

# Автомато-программато-компарадио-кружок

Артём “avp” Попцов  
memory-heap.org

31 августа 2023 г.  
Версия 1.0.0  
68f3cec1293eff41fa3c9bec08a12d33bec1d630



# Оглавление

<b>Вступление</b>	<b>7</b>
Для кого предназначена данная книга? . . . . .	7
Благодарности . . . . .	7
Исходный код . . . . .	8
Лицензия . . . . .	8
<b>1 Основные принципы электротехники</b>	<b>9</b>
1.1 Напряжение . . . . .	9
1.2 Электрическая цепь . . . . .	10
1.3 Разность потенциалов . . . . .	11
1.4 Сопротивление . . . . .	12
1.5 Сборка электрической цепи . . . . .	16
<b>2 Диалоги с компьютером</b>	<b>23</b>
2.1 Введение . . . . .	23
2.2 Алгоритмы . . . . .	23
2.3 Платформа Arduino . . . . .	25
2.4 Работа с макетной платой . . . . .	25
2.5 Подключение Arduino к компьютеру . . . . .	26
2.6 Основы работы с мультиметром . . . . .	26
2.7 Знакомство со средой разработки Arduino . . . . .	28
2.8 Структура программы на Arduino . . . . .	30
2.8.1 Задачи . . . . .	30
2.9 Переменные . . . . .	31
2.10 Константы . . . . .	33
2.11 Управляющие конструкции . . . . .	33
2.11.1 Условия . . . . .	33
2.11.2 Циклы . . . . .	34
2.11.3 Зачем столько видов циклов? . . . . .	36
2.12 Задачи . . . . .	36
<b>3 Белый шум</b>	<b>37</b>
3.1 Виды сигналов . . . . .	37
3.2 Последовательный порт . . . . .	38
3.2.1 Основы работы с Arduino через последовательный порт . . . . .	38
3.2.2 Передача данных на компьютер . . . . .	39
3.3 Аналоговые порты . . . . .	40
3.4 Аналогово-цифровое преобразование . . . . .	42

<b>4 Широтно-импульсная модуляция</b>	<b>47</b>
4.1 Общее описание принципов работы . . . . .	47
4.2 Длина волны . . . . .	47
4.3 Коэффициент заполнения . . . . .	48
4.3.1 Задачи . . . . .	51
<b>5 Синтез музыки и технологии</b>	<b>53</b>
5.1 Звук . . . . .	53
5.2 Подключение динамика . . . . .	55
5.2.1 Задачи . . . . .	55
5.3 Теория ритма . . . . .	57
5.3.1 Понятие <i>такта</i> . . . . .	57
5.3.2 Более сложные ритмы . . . . .	59
5.3.3 Музыкальная запись ритма . . . . .	59
5.3.4 Темп музыки . . . . .	60
5.4 Базовые принципы благозвучия . . . . .	65
5.5 Октачная система . . . . .	65
5.6 Программирование простых мелодий . . . . .	67
5.7 Массивы . . . . .	69
5.7.1 Двумерные массивы . . . . .	73
5.8 Нотный стан . . . . .	74
5.9 Паузы в музыке . . . . .	76
5.10 Ноты с точками . . . . .	78
5.11 Полутона, диезы и bemоли . . . . .	79
5.12 Музыкальный размер . . . . .	82
5.13 Басовый ключ . . . . .	86
5.14 Создание музыкальной группы . . . . .	87
5.14.1 Выбор названия . . . . .	87
5.14.2 Выбор дебютной композиции . . . . .	87
5.14.3 Реализация мелодии . . . . .	87
5.14.4 Синхронизация мелодий . . . . .	90
<b>6 Разработка игр</b>	<b>97</b>
6.1 ЖК-дисплей . . . . .	97
6.1.1 Подключение дисплея . . . . .	98
6.1.2 Вывод текста . . . . .	98
6.2 Выбор жанра и разработка сюжета . . . . .	100
6.3 Отображение игрового персонажа . . . . .	100
6.4 Реализация управления . . . . .	101
6.4.1 Подключение кнопки . . . . .	102
6.4.2 Обработка нажатий . . . . .	104
6.4.3 Ограничение движения игрока . . . . .	107
6.5 Реализация игровой карты . . . . .	109
6.5.1 Взаимодействие игрока с объектами . . . . .	112
6.5.2 Сбор предметов с карты . . . . .	115
6.5.3 Улучшение отрисовки карты . . . . .	115
6.5.4 Вспомогательные функции для работы с картой . . . . .	117
6.6 Добавление звуков . . . . .	118
6.6.1 Задачи . . . . .	119
6.7 Графика . . . . .	119

6.7.1	Использование готовых символов . . . . .	119
6.7.2	Создание собственных символов . . . . .	120
6.7.3	Задачи . . . . .	123
6.8	Анимация . . . . .	123
6.8.1	Добавление необходимых символов . . . . .	123
6.8.2	Реализация анимации . . . . .	123
6.9	Игровая логика . . . . .	124
6.9.1	Условие выигрыша . . . . .	124
6.9.2	Условие проигрыша . . . . .	125
6.9.3	Дополнительные квесты . . . . .	127
6.9.4	Подсчёт очков . . . . .	129
6.9.5	Отображение жизней персонажа с помощью светодиодов .	131
<b>Предметный указатель</b>		<b>132</b>
<b>Словарь терминов</b>		<b>134</b>
<b>A</b>	<b>Октачная система</b>	<b>137</b>
<b>B</b>	<b>Ноты мелодии “Кабы небыло зимы”</b>	<b>139</b>
<b>C</b>	<b>Мелодия “Twinkle Twinkle Little Star” (версия 1.0)</b>	<b>141</b>
<b>D</b>	<b>Мелодия “Twinkle Twinkle Little Star” (версия 2.0)</b>	<b>145</b>



# Вступление

## Для кого предназначена данная книга?

Дорогой читатель, добро пожаловать в наш уютный кружок технического творчества. Здесь мы учимся работать со звуком, светом, электричеством, используя наши знания для создания неожиданных, интересных и практически полезных проектов. Мы постараемся сделать ваш путь в мир электроники и программирования как можно более интересным и лёгким. Но и на вас лежит определённая ответственность – во-первых, без вашего активного участия наши усилия могут не дать желаемого эффекта. Во-вторых, мы учимся вместе с вами, и ты,уважаемый читатель, являешься активным участником работы над этой книгой. Если найдёшь ошибки или опечатки, не стесняйся писать нам по указанным в книге контактам – мы постараемся всё исправить в следующей версии книги.

Надеемся, что данная книга станет на какое-то время вашей настольной (или хотя бы *около-столной*, но во всяком случае не *под-столной*) книгой, которая поможет постичь искусство программирования, а также отчасти исследовать и понять мир вокруг нас немного лучше, чем вы понимали прежде.

## Благодарности

Основой для книги послужили годы практики в **Нижегородском радиотехническом колледже** (<https://nntc.nnov.ru/>), где велись занятия по программированию микроконтроллеров, а также ведение мастер-классов и занятий в **Нижегородском хакерспейсе “CADR”** (<https://cadrspace.ru/>). Отмечу, что именно CADR стал для меня местом, где я мог не только делиться своим опытом с другими, но и активно учиться техническому творчеству, и это сложно переоценить. Автор выражает благодарность данным организациям за предоставленную возможность к развитию и самореализации!

Также хочется выразить благодарность следующим людям, которые приняли активное участие в разработке книги:

- Денис Киселёв – вклад в разработку отдельных глав книги; вычитка текста, участие в разработке и тестирование примеров, приведённых в книге.
- Сергей Ермейкин – вычитка текста, исправление ошибок.
- Илья Маштаков – вычитка и доработка текста.
- Пётр Третьяков – вычитка текста, масса ценных советов по изложению материала.

- Алина Тараева – вычитка текста, выявление большого количества ошибок.

Само существование этой книги стало возможным благодаря вам, за что огромное спасибо.

## **Исходный код**

Исходный код книги находится по адресу:  
<https://github.com/artyom-poptsov/SPARC>.

## **Лицензия**

Copyright © 2016-2022 Артём “avp” Попцов <[poptsov.artym@gmail.com](mailto:poptsov.artym@gmail.com)>.

Права на копирование сторонних изображений и материалов, использованных в данной работе, принадлежат их владельцам.

Данная работа распространяется на условиях лицензии  
“Attribution–ShareAlike” (“Атрибуция–СохранениеУсловий”) 4.0 Всемирная (CC  
BY-SA 4.0)  
<https://creativecommons.org/licenses/by-sa/4.0/deed.ru>

# Глава 1

## Основные принципы электротехники

В первую очередь нам с вами надо рассмотреть базовые принципы того, как работает электроника, чтобы впоследствии уметь собирать простые схемы.

Начнём с рассмотрения условий, которые необходимо выполнить, чтобы через электрическую цепь шёл ток.

### 1.1 Напряжение

Представим, что у нас есть некоторая ёмкость с водой (см. рисунок 1.1.1.)

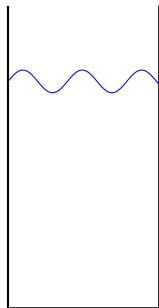


Рис. 1.1.1: Пример ёмкости с водой.

Вода в ёмкости имеет некоторую *потенциальную энергию*, которая может быть потрачена с какой-либо целью. Например, если мы внизу ведра проделаем отверстие, то вода из него будет вытекать (см. рис. 1.1.2); если при этом под струю воды подставить водяное колесо, то таким образом можно приводить в движение механизмы.

Проводя аналогию с электричеством можно сказать, что ёмкость имеет некоторое *напряжение* воды. В электрической батарее как правило запасена *химическая энергия*, которая может быть высвобождена при определённых условиях.

Напряжение в электрической цепи измеряется в *Вольтах* (В).

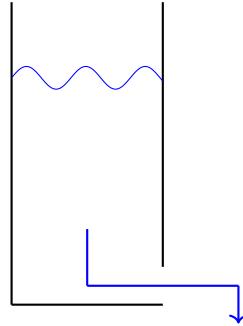


Рис. 1.1.2: Пример ёмкости с водой и отверстием снизу.

Таким образом, мы можем сделать первый вывод: для протекания тока необходим некий источник тока, обладающий некоторым напряжением.

## 1.2 Электрическая цепь

Представим теперь две ёмкости, в одном из которых десять литров воды, в другой – ноль литров (или, как говорят люди, оно пустое), как показано на рисунке 1.2.3.

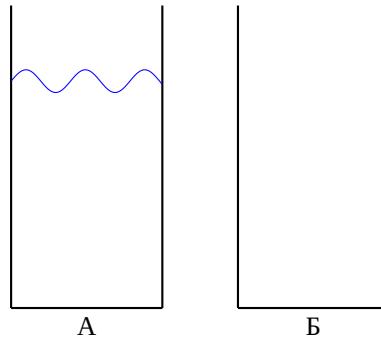


Рис. 1.2.3: Пример двух ёмкостей: с водой (А) и пустая (Б).

Если мы соединим ёмкость “А” и “Б” трубой, то вода потечёт из “А” в “Б”, по законам физики (см. рисунок 1.2.4.)

Соединив две ёмкости, мы получили замкнутую цепь, по которой возможен ток воды. Подобным образом работают электрические цепи, только вместо тока воды в электрических цепях происходит ток элементарных заряженных частиц.

Итак, второе правило, которое мы должны усвоить: электрический ток возможен в замкнутой цепи.

Ток в электрической цепи измеряется в Амперах (А).

Электрическая схема, аналогичная по своей сути рис. 1.2.4, будет выглядеть, как на рис. 1.2.5.

Как можно видеть на рис. 1.2.5, в качестве полезной нагрузки – источника света – используется светодиод. Светодиоды широко используются в современных устройствах в качестве подсветки, индикаторов или источников света.

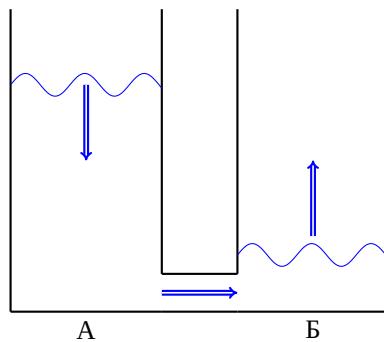


Рис. 1.2.4: Пример двух ёмкостей с разным уровнем воды, соединённые трубкой.

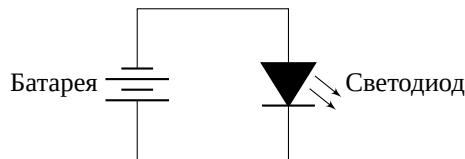


Рис. 1.2.5: Электрическая цепь с батареей в качестве источника напряжения и светодиодом в качестве нагрузки.

Светодиоды имеют разные формы и размеры, но скорее всего вы рано или поздно встретите подобные тем, что показаны на рис. 1.2.6.

У новых светодиодов такого вида обычна длинная ножка является плюсом, также называемым *анодом* (сокращённо “А”). Короткая же ножка является минусом – *катодом* (сокращённо “К”.)

### 1.3 Разность потенциалов

Посмотрим ещё раз на рис. 1.2.4. Вода перетекает из ёмкости А в ёмкость Б по трубе. Как только в уровнях воды в ёмкостях сравняются, то ток воды по трубе остановится (см. рис. 1.3.7.)

Аналогом в электронике этой ситуации, когда ток перестаёт течь, является разряженная батарейка: в процессе эксплуатации химические реакции, дающие разность уровней – или, как говорят в электронике, *разность потенциалов* – в ней постепенно замедляются, вплоть до момента, когда батарейка больше не может давать нужный ток.

Таким образом, вторым условием для электрического тока является наличие разности потенциалов.

Интересным фактом является то, что для протекания электрического тока (как и воды) не обязательно иметь разность потенциалов между каким-то положительным значением и нулём. Проводя опять же аналогию с водой, если мы возьмём две соединённые ёмкости, стоящие на одном уровне, но с разными уровнями жидкости, то ток воды будет, пока уровни жидкости не сравняются в обоих ёмкостях (см. рис. 1.3.8.)

То же самое относится к ёмкостям, стоящим на разных уровнях. В электронике для протекания тока достаточная любая разность потенциалов; примерами

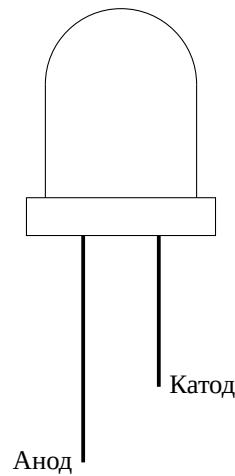


Рис. 1.2.6: Схематическое изображение одного из видов светодиода.

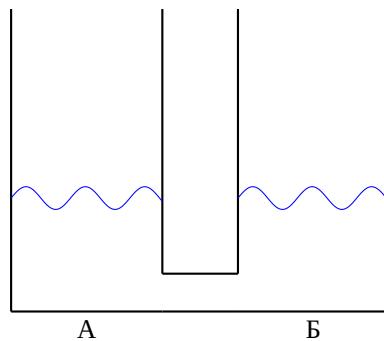


Рис. 1.3.7: Пример двух ёмкостей с равным уровнем воды.

разных потенциалов могут быть 5В и 10В (где разность будет в 5В), или же -5В и +5В (что даёт разницу в 10В.)

## 1.4 Сопротивление

Чтобы понять, с какой скоростью вода будет перетекать из сосуда в сосуд, необходимо знать параметры трубопровода, соединяющего их – иными словами, *проводника воды*. Основным параметром проводника является *сопротивление* – чем выше этот параметр, тем медленнее вода будет течь.

Примером проводника с высоким сопротивлением является трубопровод с малым сечением (или диаметром) труб, как показано на рис. 1.4.9.

Уменьшение сечения трубы приводит к увеличению её сопротивления для тока воды. Данное правило применимо также для электрических цепей, где сечение проводника (провода) и его сопротивление имеют обратную зависимость: чем больше сечение, тем меньше сопротивление; чем меньше сопротивление, тем больше ток.

