В **строке 6** начинается описание основного цикла программы — функции *loop()*. Тело функции содержит два вложенных оператора *if*.

Первый из них (**строка 7**) считывает количество принятых байтов в буфере обмена и проверяет это количество. Если в буфере есть хотя бы один байт, выполняются операторы в **строках 8, 9, 10, 11**.

В **строке 8** находится выражение, которое извлекает очередной символ из буфера, вычисляет код ноты и помещает этот код в переменную *InpCod*. Для извлечения кода используется функция *Serial.read()*. Для вычисления номера ноты из кода вычитается число 48.



ПРИМЕЧАНИЕ.

Дело в том, что данные из компьютера в модуль Ардуино, так же, как и данные из Ардуино в компьютер, передаются в виде кодов символов в кодировке ASCII. Если из ASCII кода цифр (см. ***табл. 14.1****) вычесть 48, мы получим число, равное самой этой цифре (0, 1, 2 и т. д.). Это же значение будет соответствовать и номеру ноты.*

Листинг 15.1. Программа «Звук по команде с компьютера»

```
   /*
     Получение команд с компьютера
   */

   // Определение констант:
1  const int SoundPin = 12;     // контакт звукоизлучателя

   //частоты нот
2  const int tabFreq[10] = { 1046, 987, 932, 880, 831, 784,
                             740, 698, 659, 622 };

3  void setup() {
4    pinMode(SoundPin, OUTPUT);   // режим контакта звука
5    Serial.begin(9600);
   }

6  void loop() {
7     if (Serial.available()>0) {
8       int InpCod = Serial.read()-48;
9       if ((InpCod>=0)&(InpCod<=9)) {
10        tone(SoundPin,tabFreq[InpCod],200);
11        delay(300);
        }
      }
   }
```

В результате, полученный номер ноты окажется записанным в переменную *InpCod*.

Далее, в **строке 9** проверяется, попадает ли вычисленный код в диапазон от 0 до 9. Это нужно для того, чтобы программа не реагировала на другие, не цифровые символы. В круглых скобках оператора *if* находится выражение, которое реализует сложное условие.

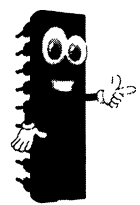
Выражение состоит из двух отдельных выражений, каждое из которых заключено в свои круглые скобки:

- **первое выражение** проверяет переменную на соответствие нижней границе диапазона (*InpCod* должно быть больше или равно нулю);
- **второе выражение** точно так же проверяет верхнюю границу (*InpCod* должно быть меньше или равно 9).

Оператор & (И) объединяет эти два условия. Благодаря нему все выражение истинно, если истинны оба входящих в него условия в круглых скобках. Если номер ноты удовлетворяет всем этим условиям (то есть попадает в нужный диапазон), то выполняются **строки 10 и 11**.

В **строке 10** находится команда формирования звука. Высота тона берется из таблицы *tabFreq[]*, знакомой нам по программе «Десять нот». В качестве указателя массива используется значение переменной *InpCod*. Длительность звукового сигнала выбрана равной 200 миллисекундам.

В **строке 11** формируется задержка в 300 миллисекунд.



ПРИМЕЧАНИЕ.

Дело в том, что в то время, когда Ардуино формирует звук, программа продолжает свою работу. Для того чтобы программа не запустила формирование нового звукового сигнала в то время, пока еще не закончилось формирование текущего, нужно выдержать паузу. Эта пауза в нашем случае выбрана на 100 миллисекунд больше, чем длительность звука. Это гарантирует паузу между двумя последовательными звуковыми сигналами в 100 миллисекунд.

Основной цикл программы *loop()* выполняется бесконечно, при этом каждый раз проверяя содержимое буфера обмена и воспроизводя звуки согласно считанным из буфера кодам.

Для того, что бы проверить работу приведенной выше программы нужно:

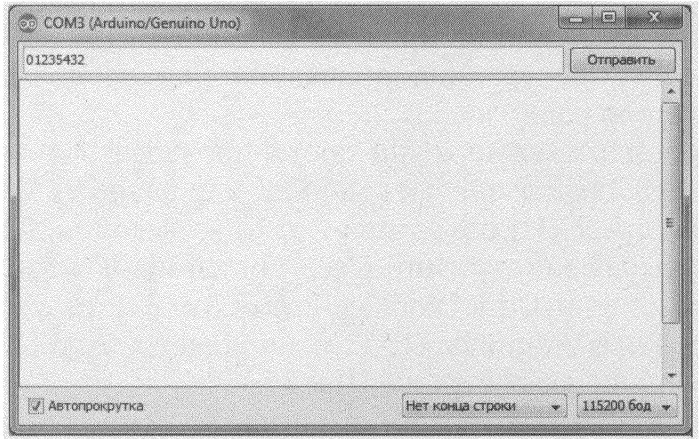
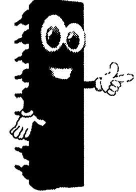


Рис. 15.2. Передача данных из окна монитора порта

- подключить модуль Ардуино к компьютеру;
- запустить среду разработки IDE;
- открыть окно монитора порта, как описано в предыдущей главе;
- набрать в верхней строке окна монитора команду (то есть вписать туда цифру от 0 до 9);
- нажать кнопку «Отправить» в верхнем правом углу окна.



ПРИМЕЧАНИЕ.

Можно также просто после набора цифры нажать клавишу «Enter». Строка отправки кода очистится, и код отправится по последовательному каналу.

Получив код ноты, модуль Ардуино издает звуковой сигнал выбранного музыкального тона. Вы можете написать целую строку цифр (как показано на **рис. 15.2**) и послать их в модуль за один раз (одно нажатие кнопки «Отправить» или клавиши «Enter»). Ардуино воспроизведет все введенные ноты по порядку, разделяя их паузой в 100 миллисекунд.

ГЛАВА 16

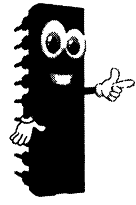
МУЗЫКАЛЬНАЯ ШКАТУЛКА

Постановка задачи

В **главе 11** мы познакомились с оператором *tone()* и узнали, что при помощи этого оператора мы легко можем сформировать на любом из выходов звуковой сигнал любой частоты в звуковом диапазоне и с заранее заданной длительностью.

Из этого следует, что для того, чтобы Ардуино воспроизвел мелодию, нужно просто сформировать последовательность звуков (нот) нужной частоты (высоты тона) и длительности. Частоты всех звуков, составляющих мелодию можно поместить в массив (что такое массив мы уже знаем).

В другой массив можно поместить все длительности звуков. И это будет самый простой способ извлечения мелодий.

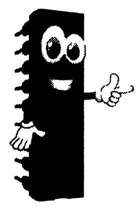


ПРИМЕЧАНИЕ.

И действительно, такой способ довольно часто применяется для простых программ, воспроизводящих одну мелодию. Мы же предлагаем вам рассмотреть более сложную задачу.

Предложенный выше способ хранения нот мелодии очень расточителен. Для хранения частоты звукового сигнала нужна переменная или даже массив переменных с типом значения

unsigned int. Этот тип значения занимает в памяти два байта. Кроме того один байт потребуется для хранения значения длительности.



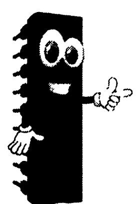
ПРИМЕЧАНИЕ.

Автор этих строк еще в самом первом издании книги [1] предложил и реализовал способ кодировки нот мелодии, при котором для хранения одной ноты программа использует всего один байт.

И сейчас, вашему вниманию предлагается программа, которая реализует тот же самый способ кодирования нот, но **уже средствами языка Ардуино**.

Для начала опишем **суть предлагаемого способа кодирования**. Во-первых, вместо частоты звукового сигнала нужно использовать просто номер ноты. Как известно, для большинства простых мелодий достаточно двух, в крайнем случае, трех октав.

Одна октава содержит семь основных и пять дополнительных нот. То есть, для простых задач будет вполне достаточно, если мы будем использовать чуть больше тридцати нот.



ПРИМЕЧАНИЕ.

Поэтому предлагается из всего нотного стана выбрать 32 ноты самого популярного диапазона и пронумеровать их по порядку. Тогда, для хранения в памяти мелодий мы можем использовать лишь номера нот, и нам потребуется значительно меньше памяти.

На **рис. 16.1** изображен фрагмент музыкальной клавиатуры (первая и вторая октава) с указанием кодов нот, которые мы будем использовать далее в нашей музыкальной программе.

На **рис. 16.2** показано, как выглядит та же кодировка на нотном стане. Для простоты на **рис. 16.2** показаны только основные ноты (ноты белых клавиш). Дополнительные ноты (черные клавиши) на нотном стане изображаются с использованием диезов или бемолей.

Присвоенные им коды вы легко можете увидеть на **рис. 16.1**, на котором принцип кодировки нот выглядит гораздо нагляднее. Все ноты (основные и дополнительные) нумеруются по порядку.

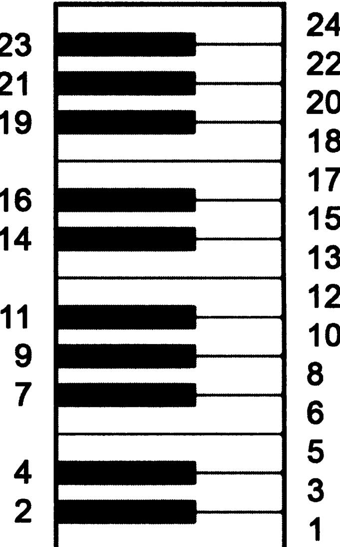
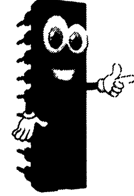


Рис. 16.1. *Коды нот на клавиатуре*



ВНИМАНИЕ!

Под номером ноты в приведенных выше рассуждениях подразумевается номер высоты тона ноты.

Такая же ситуация наблюдается в вопросе кодирования длительности звучания ноты.

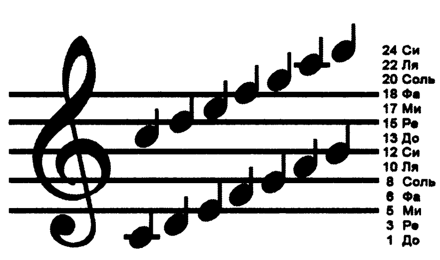
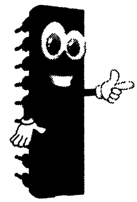


Рис. 16.2. *Коды нот на нотном стане*



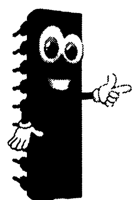
ПРИМЕЧАНИЕ.

В музыке не используется произвольная длительность. Величина длительности музыкальных звуков подчиняется ***строгим законам гармонии****. В музыке конкретное значение длительности не так важно. Длительность меняется в зависимости от темпа исполнения. Решающее значение имеет соотношение между величинами разных длительностей.*

В музыке существуют следующие термины, определяющие длительность звуков (и пауз):

- Целая;
- Половинная;
- Четверть;
- Восьмая;
- Шестнадцатая и т. д.

Достаточно пронумеровать каждую их этих типов длительности и использовать в кодировке мелодий код длительности ноты, а не ее значение. В **табл. 16.1** приведены коды всех возможных длительностей, которые мы будем использовать далее в программе.



ПРИМЕЧАНИЕ.

Обратное направление нумерации вызвано желанием использовать уже готовые таблицы кодов мелодий из программы-первоисточника, приведенной в ***[1]****. А там большей длительности соответствовал больший номер.*

Для того чтобы облегчить задачу кодировки длительностей ваших мелодий, на **рис. 16.3** показано, как обозначается длительность ноты в нотной грамоте. Ноты разной длительности отличаются хвостиками.

Кодировка длительностей *Таблица 16.1*

Код	Длительность	Величина в миллисекундах
7	1 (целая)	2560
6	1/2 (половинная)	1280
5	1/4 (четверть)	640
4	1/8 (восьмая)	320
3	1/16 (шестнадцатая)	160
2	1/32 (тридцать вторая)	80
1	1/64 (шестьдесят четвертая)	40
0	1/128 (сто двадцать восьмая)	20

Предложенный выше способ кодировки высоты тона и длительности ноты позволяют закодировать любую ноту всего одним байтом. Как известно, один байт состоит из восьми единичных разрядов (битов).

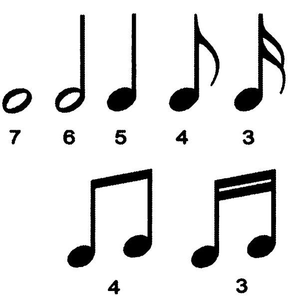


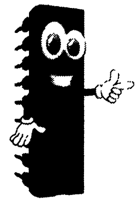
Рис. 16.3. *Кодировка длительностей*

Три старших бита мы будем использовать для хранения кода длительности ноты, а пять оставшихся битов — для хранения кода высоты тона. Вот как будет выглядеть двоичное число, в котором закодирована нота с кодом длительности 6 (половинная длительность) и кодом высоты тона 9 (соль диез первой октавы).

11001001 = 201
Код ноты (9)
Код длительности (6)

В десятичном представлении значение вышеприведенного кода равно 201. Используя описанный выше способ, можно закодировать все ноты любой мелодии.

Однако **кроме нот**, любая мелодия содержит еще и **паузы**. Длительность пауз в музыке подчиняется тем же правилам, что и длительность звуков.



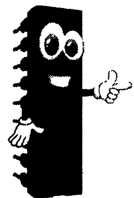
ПРИМЕЧАНИЕ.

Поэтому при кодировке мелодий мы будем представлять паузу, как ноту без звука. Логично и удобно присвоить такой ноте-паузе код, равный нулю.

Вот как будет выглядеть полный код паузы с четвертной длительностью (код длительности 5).

10100000 = 160
- 00000 — Код ноты (0)
- 101 — Код длительности (5)

Теперь для того, чтобы закодировать какую-либо мелодию, нужно составить **цепочку кодов**. В цепочку должны входить коды каждой ноты и каждой паузы кодируемой мелодии. Полученные коды нужно записать в массив.



ПРИМЕЧАНИЕ.

***Массив** может иметь тип значений – **byte**. Таким образом, каждый элемент такого массива займет в памяти всего один байт.*

Теперь, когда мы определились со способом кодировки мелодий, мы можем сформулировать **задачу**.

ЗАДАЧА.

Доработать программу «Десять нот», представленную в главе 11 этой книги (листинг 11.1) таким образом, чтобы при нажатии любой кнопки звучала мелодия. Для каждой кнопки мелодия должны быть разная. Назовем такую программу «Музыкальная шкатулка».

Схема

*Как следует из поставленной выше задачи, схема должна оставаться прежняя, предложенная в **главе 11** (см. **рис. 11.1**).*

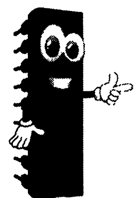
Алгоритм

1. Просканировать клавиатуру и определить номер самой первой нажатой кнопки.
2. По номеру нажатой кнопки найти в памяти таблицу закрепленной за этой кнопкой мелодии.
3. Начать цикл воспроизведения мелодии. Этот цикл должен перебирать по порядку коды нот из таблицы мелодии, выбранной в пункте 2 алгоритма.
4. Каждый извлеченный из таблицы код ноты разложить на код тона и код длительности в соответствии с правилами кодирования, описанными в начале этой главы.
5. Если код тона не равен нулю, сформировать звуковой сигнал с высотой тона, соответствующей коду тона и длительностью, соответствующей коду длительности. Код тона и код длительности использовать те, которые были получены в пункте 4 данного алгоритма.

6. Если код тона равен нулю, сформировать паузу длительностью, соответствующей коду длительности, полученному в пункте 4 данного алгоритма.
7. Повторять пункты 2...6 настоящего алгоритма многократно, до тех пор, пока нажата кнопка, выбранная в пункте 1 алгоритма.
8. После того, как перебор кодов нот в таблице мелодии закончен, перейти к началу алгоритма, то есть к пункту 1 и начать новое сканирование кнопок.
9. Если при сканировании кнопок окажется, что ни одна кнопка не нажата, запретить формирование звука и так же перейти к началу алгоритма.

Программа

Программа, реализующая поставленную задачу и описанный выше алгоритм, приведена в **листинге 16.1**. За основу этой программы взята **программа «Десять нот»** (см. **листинг 11.1**).



ПРИМЕЧАНИЕ.

Та часть программы, которая занималась сканированием кнопок, оставлена без изменений.

Но теперь, после определения номера нажатой кнопки программа вызывает добавленную в конец программы процедуру воспроизведения мелодии. Кроме того, в начало новой программы (в область определения констант и переменных) добавлен набор массивов констант. Эти массивы представляют собой таблицы мелодий (10 таблиц по одной на каждую мелодию), и несколько вспомогательных таблиц, назначение которых мы узнаем по мере рассмотрения текста программы.

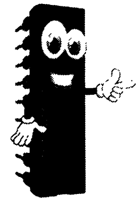
В **строке 1** программы определяется константа — номер контакта для подключения звукоизлучателя.

В **строках 2 и 3** определяются две глобальные переменные. И константа и переменные нам уже знакомы по программе «Десять нот». А вот дальше начинаются определения таблиц.