Для наглядности предположим, что ёмкости “В” и “Г” на рис. 1.4.9 соединены

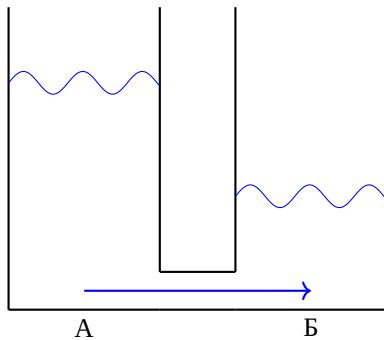


Рис. 1.3.8: Пример двух ёмкостей с разными уровнями воды, соединённые трубкой.

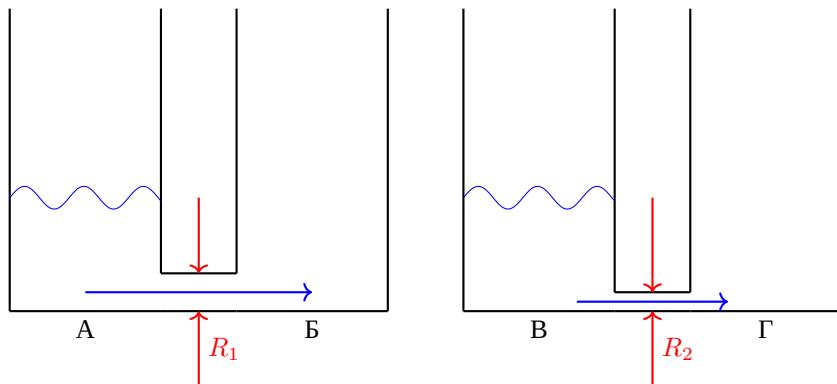


Рис. 1.4.9: Пример ёмкостей, соединённых трубами разных сопротивлений.

трубой с сечением в половину от того, что соединяет ёмкости “А” и “Б”. Тогда ток воды между ёмкостями “В” и “Г” будет в два раза меньше, чем между ёмкостями “А” и “Б”.

В электронике сопротивление проводника измеряется в Омах (англ. “Ohm”).

Взаимосвязь тока, напряжения и сопротивления описывается законом Ома.

Используя закон Ома, можно посчитать ток для электрической цепи по формуле 1.1.

$$I = \frac{U}{R} \quad (1.1)$$

Где “ $I$ ” – это сила тока (в Амперах), “ $U$ ” – это напряжение (в Вольтах), а “ $R$ ” – это сопротивление (в Омах).

Как можно видеть из данной формулы, увеличение сопротивления проводника в два раза уменьшает ток, идущий по электрической цепи, в два раза.



**Примечание:** Для запоминания закона Ома можно использовать детский стишок: “Знает каждый пионер: сила тока –  $U$  на  $R$ !”.

В обыденной жизни сопротивление есть у всего: любые провода имеют сопротивление, равно как и другие окружающие нас вещи. У некоторых материалов

сопротивление выше, чем у других. Те материалы, у которых сопротивление низкое и они хорошо пропускают ток, называют *проводниками*. Примерами хороших проводников могут служить медь, серебро и золото. Материалы, которые плохо пропускают ток, и часто используются для *изоляции*, называют *диэлектриками*. К этой категории можно отнести, например, различные пластики, из которых делают изоляцию проводов.



**Эксперимент №1:** Посмотрите вокруг – сколько различных проводников вы можете найти?

При прохождении тока через проводник, имеющий сопротивление (а следовательно, через любой “бытовой” проводник), часть энергии теряется, трансформируясь в тепло. Например, если вы включаете электрический чайник в бытовую розетку, то греется не только нагреватель чайника, который кипятит воду, но и провод подключения чайника в бытовую сеть (нагрев провода, по сравнению с нагревателем внутри чайника, будет незначительным, и вряд ли будет ощущим на ощупь.)

В некоторых случаях, нагрев проводника при прохождении тока является желаемым эффектом, как в случае с нагревателем внутри чайника; в других же случаях нагрев проводника стараются уменьшить, чтобы сократить тепловые потери энергии.



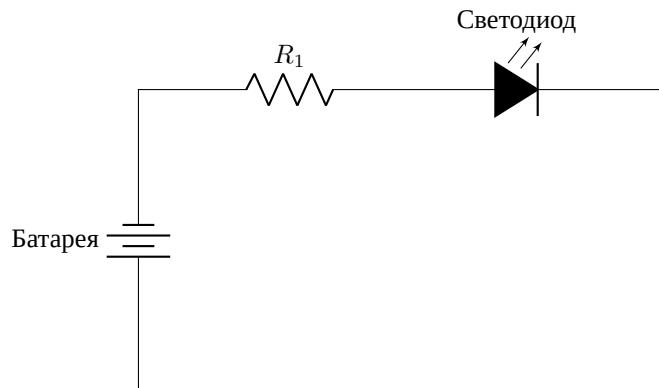
**Эксперимент №2:** Можете ли вы назвать другие примеры, где нагрев проводника является желаемым эффектом?

Существуют проводники, которые не имеют сопротивления – они называются *сверхпроводниками*. На сегодняшний день, данный вид проводников встречается только в очень специфических устройствах (например, в научном оборудовании), где требуется пропускать большой ток без потери энергии в тепло. Использование в быту таких проводников крайне затруднено из-за их огромной стоимости и требованиям к особым условиям эксплуатации (например, охлаждение до сверхнизких температур.)

Сопротивление есть не только у проводников, но и у источников напряжения – такое сопротивление называется *внутренним сопротивлением источника напряжения*.

В электронике сопротивление с нужным номиналом (значением) в пределах некоторой погрешности обычно задаётся специальным элементом – *резистором*. Слово “резистор” произошло от английского слова “resistance” – “сопротивление”. Таким образом, если буквально переводить слово “резистор”, то получится “сопротивлятор”, хотя конечно так никто обычно не говорит.

Резисторы играют важную роль в электрических схемах – они позволяют контролировать ток в цепи. Так как у каждого электронного компонента (например, у светодиода) есть определённые условия эксплуатации, то важно пропускать через них ток в пределах допустимых для данного компонента значений. Таким образом, поставив последовательно со светодиодом резистор с номиналом, рассчитанным исходя из характеристик светодиода, мы защитим его от преждевременного выхода из строя. Схема представлена на рис. 1.4.10.

Рис. 1.4.10: Схема со светодиодом и резистором  $R_1$ , ограничивающим ток.

Будучи подключенными последовательно, значения сопротивлений складываются. Провода аналогично с водой, мы можем сказать, что удлинение трубопровода, по которому течёт вода, повышает сопротивление этого трубопровода. На рис. 1.4.11 показана пара ёмкостей, соединённых последовательно двумя отрезками труб; сопротивление для тока воды между ёмкостями “А” и “Б” равно сумме сопротивлений  $R_1$  и  $R_2$ . Таким образом, суммарное сопротивление трубопровода току будет больше, чем  $R_1$  или же  $R_2$ , взятые по-отдельности.

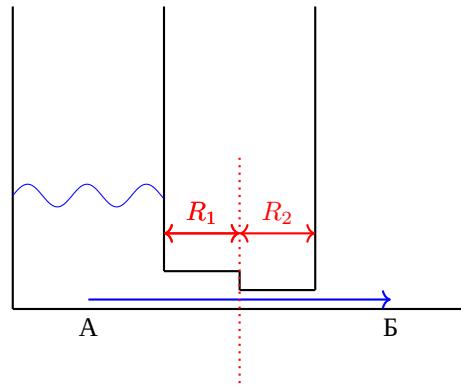


Рис. 1.4.11: Последовательное подключение сопротивлений на примере воды.

Формула 1.2 позволяет рассчитать общее сопротивление проводника при последовательном соединении отдельных сопротивлений.

$$R_{\text{общее}} = R_1 + R_2 \quad (1.2)$$

Электрическая схема с последовательным соединением резисторов показана на рис. 1.4.12.

Другим способом понизить сопротивление является использование нескольких проводников, соединённых *параллельно*, как показано на рис. 1.4.13.

При этом, общее сопротивление электрической цепи будет меньше, чем наименьшее сопротивление в группе.

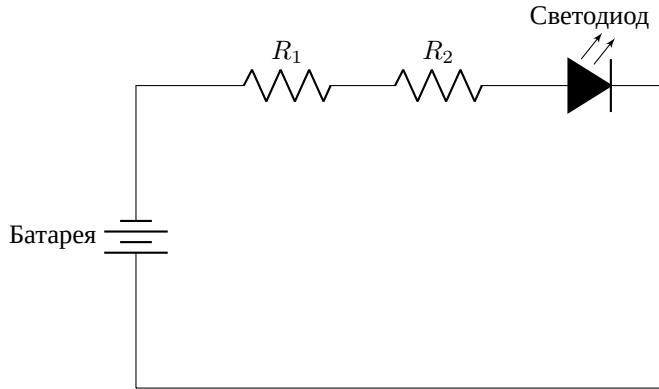
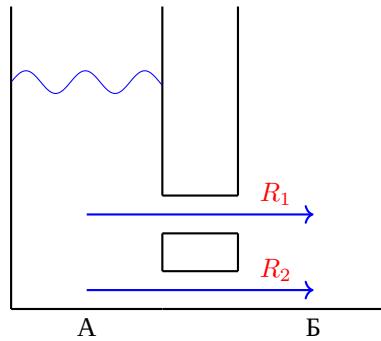
Рис. 1.4.12: Последовательное подключение сопротивлений  $R_1$  и  $R_2$ .

Рис. 1.4.13: Пример двух ёмкостей, соединённых двумя трубами, идущими параллельно.

Формула вычисления суммарного сопротивления для двух параллельно соединённых участков цепи показана ниже (см. 1.3.)

$$R_{\text{общее}} = \frac{R_1 * R_2}{R_1 + R_2} \quad (1.3)$$

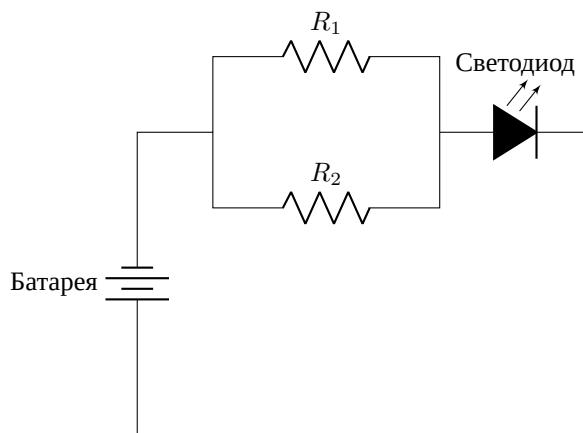
Данная формула позволяет вычислить общее сопротивление для любого количества параллельно соединённых сопротивлений в участке цепи.

Электрическая схема с параллельным соединением резисторов представлена на рис. 1.4.14.

## 1.5 Сборка электрической цепи

Сложно говорить о современной цифровой технике без упоминания *микроконтроллера* – “мозга” множества систем, с которыми мы взаимодействуем каждый день.

Микроконтроллер – это простой и обычно относительно дешёвый встраиваемый компьютер, обычно решающий одну задачу. Микроконтроллеры управляют современными бытовыми приборами, игрушками, электронными музыкальны-

Рис. 1.4.14: Параллельное подключение резисторов  $R_1$  и  $R_2$ .

ми инструментами, производственными роботами; на основе них работают 3D-принтеры и другие станки с числовым программным управлением.

Мы будем работать с платформой Arduino, которая широко доступна и предоставляет удобный интерфейс для её использования и программирования.

Внешний вид Arduino может различаться, в зависимости от модели; на рис. 1.5.15 представлен один из популярных вариантов Arduino, называемый “Arduino UNO”.

Более подробно про использование платформы Arduino и её программирование речь пойдёт в главе 2. А пока что мы будем использовать Arduino в качестве источника напряжения, вместо батарейки, которую мы указывали до этого в схемах.

Как можно видеть на рисунке 1.5.15, платформа имеет специальные разъёмы, называемые *портами*, для подключения компонентов и проводов. Есть порты, которые пронумерованы 0, 1, 2 и т.д. – это *цифровые порты*. Есть также порты, пронумерованные “A0”, “A1”, “A2” и т.д. – это *аналоговые порты*. Цифровые и аналоговые порты мы пока трогать не будем, так как для работы с ними требуется уже писать программу под микроконтроллер; о написании программ мы поговорим позже.

Сейчас стоит обратить внимание на порты, обозначенные “GND” и “5V”.

“GND” означает “Ground”, или “Земля” – это порт, на котором напряжение всегда имеет значение ноль. “5V” соответственно имеет всегда значение 5 Вольт.

Попробуем собрать электрическую цепь с резистором и светодиодом, приведённую на рис. 1.5.19.

Сборка схемы будет осуществляться на *макетной плате* (англ. *breadboard* – буквально “хлебная доска”). Внешний вид макетной платы показан на рис. 1.5.17. На левой части рисунка показана лицевая сторона макетной платы, куда вставляются детали при сборке схемы. С правой стороны рисунка показана обратная сторона макетной платы. У новых макетных плат обратная сторона обычно заклеена клейким слоем, поверх которого приклейен ещё один защитный слой бумаги – таким образом, макетную плату можно при необходимости приклеить к чему-либо. У макетной платы, показанной на рис. 1.5.17, данный клейкий слой удалён для наглядности; без особой необходимости лучше этот слой не убирать, так

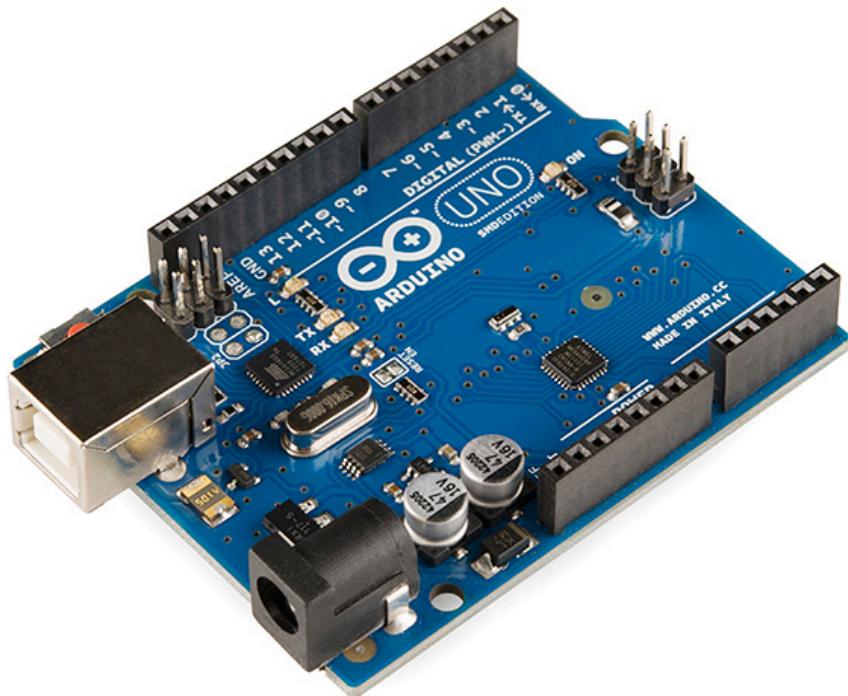


Рис. 1.5.15: Микроконтроллерная платформа Arduino UNO R3.

как он обеспечивает изоляцию металлических контактов.

Как можно видеть, макетная плата представляет собой просто сформованный кусок пластика с вставленными в него металлическими контактами, обеспечивающими соединение компонентов схемы, собираемой на ней.

Вдоль макетки, с каждой из сторон, располагаются две пары длинных линий, помеченных на лицевой стороне макетной платы, как “+” и “-” – эти линии предназначены для организации общих линий питания и земли.

На макетной плате эта схема будет выглядеть следующим образом (см. рис. 1.5.18.)

Где номинал резистора  $R_1$  должен быть не меньше 200 Ом.

Внимательно проверьте схему перед включением Arduino. В первую очередь обратите внимание на два провода, идущих от Arduino к макетной плате – они не должны замыкаться напрямую. На схеме можно видеть, что провод от 5V идет к ножке резистора, а провод от GND (GROUND) идет к ножке светодиода через общую линию, отмеченную синим цветом, на макетной плате.