В **строке 4** определяется таблица длительностей нот. Как видите, таблица выполнена в виде массива, в который помещаются значения всех используемых в программе задержек. Это те самые значения, которые мы уже приводили в **табл. 16.1**, в графе «Величина в миллисекундах». Значение задержки с кодом 0 помещается в элемент массива номер ноль, значение задержки с кодом **1**, в элемент номер **1** и так далее. Теперь, для того чтобы, например, получить величину задержки с кодом **4** нужно просто обратиться к элементу массива номер **4** (вот так: *tabz[4]*).

В **строке 5** определяется таблица частот. При помощи этой таблицы программа может быстро определить частоту (высоту тона) любой ноты. Например, частота ноты 10 определяется как *tabFreq[10]*.

В **строках 6...15** размещены таблицы мелодий. Таблицы с именами *mel1[]*, *mel2[]* и т. д. содержат коды нот разных мелодий. Каждый такой код ноты в таблице содержит номер длительности и номер ноты, закодированные способом, описанным выше в начале этой главы. В конце каждой таблицы мелодий стоит код 255. Этот код служит признаком конца мелодии.



ПРИМЕЧАНИЕ.

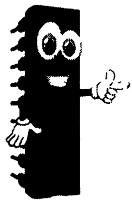
Все мелодии имеют разную длину, поэтому без специального маркера, индицирующего окончание каждой конкретной мелодии, не обойтись.

В **строке 16** мы видим определение еще одной таблицы. Она определена как **tabm[]*. Это таблица ссылок на таблицы мелодий. Она помогает программе обратиться к нужной мелодии по ее номеру. Элементами таблицы **tabm[]* имеют новые

для нас типы значения — значение типа «ссылка». Механизм ссылок — это тоже фирменная особенность языка СИ. До сих пор мы имели дело с переменными и массивами, имеющими стандартные типы значения:

- число;
- символ;
- булево;
- логическое и т. п.

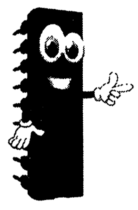
Ссылочная переменная работает с ссылками на другие переменные, функции, массивы и т. д.



ПРИМЕР.

Элемент номер 0 массива ****tabm[]*** *содержит ссылку на таблицу* ***mel1****. Элемент номер 1 содержит ссылку на таблицу mel2, элемент номер 2 содержит ссылку на* ***таблицу mel3*** *и т. д.*

Вы легко увидите это, взглянув на **строку 16** программы. В каждой таблице, на которую ссылается соответствующая ссылка, содержится набор кодов одной из мелодий. Так, например, в таблице *mel3* записаны коды мелодии, которая должна воспроизводиться при нажатии кнопки номер 3. Используя таблицу ссылок, можно обраться к любой таблице мелодий. И даже к любому элементу любой таблицы мелодий.



ПРИМЕР.

Получить значение любого элемента таблицы мелодий ***mel3[]*** *можно следующим образом:* ***tabm[2][НомерЭлемента]****.*

Результат выражение *tabm[2]* — это ссылка на таблицу *mel4[ ]*. Поэтому вместо *mel4[НомерЭлемента]* можно написать *tabm[2][НомерЭлемента]*.

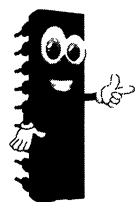
Допустим, мы хотим обратиться к элементу 5 таблицы мелодий *mel4[]*. Как видно из строки 9 программы, значение этого элемента равно 143 (не забывайте, что нумерация элементов начинается с нуля). А, значит, что мы можем прировнять следующие три выражения:

tabm[2][5] = *mel4[5]* = 143

Существует и **другой способ работы со значениями типа «ссылка»**. Ссылка на массив одновременно является ссылкой на его первый элемент (элемент номер ноль). Значение типа «ссылка» можно присвоить какой-нибудь переменной. Но эта переменная должна и сама должна иметь ссылочный тип. То есть нам нужна переменная типа «ссылка».

Например, в нашей программе мы присваиваем значения различных элементов массива *tabm[]* переменной *nota*. Предположим, что нам нужно присвоить переменной значение элемента номер 2. Для этого просто запишем *nota = tabm[2]*.

Теперь, при помощи переменной *nota*, можно обращаться к любому элементу массива, на который указывает эта ссылка (массива *mel3[]*).



ПРИМЕЧАНИЕ.

*Как говорилось выше, сразу после присвоения значения ссылка, записанная в переменную **nota**, указывает на элемент массива номер ноль (**mel3[0]**).*

Для того чтобы получить значение элемента, на который указывает ссылка, используется оператор «*» (звездочка). В нашем случае выражение **nota* будет равно 132 (**nota = 132*, см. **строку 8** программы).

Если теперь увеличить значение ссылки на единицу, то ссылка укажет на следующий элемент массива. Увеличить значение ссылки можно, просто прибавив к нему единицу или применив команду инкрементации: *nota*++.

Листинг 16.1. Программа «Музыкальная шкатулка»

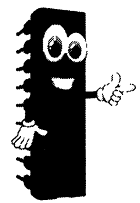
```
  /*
    Музыкальная шкатулка
  */

  // Определение констант
1 const int SoundPin = 12; // номер контакта звукоизлучателя

  // Определение глобальных переменных
2 int ButtonPin;     // указатель текущего входа от кнопки
3 boolean btFree;    // флаг «все кнопки отпущены»

  // Таблица длительностей нот
4 const unsigned int tabz[] = {20,40,80,160,320,640,1280,2560};

  // Таблица - частоты нот
5 const unsigned int tabFreq[]={0,587,622,659,698,740,784,
     831,880,932,988,1047,1109,1175,1245,1319,1397,1480,
     1568,1661,1760,1865,1976,2093,2217,2349,2489,2637,
     2794,2960,3136,3322,3520};

  // Таблицы мелодий
  // В траве сидел кузнечик
6 byte mel1[] = {109,104,109,104,109,108,108,96,108,104,
   108,104,108,109,109,96,109,104,109,104,109,108,108,96,
   108, 104,108,104,108,141,96,109,111,79,79,111,111,112,
   80,80,112,112,112,111,109,108,109,109,96,109,111,79,79,
   111,111,112,80,80,112,112,112,111,109,108,141,128,96,255};

  // Песенка крокодила Гены
7 byte mel2[] = {109,110,141,102,104,105,102,109,110,141,
   104,105,107,104,109,110,141,104,105,139,109,110,173,96,
   114,115,146,109,110,112,109,114,115,146,107,109, 110,
   114,112,110,146,109,105,136,107,105,134,128,128,102,105,
   137,136,128,104,107,139,137,128,105,109,141,139,128,110,
   109,176,112,108,109,112,144,142,128,107,110,142,141,128,
   105,109,139,128,173,134,128,128,109,112,144,142,128,107,
   110,142,141,128,105,109,139,128,173,146,128,255};
```

8	// В лесу родилась елочка byte mel3[] = {132,141,141,139,141,137,132,132,132,141, 141,142,139,176,128,144,146,146,154,154,153,151,149,144, 153,153,151,153,181,128,96,255};
9	// Happy births day to you byte mel4[] = {107,107,141,139,144,143,128,107,107,141, 139,146,144,128,107,107,151,148,146,112,111,149,117,117, 148,144,146,144,128,255};
10	// С чего начинается родина byte mel5[] = {99,175,109,107,106,102,99,144,111,175,96, 99,107,107,107,107,102,104,170,96,99,109,109,109,109, 107,106,143,109,141,99,109,109,109,109,104,106,171,96, 99,111,109,107,106,102,99,144,111,143,104,114,114,114, 114,109, 111,176, 96,104,116,112,109,107,106,64,73,143, 107,131,99,144,80,80,112,111,64,75,173,128,255};
11	// Из кинофильма «Веселые ребята» byte mel6[] = {105,109,112,149,116,64,80,148,114,64,78, 146,112,96,105,105,109,144,111,64,80,145,112,64,81,178, 96,117,117,117,149,116,64,82,146,112,64,79,146,144,96, 105,105,107,141,108,109,112,110,102,104,137,128,96,105, 105,105,137,102,64,73,142,105,107,109,64,75,137,96,105, 105,105,137,102,105,142,112,64,82,180,96,116,116,116, 148,114,112,142,109,64,78,146,144,96,105,105,107,141, 108,109,112,110,102,104,169,96,96,255};
12	// Улыбка byte mel7[] = {136,133,170,168,131,134,133,131,193,160, 133,136,138,138,138,140,143,141,140,138,173,200,138, 140,173,128,140,138,133,136,134,202,160,140,138,141,136, 140,138, 138,136,131,133,163,161,160,129,132,136,136, 136,136,168,141,129,132,131,131,131,163,131,132,136, 134,136,137,136,134,136,137,173,171,160,141,136,139, 137,137,137,169,141,134,137,136,136,136,168,141,132, 131,132,134,137,136,134,132,134,169,168,160,141,136, 139,137,137,137,169,141,134,137,136,136,136,168,141, 132,131,132,134,137,136,134,132,134,168,193,160,255};

```
   // Гимн России
13 byte mel8[] = {136,173,136,96,106,172,133,96,101,170,
    136,96,102,168,129,96,97,163,131,96,101,166,134,96,104,
    170,140,141,175,136,177,143,96,109,175,140,136,173,140,
    96,106,172,133,96,101,170,136,96,102,168,129,96,97,173,
    140,138,168,224,255};

   // Спят усталые игрушки
14 byte mel9[] = {168,128,133,168,128,133,168,168,166,165,
    163,165,200,143,145,143,145,212,168,128,131,168,128,
    131,168,166,165,163,161,165,200,145,148,145,148,214,
    168,168,170,168,177,177,170,168,161,161,166,168,170,
    170,168,166,165,165,166,165,232,198,195,193,160,224,255};

   // Гамма
15 byte mel10[] = {129,130,131,132,133,134,135,136,137,138,
    139,140,141,142,143,144,145,146,147,148,149,150,151,
    152,153, 154,155,156,157,158,159,128,159,158,157,156,
    155,154,153,152,150,149,148,147,146,145,144,143,142,
    141,140,139,138,137,136,135,134,133,132,131,130,129,
    128,255};

   // Таблица начал всех мелодий
16 byte *tabm[] = {mel1, mel2, mel3, mel4, mel5, mel6, mel7,
         mel8, mel9, mel10};

17 void setup() {
     // Установка режимов контактов:
18   pinMode(SoundPin, OUTPUT);  // контакт звука
     // цикл установки для контактов кнопок:
19   for (ButtonPin=0; ButtonPin<=9; ButtonPin++) {
20      pinMode(ButtonPin, INPUT_PULLUP);
     }
   }

21 void loop() {
     // Сканирование кнопок и воспроизведение звука:
22   btFree = true;  // флаг «все кнопки отпущены»
```

```
23     for (ButtonPin=0; ButtonPin<=9; ButtonPin++) {
24        if (digitalRead(ButtonPin)==LOW) {
25           playMelody (ButtonPin);
26           btFree = false;
27           break;
          }
       }
28     if (btFree) noTone(SoundPin);
    }

    // Функция воспроизведения мелодии
29  void playMelody (int numberMelody) {
30     byte fnota;   // код тона ноты
31     byte dnota;   // код длительности ноты
32     byte *nota;   // ссылка на текущую ноту

33     nota = tabm[numberMelody];
34     StartMelody:
35     if (digitalRead(ButtonPin)==HIGH) return;
                         // если кнопка отпущена, закончить
36     if (*nota==0xFF) return; // проверка на конец мелодии
37     fnota = (*nota)&0x1F;     // определяем код тона
38     dnota = ((*nota)>>5)&0x07; // опред. код длительности
39     if (fnota!=0) {
40        tone (SoundPin, tabFreq[fnota],tabz[dnota]);
41        delay (tabz[dnota]+10);
       }
42     else  {
43        noTone(SoundPin);   // если пауза выкл. звук
44        delay (tabz[dnota]);
       }
45     nota++;
46     goto StartMelody;
    }
```

Теперь выражение **nota* будет возвращать значение 141 (**nota = 141*). Так, увеличивая значение ссылочной переменной, можно перебрать все элементы таблицы мелодии.

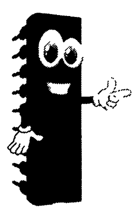


ПРИМЕЧАНИЕ.

Далее в программе мы будем использовать именно этот, последний, метод работы со ссылками.

Функция *setup()* занимает **строки 17...20** и не отличается от аналогичной функции из программы «Десять нот».

Функция *loop()* (**строки 21...28**) также является копией аналогичной функции из программы «Десять нот», но с одним небольшим отличием.

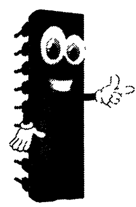


ПРИМЕЧАНИЕ.

*В **строке 25** вместо вызова команды tone() теперь производится вызов функции воспроизведения мелодии **playMelody()**. В качестве параметра в эту функцию передается номер нажатой кнопки.*

Функция *playMelody()* расположена в **строках 29...46** программы. Функция начинается с определения локальных переменных (**строки 30...32**). Переменная *fnota* предназначена для временного хранения кода высоты тона ноты, переменная *dnota* будет хранить код длительности ноты.

Обратите внимание на описание переменной в **строке 32**. В этой строке описывается переменная *nota*. Она будет использоваться для хранения ссылки на текущий элемент таблицы, выбранной для воспроизведения мелодии.

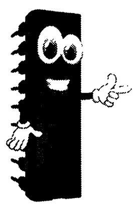


ПРИМЕЧАНИЕ.

По этой причине переменная имеет тип «ссылка», на что указывает символ звездочки перед именем переменной в строке определения.

В **строке 33** переменной *nota* как раз и присваивается значение ссылки. Ссылка извлекается из таблицы *tabm[]*.

В **строке 34** находится метка.

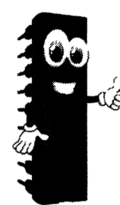


ЧТО ЕСТЬ ЧТО.

Метка *– это символьное имя, идентификатор, отмечающий место в программе, куда программа может осуществлять переход по команде* ***goto****.*

Метка состоит из имени метки, которое заканчивается двоеточием. Имя метки также должно соответствовать стандартным правилам компьютерных имен.

В **строке 35** происходит считывание и проверка состояния текущей кнопки. Номер этой кнопки был передан в функцию *playMelody()* при ее вызове через параметр *numberMelody*.

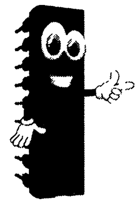


СОВЕТ.

В процессе воспроизведения мелодии нужно постоянно контролировать, нажата ли еще эта кнопка, для того, что бы немедленно прекратить воспроизведение мелодии при ее отпускании.

Оператор *if* в **строке 35** проверяет значение, определяющее состояния кнопки, и, если, это значение равно HIGH (кнопка отпущена), выполняется команда *return*. В данном случае команда *return* приводит к немедленному завершению функции *playMelody()*, а, значит, к прекращению процесса воспроизведения мелодии.

Если в результате проверки выяснится, что кнопка еще нажата, то программа начинает процесс воспроизведения мелодии. И начинается этот процесс проверкой конца мелодии.



ПРИМЕЧАНИЕ.

*Дело в том, что при воспроизведении мелодии программа будет постоянно возвращаться к метке **StartMelody**, считывая ноту за нотой. И в определенный момент мелодия закончится (очередной код ноты окажется равным 255).*

Прежде, чем начинать расшифровывать код ноты, нужно проверить, не конец ли это мелодии.

Поэтому в **строке 36** происходит проверка на конец мелодии. Если код ноты равен 0xFF (а это шестнадцатеричный аналог десятичного числа 255), то это означает, что мелодия закончилась. Если это так, выполняется оператор *return* в **строке 36**, и процедура *playMelody()* завершается.

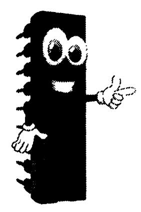
Если мелодия не окончена, то программа приступает к выделению из кода ноты:

- кода высоты тона (**строка 37**);
- кода длительности (**строка 38**).

Схематически процесс разделения показан на **рис. 16.4**.

Код тона получается путем наложения на код ноты маски 0x1F (см. выражение в **строке 37**). Число 0x1F имеет нули в трех старших разрядах и единицы во всех остальных (00011111).

Оператор «&» означает логическую операцию побитового «И».



ЧТО ЕСТЬ ЧТО.

***Операция «И»** называется **логическим умножением**.*

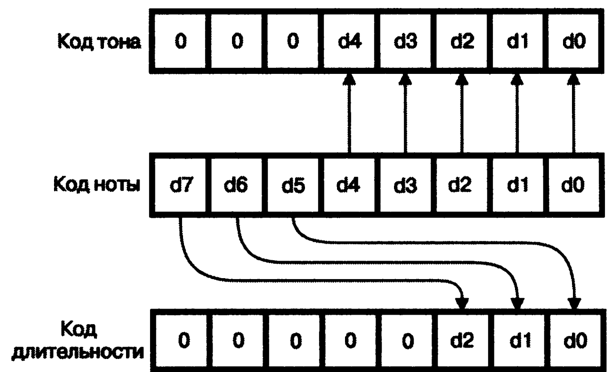


Рис. 16.4. *Процесс преобразования кодов ноты*

Если логически перемножить два бита, то результат умножения будет равен единице только в том случае, когда оба перемножаемых бита равны единице. Если хотя бы один из перемножаемых битов равен нулю, то и результат равен нулю.