После включения питания, светодиод должен постоянно светиться. Если он не светится, необходимо отключить питание и ещё раз проверить собранную схему.

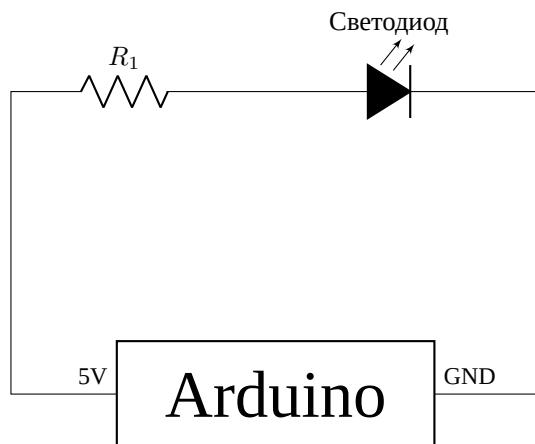


Рис. 1.5.16: Схема подключения светодиода к Arduino.

Одной из частых причин, почему светодиод не светится, является перепутанные плюс и минус (анод и катод) на светодиоде; в этом случае, достаточно просто перевернуть светодиод.



**Эксперимент №3:** Попробуйте заменить резистор  $R_1$  на резистор с большим сопротивлением (например, 500 Ом.) Как изменилась яркость светодиода?

Теперь попробуем выполнить последовательное подключение резисторов. Для этого возьмём два резистора, например, с номиналом от 200 до 300 Ом, и соберём схему 1.5.19.

Если  $R_1$  и  $R_2$  равны допустим 200 Ом, то общее сопротивление цепи будет 400 Ом, как было показано в формуле 1.2.

Итоговая схема показана на рис. 1.5.20.

Рис. 1.5.17: Макетная плата.

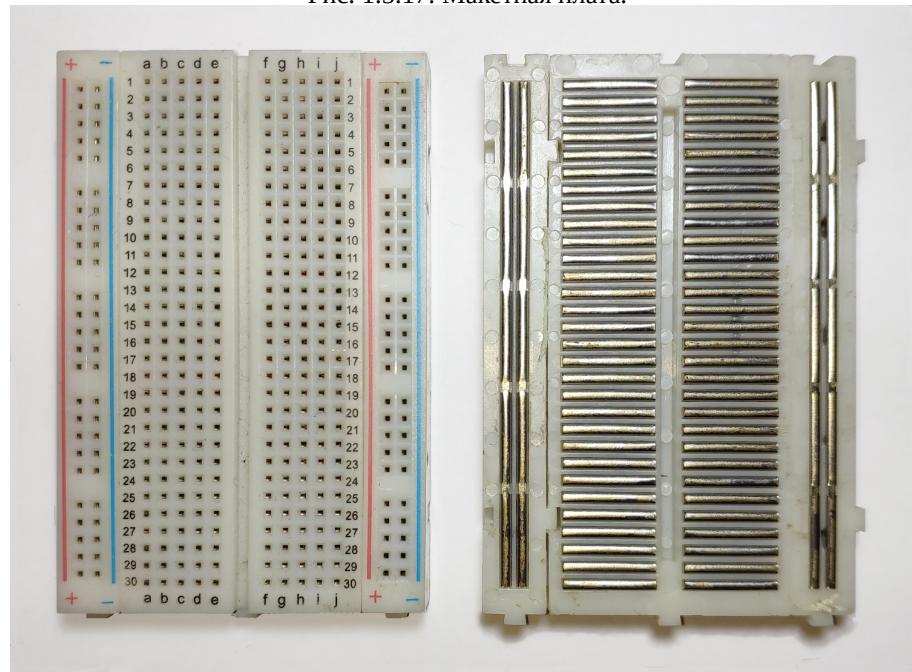
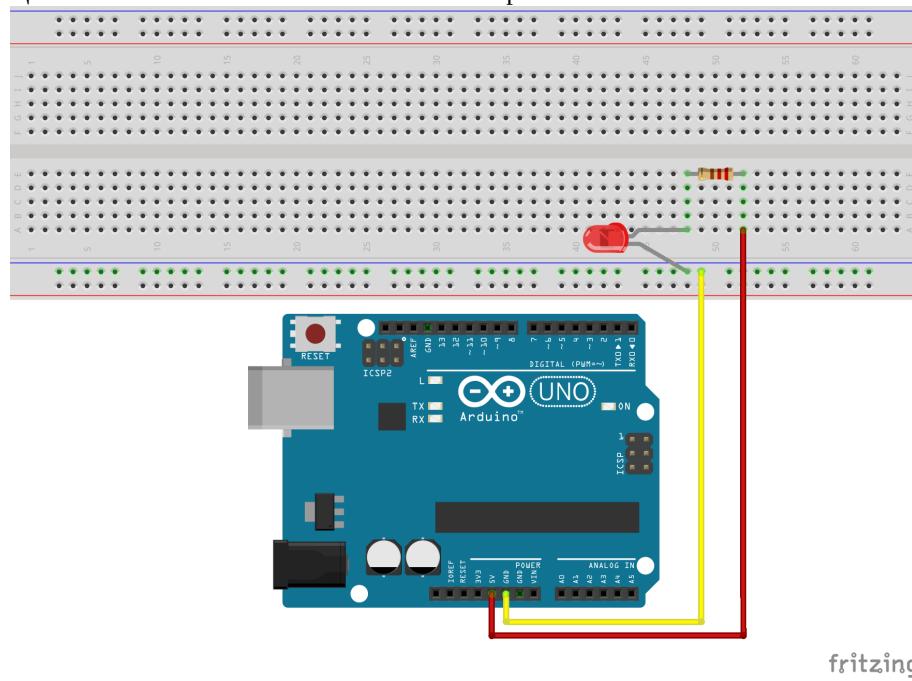


Рис. 1.5.18: Пример электрической цепи со светодиодом и резистором, использующей Arduino Uno в качестве источника напряжения.



fritzing

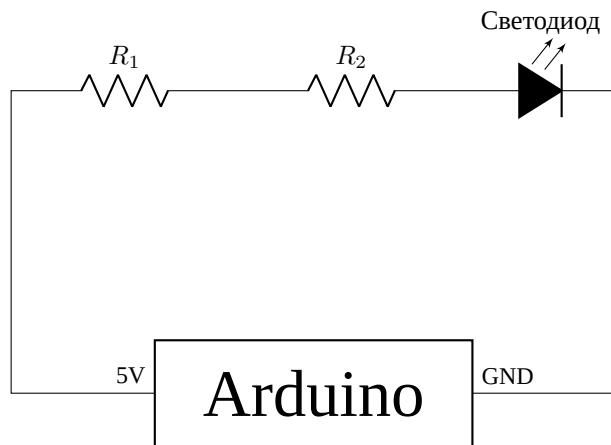
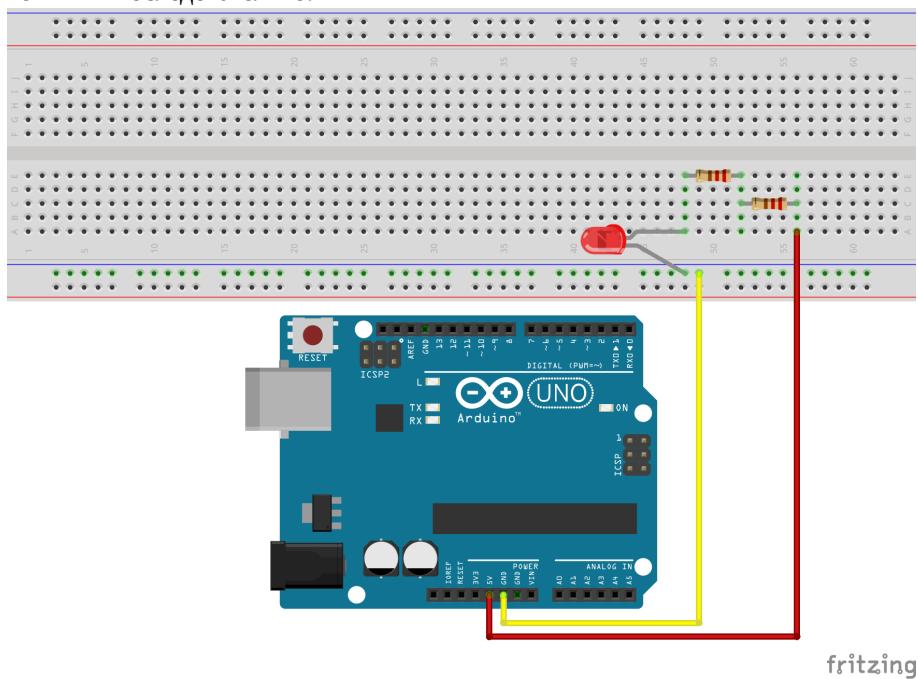


Рис. 1.5.19: Схема подключения светодиода к Arduino с последовательным подключением резисторов.

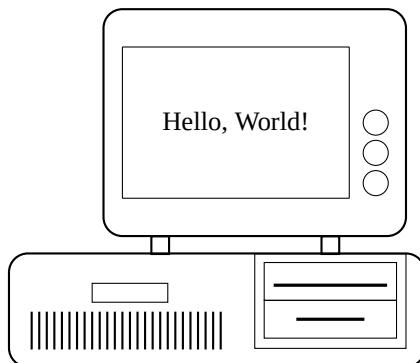
Рис. 1.5.20: Пример электрической цепи со светодиода и резисторами, подключенными последовательно.





## Глава 2

# Диалоги с компьютером



### 2.1 Введение

По нашему мнению, программирование является двойкой дисциплиной: с одной стороны, это – один из видов творчества, позволяющий человеку создать что-то необычное, новое и, возможно, полезное для общества; с другой стороны, это – инструмент, позволяющий решать практические, прикладные задачи. Как кисти и краски художника, или инструменты музыканта, инструменты программиста имеют большое разнообразие в видах и применениях. Чтобы освоить их в полной мере требуются годы. Тем не менее, долгий путь начинается с первого шага.

Данная глава позволяет людям, желающим освоить программирование, сделать первый шаг на пути в профессии программиста. Вам предлагается изучить основы программирования на C/C++, разрабатывая приложения для платформы Arduino.

### 2.2 Алгоритмы

Задолго до появления компьютеров люди писали инструкции друг для друга, которые позволяли понять последовательность действий для достижения какой-либо цели – например, “как добыть огонь”, “как сеять зерно”, “как запрячь лошадь” и т.п.

Последовательность инструкций, позволяющая достичь определённого результата, называется *алгоритмом*.

Были попытки (достаточно успешные) создания разных механизмов, выполняющих некую последовательность операций, заменяя тем самым человека. Но с появлением компьютеров разработка алгоритмов вышла на “новый уровень” – мы получили возможность записать алгоритм в память компьютера для того, чтобы он выполнил его в точности так, как мы задумывали. Современные компьютеры не понимают язык человека напрямую, поэтому им необходимо задавать алгоритмы в специальном *машинном языке*. Машинный язык представляет собой коды команд обработки данных, понимаемые процессором (главным вычислителем в компьютере.)

Написание программ на машинном языке является достаточно муторным и сложным процессом, и вскоре после появления компьютеров были придуманы первые языки программирования, более близкие человеку. Первым языком программирования был *ассемблер*, который по своей сути представлял из себя набор мнемоник – коротких человекочитаемых имён – для машинных команды процессора.

Но и ассемблер был по своей сути слишком прост и не позволял кратко и легко выражать идеи, которые люди хотели заложить в свои алгоритмы. Поэтому были придуманы высокоуровневые языки программирования, позволяющие упростить и ускорить написание программ. Язык “C”, основы которого будут рассмотрены в этой главе, является одним из старейших языков программирования, активно используемых по сей день.

Чтобы попрактиковаться в составлении алгоритмов, предположим, что у нас есть светодиод или лампочка, подключенная к компьютеру. Чтобы объяснить компьютеру, как сделать эффект мигания лампочки, мы должны сформулировать алгоритм. Это процесс называется *формализацией*. Возможный алгоритм мигания светодиодом может выглядеть так:

1. Включить светодиод.
2. Выключить светодиод
3. Повторить алгоритм.

Таким образом, на примере данного алгоритма мы видим не только действия по включению и выключению условного светодиода, но и некоторую повторяемость, цикличность этих действий.



**Эксперимент №4:** Попробуйте придумать алгоритмы для обычных операций, которые вы выполняете каждый день. Например, алгоритм заваривания чая, или же алгоритм уборки квартиры. Насколько детально вы сможете объяснить порядок выполнения простых действий? Насколько сложными получаются алгоритмы?

## 2.3 Платформа Arduino

Микроконтроллер – это простой и обычно относительно дешёвый встраиваемый компьютер, обычно решающий одну задачу. Микроконтроллеры управляют современными бытовыми приборами, игрушками, электронными музыкальными инструментами, производственными роботами; на основе них работают 3D-принтеры и другие станки с числовым программным управлением.

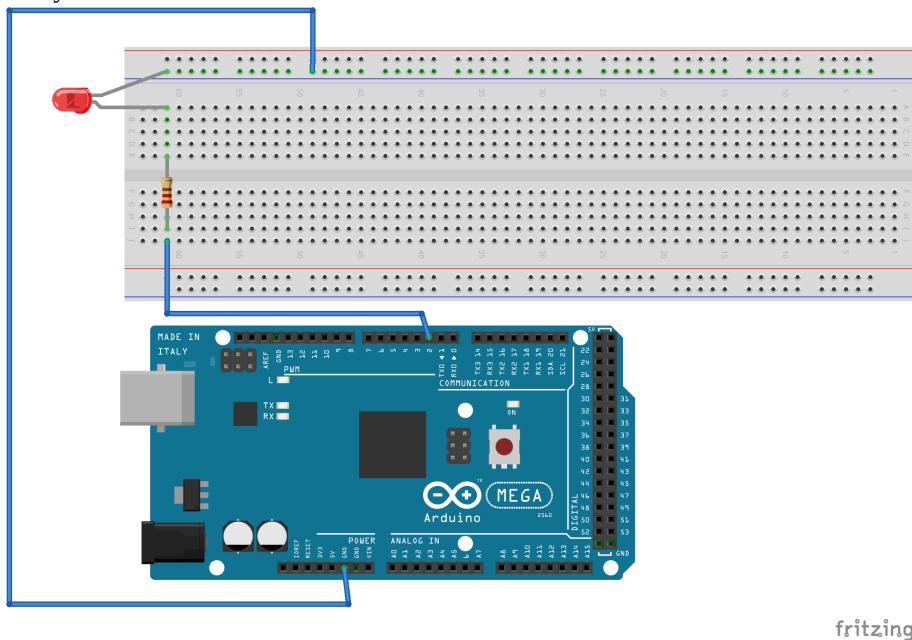
Мы будем работать с платформой Arduino, которая предоставляет удобный интерфейс для её программирования.

Большинство платформ Arduino построены на базе микроконтроллеров AVR.

## 2.4 Работа с макетной платой

Макетная плата позволяет собирать схемы (подключать электронику) без применения пайки — это упрощает прототипирование и ускоряет процесс разработки проектов. Компоненты просто вставляются в слоты на макетной плате для соединения (см. рис. 2.4.1.)

Рис. 2.4.1: Пример подключения светодиода к Arduino Mega 2560 через макетную плату.



fritzing

- Черный провод подключен к Arduino и идет на вывод “GND” (минус.)
- Синий провод подключен к Arduino и идет на вывод “5V” (плюс.)



**Примечание:** Обратите внимание, что светодиоды (и некоторые другие элементы) подключаются к платформе Arduino через резистор – это необходимо для обеспечения бесперебойной работы схемы и предупреждения всяческих поломок и ухудшения работы как отдельных деталей, так и схемы в целом.

## 2.5 Подключение Arduino к компьютеру

Чтобы подключить Arduino к компьютеру вам потребуется сама платформа Arduino (в нашем случае мы используем Arduino Mega 2560) и кабель стандарта USB-B.

Соедините Arduino с компьютером через USB-кабель. Вы увидите, как на плате загорится светодиод “ON”.