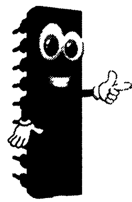
Если код ноты побитно перемножить с кодом маски, то в результате такого умножения те биты, которые в коде маски равны нулю, обнулятся и в байте результата. Те биты, которые в маске равны единице, в байте результата останутся такими же, как в исходном коде ноты (см. **рис. 16.4**).

Примерно так же вычисляется код задержки (**строка 38**). Только предварительно биты кода ноты сдвигаются вправо на 5 шагов, как показано на **рис. 16.4**. Сдвиг производится при помощи уже знакомого нам оператора «>>». После сдвига на код накладывается маска 0x07.

Шестнадцатеричное число 0x07 содержит нули в пяти старших двоичных разрядах и единицы в трех младших (00000111). Операция логического умножения «&» сдвинутого кода ноты и кода маски сбрасывает старшие биты, и, таким образом, результатом этих преобразований является код длительности ноты (см. **рис. 16.4**).

Теперь нужно воспроизвести **звук с полученными выше параметрами**. Но сначала нужно определить — это звук или пауза.

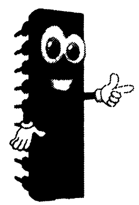
Для этого в **строке 39** проверяется значение переменной *fnota*. Если оно не равно нулю, то программа формирует звуковой сигнал с высотой тона *fnota* и длительностью *dnota* (**строка 40** программы).



ПРИМЕЧАНИЕ.

*Конкретное значение частоты сигнала берется из таблицы **tabFreq[]**, куда подставляется код высоты тона. Конкретное значение длительности звука берется из таблицы **tabz[]**, куда подставляется код задержки.*

Далее нужно приостановить работу программы на время, пока звучит звуковой сигнал. Вы, наверное, помните, что команда *tone()* работает автономно, параллельно с выполнением основной программы.



ВНИМАНИЕ!

***Если не предусмотреть задержку**, то программа запустит формирование следующего звука тогда, когда еще не закончил звучать предыдущий звук.*

Поэтому в **строке 41** программы предусмотрена **команда задержки**. Эта команда формирует задержку на 10 миллисекунд большую, чем заданная в функции *tone()* длительность звукового сигнала. Запас по длительности введен для того, чтобы между отдельными звуками всегда была небольшая микропауза и звуки не сливались в процессе воспроизведения мелодии.

Если код высоты тона (*fnota*) оказался равен нулю, то программа в **строках 43, 44** выполняет формирование музыкальной паузы с кодом длительности *dnota*. Для этого в **строке 43**

выполняется команда принудительного выключения звука *noTone()*, а в **строке 44** формируется пауза. Команда *noTone()* применяется в этом месте программы на всякий случай, для надежности. Вообще-то все звуки к этому моменту и так должны прекратиться.

В **строке 45** производится инкремент значения переменной *nota*. В результате ссылка, хранящаяся в этой переменной, теперь будет указывать на следующую ноту мелодии.

Затем, при помощи оператора *goto*, расположенного в **строке 46**, управление передается по метке *StartMelody*. Это означает, что выполнение программы продолжится со **строки 35**. То есть с первой из команд, расположенных после метки *StartMelody*.

Таким образом, образуется цикл чтения и воспроизведения кодов нот мелодии, который прекратится:

- либо в **строке 36**, если мелодия закончится (встретится код ноты 0xFF);
- либо в **строке 35**, если обнаружится, что кнопка с номером *ButtonPin* отпущена.

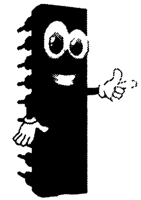
ГЛАВА 17

КОДОВЫЙ ЗАМОК

Постановка задачи

Наша книга подходит к концу, поэтому новый пример должен быть достаточно сложным с точки зрения алгоритма. Вашему вниманию предлагается **кодовый дверной замок с необычной логикой работы**. Данный пример не только поможет разобраться в некоторых новых приемах программирования для Ардуино, но может также найти практическое применение в вашем хозяйстве. Этот кодовый замок был разработан как пример еще для самого первого издания книги **[1]**.

В замке применен необычный способ набора кодовой комбинации. На первый взгляд замок устроен так же, как обычно выглядят подобные устройства. Замок имеет клавиатуру, состоящую из десяти кнопок, предназначенных для ввода цифр от 0 до 9. Также имеется переключатель режимов: «Запись кода — Работа». К выходу электронной части замка подключен электронный ключ, управляющий механизмом открывания замка (реле или соленоид).



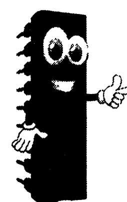
ПРИМЕЧАНИЕ.

Необычность замка *кроется в алгоритме набора кодовой комбинации. Кодовая комбинация может состоять не только из нажатий той или иной кнопки, но и из сочетаний двух или нескольких одновременно нажатых кнопок.*

При вводе кода программа просто запоминает любые изменения состояния клавиатуры. Затем, в режиме «Работа», вы должны повторить все манипуляции, которые вы выполняли при вводе кодовой последовательности в режиме «Запись кода»

Для того чтобы было понятнее, приведем пример кодовой комбинации. Допустим, сначала вы нажимаете кнопку «1». Это первое изменение состояния клавиатуры (было ничего не нажато, стало — нажата одна кнопка). Это состояние запоминается в памяти.

Далее, к примеру, не отпуская кнопки «1» вы можете, нажать кнопку «3». Это следующее изменение состояния клавиатуры (нажаты одновременно кнопки «1» и «3»). Оно так же запоминается в памяти. Далее, вы можете, например, отпустить кнопку «1». Или наоборот, не отпуская кнопки «1» и «3» нажать, к примеру, еще и кнопку «8». И все эти изменения последовательно запоминаются в памяти.



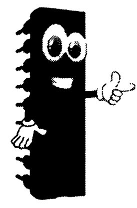
ПРИМЕЧАНИЕ.

Не важно, быстро или медленно вы набираете коды. Запоминается лишь последовательность изменения состояний.

Окончанием процесса набора кодовой комбинации является ситуация, когда все кнопки окажутся отпущенными, и в течение одной секунды не будет нажата ни одна кнопка.

Выше описанным способом набирается кодовая комбинация как в режиме «Запись кода», так и в режиме «Работа». Но в режиме «Записи кода» все эти манипуляции с клавиатурой запоминаются в памяти модуля Ардуино, а в режиме «Работа» правильно повторенная кодовая комбинация открывает замок.

Такой необычный алгоритм ввода кодовой комбинации существенно усложняет задачу подбора кода для злоумышленников.



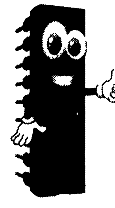
ПРИМЕЧАНИЕ.

При желании, вы всегда можете использовать электронный замок и привычным способом. Просто при наборе кодовой последовательности не нажимайте несколько кнопок одновременно.

Теперь разберемся, куда же мы будем записывать набранную в режиме записи кодовую последовательность. До сих пор мы использовали два вида памяти.

Во-первых, оперативное запоминающее устройство (ОЗУ). В нем хранятся значения всех переменных программы.

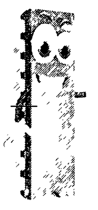
Во-вторых — программная память (здесь хранится наша исполняемая программа, а также постоянные константы). ОЗУ отличается высоким быстродействием, но информация в ОЗУ хранится лишь до тех пор, пока на модуль подано напряжение питания. При отключении питания все содержимое ОЗУ теряется.



ЧТО ЕСТЬ ЧТО.

Программная память (Flesh) *— это электрически стираемое ПЗУ (постоянное запоминающее устройство). Программа и константы записываются в эту память в процессе заливки программы и не изменяются в процессе ее работы.*

Для долговременного хранения данных, вводимых пользователем, в современных микроконтроллерах имеется специальный вид памяти, которая называется **EEPROM**.



ЧТО ЕСТЬ ЧТО.

EEPROM (Electrically Erasable Programmable Read-Only Memory) *— электрически стираемое перепрограммируемое постоянное запоминающее устройство).*

В модуле Ардуино (а, точнее, в микроконтроллере ATmega323) такая память также имеется. Для того чтобы ваша программа могла работать с этим видом памяти необходимо использовать специальную библиотеку. Эта библиотека, в отличие от использованных нами до сих пор библиотек, входит в стандартный пакет среды разработки IDE.

Библиотека так и называется «EEPROM». Как добавить библиотеку к программе мы рассматривали в **главе 6** данной книги. После подключения библиотеки в распоряжении программиста появляются следующие дополнительные функции:

- *EEPROM.write(НомерЯчейки, Значение)*; — запись байта данных в ячейку EEPROM памяти. Возвращаемого значения нет;
- *EEPROM.read(НомерЯчейки)*; — чтение байта данных из ячейки памяти EEPROM. Возвращает значение считанного байта.

Как видите, запись в энергонезависимую память EEPROM производится побайтно. Поэтому для записи значений переменных тех типов, которые занимают в памяти более одного байта (см. **Приложение 2**), вы должны самостоятельно вычислить значение каждого байта и записать каждое такое значение в отдельную ячейку EEPROM.

Чаще всего приходится иметь дело с переменными, которые занимают в памяти два байта. Для этого случая в языке Ардуино существуют специальные функции, позволяющие разбивать значение на байты, а затем объединять их обратно, в одно значение.

Так, функция *highByte()* получает из значения переменной значение ее старшего байта. Функция *lowByte()* получает значение младшего байта. А функция *word()* объединяет старший и младший байты в одно значение. Как это делается, мы увидим ниже при разборе уже готовой программы. А пока поставим задачу.

ЗАДАЧА.

Создать схему и программу на основе модуля Ардуино, реализующую вышеописанную идею электронного замка. Устройство должно иметь десять кнопок для ввода кода обозначенных цифрами от «0» до «9», переключатель режима «Запись кода - Работа», а также электронный ключ для управления исполнительным механизмом замка.

Схема

*Схема электронного замка приведена на **рис. 17.1**. Внешний вид электронного замка, выполненного с использованием универсальной монтажной платы, приведен на **рис. 17.2**.*

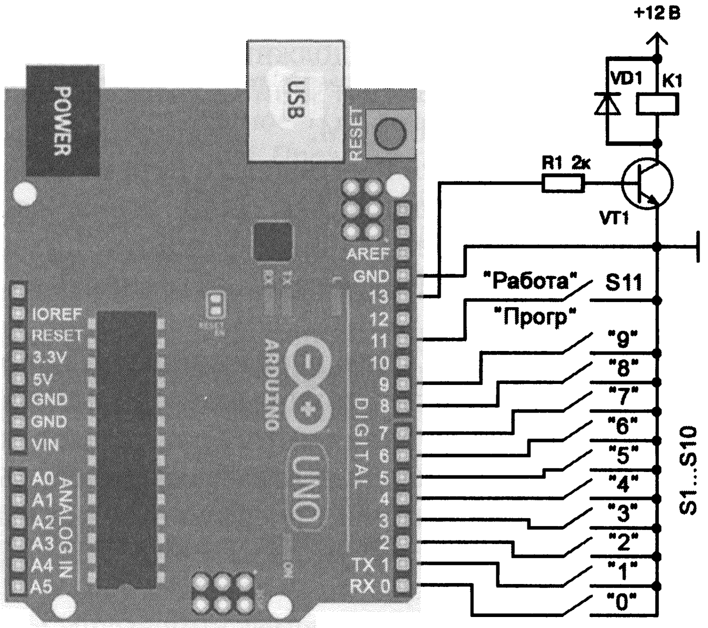


Рис. 17.1. *Схема электронного замка*

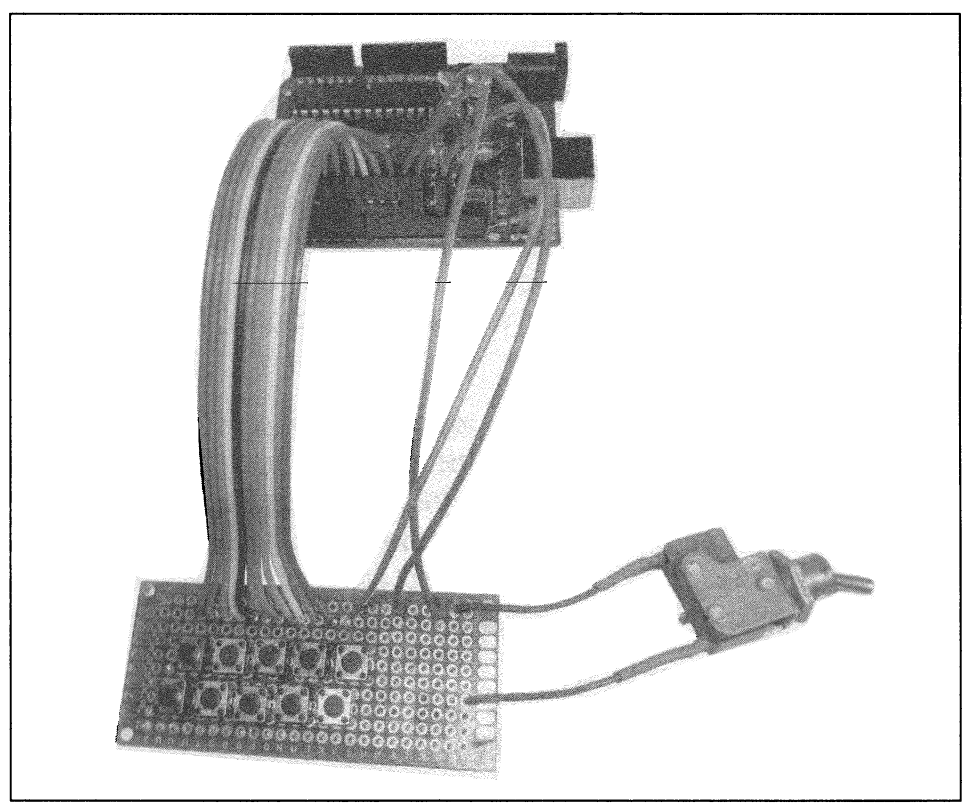


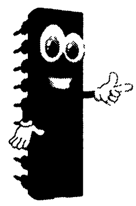
Рис. 17.2. Внешний вид электронного замка

Как видно из схемы, цифровые кнопки замка подключены к цифровым контактам 0...9 модуля. Переключатель режима представляет собой тумблер на два фиксированных положения, имеющий одну группу из двух контактов:

- в режиме «Запись кода» контакты тумблера должны быть замкнуты;
- в режиме «Работа» контакты тумблера должны быть разомкнуты.

Ключевой каскад для управления исполнительным механизмом замка подключается к контакту 13. К этому же контакту на плате Ардуино уже подключен светодиод, специально предназначенный для нужд программиста.

Как только на контакт 12 поступит напряжение, открывающее замок, светодиод загорится. И мы будем видеть, что замок открыт.

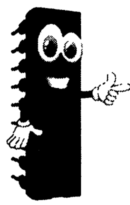


ПРИМЕЧАНИЕ.

В процессе испытания схемы и программы электронный ключ и исполнительный механизм можно не подключать. Для проверки и отладки программы достаточно светодиода.

Алгоритм

Как в режиме «Записи кода», так и в режиме «Работа» функционирование программы начинается с процедуры ввода кодовой комбинации с клавиатуры. Процедура в цикле многократно считывает состояние клавиатуры и ждет изменения этого состояния.



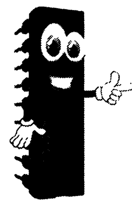
ЧТО ЕСТЬ ЧТО.

Состояние клавиатуры *– это число, состоящее из 10 двоичных битов информации.*

Каждый бит этого числа отображает состояние соответствующей кнопки клавиатуры. Младший бит (бит номер 0) отражает состояние кнопки «0». Следующий бит (бит номер 1) отображает состояние кнопки «1» и так далее.

Если какая либо кнопка нажата, то соответствующий ей бит равен нулю, если отпущена — единице. Код состояния клавиатуры записывается в переменную, имеющую тип *int*, то есть занимающую в памяти два байта.

Итак, процедура ввода кода с клавиатуры многократно считывает код состояния клавиатуры и ждет, когда при очередном считывании этот код изменится. Каждое новое значение кода записывается в буфер, который расположен в ОЗУ. В результате в буфере постепенно накапливается последовательность кодов, играющая роль ключа для открывания замка.

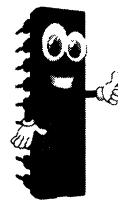


ПРИМЕЧАНИЕ.

В процессе ввода кодовой последовательности программа постоянно контролирует время между двумя последовательными нажатиями кнопок. Если это время превысит контрольное время (в 1 секунду), программа считает, что ввод кодовой последовательности завершен и выходит из цикла сканирования кнопок.

После того как ввод кодовой последовательности в буфер завершен, дальнейшие действия зависят от режима работы. Если переключатель режимов находится в **положении «Запись кода»**, программа переходит к процедуре записи кодовой последовательности из буфера (в ОЗУ) в долговременную память (EEPROM). При этом каждый код из буфера разделяется на старший и младший байты.

И все эти байты записываются по очереди в ячейки EEPROM. Затем туда же записывается и длина кодовой последовательности.



ПРИМЕЧАНИЕ.

Это необходимо, потому что длина последовательности – величина не фиксированная. В нашем кодовом замке мы можем использовать кодовую последовательность произвольной длины. Она ограничивается лишь объемом буфера.

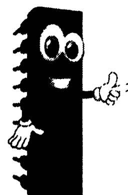
Если же переключатель режима находится в **положении «Работа»,** то после завершения процедуры ввода кодовой последовательности с клавиатуры программа переходит к процедуре чтения и сравнения кодов. Коды, записанные ранее, считываются по очереди из ячеек EEPROM и сравниваются со значениями кодов, только что введенных с клавиатуры, и находящихся в буфере в ОЗУ.

Если все коды обеих последовательностей (в EEPROM и в буфере ОЗУ) окажутся одинаковыми, программа дает команду на открывание замка.

Программа

Программа электронного замка приведена в **листинге 17.1**. При создании программы была подтверждена высокая степень совместимости языка Ардуина с другими СИ-подобными языками программирования.

Программа, которую мы видим в **листинге 17.1**, создана путем копирования программы электронного замка, приведенной в книге [1], написанной на языке СИ среды программирования Code Vision.



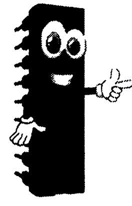
ПРИМЕЧАНИЕ.

Для адаптации программы с языка СИ Code Vision для работы в среде Ардуино пришлось сделать совсем немного изменений. Но мы не будем в данной книге заниматься сравнением двух программ. Кто хочет, может сравнить их самостоятельно. Мы просто разберем подробно работу программы, приведенной на ***листинге 17.1****.*

Как уже говорилось выше, программа начинается с присоединения библиотеки EEPROM.h (**строка 1**).

Далее, в **строках 2...8** производится определение различных констант, используемых в дальнейшем в этой программе. И тут используется другой, пока еще нам не знакомый **оператор присвоения**. Это универсальный оператор присвоения *#define*.

При помощи этого оператора можно присвоить имя любому значению, выражению или строковой константе. Оператор *#define* не требует указания типа значения, так как он определяет не значение, а только строковый литерал.



ПРИМЕЧАНИЕ.

Такой способ определение констант использовался в исходном варианте программы, взятом из книги ***[1]****. Оттуда он перешел в программу на Ардуино.*

Рассмотрим подробнее **назначение определяемых констант**.

В **строке 2** программы определяется константа *klfree*. Константе присваивается литерал 0x3FF. Это число в шестнадцатеричном формате, соответствующее коду состояния клавиатуры в случае, если ни одна кнопка не нажата. Это значение понадобится нам в процедуре «ожидания нажатия хотя бы одной из клавиш» и в процедуре «ожидания отпускания всех клавиш».

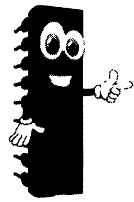
В **строке 3** определяется длительность контрольного промежутка времени *kontrTime*. Его мы будем использовать в процедуре проверки факта, окончился ли набор кодовой последовательности.

В **строке 4** определяется константа антидребезга.

В **строке 5** определен размер буфера, куда будет помещена набираемая кодовая последовательность. Эта же величина будет использоваться для определения количества ячеек в EEPROM, в которые мы будем помещать кодовую последовательность для длительного хранения.

В **строке 6** определяется константа *klen*, которая будет содержать адрес ячейки в EEPROM, в которой мы будем хранить длину нашей кодовой последовательности. Для этой цели мы выберем ячейку, расположенную сразу после блока ячеек,

зарезервированных для хранения всех байтов самой кодовой последовательности.



ПРИМЕЧАНИЕ.

Адрес не указывается напрямую, а вычисляется при помощи простого математического выражения. Чтобы узнать, сколько байтов займет кодовая последовательность, берется размер буфера в ОЗУ (константа ***bsize****) и умножается на два. Это делается потому, что для каждого элемента буфера в памяти EEPROM нужно оставить две ячейки памяти.*

Адрес ячейки, находящейся сразу после блока ячеек кодовой последовательности, будет на единицу больше, чем размер блока (то есть *bsize*2+1*). Определение адреса ячейки методом вычисления удобно тем, что при изменении размера буфера автоматически изменяется и адрес ячейки для записи длины последовательности. Это гарантирует от ошибок при модификации параметров.

В **строках 7 и 8** определяются номера контактов для подключения переключателя режимов (*modePin*) и ключа управления исполнительным механизмом (*relayPin*).

В **строках 9 и 10** определяются две переменные (*ii* и *i*), которые будут нам служить указателями при работе с буфером и ячейками в EEPROM.

В **строке 11** определяется переменная для временного хранения кода состояния клавиатуры (*codS*).

И, наконец, в **строке 12** определяется буфер для хранения кодовой последовательности. Буфер представляет собой массив с именем *bufr* и размером *bsize*.

В **строке 13** начинается описание функции *setup()*. В теле функции производится настройка режимов всех используемых контактов модуля Ардуино.

Листинг 17.1. Программа электронного замка

```
   /*
     Кодовый замок
   */

1  #include <EEPROM.h>

2  #define klfree 0x3FF     // код «Все кнопки отпущены»
3  #define kontrTime 1000   // контрольн. промежуток времени
4  #define kandr 30         // Константа антидребезга
5  #define bsize 30       // Размер буфера для хранения кода
6  #define klen bsize*2+1   // Ячейка EEPROM для длинны
                            // кодовой последовательности
7  #define modePin 11       // Пин переключателя режима
8  #define relayPin 13      // Номер контакта привода замка

9  unsigned char ii;        // Указатель массива
10 unsigned char i;         // Вспомогательный указатель
11 unsigned int codS;       // Старый код

12 unsigned int bufr[bsize]; // Буфер в ОЗУ для хранения кода

13 void setup() {
     // Определение режимов контактов:
14   pinMode(relayPin, OUTPUT);    // выход на исп. механизм
15   pinMode(modePin, INPUT_PULLUP);  // перекл. режимов
     // Цикл настройки режимов контактов кнопок:
16   for (int ButtonPin=0; ButtonPin<=9; ButtonPin++) {
17      pinMode(ButtonPin, INPUT_PULLUP);
     }
   }

18 void loop() {
     // Определение локальный переменных:
19      unsigned long itime;    // хранения текущего времени
20      unsigned int  CodC;     // код состояния клавиатуры
```

```
21 |m1:  while (incod() != klfree); // ожидание отпускания кнопок
22 |     while (incod() == klfree); // ожидание нажатия кнопок
23 |     ii=0;
24 |m2:  delay(50);          // задержка 50 мс
25 |     codS=incod();       // ввод кода и запись, как старого
26 |     bufr[ii++]=codS;  // запись очередного кода в буфер
27 |     if (ii>=bsize) goto m4; // проверка конца буфера
28 |     itime = millis();   // запись текущего времени
29 |m3:  if (incod() != codS) goto m2; // изменилось ли состояние
30 |     if (millis()-itime< kontrTime) goto m3;
   |                                   // пров.контр.времени
31 |m4:  // Проверка перекл. режимов
   |     if (digitalRead(modePin)==HIGH) goto comp;
   |
   |//----------------------------- Запись кода в EEPROM
32 |     EEPROM.write(klen,ii);        // Запись длины кода
   |     // Запись всех байтов кода
33 |     for (i=0; i<ii; i++)  {
34 |        EEPROM.write(i*2,highByte(bufr[i]));
35 |        EEPROM.write(i*2+1,lowByte(bufr[i]));
   |     }
36 |     goto zamok;
   |
   |//----------------------------- Проверка кода
37 |comp:   // Проверка длины кода:
   |     if (EEPROM.read(klen)!=ii) goto m1;
38 |        for (i=0; i<ii; i++)  {
39 |          CodC = EEPROM.read(i*2)*256 +
   |                              EEPROM.read(i*2+1);
40 |         if (CodC!=bufr[i]) goto m1;// Проверка самого кода
   |        }
   |
   |//----------------------------- Открывание замка
41 |zamok:  digitalWrite(relayPin,HIGH);  // Открываем замок
42 |        delay(1000);            // Задержка на 1 сек
43 |        digitalWrite(relayPin,LOW);  // Закрываем замок
   |}
```

```
   // Функция опроса клавиатуры и антидребезга
44 unsigned int incod (void)  {
45    unsigned int cod0=0;   // Локальные переменные
46    unsigned int cod1;
47    byte k;

   // --------------------------- Цикл антидребезга
48    for (k=0; k<kandr; k++)  {
49        cod1=PINB&0x3;   // Формируем первый байт кода
50        cod1=(cod1<<8)+PIND;  // Формируем полный
                               // код состояния клавиатуры
51        if (cod0!=cod1)    // Сравниваем со старым кодом
          {
52           k=0;        // Если не равны, сбрасываем счетчик
53           cod0=cod1;  // И присваиваем новое значение
                         // старому коду
          }
       }
54    return cod1;
   }
```

В **строке 14** контакт *relayPin* (управление исполнительным механизмом) переводится в режим вывода информации.

В **строке 15** определяется режим работы контакта переключателя режимов (*modePin*).

В **строках 16, 17** находится цикл, который определяет режимы работы всех десяти контактов, предназначенных для подключения цифровых кнопок.

Основной цикл программы *loop()* начинается в **строке 18**. В теле функции *loop()* выполняются практически все основные действия программы. Отдельно выполняется лишь задача считывания кода состояния кнопок.

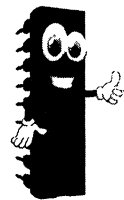
Для считывания кода состояния используется функция *incod()*. Функция считывает информацию с кнопок и возвращает код состояния, в котором, как уже говорилось, каждый из десяти младших битов отвечает за состояние своей конкретной кнопки. Функция *incod()* так же выполняет чистовой этап борьбы с дребезгом контактов.

Основная функция *loop()* начинается с определения двух локальных переменных. Переменная *itime* (см. **строку 19**) введена для того, чтобы можно было запоминать текущее значение системных часов. Это значение затем используется для определения временных интервалов.

Переменная *CodC* (см. **строку 20**) используется как вспомогательная в операции сравнения кодов.

В программе широко используются метки. При помощи меток решена вся логика работы основной задачи. Надеюсь, что вы еще не забыли: метка используется для того, чтобы переходить на строку, следующую непосредственно за меткой из других мест программы.

Так, **строка 21** помечена меткой «*m1:*». Кроме метки, в **строке 21** находится цикл ожидания отпускания всех кнопок. Если в момент запуска программы одна или несколько кнопок окажутся нажатыми, то это может означать, что набор кодов начался преждевременно и часть нажатий уже пропущено.



ПРИМЕЧАНИЕ.

Чтобы предотвратить некорректный набор, работа программы приостанавливается до тех пор, пока все кнопки не будут отпущены.

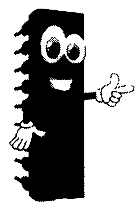
Цикл ожидания отпускания кнопок многократно обращается к функции *incod()*, которая возвращает код состояния кнопок. Полученный код сравнивается с константой *klfree*, значение которой равно коду полностью отпущенных кнопок. Цикл длится до тех пор, пока эти два кода отличаются друг от друга. Когда коды окажутся одинаковыми, цикл закончится.

Далее, в **строке 22** находится другой цикл — цикл **ожидания нажатия хотя бы одной из кнопок**. Он очень похож на цикл из **строки 21**. Отличается только условием. Цикл длится до тех пор, пока код состояния кнопок равен значению *klfree*.

После обнаружения факта нажатия хотя бы одной кнопки программа входит в цикл записи кодовой после-

довательности в буфер. Для начала обнуляется указатель буфера (**строка 23**).

В **строке 24** выполняется небольшая задержка на 50 миллисекунд. Это первый уровень антидребезга. В строке 24 еще раз считывается код состояния кнопок (уже чистый от дребезга). Считанный код записывается в переменную *codS*. Эта переменная предназначена для хранения старого значения кода.



ПРИМЕЧАНИЕ.

Далее мы будем многократно перечитывать код, и сравнивать с этим старым значением.

Считанное значение кода состояния записывается также и в первую ячейку буфера (**строка 26**). Одновременно значение указателя буфера увеличивается на единицу (благодаря оператору ++). Двойной плюс, как вы можете видеть, стоить после имени переменной. Это значит, что сначала происходит запись значения *codS* в элемент номер *ii*, и только затем происходит инкрементация *ii*. Новое значение кода состояния будет записываться уже в следующую ячейку буфера.

В **строке 27** производится проверка конца буфера. Если пользователь попытается ввести слишком длинную кодовую последовательность, при достижении конца буфера ввод кода прекращается. Оператор *if* в **строке 27** сравнивает значение указателя *ii* с заданным размером буфера. И если конец достигнут, передает управление по метке *m4*, в конец операции ввода кодовой последовательности.

Если указатель *ii* не достиг конца буфера, то программа переходит к **строке 28**. В этой строке считывается и запоминается в переменную *itime* текущее системное время.

Далее, в **строке 29** считывается новое значение кода состояния кнопок и сравнивается со старым. Смысл этой операции — ожидание изменения кода состояния. Пока код состояния при каждом считывании будет неизменным, условие оператора *if* в

строке 29 будет иметь значение «Ложь» и программа каждый раз будет переходить к **строке 30**.

В этой строке происходит проверка контрольного времени. Для этого считывается новое значение системного времени, из него вычитается время, сохраненное в **строке 28**, и полученная разность сравнивается с величиной контрольного промежутка времени.

Если контрольное время не достигнуто, то управление передается к метке *m3*, и снова выполняется **строка 29**. В ней снова считывается код состояния кнопок и сравнивается со старым значением. И снова, если значение не изменилось, в **строке 30** производится проверка на истечение контрольного промежутка времени.

Из этого цикла **два выхода**:

- **во-первых**, если истечет контрольный промежуток времени, программа перестанет переходить к *m3*, и перейдет к **строке 31** (то есть закончит процедуру ввода кодовой последовательности);
- **во-вторых**, если в **строке 29** программа обнаружит факт изменения кода состояния.

Тогда произойдет переход по метке *m2*. Там новый код запишется как старый (в переменную *codS*), затем он запишется в очередную ячейку буфера, далее произойдет проверка буфера на исчерпание его объема (**строка 27**) и, наконец, в **строке 28** будет считано и записано в переменную новое текущее значение системного времени.

Далее программа войдет в новый цикл ожидания изменения состояния кнопок с проверкой исчерпания контрольного периода времени.

Как бы ни завершился процесс ввода кодовой последовательности, в любом случае управление переходит к **строке 31**. В этой строке находится команда, которая проверяет состояние переключателя режимов. Если контакты переключателя не замкнуты (значение, считанное с контакта *modePin* равно HIGH), это означает, что включен режим «Работа».

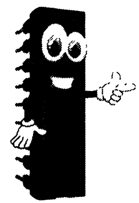
Управление передается по метке *comp*, к процедуре сравнения кодовых последовательностей. Эта процедура зани-

мает **строки 37...40**. Если контакты переключателя режимов замкнуты, то управление переходит к **строке 32**, где начинается процедура записи кодовой последовательности в EEPROM.

Выполнив одну из этих процедур (записи или сравнения кодовых последовательностей) программа переходит к небольшой процедуре, завершающей выполнение основного цикла, которая начинается в **строке 41**. Эта маленькая процедура, отмеченная меткой ***zamok*:**, вызывает срабатывание механизма замка, и удержание его в открытом состоянии в течение 1 секунды.

Рассмотрим все три описанные выше процедуры по порядку.

Процедура записи в EEPROM начинается в **строке 32**. В этой строке в ячейку *klen* долговременной памяти EEPROM записывается значение длины ключевой кодовой последовательности. После завершения процедуры ввода кода длина последовательности содержится в переменной *ii*.



ПРИМЕЧАНИЕ.

Это происходит потому, что указатель буфера (ii) в процессе ввода кодов всегда указывает на ячейку буфера, куда должен записываться очередной код.

Учитывая, что нумерация ячеек начинается с нуля, значение этой ячейки всегда равно текущей длине кодовой последовательности.

В **строке 33** начинается цикл записи кодов из буфера в ячейки EEPROM. В качестве указателя цикла используется переменная *i*. При помощи этого указателя мы будем считывать коды из буфера, начиная с элемента 0 и заканчивая последним элементом введенной кодовой последовательности.

Каждый считанный код мы разобьем на два байта (старший и младший) и запишем каждый из них в долговремен-

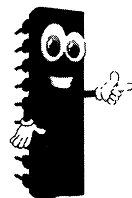
ную память в две последовательные ячейки. А начнем запись с ячейки номер 0. Эта операция выполняется в два этапа.

В **строке 34** в ячейку памяти с адресом *i*2* записывается старший байт очередного элемента буфера. Для извлечения старшего байта из очередного элемента буфера используется стандартная функция языка Ардуино *highByte()*.