Теперь необходимо настроить Arduino IDE для работы с подключенной Arduino, для этого нужно войти в панель “Инструменты” затем “Плата” – в этом меню необходимо выбрать вариант Arduino, с которой вы сейчас работаете, затем в подменю “Порт” необходимо выбрать порт, к которому подключена Arduino.

## 2.6 Основы работы с мультиметром

Мультиметр – незаменимый прибор, с его помощью можно узнать сопротивление резистора, измерить напряжение, произвести проверку на проводимость (“прозвонка”), узнать цвет и полярность светодиода и многое другое.

На рис. 2.6.2 показан один из вариантов мультиметра.

Рис. 2.6.2: Пример мультиметра.



Далее приведена таблица на которой отражены основные символы, встречающиеся на корпусе прибора, необходимые для работы с мультиметром:

Обозначение	Описание
V~	Измерение напряжения переменного тока.
mV~	Измерение напряжения переменного тока, милливольты (мВ.)
V—	Измерение напряжения постоянного тока.
mV	Измерение напряжения постоянного тока, милливольты (мВ.)
A—	Измерение постоянного тока.
A~	Измерение переменного тока.
$\Omega$	Измерение сопротивления. Как правило, доступны следующие диапазоны: “2k” (2000 Ом), “20k” (20000 Ом), “200k” (200000 Ом) и “2M” (два мегаома, или два миллиона Ом.)
HOLD	“Заморозить” текущее показание на дисплее.
	Тестирование диодов.
Hz	“Hertz”, Герцы – измерение частоты.
	Измерение ёмкости.
•))	Измерение целостности цепи (так называемая “прозвонка”.)
hFE	“Hybrid parameter forward current gain, common emitter” – режим тестирования транзисторов.



**Эксперимент №5:** Попробуйте померять с помощью мультиметра сопротивление резисторов, соединяя их последовательно и параллельно через макетную плату. Как меняется сопротивление собранной цепи?



**Эксперимент №6:** Померяйте сопротивление других проводников – например, проводов для макетной платы Arduino, металлических предметов. Какое у них сопротивление?



**Эксперимент №7:** Известно, что графит является проводником. Померяйте сопротивление графитового стержня в карандаше. Также попробуйте на листе бумаги нарисовать жирную графитовую линию и померяйте её сопротивление. Какое сопротивление вы видите?

## 2.7 Знакомство со средой разработки Arduino

Среда разработки Arduino (Arduino IDE) состоит из встроенного текстового редактора программного кода, области сообщений, окна вывода текста (консоли), панели инструментов с кнопками часто используемых команд и нескольких меню. Для загрузки программ и связи с компьютером среда разработки подключается к аппаратной части Arduino.

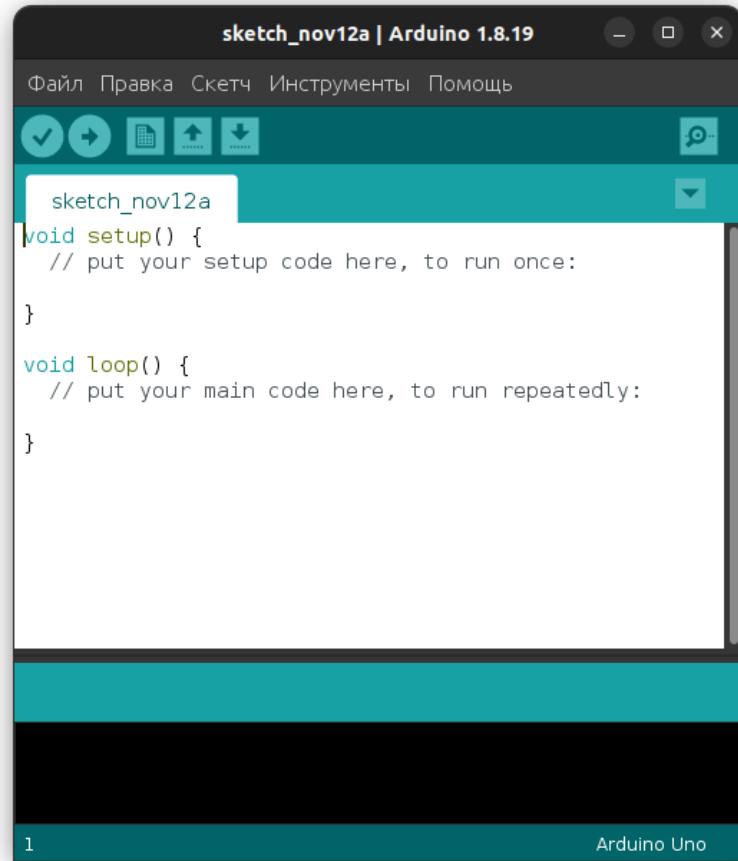
Скачать среду разработки можно с официального сайта Arduino: <https://www.arduino.cc/en/Main/Software>.

Перед скачиванием будет предложено пожертвовать денег проекту Arduino для дальнейшего развития, но этот шаг необязателен и может быть выполнен на ваше усмотрение.

После скачивания необходимо установить программу на вашу операционную систему. Самый простой способ установки на GNU/Linux – это скачивание архива и его последующая распаковка в удобный для вас каталог. В архиве содержится всё необходимое для запуска Arduino IDE.

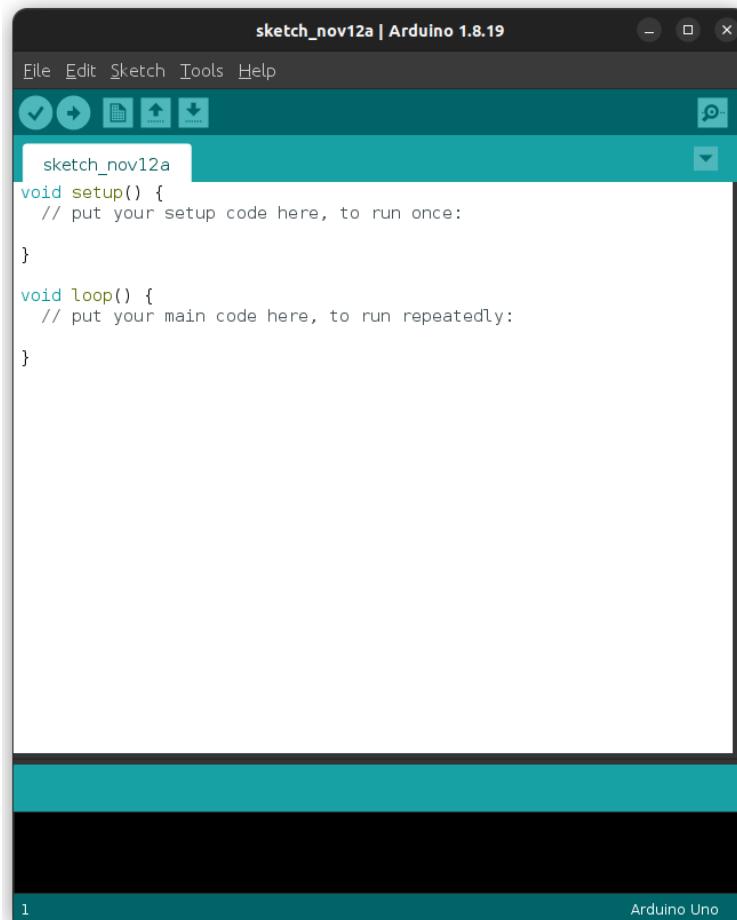
Если ваша операционная система настроена на использование русского языка в интерфейсе, то скорее всего интерфейс Arduino IDE будет русифицирован, как показано на рис. 2.7.3.

Рис. 2.7.3: Главное окно Arduino IDE.



В англоязычной версии главное окно программы Arduino IDE выглядит, как показано на рис. 2.7.4.

Рис. 2.7.4: Главное окно Arduino IDE (английская версия.)



Сменить язык интерфейса можно через меню “Файл” (“File”) → “Настройки” (“Preferences”), где в открывшемся диалоговом окне необходимо выбрать из выпадающего списка “Язык редактора” (“Editor language”) нужный вам вариант.

Ниже приведено описание кнопок в интерфейсе Arduino IDE.

Название	Описание
Verify/Compile (Проверка)	Проверка программного кода.
Upload (Загрузка)	Компилирует программный код.
New (Создать)	Создание нового скетча.
Open (Открыть)	Открыть скетч.
Save (Сохранить)	Сохранить скетч.
Serial Monitor (Монитор порта)	Открыть монитор порта.

## 2.8 Структура программы на Arduino

Программа для Arduino обычно состоит из двух основных частей, также называемых функциями: `setup` и `loop`. Пример программы, которая мигает одним светодиодом:

```
void setup() {
    pinMode(2, OUTPUT);
}

void loop() {
    digitalWrite(2, HIGH);
    delay(500);
    digitalWrite(2, LOW);
    delay(500);
}
```

Функция `setup` производит инициализацию микроконтроллера при его включении. В неё следует помещать все команды, которые должны выполняться единожды на старте системы.

Цифровой порт (или, по-другому, пин) Arduino может находиться в двух состояниях. В режиме входа пин считывает напряжение, а в режиме выхода – позволяет выдавать на пине такое же напряжение.

Рассмотрим приведённый выше пример. В `setup` выполняется функция `pinMode`, которая позволяет настроить режим работы указанного пина как вход или выход:

```
pinMode(pin, mode);
```

где `pin` – номер пина, `mode` – режим работы (`INPUT/OUTPUT`).

В `loop` вызываются две функции: `digitalWrite` и `delay`.

Функция

```
digitalWrite(pin, value);
```

где `pin` – номер пина, `value` – уровень сигнала (`HIGH/LOW`), подаёт на пин высокое или низкое напряжение.

Функция

```
delay(value);
```

где `value` – количество миллисекунд, останавливает выполнение программы на указанное время.

### 2.8.1 Задачи

- Соберите на макетной плате “бегущий огонь”: светодиоды должны поочерёдно включаться и выключаться, один за другим.
- Модифицируйте “бегущий огонь” так, чтобы он бежал сначала в одну сторону, затем в другую.

## 2.9 Переменные

*Переменная* – это ключевое понятие в программировании. Любая программа работает с данными. Возьмём для наглядности некую программу- калькулятор, умеющую складывать два числа. Чтобы микроконтроллер мог работать с этими числами их нужно где-то хранить. Где? В оперативной памяти. Все данные, которые используются микроконтроллером во время работы, хранятся именно там. Для работы нашего калькулятора нужно загрузить в ячейки оперативной памяти два числа-операнда, которые нужно сложить, например 15 и 3:

Адрес ячейки	Значение ячейки
0000	15
0001	3
0003	0
...	...

*Переменная* – это ячейка данных в оперативной памяти (ОЗУ). Объявить переменную – значит сказать компьютеру выделить какую-нибудь ячейку памяти для наших нужд.

Переменная в языке C++ имеет определённый тип и уникальное имя. Объявление (*инициализация*) переменной выглядит следующим образом:

```
тип имя = значение;
```

То есть, чтобы загрузить в оперативную память два числа 15 и 3, мы должны написать следующее:

```
int a = 15;
int b = 3;
```

Слово **int** это тип переменной, означает, что эта переменная является числом.

Также следует объявить переменную для хранения результата сложения:

Дальше – складываем значения двух переменных **a** и **b**:

```
int result = a + b;
```

Здесь мы присвоили переменной **result** результат операции сложения двух переменных.

**ВАЖНО!** Имя переменной может состоять только из букв, цифр и нижнего подчёркивания, причём имя не может начинаться с цифры.

Вернёмся к нашим светодиодам. Объявим новую переменную:

```
int k = 500;
```

Что мы можем с ней сделать? Например, в программе мигания светодиодом заменим ею значение задержки в функции **delay**. Вообще, переменным следует давать осмысленные имена, в нашем случае пусть это будет не **k**, а **delay\_val**:

```
void loop() {
    int delay_val = 500;
    digitalWrite(2, HIGH);
    delay(delay_val);
    digitalWrite(2, LOW);
    delay(delay_val);
}
```

Таким образом, мы сможем поменять значения всех задержек одной заменой значения `delay_val`:

```
void loop() {
    int delay_val = 600;
    // ...
}
```

Можно, например, увеличивать `delay_val` на 100 при каждом выполнении `loop`:

```
void loop() {
    int delay_val = 100;
    digitalWrite(2, HIGH);
    delay(delay_val);
    digitalWrite(2, LOW);
    delay(delay_val);
    delay_val = delay_val + 100;
}
```

Кстати, строчку `delay_val = delay_val + 100` можно заменить на `delay_val += 100` и результат будет тем же, но запись короче.

`+=` – оператор присваивания, совмещённый со сложением.

Существуют также другие операторы подобного рода – например, “`-=`” (читается “минус-равно”). Если мы запустим этот код, то увидим, что задержка переключения светодиодов... не меняется. Почему? При каждом выполнении `loop` каждый раз объявляется новая переменная `delay_val` со значением 100 и потому задержка остаётся той же. Сейчас `delay_val` объявлена как локальная переменная внутри `loop`, следует объявить её за пределами функции, чтобы она стала глобальной:

```
int delay_val = 100;

void loop() {
    digitalWrite(2, HIGH);
    delay(delay_val);
    digitalWrite(2, LOW);
    delay(delay_val);

    delay_val += 100;
}
```

Теперь всё будет работать. Но так задержка будет бесконтрольно расти. Решением будет сделать так, чтобы `delay_val` увеличивалась до какого-то порогового значения, например, до 600. Для этого нужно добавить условие (`if`):

```
int delay_val = 100;

void loop() {
    digitalWrite(2, HIGH);
    delay(delay_val);
```

```

digitalWrite(2, LOW);
delay(delay_val);

if (delay_val < 600){
    delay_val += 100;
}
}

```

Об условиях и других управляющих конструкциях будет подробнее рассказано в главе “Управляющие конструкции”.

## 2.10 Константы

*Константа* – это некоторое фиксированное, неизменяемое значение. Константы бывают двух видов: безымянные и именованные. Безымянные константы, как следует из их названия (вот вам и каламбур!) не имеют имени в программе, и представляют собой явно записанные значения – например, значение “500” в коде ниже является безымянной константой:

```
delay(500);
```

*Именованные константы* же имеют явно заданное имя. Например, в приведённом ниже примере кода D является именованной константой, равной числу 500:

```

void loop() {
    const int D = 500;
    digitalWrite(2, HIGH);
    delay(D);
    digitalWrite(2, LOW);
    delay(D);
}

```

Обратите внимание, что структура объявления константы практически повторяет структуру объявления переменных, добавляется только слово “const” перед типом.

```
const тип имя = значение;
```

Как и в случае с переменными, мы можем ссылаться на константу по её имени; основное различие между переменными и константами в том, что после объявления константу нельзя поменять. Попытка изменить константу после объявления является ошибкой в языке программирования C++.

## 2.11 Управляющие конструкции

### 2.11.1 Условия

Иногда во время выполнения программы следует принять решение о том, что делать дальше. Для того, чтобы компьютер мог сделать правильный выбор, по какому пути пойти, нам, как программистам, следует описать условия в коде программы: если условие выполняется, делаем одно, иначе – делаем другое.

Условия в программах описываются при помощи специальных управляющих конструкций. В языке C++ у нас есть две основные конструкции. Первая из них – оператор **if** (буквально в переводе с английского “если”). Пример использования:

```
if (a > 10) {
    // действие, выполняемое, если значение
    // переменной 'a' больше 10.
}
```

Если нужно проверить равно ли значение переменной чему-либо, используют оператор сравнения “==”:

```
if (a == 10) {
    // действие, выполняемое, если значение
    // переменной 'a' равно 10.
}
```

Не путайте оператор сравнения “==” с оператором присваивания “=” - это важно!

Часто необходимо не только делать что-либо при выполнении условия, но и предоставить альтернативную инструкцию (или набор инструкций), выполняемую тогда, когда условие не выполняется. В этом случае используют конструкцию **if ... else**:

```
if (a > 10) {
    // действие, выполняемое, если значение
    // переменной 'a' больше 10.
} else {
    // действие, выполняемое, если значение
    // переменной 'a' меньше или равно 10.
}
```

Второй оператор, который нам будет встречаться, это так называемый *оператор выбора switch*. С ним познакомимся позже. Он удобен, например, тогда, когда нам нужно выполнять несколько разных действий в зависимости от значения переменной, и этих действий много.