В **строке 34** в ячейку долговременной памяти с адресом *i*2+1* записывается младший байт значения текущего элемента буфера. Для получения младшего байта используется функция *lowByte()*. Таким образом, нулевой элемент массива (*i=0*) будет разделен на два байта. Старший байт будет записан в ячейку с номером *0*2=0*. Младший в ячейку с номером *0*2+1=1*.

Следующий элемент буфера (*i=1*) будет разделен и записан так: старший байт в ячейку с номером *1*2=2*, младший байт в ячейку с номером *1*2+1=3*. И так далее, для всех кодов последовательности.

После записи всей кодовой последовательности программа переходит к процедуре открывания замка по метке «zamok:». Замок на 1 секунду открывается и закрывается. Это служит индикатором того, что ввод кодовой последовательности успешно завершен.



ПРИМЕЧАНИЕ.

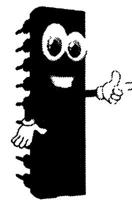
Такой способ индикации выбран для того, чтобы ни вводить в схему лишних светодиодов.

Процедура проверки кодов начинается в **строке 37**. В этой строке из ячейки *klen* долговременной памяти (EEPROM) считывается значение длины кодовой последовательности, записанной туда ранее в режиме «Запись кода». Оно сравнивается с только что полученным значением длины уже новой кодовой последовательности, хранящейся в буфере.

Если два этих значения не равны, то это значит, что код в буфере набран неправильно (не та длина). Управление передается по метке *m1* в самое начало программы. Открывание замка не происходит.

В случае если первая проверка прошла успешно (длина кода совпадает), программа переходит к сравнению самих кодовых последовательностей. Цикл сравнения формируется в **строке 38**. В качестве указателя цикла так же используется переменная *i*.

В **строке 39** считываются оба байта очередного кода последовательности. Значение старшего байта умножается на 256 и к нему прибавляется значение младшего байта. В результате мы получаем искомый код из очередной позиции кодовой последовательности. Этот код записывается в переменную *CodC*.



ПРИМЕЧАНИЕ.

Нужно отметить, что язык Ардуино предлагает специальную функцию объединения старшего и младшего байтов ***word(СтаршийБайт, МладшийБайт).***

Если использовать эту функцию, то **строка 39** программы может выглядеть следующим образом:

*CodC = word(EEPROM.read(i*2), EEPROM.read(i*2+1));*

В **строке 40** значение *CodC* сравнивается со значением текущего элемента буфера. Они должны совпадать. Если значения не совпадают, управление тут же передается по метке *m1* и открывание замка не происходит.

И только в том случае, когда цикл полностью переберет и сравнит все коды последовательности, все они окажутся попарно равными, цикл завершится нормальным образом и управление плавно перейдет к **строке 41**, то есть к процедуре открывания замка.

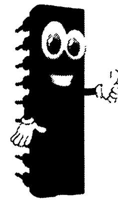
<u>Процедура открывания замка</u> выполняется как в режиме «Запись кода», так и в режиме «Работа». Только при «Записи кода» процедура выполняется всегда по завершении набора кодовой последовательности. И просто индицирует то, что набранная кодовая последовательность благополучно записана в память.

В режиме «Работа» открывание замка происходит только в случае полного совпадения введенной кодовой комбинации с комбинацией, записанной ранее в EEPROM. Процедура открывания замка очень проста.

В **строке 41** на выход исполнительного механизма (*relayPin*) подается сигнал высокого логического уровня. При этом загорается светодиод на плате Ардуино, помеченный литерой «L». Одновременно срабатывает электронный ключ (если, конечно, вы его подключили). Через ключ подается питание на исполнительный механизм и открывается замок.

В **строке 42** формируется задержка длительностью в 1 секунду. За это время вы должны успеть открыть (или хотя бы приоткрыть) дверь так, чтобы замок уже не мог защелкнуться.

В **строке 43** на выход *relayPin* подается низкий логический уровень. Исполнительный механизм отпускает щеколду замка.



СОВЕТ.

Если одной секунды вам не достаточно, чтобы успеть открыть дверь, вы можете увеличить значение задержки в ***строке 42****.*

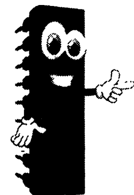
Функция опроса клавиатуры и антидребезга выполняет две задачи:

- считывает состояние всех кнопок клавиатуры и формирует код состояния;
- реализует второй уровень антидребезговой защиты.

Для формирования кода состояния программа использует прямое чтение из регистров. Из **главы 2** нашей книги, а также из схемы распиновки модуля Ардуино (см. **рис. 2.3**), мы знаем, что цифровые контакты с 0 по 7 подключены к разрядам PD0...PD7 порта PD основного микроконтроллера ATmega323.

Именно к этим контактам в нашей схеме подключены кнопки с «0» по «7». Еще две кнопки подключены к контактам 8 и 9 модуля. Эти два контакта подключены уже к другому порту

микроконтроллера. Контакт 8 подключен к разряду PB0, а контакт 9 к разряду PB1 порта PB.

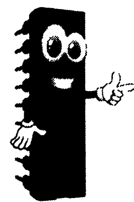


ПРИМЕЧАНИЕ.

Для того чтобы получить состояние сразу всех кнопок с 0 по 7, можно просто прочитать содержимое порта PD. Прочитав восьмиразрядное число из этого порта, мы получим код, у которого каждый бит будет отражать состояние своей кнопки. Еще два бита, соответствующих оставшимся двум кнопкам («8» и «9»), можно получить, прочитав содержимое порта PB.

Два младших бита полученного числа будут соответствовать оставшимся двум кнопкам. При этом нам придется обнулить шесть старших битов прочитанного числа для того, чтобы случайные сигналы не влияли на результат. Для этого можно наложить на число соответствующую маску.

Затем два кода (считанный из порта PD и считанный из порта PB) объединяются в одно двухбайтовое число и записываются в переменную типа *int*. Такой способ работы с входной информацией называется «**работа на уровне портов**».



ПРИМЕЧАНИЕ.

*Этот способ **не характерен для программ, написанных в программной среде Ардуино**, но поддерживается в ней. Работа на уровне портов часто используется в других системах программирования для микроконтроллеров. Например, в среде Code Vision, в программе, описанной в книге **[1]**.*

В нашу программу такой метод перекочевал из книги **[1]**, программу из которой мы взяли за основу. Таким образом, мы

хотели показать, что язык Ардуино достаточно гибок и поддерживает самые разные приемы программирования, что гарантирует высокую переносимость программ.

Функция *incod()* занимает в программе **строки 44...54**. Начинается функция с определения локальных переменных. Переменные *cod0* и *cod1* (см. **строки 45 и 46**) предназначены для временного хранения старого и нового значения кода состояния кнопок.

Переменная *k* (**строка 47**) — это счетчик цикла антидребезга.

В **строке 48** как раз и объявляется цикл антидребезга. Внутри цикла происходит многократное чтение и формирование кода состояния. В нем реализуется алгоритм антидребезга, который состоит в том, что определенное число раз считанный код состояния должен быть неизменным. Число таких считываний определено константой антидребезга *kandr*.

Если при очередном считывании код изменился, то новое значение кода запоминается, а программа заново начинает отсчет последовательности, в которой коды должны быть одинаковыми. Функция завершается и возвращает значение кода состояния кнопок только в том случае, когда этот код не изменялся при считывании его *kandr* число раз подряд.

Собственно считывание и формирование кода происходит в **строках 49 и 50**. Сначала считывается значение из порта *PB* (**строка 49**). На считанное значение накладывается маска *0x3 (00000011)*, которая обнуляет все старшие разряды, оставляя без изменения только два младших. Полученное число записывается в переменную *cod1*.

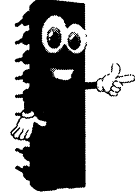
Затем в **строке 50** программа считывает значение из порта *PD*, и складывает его со значением *cod1* сдвинутым на 8 разрядов влево. Результат всех этих действий снова записывается в *cod1*. После всех этих преобразований два разряда, считанные из порта PB, окажутся в разрядах 8 и 9 полученного кода. А разряды с 0 по 7 займет код, считанный из порта PD. Это и будет искомый код состояния кнопок, и храниться он будет в переменной *cod1*.

Далее, в **строке 51** производится сравнение только что считанного кода состояния из *cod1* и старого значения кода, который должен храниться в переменной *cod0*. При первом считывании переменная *cod0* содержит нулевое значение, записанное туда при создании этой переменной в **строке 45** программы. Это равносильно тому, что код состояния изменился. Поэтому счетчик цикла сразу же будет обнулен, и цикл начнется сначала.

Это произойдет следующим образом: в **строке 51** сравниваются значения переменных *cod0* и *cod1*. Если они не равны, программа выполняет команды в **строках 52 и 53.** В **строке 52** счетчику цикла антидребезга *k* присваивается нулевое значение. Таким образом, цикл начнется сначала.

В **строке 53** в переменную *cod0* записывается значение переменной *cod1*. Теперь новое значение кода будет использоваться в качестве старого. Цикл в **строке 48** будет повторяться *kandr* раз в том случае, если все эти разы *cod1* будет равен *cod0*. Как только новый код окажется не равным старому, счетчик *k* опять обнулится и цикл начнется сначала.

Если код ни разу не изменялся, цикл завершается, и программа переходит к **строке 54.** В этой строке находится оператор *return* (возврат). Параметром оператора является возвращаемое значение. Оператор *return* завершает работу функции *incod()*, и возвращает значение *cod1*.



ПРИМЕЧАНИЕ.

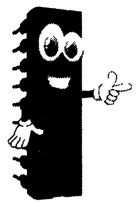
При желании функцию incod() можно написать, не используя прямой доступ к регистрам. В ***листинге 17.2*** *приведен вариант той же функции, в котором для считывания состояния кнопок используется стандартный оператор языка Ардуино* ***digitalRead()****.*

Новая редакция функции *incod()* отличается от старого только в той части, где происходит считывание и формирование кода состояния клавиатуры. Вся операция формирования кода занимает **строки 6, 7 и 8** (см. **листинг 17.2**).

Код состояния формируется в переменной *cod1*. Перед началом операции формирования кода эта переменная обнуляется (см. **строку 6** программы). Затем, в **строке 7**, начинается цикл опроса кнопок. В качестве указателя цикла используется переменная *ButtonPin*.

Переменная в цикле изменяется от 0 до 9 и поочередно указывает на контакты, к которым подключены кнопки. В **строке 8** значение, записанное в переменной *cod1*, сдвигается на один бит влево. При этом к полученному в результате сдвига значению прибавляется бит состояния текущей кнопки, считанный с входа, на который указывает переменная *ButtonPin*.

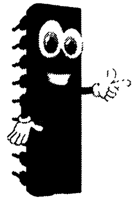
В результате за десять шагов цикла все десять битов состояния кнопок выстроятся в единое число — **код состояния кнопок**. В остальном новая редакция функции *incod()* ничем не отличается от старой.



ВНИМАНИЕ!

*В **листинге 17.2** используется своя локальная нумерация строк, не связанная с нумерацией в **листинге 17.1**.*

Прежде, чем закончить данную главу, несколько слов о таком важном вопросе, как **отладка программ**.



ПРИМЕЧАНИЕ.

К сожалению, среда разработки Ардуино не имеет практически никаких средств для отладки программ. Поэтому программистам приходится выкручиваться.

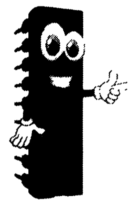
Листинг 17.2. Альтернативный вариант функции опроса клавиатуры и антидребезга

```
   // Функция опроса клавиатуры и антидребезга
1  unsigned int incod (void)  {
2     unsigned int cod0=0;       // локальные переменные
3     unsigned int cod1;
4     byte k;

      // Цикл антидребезга
5     for (k=0; k<kandr; k++) {
6       cod1=0;
7       for (int ButtonPin=0; ButtonPin<=9; ButtonPin++) {
8         cod1=(cod1<<1)+ digitalRead(ButtonPin);
        }
9       if (cod0!=cod1) {    // сравниваем со старым кодом
10        k=0;             // если не равны, сбрасываем счетчик
11        cod0=cod1;       // новое значение старому коду
         }
      }
12    return cod1;
   }
```

Простые программы можно отлаживать, применяя следующие **приемы**:

- написание программы отдельными модулями с опробованием каждого модуля по отдельности;
- применение метода пробных изменений программы, проверяющих разные гипотезы возникновения ошибок;
- внимательный просмотр текста программы с целью понять причину ошибки чисто логически.

В число таких приемов также можно включить временное внедрение в программу специальных отладочных программных блоков, выводящих любым доступным способом промежуточные результаты работы программы.



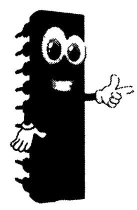
ПРИМЕР.

*В процессе отладки программы электронного замка был применен следующий прием. В программу в отладочных целях был временно введен **блок операторов**, который каждый раз после набора кодовой последовательности выводил в компьютер через последовательный канал содержимое буфера, а также кодовую комбинацию, считанную из EEPROM.*

Эту информацию можно увидеть на компьютере при помощи монитора порта, как мы это делали в **главе 14**. В процессе отладки блок команд перемещался в разные места отлаживаемой программы. Это позволяло наблюдать формирование кодовых последовательностей на разных этапах работы программы.

Для того чтобы кнопки «0» и «1» не мешали при передаче данных по последовательному порту, они были временно отключены на период отладки. После завершения процесса отладки все отладочные изменения из программы были удалены.

В набор электронных версий программных примеров, находящийся на **виртуальном диске**, включена версия программы электронного замка, в которую уже включены отладочные блоки. Скачав пакет электронных версий программных примеров, вы можете, при желании, самостоятельно запустить и попробовать поработать с этим вариантом программы. То есть освоить работу в режиме отладки.



ВНИМАНИЕ.

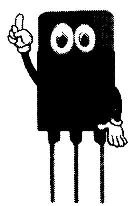
*Версии программ, снабженные отладочными модулями, помечены словом **debug**.*

ГЛАВА 18

КОДОВЫЙ ЗАМОК С МУЗЫКАЛЬНЫМ ЗВОНКОМ

Постановка задачи

В качестве последнего примера покажем, как можно объединить две самые сложные схемы и программы, описанные в этой книге в одну. А именно, описанный в предыдущей главе кодовый замок, мы объединим с программой музыкальной шкатулки, превратив «Музыкальную шкатулку» в **музыкальный дверной звонок.** Не смотря на то, что указанные две программы являются самыми сложными из всех, приведенных в книге программ, их совсем не трудно объединить в одну.



ЗАДАЧА.

Доработать схему и программу кодового замка, внедрив в него алгоритм воспроизведения мелодий, который должен служить в качестве музыкального дверного звонка. Для этого в схему необходимо добавить кнопку звонка и звукоизлучающее устройство. При этом кодовый замок должен работать так же, как он работал при

отсутствии функции звонка. Однако при нажатии кнопки звонка устройство должно воспроизводить при помощи звукового излучателя одну из десяти заложенных в память мелодий. При каждом нажатии кнопки звонка должна звучать новая мелодия.

Схема

Схема доработанного электронного замка приведена на ***рис. 18.1****.*

В схему, в соответствии с поставленной задачей, добавлены:

- кнопка звонка, которая подключена к цифровому контакту 10;
- звукоизлучатель (динамик), который подключен так же, как это было сделано в схеме музыкальной шкатулки, к контакту 12.

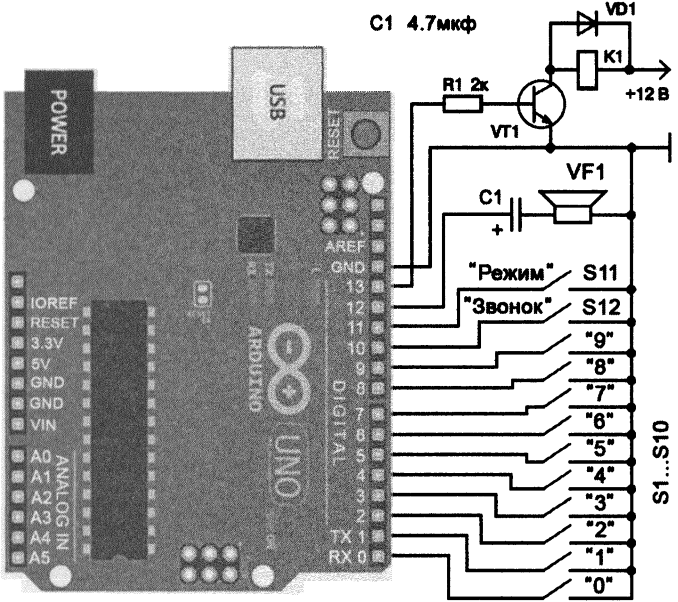


Рис. 18.1. *Схема электронного замка со звонком*

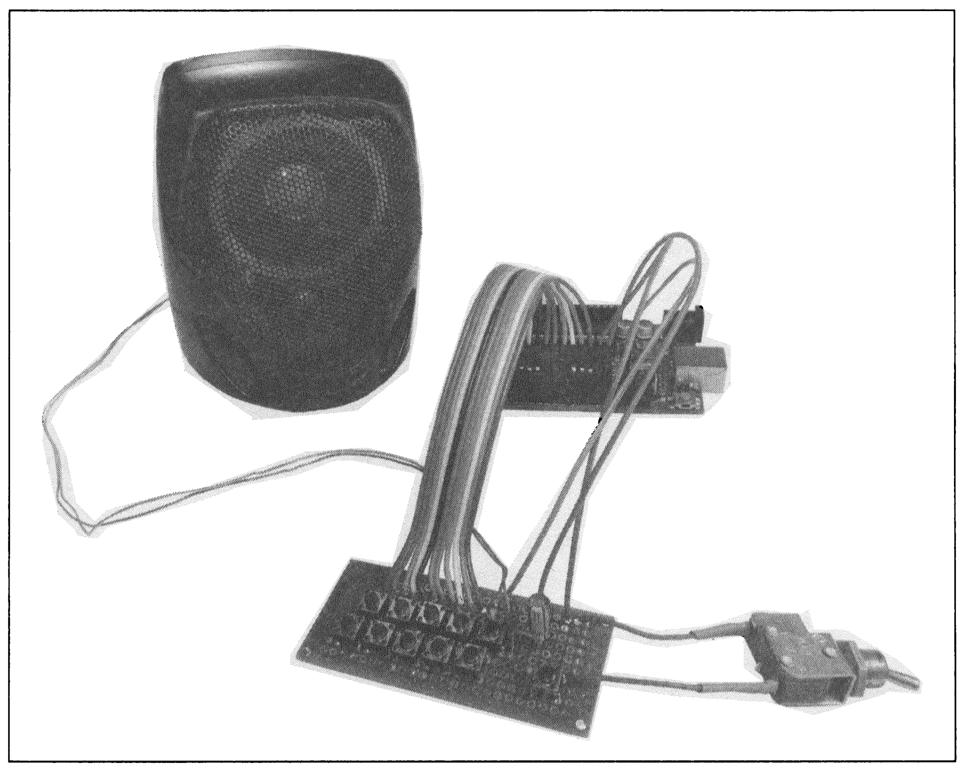


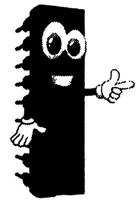
Рис. 18.2. Внешний вид электронного замка со звонком

Внешний вид собранного устройства показан на **рис. 18.2.**

Алгоритм

Алгоритм работы электронного замка и алгоритм воспроизведения музыки в новой программе будут взяты из программ, описанных в **главе 16** и **главе 17**. Они будут использоваться без изменений. Нам нужно только объединить два алгоритма и заставить их работать совместно.

Как мы знаем из **главы 17**, пока не нажата не одна из цифровых кнопок, программа электронного замка постоянно находится в режиме опроса состояния клавиатуры и ожидает нажатия.



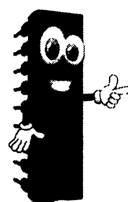
ПРИМЕЧАНИЕ.

Именно в этот цикл опроса нужно включить еще одну проверку: проверку состояния кнопки звонка. В результате цикл по очереди будет проверять то факт нажатия любой из цифровых кнопок замка, то факт нажатия кнопки звонка.

При обнаружении нажатия дальнейшие действия программы зависят от того, что именно было нажато.

Если нажата одна из кнопок замка, то выполняется алгоритм ввода кодовой последовательности и далее весь алгоритм замка.

Если окажется нажатой кнопка звонка, то выполняется алгоритм воспроизведения мелодии.



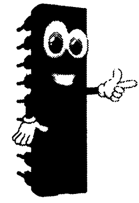
ПРИМЕЧАНИЕ.

От музыкальной шкатулки этот алгоритм отличается тем, что мелодии воспроизводятся по очереди. Номер воспроизводимой мелодии записывается в специально выделенную ячейку в EEPROM. При очередном нажатии кнопки звонка, воспроизводится следующая по порядку мелодия.

Программа

Объединенная программа электронного замка и музыкального звонка приведена в **листинге 18.1**. За основу взята программа электронного замка. Программа воспроизведения мелодии оформлена в виде отдельной функции *playMelody()*, которая находится в конце **листинга 18.1** и занимает **строки 78...96**.

В начале программы в блоке определений всех переменных, констант и массивов присутствуют определения как для одной, так и для второй задачи.



ПРИМЕЧАНИЕ.

Так как большинство строк в блоке определений повторяют аналогичные строки в программах-источниках: программы из ***листинга 17.1*** *(электронного замка), и программы из* ***листинга 16.1*** *(музыкальной шкатулки), мы рассмотрим только те строки описания, которые требуют особых пояснений.*

И **первое отличие** мы видим в **строке 7 листинга 18.1**. Здесь вводится новая константа *numuz*. Этой константой определяется адрес ячейки в EEPROM для хранения номера текущей мелодии. Хранение номера текущей мелодии в энергонезависимой памяти позволит программе даже после выключения питания хранить этот номер.

При очередном нажатии кнопки звонка номер мелодии увеличивается на единицу, и звонок каждый раз воспроизводит новую мелодию. Мелодии нумеруются числами от 0 до 9.

Когда, при очередном нажатии кнопки звонка номер мелодии окажется больше девяти, программа сбрасывает его в ноль. В результате, после девятой мелодии воспроизводится мелодия номер ноль. Номер ячейка *numuz* выбирается на единицу большим, чем номер для хранения длины кодовой последовательности (вычисляется по формуле *klen+1*), так как все предыдущие ячейки в EEPROM уже заняты.

В **строке 8** определяется еще одна новая константа *ringPin*. Это номер контакта для подключения кнопки звонка. Еще одна новая константа *SoundPin* определяется в **строке 10**. Она представляет собой номер контакта для подключения динамика.

В **строках 16...28** мы видим знакомые нам таблицу высоты тона, таблицу длительностей нот и таблицы мелодий.

Листинг 18.1. Программа электронного замка

```
   /*
     Кодовый замок плюс музыкальный звонок
   */

1  #include <EEPROM.h>

2  #define klfree 0x3FF      // код «Все кнопки отпущены»
3  #define kzad 3000         // код задержки при сканировании
4  #define kandr 30          // константа антидребезга
5  #define bsize 30          // размер буфера
      // Два адреса ячеек в EEPROM:
6  #define klen bsize*2+1 // длинна кодовой последовательности
7  #define numuz klen+1      // номер текущей мелодии
      // Номера контактов:
8  #define ringPin 10        // конт. кнопки звонка
9  #define modePin 11        // конт. переключателя режима
10 #define SoundPin 12       // конт. звукоизлучателя
11 #define relayPin 13       // конт. исполнит. механизма
   замка

     // Определение переменных:
12 unsigned char ii;         // Указатель массива
13 unsigned char i;          // Вспомогательный указатель
14 unsigned int codS;        // Старый код


15 unsigned int bufr[bsize]; // Буфер в ОЗУ для хранения кода

   // Таблица длительностей нот
16 const unsigned int tabz[] = {20,40,80,160,320,640,1280,2560};

   // Таблица - частоты нот
17 const unsigned int tabFreq[]={0,587,622,659,698,740,784,
      831,880,932,988,1047,1109,1175,1245,1319,1397,1480,
      1568,1661,1760,1865,1976,2093,2217,2349,2489,2637,
      2794,2960,3136,3322,3520};
```

	// Таблицы мелодий
	// В траве сидел кузнечик
18	byte mel1[] = {109,104,109,104,109,108,108,96,108,104, 108,104,108,109,109,96,109,104,109,104,109,108,108,96, 108,104,108,104,108,141,96,109,111,79,79,111,111,112,80, 80,112,112,112,111,109,108,109,109,96,109,111,79,79,111, 111,112,80,80,112,112,112,111,109,108,141,128,96,255};
	// Песенка крокодила Гены
19	byte mel2[] = {109,110,141,102,104,105,102,109,110,141, 104,105,107,104,109,110,141,104,105,139,109,110,173,96, 114,115,146,109,110,112,109,114,115,146,107,109,110,114, 112,110,146,109,105,136,107,105,134,128,128,102,105,137, 136,128,104,107,139,137,128,105,109,141,139,128,110,109, 176,112,108,109,112,144,142,128,107,110,142,141,128,105, 109,139,128,173,134,128,128,109,112,144,142,128,107,110, 142,141,128,105,109,139,128,173,146,128,255};
	// В лесу родилась елочка
20	byte mel3[] = {132,141,141,139,141,137,132,132,132,141, 141,142,139,176,128,144,146,146,154,154,153,151,149,144, 153,153,151,153,181,128,96,255};
	// Happy births day to you
21	byte mel4[] = {107,107,141,139,144,143,128,107,107,141, 139,146,144,128,107,107,151,148,146,112,111,149,117, 117,148,144,146,144,128,255};
	// С чего начинается родина
22	byte mel5[] = {99,175,109,107,106,102,99,144,111,175,96, 99,107,107,107,107,102,104,170,96,99,109,109,109,109, 107,106,143,109,141,99,109,109,109,109,104,106,171,96, 99,111,109,107,106,102,99,144,111,143,104,114,114,114, 114,109,111,176, 96,104,116,112,109,107,106,64,73,143, 107,131,99,144,80,80,112,111,64,75,173,128,255};

```
   // Из кинофильма «Веселые ребята»
23 byte mel6[] = {105,109,112,149,116,64,80,148,114,64,78,
    146,112,96,105,105,109,144,111,64,80,145,112,64,81,178,
    96,117,117,117,149,116,64,82,146,112,64,79,146,144,96,
    105,105,107,141,108,109,112,110,102,104,137,128,96,105,
    105,105,137,102,64,73,142,105,107,109,64,75,137,96,105,
    105,105,137,102,105,142,112,64,82,180,96,116,116,116,
    148,114,112,142,109,64,78,146,144,96,105,105,107,141,
    108,109,112,110,102,104,169,96,96,255};

   // Улыбка
24 byte mel7[] = {136,133,170,168,131,134,133,131,193,160,
    133,136,138,138,138,140,143,141,140,138,173,200,138,140,
    173,128,140,138,133,136,134,202,160,140,138,141,136,140,
    138,138,136,131,133,163,161,160,129,132,136,136,136,136,
    168,141,129,132,131,131,131,163,131,132,136,134,136,137,
    136,134,136,137,173,171,160,141,136,139,137,137,137,169,
    141,134,137,136,136,136,168,141,132,131,132,134,137,136,
    134,132,134,169,168,160,141,136,139,137,137,137,169,141,
    134,137,136,136,136,168,141,132,131,132,134,137,136,134,
    132,134,168,193,160,255};

   // Гимн России
25 byte mel8[] = {136,173,136,96,106,172,133,96,101,170,
    136,96,102,168,129,96,97,163,131,96,101,166,134,96,104,
    170,140,141,175,136,177,143,96,109,175,140,136,173,140,
    96,106,172,133,96,101,170,136,96,102,168,129,96,97,173,
    140,138,168,224,255};

   // Спят усталые игрушки
26 byte mel9[] = {168,128,133,168,128,133,168,168,166,165,
    163,165,200,143,145,143,145,212,168,128,131,168,128,131,
    168,166,165,163,161,165,200,145,148,145,148,214,168,168,
    170,168,177,177,170,168,161,161,166,168,170,170,168,166,
    165,165,166,165,232,198,195,193,160,224,255};

   // Гамма
27 byte mel10[] = {129,130,131,132,133,134,135,136,137,
    138,139,140,141,142,143,144,145,146,147,148,149,150,
```

```
   151,152,153,154,155,156,157,158,159,128,159,158,
   157,156,155,154,153,152,150,149,148,147,146,145,144,
   143,142,141,140,139,138,137,136,135,134,133,132,131,
   130,129,128,255};

   // Таблица начал всех мелодий
28 byte *tabm[] = {mel1, mel2, mel3, mel4, mel5, mel6, mel7,
         mel8, mel9, mel10};

   // =================
29 void setup() {
     // Инициализация контактов:
30   pinMode(relayPin, OUTPUT);    // конт. механизма замка
31   pinMode(SoundPin, OUTPUT); // конт. звукоизлучателя
32   pinMode(modePin, INPUT_PULLUP);     // перекл. режимов
33   pinMode(ringPin, INPUT_PULLUP);     // кнопка звонка
     // Инициализация пинов цифровых кнопок:
34   for (int ButtonPin=0; ButtonPin<=9; ButtonPin++) {
35     pinMode(ButtonPin, INPUT_PULLUP);
     }
   }

   // =================
36 void loop() {
37     unsigned long itime;
38     unsigned int  CodC;

39 m1:  while (incod() != klfree); // ожидание отпускания кнопок
         // Ожидание нажатия кнопок:
40      while (incod() == klfree) {
41         if (digitalRead(ringPin)==LOW) {
42           byte imel = EEPROM.read(numuz);
                                    // зап.номер мелодии
43           if (imel>9) imel=0;
44           playMelody (imel++);
45           EEPROM.write(numuz,imel);
           }
```

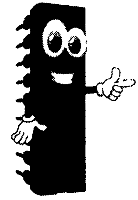
```
   |     }
46 |     ii=0;
47 |m2:  delay(50);                  // задержка 50 мс
48 |     codS=incod();     // ввод кода и запись, как старого
49 |     bufr[ii++]=codS;  // запись очередного кода в буфер
50 |     if (ii>=bsize) goto m4;    // проверка конца буфера
   |       // сканируем состояние кнопок в течении 1 секунды:
51 |     itime = millis();
52 |m3:  if (incod() != codS) goto m2; // изменилось ли состояние?
   |      // Проверка контрольн. промежутка времени:
53 |     if (millis()-itime<1000) goto m3;
   |       // Проверка переключателя режимов:
54 |m4:  if (digitalRead(modePin)==HIGH) goto comp;
   |
   |//------------------------------ Запись кода в EEPROM
55 |     EEPROM.write(klen,ii);       // запись длины кода
   |     // Запись всех байтов кода
56 |     for (i=0; i<ii; i++)  {
57 |        EEPROM.write(i*2,highByte(bufr[i]));
58 |        EEPROM.write(i*2+1,lowByte(bufr[i]));
   |     }
59 |     goto zamok;
   |
   |//------------------------------ Проверка кода
60 |comp:   if (EEPROM.read(klen)!=ii) goto m1;
   |// Проверка длины кода
61 |        for (i=0; i<ii; i++)  {
62 |          CodC = EEPROM.read(i*2)*256 +
   |                                    EEPROM.read(i*2+1);
63 |          if (CodC!=bufr[i]) goto m1;  // проверка кода
   |        }
   |
   |//------------------------------ Открывание замка
64 |zamok:  digitalWrite(relayPin,HIGH);   // открываем замок
65 |        delay(1000);
   |// задержка на 1 сек
66 |        digitalWrite(relayPin,LOW);  // закрываем замок
   |}
```

```
   // Функция опроса клавиатуры и антидребезга
67 unsigned int incod (void)  {
68    unsigned int cod0=0;  // локальные переменные
69    unsigned int cod1;
70    byte k;
   // Цикл антидребезга
71    for (k=0; k<kandr; k++)  {
72        cod1=PINB&0x3;     // формируем первый байт кода
            // Формируем полный код состояния клавиатуры:
73        cod1=(cod1<<8)+PIND;
          // Сравниваем со старым кодом
74        if (cod0!=cod1)  {
75           k=0;    // если не равны, сбрасываем счетчик
76           cod0=cod1;    // новое значение старому коду
          }
      }
77    return cod1;
   }

   // ========================
   // Функция воспроизведения мелодии
78 void playMelody (int numberMelody) {
79   byte count; // определяем переменную (счетчик)
80   byte fnota; // код тона ноты
81   byte dnota; // код длительности ноты
82   byte *nota; // ссылка на текущую ноту

83   nota = tabm[numberMelody];
84   StartMelody:
85   if (digitalRead(ringPin)==HIGH) return;  // Если кнопка
                                    // звонка не нажата, закончить
86   if (*nota==0xFF) return; // проверка на конец мелодии
87   fnota = (*nota)&0x1F;           // определяем код тона
88   dnota = ((*nota)>>5)&0x07; // определяем код длительности
89   if (fnota!=0) {
90      tone (SoundPin, tabFreq[fnota],tabz[dnota]);
91     delay (tabz[dnota]+10);
```

```
       }
92     else  {
          noTone(SoundPin);    // если пауза выкл. звук
93        delay (tabz[dnota]);
94     }
95     nota++;

96     goto StartMelody;
    }
```

В функцию *setup()*, которая занимает **строки 29...35**, включены команды, определяющие режимы всех используемых в программе контактов модуля Ардуино.

Строки 36...66 занимает основной цикл программы (т. е. функция *loop()*). Она практически полностью повторяет основной цикл программы кодового замка (см. **листинг 17.1**).



ПРИМЕЧАНИЕ.

*Отличие только в том, что в цикл ожидания нажатия кнопок (который начинается в **строке 40**) включены пять новых команд (**строки 41...45**), которых не было в оригинальной программе замка.*

В **строке 41** находится команда чтения и контроля состояния кнопки звонка. Если кнопка не нажата (считанное значение равно HIGH), то команды в фигурных скобках оператора *if* не выполняются, и цикл ожидания нажатия кнопок продолжается.