### 2.11.2 Циклы

Простые программы, вроде “бегущего огня”, могут быть написаны простым копированием и вставкой алгоритма мигания светодиода (возможно, с небольшими модификациями).

А теперь представьте, что вам требуется запрограммировать “бегущий огонь” на 100 светодиодов. Утомительная задача, не правда ли? Для того, чтобы не делать тупую работу по копированию одного и того же кода много раз, программистами придуманы специальные управляющие конструкции, называемые *циклами*.

Циклы бывают разные. Основные виды циклов, которые вам будут встречаться практически в любом языке программирования:

- Цикл со счётчиком (также называемый “параметрический цикл”).
- Цикл с предусловием.

- Цикл с постусловием.

Каждый вид циклов имеет собственную реализацию в языке программирования, который мы используем (C++).

### Цикл со счётчиком

Цикл со счётчиком реализуется конструкцией `for` – она позволяет нам создать счётчик, задать его начальное значение, описать условие выполнения цикла и операцию изменения счётчика:

```
//      5.
//      1.       2.           4.
for (int pin = 0; pin < 10; pin = pin + 1) {
    // 3. (тело цикла)
}
```

Выполняется эта конструкция в следующем порядке:

1. объявляем переменную и присваиваем ей значение 0 (шаг 1);
2. переходим к проверке, где смотрим, выполняется ли условие (шаг 2);
3. после этого, если условие 2 выполняется, мы переходим к телу цикла (шаг 3);
4. после выполнения тела цикла, мы переходим к изменению значения счётчика (шаг 4);
5. после шага 4 мы опять возвращаемся к шагу 2, если условие выполняется, то переходим к шагу 3 и т.д.

### Цикл с предусловием

Другим распространённым видом цикла является цикл с предусловием, реализуемый в C++ конструкцией `while` – данный вид цикла удобен в тех случаях, когда мы не знаем точного количества раз, сколько нужно повторить тело цикла (не знаем количество итераций.)

Общий вид цикла `while` таков:

```
int pin = 2;

while (pin < 10) {
    // тело цикла
}
```

### Цикл с постусловием

Кроме вышеперечисленных видов циклов, есть ещё цикл с постусловием, где проверка условия выполнения цикла осуществляется после выполнения тела цикла. Реализуется данный вид циклов конструкцией `do .. while`.

Он достаточно редко используется и мы не будем на нём здесь останавливаться.

### 2.11.3 Зачем столько видов циклов?

Обратите внимание, что один вид цикла может быть реализован через другой, т.к. данные управляющие конструкции взаимозаменяемы. Возникает вопрос – зачем же нам нужно столько видов циклов? Всё дело в удобстве. В одних случаях удобнее использовать один вид циклов, в других случаях – другой.

У программистов есть специальный термин для описания подобных конструкций языка программирования: синтаксический сахар. Синтаксический сахар – это конструкции языка, без которых в принципе можно обойтись при разработке программ, но с ними всё проще (“слаще”).

## 2.12 Задачи

1. Перепишите “бегущий огонь” с использованием цикла.
2. Модифицируйте алгоритм “бегущего огня” таким образом, чтобы светодиоды начинали загораться с обоих концов гирлянды, и огни “бежали” навстречу друг другу.

## Глава 3

# Белый шум



Как правило, самая интересная часть при работе с микроконтроллерами – это возможность взаимодействовать с физическим миром посредством портов ввода/вывода, расположенных на отладочной плате.

В этой главе мы посмотрим, как микроконтроллер преобразует сигналы из реального мира в цифровое представление, которое можно использовать в программе для управления чем-либо.

### 3.1 Виды сигналов

Существуют два основных вида сигналов – *аналоговые* и *цифровые*. Для их сравнения посмотрим на графики 3.1.1 и 3.1.2.

По оси “Y” на обоих графиках отложены Вольты, по оси “X” – время.

Как можно видеть, аналоговый сигнал не имеет чётко выделенных уровней и меняется произвольно во времени в некоторых пределах. Цифровой сигнал имеет чётко выраженные логические уровни, соответствующие нулю и единице.

Примером аналоговых сигналов является голос человека. Если мы попробуем записать голос на цифровой диктофон, то на микрофон диктофона будет приходить звуковая волна, подобная той, что на рисунке 3.1.1.

Примером цифрового сигнала является записанный звуковой файл, передаваемый по сети Internet. Если сохранить цифровой сигнал на накопитель, то получатся цифровые данные, состоящие из нулей и единиц – на обычных компьютерах подобные данные, как правило, хранятся в виде файлов.

Также компьютеры обладают возможностью принимать и оцифровывать аналоговые сигналы.

Благодаря возможности считывать аналоговые сигналы, компьютер или микроконтроллер может получать самую различную информацию об окружающем

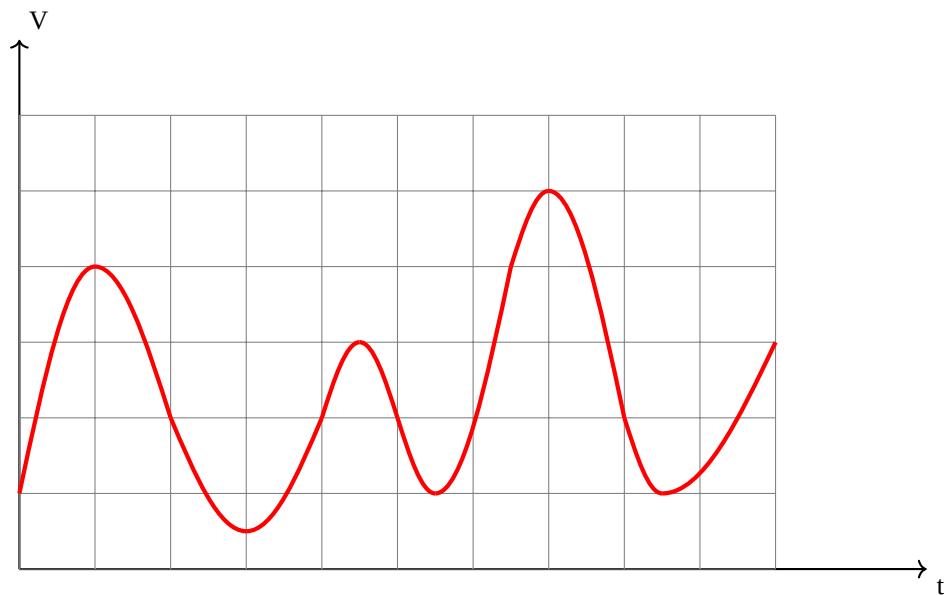


Рис. 3.1.1: Пример аналогового сигнала.

мире, ведь масса различных параметров окружающей среды представлены именно некоторым диапазоном значений, которые не имеют чётких делений: температура, влажность, освещённость, атмосферное давление и т.п.

Именно поэтому большинство микроконтроллеров имеют аналоговые входы для работы с аналоговыми сигналами, и в последующих разделах мы рассмотрим эти возможности на примере платформы Arduino.

Но для начала мы должны изучить инструменты, которые позволяют нам видеть сигнал, который принимает и обрабатывает микроконтроллер.

## 3.2 Последовательный порт

Последовательный порт в Arduino — это тот самый USB-B, который мы подключаем всякий раз, когда желаем включить наш микроконтроллер или загрузить в Arduino какую-либо программу. С помощью последовательного порта можно передавать данные с Arduino на компьютер и наоборот.

### 3.2.1 Основы работы с Arduino через последовательный порт

Прежде, чем начать работать с последовательным портом, нам необходимо его настроить; делается это следующим образом: в теле функции `setup` мы должны написать:

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

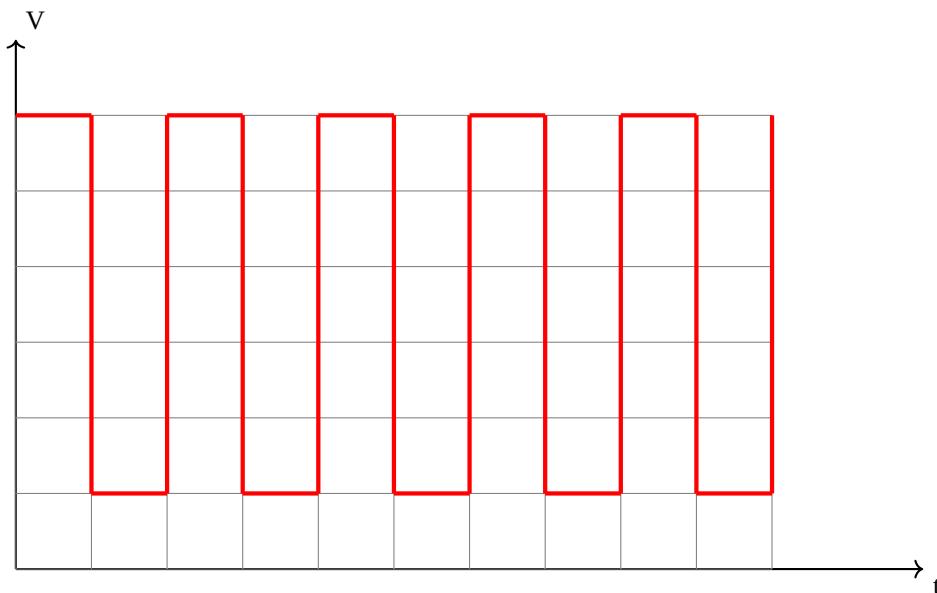


Рис. 3.1.2: Пример цифрового сигнала.

В этом случае мы обеспечиваем обмен данными между компьютером и Arduino с указанной скоростью, где 9600 – это скорость, с которой мы передаем данные на персональный компьютер в бодах (битах в секунду.) Обычно данный параметр принимает одно из следующих значений: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 115200.

### 3.2.2 Передача данных на компьютер

Теперь попробуем передать какие-нибудь данные на компьютер. В качестве примера мы просто отправим строку “Hello, world!” по последовательному порту. Для начала пропишем настройку порта в функции `setup`:

```
void setup() {
    Serial.begin(9600); // устанавливаем скорость порта
}
```

А вот так в нашем случае выглядит функция `loop`:

```
void loop() {
    Serial.println("Hello World");
    delay(1000); // ждём 1000 мс перед следующей отправкой
}
```

Результат выполнения программы можно увидеть, открыв монитор порта в Arduino IDE – это можно сделать из меню “Инструменты” (“Tools”), выбрав пункт “Монитор порта”.

Кроме того, открыть монитор порта можно, нажав комбинацию клавиш **Ctrl + Shift + M**.

Передачу данных с Arduino на компьютер можно использовать в множестве разных задач. Примером одной из таких задач является простейший способ отладки программ – с помощью вывода информации о работе программы в Arduino на последовательный порт. Иными словами, вместо того, чтобы пытаться самим понять, что же пошло не так и почему что-то не работает, мы просим Arduino саму рассказывать нам, что она делает.

### 3.3 Аналоговые порты

На отладочной плате Arduino присутствуют так называемые *аналоговые порты*, которые позволяют считывать аналоговый сигнал. На плате они подписаны как A0, A1, и т.д. Если цифровые порты рассчитаны на цифровой сигнал, который может быть в двух состояниях - 0 В или 5 В, то аналоговые, соответственно, рассчитаны на аналоговый, непрерывный сигнал, который может принимать любые значения от 0 до 5 В.

В качестве первого эксперимента мы попробуем вывести аналоговый сигнал на компьютер с аналогового порта, который никуда не подключен.

В коде будем использовать функцию `analogRead`:

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

void loop() {
    // Получаем значение с аналогового порта 0
    int value = analogRead(A0);

    Serial.println(value);

    delay(100);
}
```

Если мы посмотрим на монитор порта в Arduino IDE, то увидим, что сигнал принимает значения, которые на вид совершенно случайны. Тем не менее, мы можем на них влиять – если поднесём руку, то скорее всего увидим, что сигнал изменился. Означает ли это, что у нас открылись супер-способности и нам пора записываться в лигу супер-героев? К сожалению, нет – но всё же мы только что открыли способность в Arduino считывать электромагнитный шум, который в обычной жизни, как правило, невидим для нас.

Есть несколько способов “увидеть” этот шум в обыденной жизни. Например, можно настроить радио на частоту между радио-станциями, и услышать “шипение” из динамика. Пример такого шума показан на рис. 3.3.3.

Как бы странно это не звучало, но есть разные “цвета” шума: “белый”, “розовый”, “красный”, “фиолетовый” и “серый”. Шумы различных “цветов” различаются спектром сигнала, и характеристика “цвета” дана по аналогии со спектром видимого света. В нашем случае мы рассматриваем “белый” шум, который и дал название этой главе.

“Белый” шум – это сигнал, составляющие которого распределены равномерно по всему диапазону используемых частот. Если “белый” шум воспроизвести

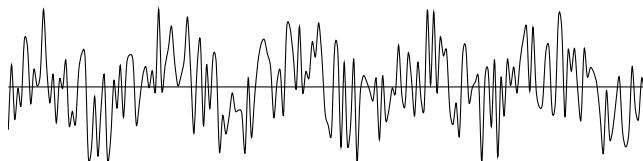


Рис. 3.3.3: Белый шум.

через динамик, чтобы мы могли его услышать, то мы сможем услышать, что все слышимые нами частоты звука в нём распределены равномерно – иными словами, мы услышим просто “ш-ш-ш-ш” из динамика или аудио-колонок.

**Примечание:** Изображение 3.3.3 сгенерировано в момент создания электронной версии книги в формате PDF, и в разных версиях книги оно будет различаться. Это вызвано тем, что для генерации изображения используется генератор случайных чисел, а случайность в компьютере обычно берётся из непредсказуемости окружающего мира – в частности, как уже говорилось выше, источником такой непредсказуемости является “Белый шум”.

В случае с Arduino, эти шумы улавливаются самой схемой, и преобразуются в набор чисел определённого диапазона.

Данный электромагнитный фон постоянно присутствует вокруг нас; у него есть много разных источников. Во-первых, шум создаёт человеческая цивилизация в целом – телевизионные и радиовышки; базовые станции, обеспечивающие работу мобильных телефонов; сами телефоны, передающие данные по беспроводным сетям и многое другое. Во-вторых, существуют совершенно естественные, природные, источники радиоволн – например, некоторые виды звёзд производят очень мощный радиосигнал.

Для более удобного просмотра этого сигнала удобно воспользоваться “плоттером по последовательному соединению”, доступного из меню “Инструменты” (“Tools”) – таким образом, вы можете увидеть график, подобный 3.3.3.



**Эксперимент №8:** Попробуйте поднести мобильный телефон к Arduino. Как изменится сигнал?



**Эксперимент №9:** Подключите порт A0 сначала к земле (порту “GND”), потом отключите его и подключите A0 к порту 5V на Arduino. Что происходит с сигналом при этих действиях?

Есть ли какое-то применение этому “шуму”? Оказывается, да. Благодаря своей непредсказуемости, он может быть использован в качестве источника случайных чисел – что бывает полезно например при разработке игр, о чём будет рассказано в последующих главах книги.

### 3.4 Аналогово-цифровое преобразование

Вы могли заметить в главе 3.3, что при считывании с аналогового порта мы получаем значения от 0 до 1023. Число 1023 подозрительно похоже на число 1024, которое в программировании встречается достаточно часто – не удивительно, ведь 1024 это степень двойки ( $2^{10}$ ).

Если мы проследим дальше этот след, то увидим, что аналоговый порт каким-то образом преобразует входящий аналоговый сигнал в цифровое (двоичное) представление, которое мы и видим в программе.

Если мы возьмём 10 бит для хранения информации, то в них мы можем закодировать  $2^{10} = 1024$  разных комбинации из 10 нулей и единиц – отсюда и максимальное значение 1023, ведь мы должны учитывать ещё ноль, который также является одним из возможных значений.

Операцию преобразования аналогового сигнала в цифровой выполняет компонент, называемый *Аналогово-Цифровой Преобразователь* (сокращённо “АЦП”.) В роли АЦП может выступать как отдельная микросхема, так и сам микроконтроллер. Схематически аналогово-цифровой преобразователь можно изобразить, как показано на рис. 3.4.4.

По-английски “АЦП” – “Analog-to-Digital Converter” (сокращённо “ADC”.)

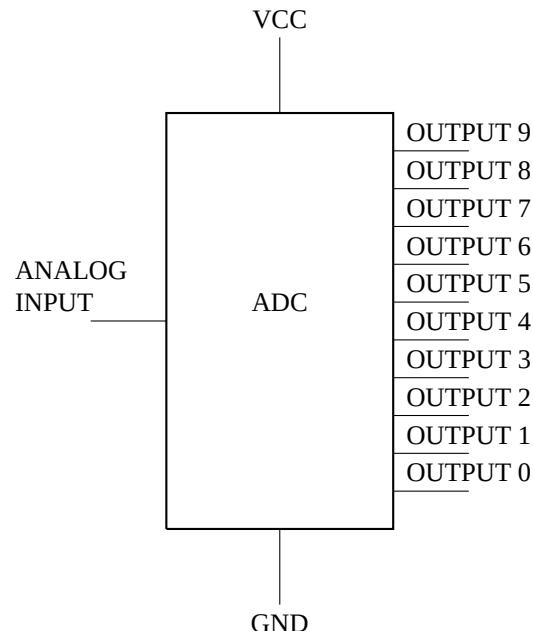


Рис. 3.4.4: Схематическое изображение аналогово-цифрового преобразователя.

На вход АЦП (“ANALOG INPUT”) подаётся аналоговый сигнал, а на выходах (“OUTPUT 0” .. “OUTPUT 9”) кодируется значение входного сигнала в каждый момент времени в виде набора логических уровней “HIGH” (“1”) / “LOW” (“0.”). Самому АЦП требуется также питание – для этого как раз предназначены выводы “VCC” и “GND”.



**Пример 1:** На вход АЦП подаётся 2.5В. На выходах формируется двоичное значение “1000000000”, что соответствует числу  $2^9 = 512$ , которое может быть получено в программе микроконтроллера.

Преобразование аналогового сигнала в цифровой внутри АЦП проходит в три этапа:

1. **Дискретизация.** Выбираются значения из исходного аналогового сигнала через равные временные промежутки (1.).

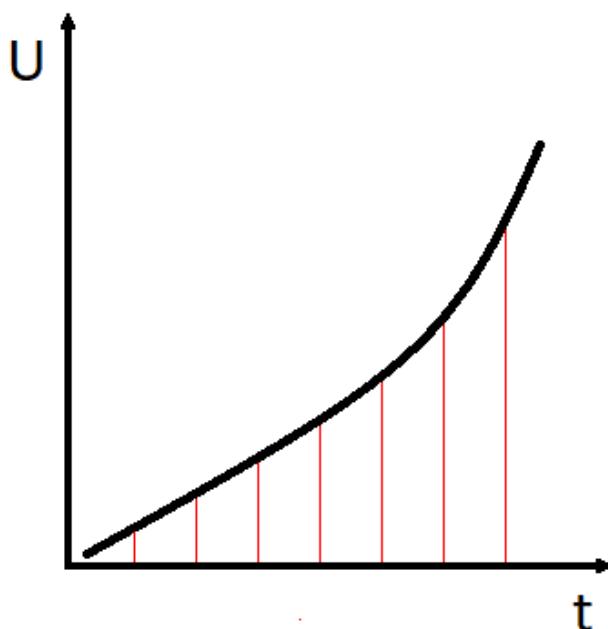


Рис. 3.4.5: Дискретизация.

Характеристика, отражающая эти временные промежутки, называется *частотой дискретизации*.

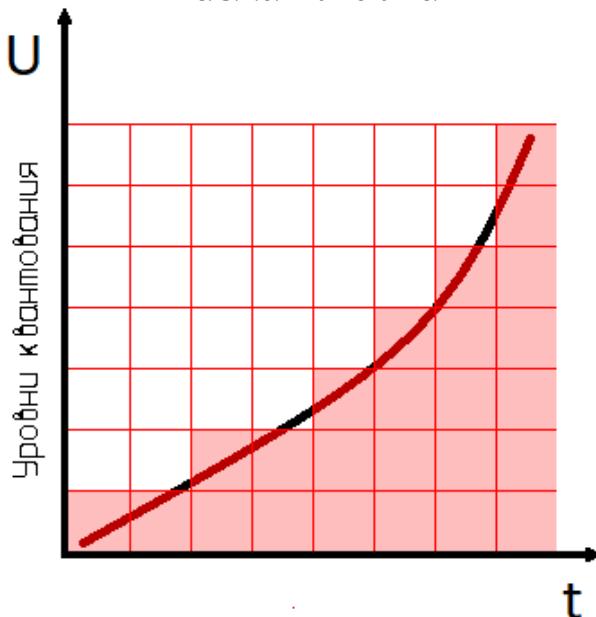
2. **Квантование.** Полученные значения заменяются ближайшим значением из набора фиксированных величин – *уровней квантования* (3.4.6.)
3. **Кодирование.** Квантованным значениям присваивается цифровой код (3.4.7.)

Чем выше частота дискретизации и чем больше уровней квантования, тем точнее преобразование.

Одной из характеристик АЦП является *разрядность*. Она определяет количество значений, которое может выдать АЦП. Посмотрим на последний график: для кодирования значений используется три бита, значит АЦП, описываемый таким графиком, имеет, соответственно, разрядность 3 бита. То есть  $2^3 = 8$ , что равно количеству уровней квантования.

Вот и ответ на поставленный вопрос. АЦП Arduino 10-ти разрядный,  $2^{10} = 1024$ . Именно столько значений АЦП Arduino может выдать.

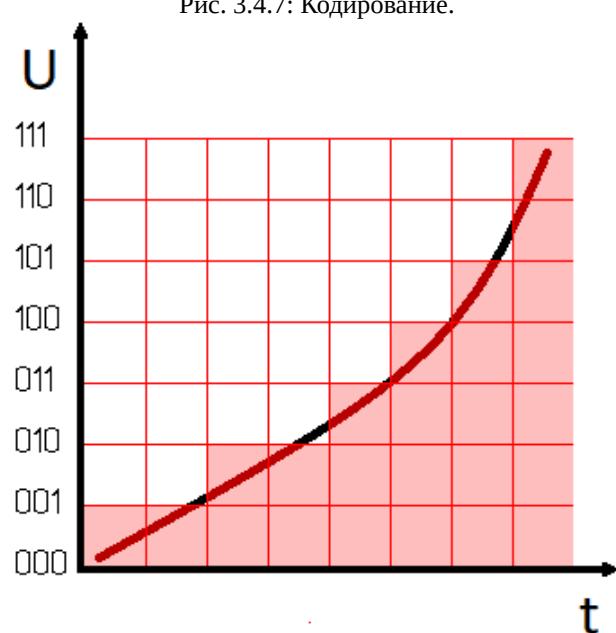
Рис. 3.4.6: Квантование.



Ещё есть такое устройство как ЦАП - Цифро-Аналоговый Преобразователь, который, как нетрудно догадаться, выполняет функцию, обратную функции АЦП - преобразует цифровой сигнал в аналоговый. Область применения ЦАП и АЦП достаточно широка: в звуковых и видеокартах, в мониторах, в различной акустической аппаратуре, в измерительных приборах, и многих других видах техники.

Стоит упомянуть про 8-битную музыку в древних игровых консолях. Её название отражает разрядность ЦАП звуковых чипов тех консолей – 8 бит. Именно такой ЦАП позволял выдавать тот самый резковатый, хлопающий и шипящий звук.

Рис. 3.4.7: Кодирование.





## Глава 4

# Широтно-импульсная модуляция

### 4.1 Общее описание принципов работы

Широтно-импульсная модуляция, или сокращённо *ШИМ*, позволяет выдавать на цифровом порту Arduino напряжение в диапазоне от 0 до 5 вольт, используя при этом только два сигнала – **HIGH** (логическая единица, при которой напорт подается 5 В) и **LOW** (логический ноль, при котором на порт подается 0 В.) Меняя быстро данные значения на порту, можно добиться, например, напряжения в 2.5 В.

### 4.2 Длина волны

При создании “мигающего светодиода” мы попеременно подавали на цифровой порт сигналы **HIGH** и **LOW**, с указанием задержки (в миллисекундах). Если мы посмотрим на вид сигнала на цифровом порту во времени (скажем, с помощью осциллографа), то увидим график, примерно как на рис. 4.2.1.

Где *длина периода* – расстояние между двумя ближайшими друг к другу точками в пространстве, в которых колебания происходят в одинаковой фазе.

Зная длину периода, можно рассчитать *частоту колебаний*, и наоборот – зная частоту, можно рассчитать длину волны.

При работе с ШИМ мы будем использовать длину периода, заданную в микросекундах (мкс). 1 микросекунда – это одна миллионная часть секунды. Для краткости записи подобных маленьких величин часто используется возвведение числа 10 в отрицательную степень. Ниже приведена таблица с указанием различных долей секунды<sup>1</sup>:

Название	Величина	Пример
секунда (с)	1с или $10^0$ с	$500 * 10^0 = 500$ с
миллисекунда (мс)	0.001с или $10^{-3}$ с	$500 * 10^{-3} = 500$ мс
микросекунда (мкс)	0.000001с или $10^{-6}$ с	$500 * 10^{-6} = 500$ мкс
наносекунда (нс)	0.000000001с или $10^{-9}$ с	$500 * 10^{-9} = 500$ нс

<sup>1</sup>Для полного списка кратных и дольных единиц см. статью Секунда в Википедии.

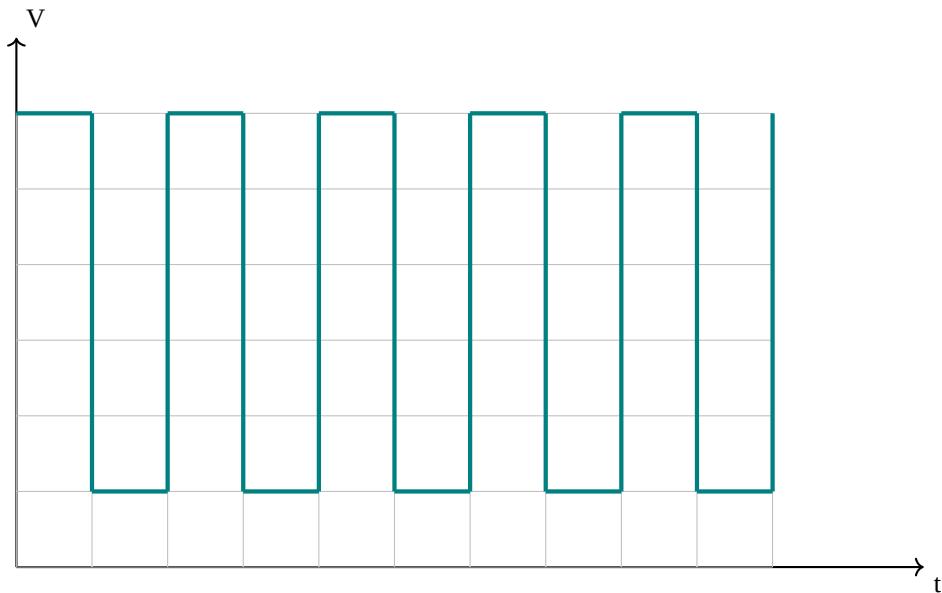


Рис. 4.2.1: Графическое отображение сигнала, меняющегося во времени, на цифровом порту.

### 4.3 Коэффициент заполнения

Каким образом задается напряжение из диапазона? Очень просто: путём изменения времени подачи того или иного сигнала. Чем больше времени на порту сигнал HIGH, тем выше напряжение. При этом, длина периода P остаётся фиксированной (например, 1000 микросекунд). Таким образом для ШИМ важно процентное отношение одного сигнала к другому, и, увеличивая время подачи одного сигнала, следует уменьшать время подачи другого (следовательно, если мы подаем сигнал HIGH 60% от отведенного времени, нужно заполнить оставшиеся 40% сигналом LOW).

Отношение периода следования сигнала к длительности импульса называется *скважностью*. В англоязычной литературе величина, обратная скважности, называется *коэффициентом заполнения* (англ. *duty cycle*).

Мы будем использовать термин “коэффициент заполнения”.



**Пример 2:** Мы хотим получить 2.5 вольта на цифровом порту 2, имея в распоряжении только два значения – 0 В и 5 В. Для этого нам потребуется реализовать ШИМ с коэффициентом заполнения 50%. При длине волны в 1000 микросекунд мы должны половину времени (500 микросекунд) заполнить положительным сигналом (сигнал HIGH) и подать его на выбранный порт, затем остальную часть заполним отрицательным сигналом (сигналом LOW). Если всё сделано правильно, то на 2 цифровом порту получим 2.5 вольта.

Для генерации нужного сигнала нам потребуется создать инструмент (функцию).

цию), который впоследствии мы будем использовать. Мы уже говорили о важности написания и использования собственных функций в программе – функции позволяют нам создавать модульные программы, упрощают поддержку существующего программного кода и написание нового кода.

Подумаем над тем, какой должна быть наша функция, реализующая ШИМ. Ниже приведён знакомый нам график, отображающий сигнал на цифровом порту – мы будем использовать этот график, как основу для написания функции.

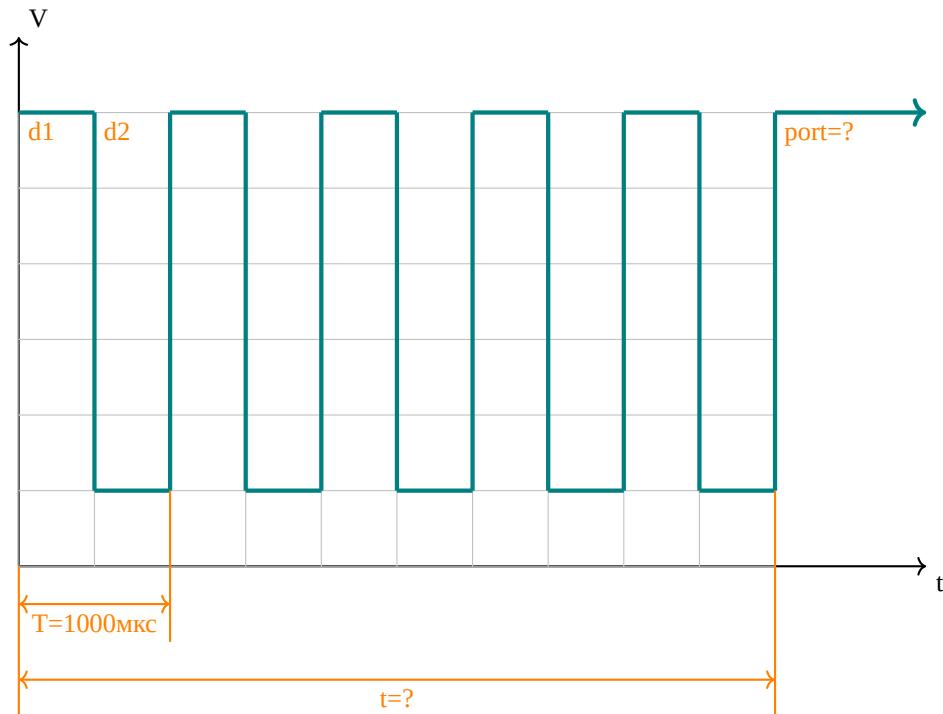


Рис. 4.3.2: Графическое отображение процесса генерации ШИМ-сигнала.

Имея перед глазами этот график, нетрудно набросать словесное описание функции (назовём функцию `rwm`, как сокращение от англ. *pulse width modulation*.)

Начнём с того, что, скорее всего, подобная функция должна принимать три параметра:

1. Номер порта (обозначаемый целым числом), на котором следует сгенерировать ШИМ сигнал; назовём этот параметр `port`.
2. Коэффициент заполнения, заданный дробным числом – к примеру, 50% будет задано, как 0.5; назовём этот параметр `dc` (от слов “duty cycle” – коэффициент заполнения.)
3. Длина ШИМ-сигнала в микросекундах; назовём этот параметр `t`.

Запишем тоже самое на языке C++:

```
void pwm(int port, float dc, long t) {
    // тело функции
}
```

О том, что такое **void**, будет сказано позже, пока что стоит принять как факт, что это начало объявления функции. Наверняка появились вопросы по новым типам переменных – **float** и **long**. Почему нельзя использовать привычный **int**? Дело в том, что переменная типа **int** не может хранить дробные числа, а также имеет диапазон значений от -32 768 до 32 767. Для хранения дробных чисел используется **float**, а для чисел, не входящих в диапазон **int**, стоит использовать **long**, имеющий диапазон от -2 147 483 648 до 2 147 483 647. Теперь подумаем над телом функции.

Обозначим длину периода константой **T**, равной 1000 мкс.