Если нажать одну из цифровых кнопок, то условие в круглых скобках оператора *while* в **строке 40** перестанет выполняться, цикл прервется, а управление перейдет к **строке 46**, то есть к реализации алгоритма кодового замка.

Если же вместо цифровых кнопок нажать кнопку звонка, то в **строке 41** считанное с входа *ringPin* состояние кнопки звонка окажется равным LOW. То есть условие оператора *if* (в **строке 41**) окажется выполнено.

Поэтому управление перейдет к операторам тела функции *if* (**строки 42....45**).

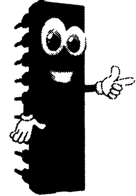
Сначала в **строке 42** из ячейки EEPROM с адресом *numuz* считывается номер текущей мелодии и записывается в переменную *imel*. Этот номер по условиям задачи должен принимать значения от 0 до 9, так как в таблицах мелодий у нас заложено десять мелодий, пронумерованных именно таким образом (начиная с нуля).

Однако при первом запуске программы либо в процессе предыдущей работы переменная может иметь и другие значения. Нам нужно проверить, попадает ли значение в заданный диапазон и исправить его, если не попадает.

Переменная *imel* имеет тип *byte*, а значение этого типа никогда не бывает отрицательным. Поэтому достаточно одной проверки: в **строке 43** проверяется, не больше ли *imel* девяти. Если *imel* больше девяти, то ему присваивается нулевое значение.

В **строке 44** вызывается функция воспроизведения мелодии *playMelody()*. В качестве параметра в функцию передается только что полученный номер мелодии *imel*. Одновременно производится инкрементация номера мелодии при помощи оператора «++».

В **строке 45** новый, увеличенный на единицу, номер мелодии записывается в ячейку *numuz* долговременной памяти EEPROM. При следующем нажатии кнопки звонка будет воспроизводиться уже новая мелодия.

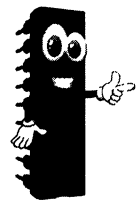


ПРИМЕЧАНИЕ.

Никаких других отличий от оригинальной версии в той части программы, которая реализует функцию электронного замка, нет.

Функция воспроизведения мелодии *playMelody()* начинается в **строке 78**. Она представляет собой копию основного цикла программы музыкальной шкатулки (см. **листинг 16.1**), из которой исключены команды, которые опрашивают кнопки.

Номер мелодии функция получает через свой единственный входной параметр *numberMelody*.



ПРИМЕЧАНИЕ.

Полученная выше объединенная программа рассчитана на то, что пользователь будет в каждый момент времени либо набирать кодовую последовательность, либо звонить в звонок. Если попытаться два этих действия выполнить одновременно, то одно действие будет мешать другому. Нажатие кнопки звонка помешает набору кода.

Мелодия звонка будет звучать до тех пор, пока нажата кнопка звонка, и нажатие цифровых кнопок при этом будет просто игнорироваться.

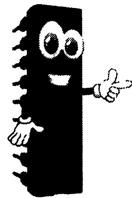
ГЛАВА 19

ПЛАТЫ ARDUINO: ОСОБЕННОСТИ И ВОЗМОЖНОСТИ

Arduino Due

Arduino Due — плата микроконтроллера на базе процессора Atmel SAM3X8E ARM Cortex-M3. Это первая плата Arduino на основе 32-битного микроконтроллера с ARM ядром.

На ней имеется 54 цифровых вход/выхода (из них 12 можно задействовать под выходы ШИМ), 12 аналоговых входов, 4 последовательных порта UART, генератор тактовой частоты 84 МГц, связь по USB с поддержкой OTG, 2 ЦАП, 2 TWI, разъем SPI, разъем JTAG, кнопка сброса и кнопка стирания.



ВНИМАНИЕ!

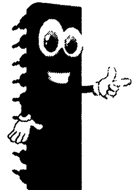
В отличие от других плат Arduino, ***Arduino Due работает от 3,3 В****. Т. е. максимальное напряжение, которое выдерживают вход/выходы, составляет 3,3 В. Подав более высокое напряжение, например, 5 В на выводы Arduino Due,* ***можно повредить плату****.*

Arduino Leonardo

Arduino Leonardo — контроллер на базе ATmega32u4. Платформа имеет 20 цифровых вход/выходов, из них могут использоваться:

- 7 — как выходы ШИМ;
- 12 — как аналоговые входы.

Частота кварцевого генератора 16 МГц. Плата имеет: разъем микро-USB, силовой разъем, разъем ICSP и кнопку перезагрузки.



ПРИМЕЧАНИЕ.

В отличие от всех предыдущих плат, ATmega32u4 имеет ***встроенную поддержку для USB соединения****. Такая поддержка позволяет задать, как Leonardo будет виден при подключении к компьютеру (это может быть клавиатура, мышь, виртуальный COM порт).*

Arduino Yun

Arduino Yun — контроллер на базе ATmega32u4 и AR9331 от фирмы Atheros. Процессор от Atheros поддерживает ОС Linino (версия операционной системы Linux).

Плата имеет встроенные модули Ethernet и WiFi, а также USB-порт, слот для карты микро-SD, 20 цифровых вход/выходов (7 из которых могут использоваться как выходы ШИМ и 12 как аналоговые входы), работает на частоте 16 МГц, разъем микро USB, разъем ICSP и 3 кнопки сброса.

Arduino Micro

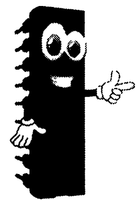
Arduino Micro — плата микроконтроллера на базе ATmega32u4, была разработана совместно с фирмой Adafruit.

Плата имеет 20 цифровых вход/выходов (из них 7 могут использоваться в качестве выходов ШИМ и 12 — как аналоговые входы). Частота кварцевого генератора 16 МГц.

Плата имеет: гнездо микро-USB, разъем ICSP и кнопку перезагрузки. Малый размер контроллера позволяет легко поместить его на макетной плате.

Arduino UNO

Arduino UNO — самая популярная версия базовой платформы Arduino USB. Плата Uno имеет стандартный порт USB.



ПРИМЕЧАНИЕ.

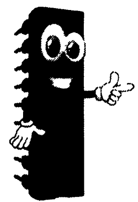
Arduino Uno во многом схожа с Duemilanove, но имеет новый чип ATMega8U2 для последовательного подключения по USB и новую, более удобную маркировку вход/выходов.

Платформа может быть дополнена **платами расширения**, например, пользовательскими платами с различными функциями.

Arduino Ethernet

Arduino Ethernet — это плата микроконтроллера на базе ATmega328. Она имеет 14 цифровых входов/выходов, 6 аналоговых входов. Частота кварцевого генератора 16 МГц. Имеет

возможность подключить: RJ45 разъем, разъем питания, соединитель ICSP и кнопку «Reset».



ПРИМЕЧАНИЕ.

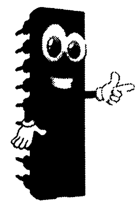
Выводы 10, 11, 12 и 13 зарезервированы для сопряжения с модулем Ethernet и не могут использоваться никак иначе. Таким образом, число доступных выводов уменьшается до 9, 4 из которых могут использоваться как выходы ШИМ.

На плату может быть добавлен дополнительный модуль питания через Ethernet (PoE).

Arduino Duemilanove

Arduino Duemilanove («2009») построена на одном из микроконтроллеров: ATmega168 или ATmega328.

Платформа содержит 14 цифровых входов/выходов (6 из которых могут использоваться как выходы ШИМ), 6 аналоговых входов, кварцевый генератор 16 МГц, разъем USB, силовой разъем, разъем ICSP и кнопку перезагрузки.



ПРИМЕЧАНИЕ.

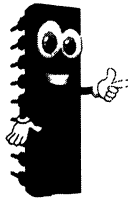
Duemilanove (в переводе с итальянского – 2009) была названа в честь года своего выпуска – 2009 год.

Является предпоследней версией базовой платформы Arduino USB.

Arduino Diecimila

Arduino Diecimila построена на микроконтроллере ATmega168. Платформа содержит 14 цифровых входов/выходов (6 из которых могут использоваться как выходы ШИМ), 6 аналоговых входов, частота кварцевого генератора 16 МГц.

На плате имеются: разъем USB, силовой разъем, разъем ICSP и кнопка перезагрузки.

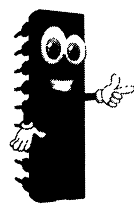


ПРИМЕЧАНИЕ.

*Diecimila (в переводе с итальянского – **10000**) была названа в честь выпуска 10000-ой платы Arduino.*

Arduino Nano

Arduino Nano построенна на микроконтроллере ATmega328 (Arduino Nano 3.0) или ATmega168 (Arduino Nano 2.x), имеет небольшие размеры и может использоваться в лабораторных работах.



ПРИМЕЧАНИЕ.

***Arduino Nano** имеет схожую с Arduino Duemilanove функциональность, однако отличается сборкой. Отличие заключается в отсутствии силового разъема и работе через кабель Mini-B USB.*

Плата Nano разработана и производится компанией Gravitech.

Arduino Mega

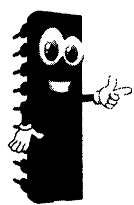
Arduino Mega построена на микроконтроллере ATmega1280. Платформа содержит 54 цифровых входов/выходов (14 из которых могут использоваться как выходы ШИМ), 16 аналоговых входов, 4 последовательных порта UART. Частота кварцевого генератора 16 МГц.

Плата имеет: разъем USB, силовой разъем, разъем ICSP и кнопку перезагрузки. Arduino Mega совместима со всеми устройствами расширения, разработанными для модулей Duemilanove или Diecimila.

Arduino Mega 2560

Arduino Mega 2560 построена на микроконтроллере ATmega2560. Плата имеет: 54 цифровых входа/выхода (14 из которых могут использоваться как выходы ШИМ), 16 аналоговых входов, 4 последовательных порта UART. Частота кварцевого генератора 16 МГц.

Плата содержит: USB коннектор, разъем питания, разъем ICSP и кнопку перезагрузки.

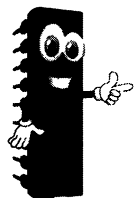


ПРИМЕЧАНИЕ.

Плата Arduino Mega 2560 совместима со всеми платами расширения, разработанными для платформ Uno или Duemilanove.

Arduino ADK

Arduino ADK во многом повторяет Arduino Mega 2560 и построена на микросхеме ATmega2560.



ПРИМЕЧАНИЕ.

Ключевое отличие заключается в наличие USB Host интерфейса, который позволяет подключать контроллер к различным устройствам с интерфейсом USB, включая телефоны и другие устройства на базе Android.

USB Host интерфейс реализован на микросхеме MAX3421e. Также, как Mega 2560, плата имеет 54 цифровых входов/выходов (14 из которых могут использоваться как выходы ШИМ), 16 аналоговых входов, 4 последовательных порта UART. Частота кварцевого генератора 16 МГц.

Плата имеет: USB коннектор, разъем питания, разъем ICSP и кнопка перезагрузки. Последовательное подключение через USB реализовано на микросхеме Atmega8U2, также как в платах UNO и Mega.

Arduino LilyPad

Arduino LilyPad разработана с целью использования как часть одежды. Она может быть зашита в ткань со встроенными источниками питания, датчиками и приводами с проводкой.

Плата построена на микроконтроллере ATmega168V (маломощная версия с ATmega168) или ATmega328V. Arduino LilyPad была создана компаниями Leah Buechley и SparkFun Electronics.

Arduino Fio

Arduino Fio, построенная на микроконтроллере ATmega328P, работает при напряжении 3,3 В, с тактовой частотой 8 МГц.

Плата содержит: 14 цифровых входов и выходов (6 из которых могут использоваться как выходы ШИМ), 8 аналоговых входов, резонатор, кнопку перезагрузки и отверстия для монтажа выводов.

Плата Fio также содержит схему зарядки через разъем USB и позволяет подключить литий-полимерную батарею.

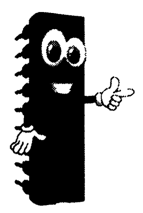
На лицевой поверхности платформы установлен разъем XBee. Arduino Fio может применяться в беспроводных сетях. Загрузка скетчей может производиться через кабель FTDI или плату-конвертер Sparkfun.

Дополнительно имеется возможность загружать скетчи по беспроводной связи при использовании адаптера USB-to-XBee, например, XBee Explorer USB.

Arduino Mini

Arduino Mini построена на микроконтроллере ATmega168 и предназначена для использования в тех случаях, когда требуются минимальные размеры.

Платформа содержит 14 цифровых входов и выходов (6 из которых могут использоваться как выходы ШИМ), 8 аналоговых входов. Частота кварцевого генератора 16 МГц.



ВНИМАНИЕ!

Недопустимы как превышение питающего напряжения, так и переполюсовка выводов питания.

Программируется при помощи адаптера Mini USB или любого преобразователя USB или RS232 в TTL.

Arduino Pro

Arduino Pro построена на одном из микроконтроллеров: ATmega168 или ATmega328. Плата Pro производится в обоих исполнениях 3,3 В / 8 МГц и 5 В / 16 МГц.

Плата содержит: 14 цифровых входов и выходов (6 из которых могут использоваться как выходы ШИМ), 6 аналоговых входов, силовой разъем батареи, силовой выключатель, кнопку перезагрузки, отверстия для монтажа силового разъема, блок ICSP и блоки выводов.

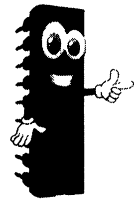
Шестипиновый блок может подключаться к кабелю FTDI или плате-конвертеру Sparkfun для обеспечения питания и связи через USB.

Arduino Pro предназначена для непостоянной установки в объекты или экспонаты. Расположение выводов совместимо с платами расширения Arduino.

Arduino Pro Mini

Arduino Pro Mini построена на микроконтроллере ATmega168. Платформа содержит 14 цифровых входов и выходов (6 из которых могут использоваться как выходы ШИМ), 6 аналоговых входов, резонатор, кнопку перезагрузки и отверстия для монтажа выводов.

Блок из шести выводов может подключаться к кабелю FTDI или плате-конвертеру Sparkfun для обеспечения питания и связи через USB.



ПРИМЕЧАНИЕ.

Плата предназначена для непостоянной установки в объекты или экспонаты. Расположение выводов совместимо с платформой Arduino Mini.

Существует две версии платформы Pro Mini. Одна версия работает при напряжении 3,3 В и частоте 8 МГц, другая — при напряжения 5 В и частоте 16 МГц.

USB Serial Light Адаптер

USB Serial Light Адаптер преобразует USB-канал в последовательный RS-232 канал TTL уровня, который можно подключить непосредственно к Arduino Mini, Arduino Ethernet или другим платам Arduino, не имеющим своего USB адаптера. Обеспечивает указанным платам связь с компьютером и загрузку скетчи.

Логика конвертора реализована на базе чипа Atmega8U2, запрограммированном как конвертер из USB в последовательный сигнал, такой же как на Arduino Uno. Для Windows требуется файл .inf с драйверами.

Плата имеет встроенный разъем мини-USB. Вы также можете использовать 5 выводов: RX (для приема данных с компьютера), TX (для передачи данных), 5V, Ground (общий провод) и Reset (Сброс).

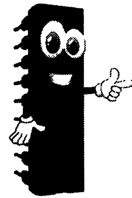
Имеются светодиодные индикаторы питания и активности на линиях RX и TX.

ГЛАВА 20

ARDUINO SHIELDS ИЛИ ПЛАТЫ РАСШИРЕНИЯ

Для чего нужны платы расширения?

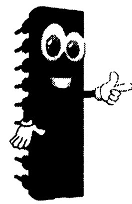
Платы расширения представляют собой законченные устройства, предназначенные для установки на модули Arduino, которые расширяют их функциональность. Плата расширения Ардуино имеет еще англоязычное название **Arduino shield** или просто **шилд**.



ПРИМЕЧАНИЕ.

Англоязычное слово Shield *переводится как щит, экран, ширма. Это устройство, которое как бы покрывает плату контроллера, создает дополнительный слой устройства, ширму, за которой скрываются различные элементы.*

Подключаются платы расширения к основному контроллеру **с помощью стандартных разъемов.** На плате расширения установлены все необходимые электронные компоненты, а взаимодействие с микроконтроллером и другими элементами основной платы происходят через **стандартные пины Arduino.**



ПРИМЕЧАНИЕ.

Чаще всего питание на плату расширения тоже подается с основной платы Arduino, хотя во многих случаях есть возможность запитать плату и от других источников. В любой плате расширения остаются несколько свободных пинов, которые можно использовать по своему усмотрению, подключив к ним любые другие компоненты.

Плата расширения Arduino WiFi

Плата расширения Arduino WiFi позволяет контроллерам Arduino осуществлять сетевое соединение, используя беспроводную сеть формата 802.11. Плата построена на базе чипа HDG104 Wireless LAN 802.11b/g System in-Package.

Микроконтроллер Atmega 32UC3 обеспечивает поддержку сетевого стэка (IP) как для TCP, так и для UDP протокола. Плата WiFi, как и большинство плат расширения, соединяется с платой контроллера Arduino посредством контактных колодок, расположенных по краям платы.