```
const int T = 1000; // мкс
```

Обратите внимание, что мы используем ключевое слово **const** для того, чтобы пометить **T**, как константу – мы всё равно не собираемся менять это значение. Кроме того, мы указали в комментарии, что значение задано в микросекундах (мкс), что упрощает чтение кода. В C++, да и в других языках, константа – это та же переменная, но её значение нельзя менять после её объявления.

Следующим этапом будет вычисление в теле функции значений переменных **d1** и **d2** на основе значения **dc**, заданного при вызове функции.

Значения задержек, заданных переменными **d1** и **d2**, необходимо вычислить на основе коэффициента заполнения, задаваемого параметром функции **dc** (который задаётся дробным значением – к примеру, 0.5.)

Формулы вычисления **d1** и **d2** представлены ниже.

$$d1 = T * dc \quad (4.1)$$

$$d2 = T - d1 \quad (4.2)$$

В коде это будет выглядеть следующим образом.

```
int d1 = T * dc;
int d2 = T - d1;
```

Видно, что как только мы вычислили **d1**, вычислить **d2** не составляет труда. Осталось только посчитать, сколько раз нужно повторить волну длиной **T**, чтобы сгенерировать сигнал длиной **t**:

```
int count = t / T;
```

Теперь у нас есть всё, что нужно для генерации нужного нам сигнала. Поскольку, исходя из описания выше, нам нужно будет повторять волну **count** раз, то для этого удобно использовать цикл **for** (цикл со счётчиком):

```
for (int c = 0; c < count; c++) {
    digitalWrite(port, HIGH);
    delayMicroseconds(d1);
    digitalWrite(port, LOW);
    delayMicroseconds(d2);
}
```

Полностью функция `pwm` будет выглядеть вот так:

```
void pwm(int port, float dc, long t) {  
    const int T = 1000; // мкс  
    int d1 = T * dc;  
    int d2 = T - d1;  
    int count = t / T;  
    for (int c = 0; c < count; c++) {  
        digitalWrite(port, HIGH);  
        delayMicroseconds(d1);  
        digitalWrite(port, LOW);  
        delayMicroseconds(d2);  
    }  
}
```

Готово! Осталось только задействовать функцию `pwm` в нашей программе.

### 4.3.1 Задачи

1. Написать программу, плавно включающую и выключающую светодиод. Собрать и протестировать схему.
2. Написать программу, реализующую “бегущий огонь” с использованием ШИМ. Собрать и протестировать схему.
3. Используя потенциометр, модифицировать систему из задания №2 таким образом, чтобы можно было регулировать яркость “бегущего огня”.
4. Разработать “бегущий огонь”, где следующий светодиод начинает плавно разгораться одновременно с затуханием предыдущего светодиода.



## Глава 5

# Синтез музыки и технологии

### 5.1 Звук

Как известно, звук – это колебания (вид сигнала), и каждому определённому звуку соответствует своя частота колебаний.

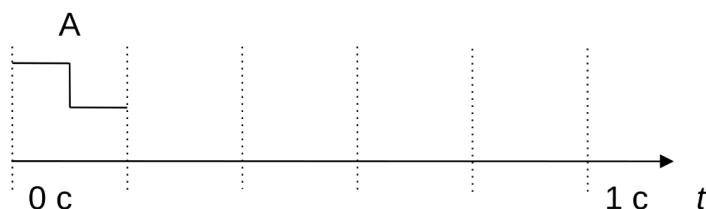
Частоты измеряются в Герцах (Гц), и один Герц (1 Гц) означает одно колебание в секунду. 10 колебаний в секунду – 10 Гц, 100 колебаний в секунду – 100 Гц и т. д. Если же мы говорим про частоты в 100 Гц и более, то удобнее использовать приставки Кило- (КГц), Мега- (МГц) и Гига- (ГГц): сигнал с частотой 1 КГц по-данный на цифровой порт колеблет мембранию динамика 1000 раз в секунду. Ниже приведена таблица некоторых кратных единиц частот в Герцах:

Название	Величина	Пример
Герц (Гц)	1 Гц или $10^0$ Гц	$100 * 10^0$ Гц = 100 Гц
Килогерц (КГц)	1000 Гц или $10^3$ Гц	$100 * 10^3$ Гц = 1 КГц
Мегагерц (МГц)	1 000 000 Гц или $10^6$ Гц	$100 * 10^6$ Гц = 1 МГц
Гигагерц (ГГц)	1 000 000 000 Гц или $10^9$ Гц	$100 * 10^9$ Гц = 1 ГГц

Таким образом, для генерации сигнала нам необходимо знать его частоту в Герцах, либо знать длину волны.

Зная период, мы можем узнать частоту, и наоборот – поскольку частота является ничем иным, как количеством повторений заданных колебаний в секунду. Это удобно представить визуально (5.1.1.)

Рис. 5.1.1: Визуальное представление частоты колебаний 5 Гц.



Если известно, что колебание А помещается 5 раз в 1 секунду, то говорят, что частота данного сигнала равна 5 Гц. Узнать период можно, разделив 1 секунду (заданную в микросекундах) на частоту (5 Гц):

$$\frac{1000000\text{мкс}}{5\text{Гц}} = 200000\text{мкс} \quad (5.1)$$

Получается, что длина волны равна 200000 мкс, или  $200 * 10^3$  мс. Если же нам известна длина волны и нужно узнать частоту, то необходимо разделить 1 секунду (в микросекундах) на длину волны – таким образом, получим частоту в Герцах. Всё просто.

Метод генерации звука похож на ШИМ. Основные отличия заключаются в том, что теперь мы должны изменять длину волны `len`, оставляя коэффициент заполнения неизменным – он описывается константой `DC` и всегда равен 0.5. Поскольку коэффициент заполнения всегда равен 50% (0.5), то время подачи сигналов `HIGH` и `LOW` всегда одинаково – иными словами, нам достаточно вычислить только задержку `d1`. Это показано на графике ниже:

Как и в случае с ШИМ, начнём писать функцию, которая будет реализовывать вышеописанные принципы. Функция будет называться `play_tone` и будет позволять генерировать звук с нужной частотой на указанном цифровом порту.

Посмотрим, что данная функция должна принимать в качестве параметров:

1. Номер цифрового порта, к которому подключен динамик и куда будет выводиться звук; назовём этот параметр “`port`”;
2. Частота “`f`”, измеряемая в Герцах.
3. Длина звукового сигнала; назовём этот параметр “`t`”.

На языке C++ это будет выглядеть примерно так:

```
void play_tone(int port, float f, long t) {
    // тело функции
}
```

Первым делом в теле функции из частоты найдём период `T`:

```
const long T = 1000000 / f;
```

Далее посчитаем длину задержки `d` – для этого мы должны поделить период на 2, то есть, найти *полупериод*:

```
long d = T / 2;
```

И посчитаем, сколько раз нам нужно повторить период длиной `p` микросекунд, чтобы заполнить время `t`:

```
int count = t / T;
```

Почти всё готово. Осталось только написать цикл, который будет генерировать заданную волну нужное количество раз. Здесь отлично подойдёт цикл со счётчиком (`for`):

```
for (int i = 0; i < count; i++) {
    digitalWrite(port, HIGH);
    delayMicroseconds(d);
    digitalWrite(port, LOW);
    delayMicroseconds(d);
}
```

В общем виде, функция выглядит так:

```
void play_tone(int port, float f, long t) {
    const long T = 1000000 / f;
    long d = T / 2;
    int count = t / T;
    for (int i = 0; i < count; i++) {
        digitalWrite(port, HIGH);
        delayMicroseconds(d);
        digitalWrite(port, LOW);
        delayMicroseconds(d);
    }
}
```

Наша функция генерации звука завершена. Теперь нам нужно подключить динамик к Arduino и протестировать нашу систему.



**Примечание:** Во многих случаях одна и та же задача может быть решена несколькими способами. К примеру, функция `play_tone` может быть реализована иначе; предложенная нами реализация является только одной из корректных.

## 5.2 Подключение динамика

Есть несколько вариантов динамиков, которые вы можете встретить. Например, есть обычные динамики, где мембрана колеблется магнитным полем и тем самым создаёт колебания воздуха, которые мы слышим, как звук. Есть пьезодинамики, в которых звук генерируется за счёт обратного пьезоэлектрического эффекта – механической деформации пьезоэлектрика под действием электрического поля.<sup>1</sup>

Подключение и обычных динамиков и пьезодинамиков похоже; для наших задач подойдёт как пьезодинамик “для Arduino”, так и обычный динамик-пищалка из персонального компьютера (а вы знали, что у вас в компьютере к системной плате подключен динамик?)

Схема подключения представлена на рис. 5.2.2.

Соберём указанную схему на макетной плате, и загрузим нашу программу генерации звука в Arduino. Не забудьте добавить в тело функции `loop` вызов нашей функции `play_tone` и настроить цифровой порт, к которому подключен динамик, на вывод.

Порт, к которому подключен динамик, лучше задать в виде константы `SPEAKER_PIN` в самом начале программы (до функции `setup`.)

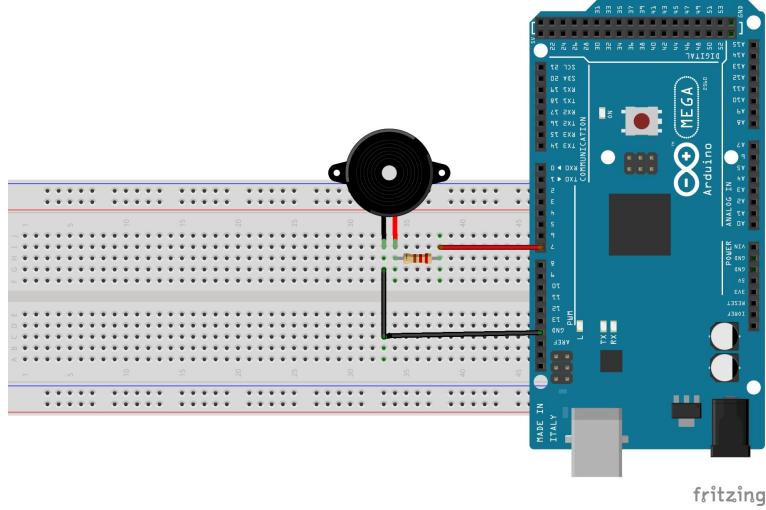
### 5.2.1 Задачи

1. Сгенерируйте постоянный сигнал с частотой 261.63 Гц.
2. Сделайте так, чтобы сигнал менялся между 261.63 Гц и 349.23 Гц с частотой в 1 секунду.

---

<sup>1</sup>См. статью “Пьезоэлектрический эффект” в Википедии для более подробного описания эффекта.

Рис. 5.2.2: Подключение динамика-”пищалки” к Arduino Mega 2560.



fritzing

3. Модифицируйте систему таким образом, чтобы частота сигнала зависела от положения ручки потенциометра.
4. Сделайте включение звукового сигнала по нажатию кнопки.

Теперь мы можем генерировать звуковой сигнал с нужной нам частотой. Однако, если мы хотим сгенерировать что-нибудь интересное – вроде мелодии – то нам потребуется использовать вполне определённые частоты. Здесь нам очень кстати будет хотя бы начальное знание музыкальной теории, но если таких знаний нет – не беда, разберём по ходу дела.

## 5.3 Теория ритма

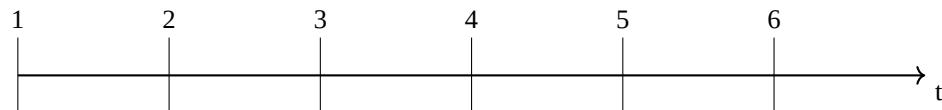
Наш путь в музыку начнётся с разбора теории построения *ритмов*. Думаю, большинство из нас представляют, что такое ритм – у многих именно он вызывает например рефлекс покачивания головой в такт, или машинальное отстукивание ритма на столешница стола, когда думаем над чем-то.

Чтобы понять, как строится ритм, необходимо знать две простые вещи, о которых мы сейчас поговорим.

### 5.3.1 Понятие такта

Во-первых, музыкальное произведение делится на отрезки времени, называемые *тактами* – как правило одинаковой длительности.<sup>2</sup>

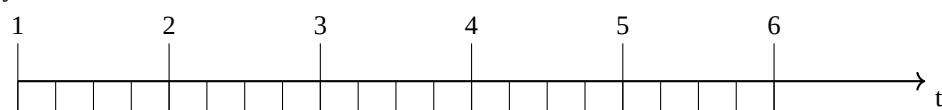
На рисунке ниже показано, как может выглядеть музыкальная композиция из шести тактов:



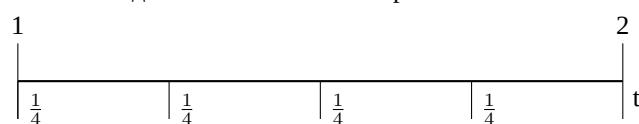
Протяжённость одного такта во времени определяется быстротой ритма, и про это мы поговорим позже. Пока можно представить, что один отрезок занимает одну условную единицу времени. Можете принять эту условную единицу за отрезок времени, удобный для вашего восприятия – например, одну секунду.

Во-вторых, эти такты делятся на ещё более мелкие части, по которым, как по “ячейкам”, раскладываются различные звуки. Достаточно большая часть музыки пронизана математикой, и первое математическое, с чем мы столкнёмся – это простые дроби. Одним из популярных способов деления такта на части является  $\frac{4}{4}$  или, по-другому называемый “четыре четверти”. В таком способе деления такта в него умещается ровно единица некоторого условного времени, про которое мы говорили выше.<sup>3</sup>

Если мы поделим каждый такт на четыре части, то получим следующую картину:



Возьмём отдельный такт и посмотрим на него внимательно:



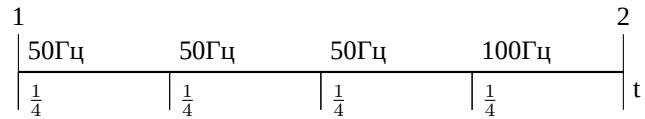
Если просуммируем все части, то снова получим единицу:

$$\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{1}{1} \quad (5.2)$$

<sup>2</sup>Музыка отличается большим разнообразием, и композиторы придумывают всё новые трюки, как получить желаемое впечатление у слушателя, поэтому здесь мы говорим с некоторыми допущениями.

<sup>3</sup>Существуют более сложные способы деления такта на части, которые дают суммарное значение меньше или больше единицы – про это мы поговорим позднее.

На каждую из четвертей можно задать какой-то звук – пока нам не важно, насколько он будет музыкальный. Допустим, что три четверти будут звучать с частотой 50Гц, и одна часть – с частотой 100Гц:



Поздравляю – мы только что получили простой ритм. Попробуем запрограммировать его, беря за длину такта  $T$  одну секунду, или 1000000 (миллион) микросекунд.

Между “ударами” необходимо поставить короткую задержку (например, в 100 мс), чтобы удары одинаковой частоты, идущие один за другим, несливались в один долгий звук.