Размеры соответствуют контактам на контроллерах Arduino UNO и Arduino Mega2560. На плате WiFi имеется слот для micro-SD карт, которые могут быть использованы для хранения и передачи файлов по сети.

Плата расширения Xbee Shield

Плата расширения Xbee Shield при помощи модуля Maxstream Xbee Zigbee обеспечивает беспроводную связь нескольким устройствам Arduino в радиусе до 35 метров (в помещении) и до 90 метров (вне помещения).

Плата расширения Arduino Motor

Плата расширения Arduino Motor (управление моторами) изготовлена на основе микросхемы L298. Данная микросхема являющейся двойным полномостовым драйвером. Он разработан специально для управления индуктивными нагрузками, такими как реле, соленоиды, двигатели постоянного тока и шаговые двигатели.

Она позволяет управлять двумя двигателями постоянного тока с помощью вашей платы Arduino, независимо регулируя скорость и направление каждого из них.

Кроме всего прочего, плата позволяет измерять ток, потребляемый каждым двигателем. Плата совместима с модулями TinkerKit.

Плата расширения Ethernet Shield

Плата расширения Ethernet Shield обеспечивает подключение к локальной сети посредством сетевого кабеля (витая пара), а через локальную сеть обеспечивается связь с Интернет.

Для подключения достаточно всего лишь подключить плату расширения к Ардуино, подсоединить ее к сети кабелем RJ-45 и выполнить несколько простых действий.

ГЛАВА 21

ПОДВОДЯ ИТОГИ...

Подводя итоги, мы смеем надеяться, что данная книга дала вам достаточно полное представление о возможностях такого замечательного устройства, как модуль Ардуино. Надеемся, что вы освоили основные приемы схемотехники и принципы составления программ.

Мы постарались осветить как можно полнее все возможности модуля. При этом число примеров оказалось даже больше, чем число примеров использованных в книге [1]. Разобравшись и попробовав самостоятельно собрать и испытать каждую из схем, предложенных в книге вы, как мы надеемся, уже стали достаточным специалистом в этой отдельной, но очень перспективной области микроэлектроники. И теперь уже самостоятельно сможете осуществлять свои собственные разработки и реализовать свои идеи.

Большим подспорьем вам будет **виртуальный диск**, специально разработанный для данной книги. Диск находится в свободном доступе в Интернете на сайте автора книги по адресу ***book.mirmk.ru/wdisk1*** и на сайте Издательства ***www.nit.com.ru.***

Кроме текстов всех программных примеров из книги в электронном виде, инсталляционного пакета среды разработки IDE, архивов всех используемых в книге программных библиотек, на диске вы найдете видеоролики с авторскими обзорами различных вопросов, помогающими полнее освоить материал из книги, а также набор вспомогательной справочной информации.

Автор данной книги будет благодарен за любые отзывы и замечания. Отзывы присылайте по E-mail: *belov@mirmk.ru* или на сайт *http://book.mirmk.ru*.

Приложение 1
Основные операторы языка Ардуино

Главные функции

Оператор	Синтаксис	Описание

setup()	void **setup**()
	{
	Строки программы;
	}
		Функция используется для инициализации переменных, определения режимов работы линий ввода/вывода и т. п. Функция запускается однократно, либо после нажатия кнопки «Сброс», либо сразу после подачи питания, а также сразу после загрузки в модуль новой версии программы

loop()	void **loop**()
	{
	Строки программы;
	}
		Функция **loop**() многократно выполняется в цикле. В теле этого цикла и выполняются все основные операции программы. Функции **setup**() и **loop**() должны присутствовать в каждом скетче, даже если эти функции не используются

Управляющие операторы

Оператор	Синтаксис	Описание

if	**if** (Условие) {
	Оператор;
	}
		Оператор **if** позволяет выполнять ряд операторов в зависимости от Условия. Условие – это математическое выражение, возвращающее значение Ложь (False) или Истина (True). Для этого используется один из операторов сравнения (<, >, ==, !=).

if ... else	**if** (Условие) Оператор; else Оператор;	Оператор **else** дополняет оператор **if** и позволяет выполнить альтернативный набор операторов в случае не выполнения Условия в соответствующем ему операторе **if**.

switch ... case	**switch** (Переменная)
	{
	case Значение1: Оператор;
	case Значение2: Оператор;
	case Значение3: Оператор;
	default: Оператор);
	}
		Программный переключатель. Оператор позволяет задавать действия, которые будут выполняться при разных условиях. Если значение переменной в скобках у слова **swich** равно значению у одного из слов **case**, то оператор(ы) напротив этого слова выполняется(ются). В противном случае соответствующий оператор игнорируется.
		Команда **break**, поставленная в любом месте переключателя, досрочно прекращает перебор case и приводит к выходу из **swich**.
		Оператор(ы) после управляющего слова **default** выполняется(ются), если не выбрана ни одна из опций **case**

Оператор	Синтаксис	Описание

for	**for** (i=0; i <= Значение; i++){
	Оператор;
	Оператор;
	}
		Оператор цикла. В **i** – переменная цикла. Оператор имеет три параметра.
		В первом параметре должно быть указано выражение, которое присваивает переменной цикла начальное значение. Второй параметр – это логическое выражение, при выполнении которого цикл продолжается. В третьем параметре размещается выражение, которое выполняет итерацию цикла. В приведенном примере после каждой итерации переменная **i** увеличивается на единицу

while	**while** (Условие)
	{
	Оператор;
	Оператор;
	}
		Оператор **while** (Пока) используется, как цикл, который будет выполняться, пока Условие в круглых скобках – истина.
		Операторы в теле цикла должны обязательно в какой-то момент так изменять условие, чтобы оно приняло значение Ложь. Иначе программа войдет в бесконечный цикл и зависнет

do … while	**do**
	{
	Оператор;
	Оператор;
	}
	while (Условие);
		Оператор цикла **do** … **while** работает так же, как и цикл **while**, только проверка условия происходит не в начале, а в конце цикла. Такой цикл всегда выполняется хотя бы один раз

break **continue**		Оператор **Break** используется для принудительного выхода из циклов **switch**, **do**, **for** и **while**. Оператор **continue** пропускает оставшиеся операторы в текущей итерации цикла и передает управление в его начало

Операторы цифрового ввода/вывода

Оператор	Синтаксис	Описание
pinMode()	**pinMode(**pin, mode**)**	Установка режима работы цифрового входа/выхода. **pin** — номер контакта. Параметр может принимать значения 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 (а также 14, 15, 16, 17, 18, 19); **mode** — режим работы. Параметр mode может принимать значения: OUTPUT — режим вывода информации (устанавливается по умолчанию); INPUT — режим ввода информации; INPUT_PULLAP — режим ввода с включением внутреннего подтягивающего резистора
digitalRead()	**digitalRead(**pin**)**	Функция чтения из цифрового входа. **pin** — номер контакта (принимает те же значение, что и в двух предыдущих операторах). Функция возвращает значение сигнала на выбранном входе. Возвращаемые значения HIGH или LOW (или, что тоже самое, 1 или 0)
digitalWrite()	**digitalWrite(**pin, value**)**	Оператор вывода информации на цифровой контакт Ардуино. Параметр pin — номер контакта. **pin** может принимать те же значения, что и в предыдущем операторе; **value** — выводимое значение. Value может принимать значение HIGH или LOW (или, что тоже самое, 1 или 0)

Операторы аналогового ввода/вывода

Оператор	Синтаксис	Описание
analogRead()	**analogRead(**An**)**	Функция аналогового чтения с аналогового входа An. Параметр может принимать значения: A0, A1, A2, A3, A4, A5, A6
analogWrite()	**analogWrite(**pin, value**)**	Процедура аналогового вывода. **pin** – номер контакта. **value** – выводимое значение. Вывод производится на цифровые выходы модуля в формате широтно-импульсной модуляции (PWM). Номер контакта (pin) может принимать значение (для Arduino UNO) – 3, 5, 6, 9, 10, 11. Значение параметра value может принимать значения от 0 до 255

Операторы времени

Оператор	Синтаксис	Описание
millis()		Функция возвращает количество миллисекунд, прошедшее с запуска программы
micros()		Функция возвращает количество микросекунд, прошедшее с запуска программы
delay(tz)		Функция программной задержки. Параметр **tz** определяет время задержки в миллисекундах
delayMicroseconds(tz)		Функция программной задержки. Параметр tz определяет время задержки в микросекундах

Расширенный ввод/вывод

Оператор	Синтаксис	Описание
tone()	**tone(**pin, fq**)** **tone(**pin, fq, tz**)**	Процедура генерации звукового сигнала на цифровом выходе. **pin** — вывод, на котором будет генерироваться сигнал. **fq** — частота сигнала в герцах (unsigned int). **tz** — длительность сигнала в миллисекундах (unsigned long). Если не указан, звук генерируется, пока не поступит команда **noTone**()
noTone()	**noTone(**pin**)**	Прекращение генерации звука на указанном контакте
shiftOut()	**shiftOut(**dataPin, clockPin, bitOrder, value**)**	Функция последовательного вывода байта данных. Программно реализует SPI канал, используя любые два цифровых контакта. **dataPin** — контакт, через который побитно выводятся передаваемые данные. **clockPin** — контакт, который используется для вывода сигнала синхронизации. **clockPin** — порядок передачи битов. Принимает значения: MSBFIRST (старший бит вперед), LSBFIRST (младший бит вперед). **value** — передаваемый байт данных. Возвращаемого значения нет
shiftIn()	shiftIn**(**dataPin, clockPin, bitOrder**)**	Функция последовательного ввода байта данных. Работает в паре с функцией **shiftOut()**. Параметры dataPin, clockPin и bitOrder аналогичны соответствующим параметрам **shiftOut.** Функция возвращает считанный байт

Оператор	Синтаксис	Описание
pulseIn()	**pulseIn(**pin, value**)** **pulseIn(**pin, value, timeout**)**	Функция измерения длительности входного импульса. **pin** — контакт, на котором происходит измерение. **value** — полярность измеряемого импульса: Если value = HIGH — измеряется длительность положительного импульса. Если value = LOW, измеряется длительность отрицательного импульса. **timeout** — время ожидания начала импульса (по умолчанию 1 с). Функция возвращает длительность импульса. Функция работает с импульсами длительностью от 10 микросекунд до 3 минут

Работа с последовательным портом

Оператор	Синтаксис	Описание
Serial.begin()	**Serial.begin(**rate)	Инициализация последовательного порта. **rate** — скорость передачи информации. Может принимать значения: 300, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600 или 115200
Serial.print()	**Serial.print(**data)	Передача строки данных в последовательный порт (печать текста в последовательный порт)
Serial.println()	**Serial.println(**data)	Передача строки данных в последовательный порт, завершаемый символом перевода строки
Serial.available()	**Serial.available()**	Функция проверки количества считанных по последовательному порту байт в буфере. Байты помещаются в буфер автоматически по поступлению
Serial.read()	**Serial.read()**	Считывание очередного байта из буфера последовательного порта. Если данные отсутствуют, функция возвращает −1 (минус один)
Serial.write()	**Serial.write(**val) **Serial.write(**str) **Serial.write(**buf, len)	Запись данных в последовательный порт. **val** — число от 0 до 255 (байт). **str** — текстовая строка в виде набора байтов. **buf** — массив данных в виде набора байтов. **len** — длина массива данных
Serial.flush()	**Serial.flush()**	Ожидание окончания процесса передачи данных

Приложение 2. Типы данных в Arduino IDE

Тип	Кол-во байт	Диапазон значений	Примечание
boolean	1 байт	0 или 1	Логическое значение TRUE или FALSE (правда/ложь)
byte	1 байт	0 ... 255	число в диапазоне от 0...255
char	1 байт	−128 ... 127	Хранит код символа в кодировке ASCII
unsigned char	1 байт	0 ... 255	Хранит код символа в кодировке ASCII
int	2 байта	−32768 ... 32767	Целое число со знаком
unsigned int	2 байта	0 до 65535	Целое число без знака
word	2 байта	0 до 65535	То же самое, что unsigned int
long	4 байта	−2 147 483 648 ... 2 147 483 647	
unsigned long	4 байта	0 ... 4 294 967 295	
float	4 байта	−3.4028235E38 ... 3.4028235E38	Числа с плавающей точкой. Точность 6...7 знаков
double	4 байта		То же самое, что float

Примечания:

unsigned char — то же что и **byte**. Для лучшей читаемости удобнее использовать **byte**.

unsigned int — то же самое, что и **word**. Для лучшей читаемости используйте **word**.

int — один из самых распространенных типов данных, который очень часто используется для объявления переменных в скетчах для Arduino.

unsigned long — чаще всего этот тип данных используется для хранения результатов функции **millis()**, которая возвращает количество миллисекунд, прошедших с момента начала работы программы.

float — числа с плавающей запятой не характерны для Arduino. Компилятор обрабатывает программы, использующие этот тип данных работают очень долго. Рекомендуется по возможности избегать применения этого типа данных.

Список литературы

1. *Белов А.В.* Микроконтроллеры AVR от азов программирования до создания практических устройств. — СПб.: Наука и Техника, 2016.— ISBN: 978-5-94387-854-1

2. *Белов А.В.* Программирование микроконтроллеров для начинающих и не только... — СПб.: Наука и Техника. — ISBN: 978-5-94387-867-1

3. *Белов А.В.* Разработка устройств на микроконтроллерах AVR, шаг за шагом от «чайника» до профи. — СПб.: Наука и Техника, 2013. — ISBN: 978-5-94387-825-1

4. *Белов А.В.* Самоучитель разработчика устройств на микроконтроллерах AVR, второе издание. — СПб.: Наука и Техника, 2010 г. — ISBN: 978-5-94387-808-4

5. *Белов А.В.* Самоучитель разработчика устройств на микроконтроллерах AVR.. — СПб.: Наука и Техника, 2008. — ISBN: 978-5-94387-363-8

6. *Белов А.В.* Самоучитель по микропроцессорной технике. Изд. 2. — СПб.: Наука и Техника, 2007. — ISBN: 978-5-94387-190-0

7. *Белов А.В.* Создаем устройства на микроконтроллерах. — СПб.: Наука и Техника, 2007. — ISBN: 978-5-94387-364-3

8. *Белов А.В.* Микроконтроллеры AVR в радиолюбительской практике. — СПб.: Наука и Техника, 2007. — ISBN: 978-5-94387-365-2

Список ссылок на ресурсы в интернет

1. http://book.mirmk.ru

Сайт поддержки всех книг автора (Белова Александра). Здесь вы найдете дополнительные материалы к книге, тексты всех программных примеров в электронном виде, описание других книг на ту же тему.

2. https://www.arduino.cc/

Сайт разработчика проекта Ардуино (англоязычный). На этом сайте вы можете подробно прочитать обо всех вариантах модуля Ардуино, а также скачать пакет для установки интегрированной среды разработчика (IDE) и любую его Бетта версию.

Издательство «Наука и Техника»

КНИГИ ПО КОМПЬЮТЕРНЫМ ТЕХНОЛОГИЯМ, МЕДИЦИНЕ, РАДИОЭЛЕКТРОНИКЕ

Scan & DjVu
Bookingolz

Уважаемые читатели!

Книги издательства «Наука и Техника» вы можете:

- **заказать в нашем интернет-магазине**

www.nit.com.ru (более 100 пунктов выдачи на территории РФ)

- **приобрести в Москве:**

«Новый книжный» Сеть магазинов	тел. (495) 937-85-81, (499) 177-22-11
ТД «БИБЛИО-ГЛОБУС»	ул. Мясницкая, д. 6/3, стр. 1, ст. М «Лубянка» тел. (495) 781-19-00, 624-46-80
Московский Дом Книги, «ДК на Новом Арбате»	ул.Новый Арбат, 8, ст. М «Арбатская», тел. (495) 789-35-91
Московский Дом Книги, «Дом технической книги»	Ленинский пр., д.40, ст. М «Ленинский пр.», тел. (499) 137-60-19
Московский Дом Книги, «Дом медицинской книги»	Комсомольский пр., д. 25, ст. М «Фрунзенская», тел. (499) 245-39-27
Дом книги «Молодая гвардия»	ул. Б. Полянка, д. 28, стр. 1, ст. М «Полянка» тел. (499) 238-50-01

- **приобрести в Санкт-Петербурге:**

Санкт-Петербургский Дом Книги	Невский пр. 28, тел. (812) 448-23-57
Буквоед. Сеть магазинов	тел. (812) 601-0-601

- **приобрести в регионах России:**

г. Воронеж, «Амиталь» Сеть магазинов	тел. (473) 224-24-90
г. Екатеринбург, «Дом книги» Сеть магазинов	тел. (343) 289-40-45
г. Нижний Новгород, «Дом книги» Сеть магазинов	тел. (831) 246-22-92
г. Владивосток, «Дом книги» Сеть магазинов	тел. (423) 263-10-54
г. Иркутск, «Продалить» Сеть магазинов	тел. (395) 298-88-82
г. Омск, «Техническая книга» ул. Пушкина, д.101	тел. (381) 230-13-64

Мы рады сотрудничеству с Вами!