```
// Номер порта, куда подключен динамик.
const int SPEAKER = 2;

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

// Функция воспроизведения звука указанной частоты.
void play_tone(int port, float f, long t) {
    const int T = 1000000 / f;
    int d = T / 2;
    int count = t / T;
    for (int i = 0; i < count; i++) {
        digitalWrite(port, HIGH);
        delayMicroseconds(d);
        digitalWrite(port, LOW);
        delayMicroseconds(d);
    }
}

void loop() {
    const long T = 1000000; // Длина такта в микросекундах
    play_tone(SPEAKER, 50, T / 4); // Четверть
    delay(100); // Задержка между звуками
    play_tone(SPEAKER, 50, T / 4); // Четверть
    delay(100);
    play_tone(SPEAKER, 50, T / 4); // Четверть
    delay(100);
    play_tone(SPEAKER, 100, T / 4); // Четверть
    delay(100);
}
```

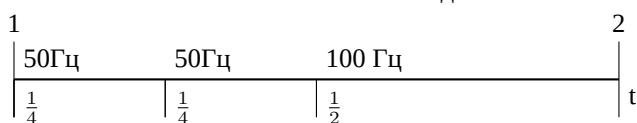
Можно теперь взять что-то посложнее.

### 5.3.2 Более сложные ритмы

Возможно некоторые из вас знают такую зажигательную композицию, как “We Will Rock You” в исполнении группы Queen<sup>4</sup>. Данная композиция имеет чётко выраженный, легко узнаваемый ритм, который можно упрощенно описать как “два притопа, один прихлоп”, вокруг которого строится вся композиция – вы сами можете достаточно легко повторить этот ритм, коротко топая ногой два раза, и потом делая “долгий” хлопок руками.

Структура этого ритма может быть описана следующим набором простых дробей (5.3.2.) Частоты звуков мы опять же берём произвольно.

Рис. 5.3.3: Ритм мелодии “We Will Rock You”



Как мы видимо из рисунка 5.3.2, деление такта на части не обязательно должно быть всегда равномерным – здесь мы получили две четверти, и одну половину. Суммарно же у нас опять выходит единица (см. формулу 5.3.) Попутно вспоминаем, что для сложения простых дробей нам необходимо их привести к общему знаменателю.

$$\frac{1}{4} + \frac{1}{4} + \frac{1}{2} = \frac{1}{4} + \frac{1}{4} + \frac{2}{4} = \frac{4}{4} = \frac{1}{1} = 1 \quad (5.3)$$

### 5.3.3 Музыкальная запись ритма

С точки зрения музыкальной нотации, данный ритм можно записать примерно следующим образом (см. рисунок 5.3.5.)

Рис. 5.3.4: Ритм мелодии “We Will Rock You” в музыкальной нотации (упрощенная версия.)



Если вы не знакомы с музыкальной нотацией (т.е. способом записи музыки), и изображение выглядит для вас совершенно непонятно, то не отчайрайтесь – на данном этапе нам достаточно увидеть, что есть три “закорючки”, которые обозначают звуки разной длительности: “♪” ( $\frac{1}{4}$ ), “♪” ( $\frac{1}{4}$ ) и “♪” ( $\frac{1}{2}$ ).

Расположение “закорючек” слева направо говорит нам о порядке “извлечения” звуков из музыкальных инструментов, а каждый вид начертания “закорючки” соответствует длительности звука, согласно следующей таблице (5.1.)

Существуют и более длинные и более короткие ноты, но они встречаются реже, чем перечисленные в таблице 5.1, поэтому для краткости мы их рассматривать не будем.

<sup>4</sup>Официальное музыкальное видео на данную композицию можно посмотреть здесь: <https://www.youtube.com/watch?v=-tJYN-eG1zk>

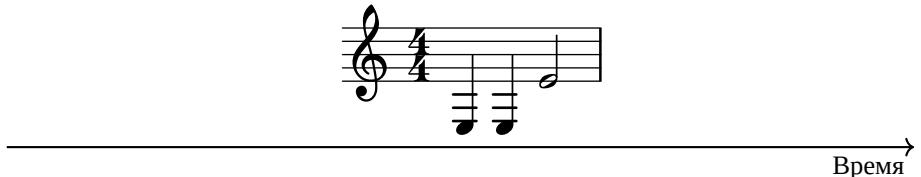
Таблица 5.1: Некоторые возможные длительности нот.

Начертание	Длительность	Название
♩	$\frac{1}{1}$	“Целая”
♪	$\frac{1}{2}$	“Половина”
♪	$\frac{1}{4}$	“Четверть”
♪	$\frac{1}{8}$	“Восьмая”
♪	$\frac{1}{16}$	“Шестнадцатая”

Какие выводы мы можем сделать в итоге? Что в музыкальной нотации звуки, которые необходимо “извлечь” из музыкального инструмента, записываются слева на право, как и обычный, привычный нам русский текст (или английский, к слову говоря.)

Мы можем расположить музыкальную запись для удобства нашего понимания на графике, где ось X, будет обозначать время, идущее слева направо:

Рис. 5.3.5: Музыкальный “график”.



С точки зрения программирования, код воспроизведения ритма может быть следующим:

```
// Здесь пропущен уже известный вам код настройки системы и реализации
// функции воспроизведения звука.
```

```
void loop() {
    const long T = 1000000; // Длина такта в микросекундах
    play_tone(SPEAKER, 50, T / 4); // Четверть
    delay(100);
    play_tone(SPEAKER, 50, T / 4); // Четверть
    delay(100);
    play_tone(SPEAKER, 100, T / 2); // Половина
    delay(100);
}
```

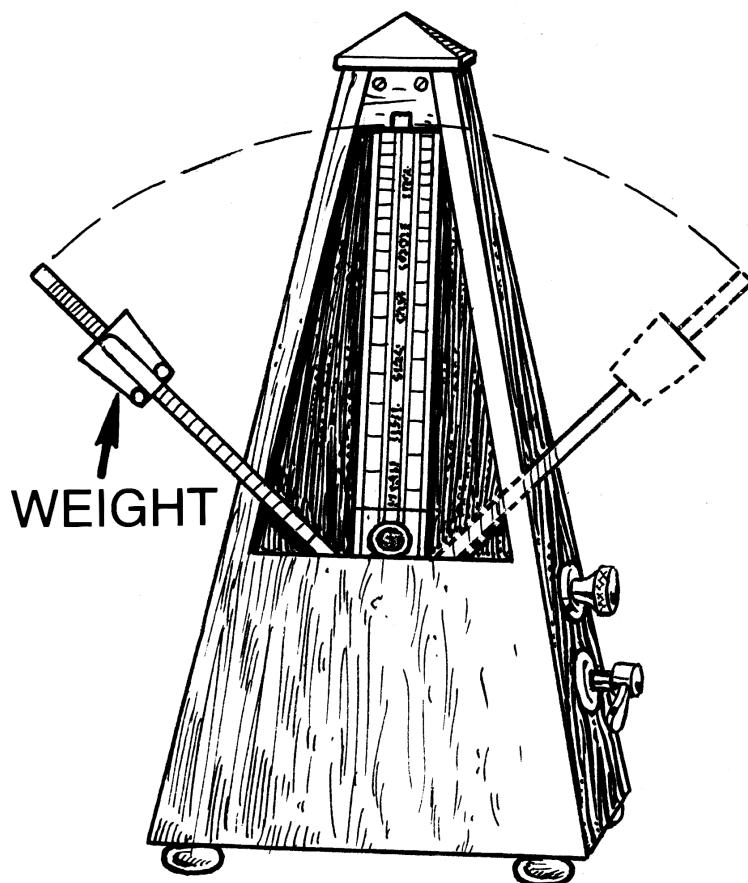
Теперь нам нужно более точно определить, какова длительность целого отрезка времени, чтобы посчитать длину его частей.

### 5.3.4 Темп музыки

Темпом в музыке называется скорость исполнения музыки, если говорить упрощенно. В современной музыке темп измеряется в *ударах в минуту* (буквальный перевод английского “Beats per Minute”, сокращённо называемый *BPM*.)

Темп в музыке может задаваться с помощью специального маятника, который раскачивается из стороны в сторону с равномерной скоростью, делая удары (или щелчки) в крайних положениях. Такой маятник называется *метрономом*.

Рис. 5.3.6: Схематическое изображение механического метронома (источник: [https://commons.wikimedia.org/wiki/File:Metronome\\_\(PSF\).png](https://commons.wikimedia.org/wiki/File:Metronome_(PSF).png)).

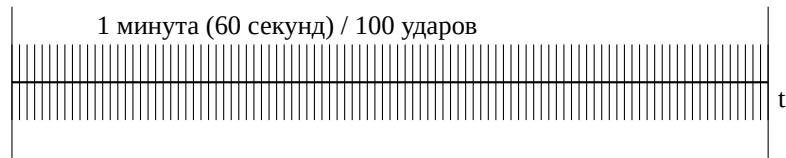


В современном мире роль механического маятника часто заменяет специальное компактное электронное устройство или даже приложение для мобильного телефона, издающее щелчки через равные промежутки времени с заданной скоростью.

Чем больше ударов в минуту, тем выше темп – и выше скорость ритма. Примерами BPM могут служить 60, 80 и 100 ударов в минуту.

Но как связать скорость ритма с делением такта на части? Оказывается, равномерные удары метронома задают длину одной четверти ( $\frac{1}{4}$ ), на которые у нас в примерах выше делился такт.

Таким образом, если темп ритма задан в 100 ударов в минуту, то это значит, что в минуту помещается ровно 100 четвертей.



Расстояние между двумя ударами можно рассчитать по формуле 5.4.

$$\frac{60 \text{ с}}{100 \text{ ударов}} = 0.6 \text{ с} = 60 \text{ мс} = 60000 \text{ мкс} \quad (5.4)$$

Исходя из формулы 5.4 мы получаем длительность одной четверти в 0.6с. Но для удобства расчётов мы должны посчитать длину целого такта, а не его части. Поскольку в такт, который мы рассматриваем в примере, помещается четыре четверти, для нахождения целого достаточно умножить четверть на четыре (см. формулу 5.5).

$$\frac{1}{4} * 4 = \frac{4}{4} = \frac{1}{1} = 1 \quad (5.5)$$

Зная это, можно вывести общую формулу 5.6 для вычисления длительности целого такта.

$$\frac{60 \text{ с}}{\text{BPM}} * 4 = \text{Длина целого такта} \quad (5.6)$$

Теперь мы можем музыкально точно рассчитать длину звуков, из которых строится ритм произведения – в том числе, композиции “We Will Rock You”, которую мы рассматривали выше. Для неё BPM равен примерно 80 ударов в минуту.<sup>5</sup>

Попробуем запрограммировать его.

```
// Номер порта, куда подключен динамик.
const int SPEAKER = 2;

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

// Функция воспроизведения звука указанной частоты.
void play_tone(int port, float f, long t) {
    const int T = 1000000 / f;
    int d = T / 2;
    int count = t / T;
    for (int i = 0; i < count; i++) {
        digitalWrite(port, HIGH);
        delayMicroseconds(d);
        digitalWrite(port, LOW);
        delayMicroseconds(d);
    }
}

void loop() {
```

<sup>5</sup> В разных источниках могут быть указаны разные варианты BPM, но в целом темп этой композиции обычно находится в диапазоне от 84 до 88 ударов в минуту.

```

const int BPM = 80; // Удары в минуту

// Одна минута в микросекундах
const long MINUTE = 60000000;

// Длина целого такта в микросекундах
const long T = (MINUTE / BPM) * 4;

play_tone(SPEAKER, 50, T / 4); // Четверть
delay(100);
play_tone(SPEAKER, 50, T / 4); // Четверть
delay(100);
play_tone(SPEAKER, 100, T / 2); // Половина
delay(100);
}

```

Если вы загрузите данный проект в Arduino, то обнаружите, что ритм получился слишком медленный и не совпадает с темпом оригинальной композиции. Так произошло потому, что мы не учли один важный момент – в такте должно быть две группы “два притопа, один прихлоп” (см. рисунок 5.3.7.)

Рис. 5.3.7: Ритм мелодии “We Will Rock You” в музыкальной нотации (полная версия.)



Из-за этого в оригинал композиция кажется в два раза быстрее – не потому, что там выше BPM, а просто по тому, что сами ноты короче.

Адаптируем наш код соответствующим образом.

```

// ...

void loop() {
const int BPM = 80; // Удары в минуту

// Одна минута в микросекундах
const long MINUTE = 60000000;

// Длина целого такта в микросекундах
const long T = (MINUTE / BPM) * 4;

play_tone(SPEAKER, 50, T / 8); // Восьмая
delay(100);
play_tone(SPEAKER, 50, T / 8); // Восьмая
delay(100);
play_tone(SPEAKER, 100, T / 4); // Четверть
delay(100);
play_tone(SPEAKER, 50, T / 8); // Восьмая

```

```
delay(100);
play_tone(SPEAKER, 50, T / 8); // Восьмая
delay(100);
play_tone(SPEAKER, 100, T / 4); // Четверть
delay(100);
}
```

Ритм готов, однако мы всё ещё выбираем частоты наших “инструментов” произвольно – пришло время это исправить.

## 5.4 Базовые принципы благозвучия

Человеческий слух устроен так, что нам может нравиться сочетание одних частот звуков, и не нравиться сочетание других. Говорят, что те звуки, которые хорошо сочетаются, образуют **консонанс**, тогда как “неприятные” комбинации образуют **диссонанс**.

Например, частоты 50 Гц и 100 Гц звучат в сочетании довольно неплохо, хотя и не являются музыкальными – а всё потому, что одна частота ровно в два раза больше другой. Понимание такого малозначимого, казалось бы, факта, позволяет нам строить достаточно мелодичные произведения путём сочетания звуков, кратных по частоте друг другу.

Воспроизведение одинаковых по частоте звуков образует идеальный консонанс, так как их волны накладываются друг на друга, при этом усиливаясь.



**Эксперимент №10:** Запрограммируйте две Arduino таким образом, чтобы они воспроизводили одинаковые частоты и включите их одновременно. Сможете ли вы услышать, что частоты совпадают? Что вы чувствуете, когда это происходит?

Перепрограммируйте Arduino, чтобы частоты различались на 10Гц, 20Гц, 30Гц и т.д., увеличивая шаг сначала на 10Гц, потом на 100Гц. Какие комбинации получились неприятные, какие терпимые, какие приятные?



**Эксперимент №11:** Попробуйте воспроизвести какую-нибудь звуковую частоту на Arduino, и одновременно подстроить звук вашего голоса под этот звук, пропев какую-нибудь гласную (например, “А”) – возможно у вас произойдёт тот самый “Ага!” момент, когда вы услышите совпадение частоты вашего голоса и звука динамика.

## 5.5 Октачная система

Как вы, возможно, знаете, музыка строится из нот – их всего семь: “до”, “ре”, “ми”, “фа”, “соль”, “ля”, “си”. Каждой ноте соответствует определённая частота. Но если мы с вами возьмём пианино (или синтезатор) и посмотрим на его клавиатуру, то увидим, что клавиш на пианино гораздо больше, чем нот. Почему?

Оказывается, ноты объединяются в группы, которые называются **октавами**. В одной октаве семь нот (от “до” до “си”), всего октав девять.

Для удобства мы пронумеруем все октавы от 0 (октава с самыми низкими частотами) и до 8 (октава с самыми высокими частотами.)

Таким образом мы получаем  $7 * 9 = 63$  разных нот<sup>6</sup> в октавной системе.

Каким же образом ноты можно легко различить, если они называются одинаково и различаются только номером октавы? Оказывается, для этого есть достаточно удобная **научная нотация** – в таблице 5.5 показано соответствие слоговых

<sup>6</sup>На самом деле, разных звуков в октавной системе больше, чем нот – об этом речь пойдёт чуть позже.

названий нот (“до”, “ре”, “ми”, “фа”, “соль”, “ля”, “си”) и обозначений в научной нотации.

Слогоное обозначение	Научное обозначение
До	C
Ре	D
Ми	E
Фа	F
Соль	G
Ля	A
Си	B (H)

Обратите внимание, что нота “Си” может быть обозначена либо буквой “B” (английский вариант), либо буквой “H” (немецкий вариант.) Мы с вами будем использовать только английский вариант и всегда записывать “Си” как “B”.

Прелесть научной нотации не только в краткости записи нот (что удобно для их программирования), но и в том, что после буквы обычно ставится цифра – номер октавы, к которой принадлежит нота. Например, “C0” – то нота “До” нулевой октавы, а нота “G5” – это нота “Соль” пятой октавы.

Однаковые ноты из разных октав различаются своей частотой звука – прям, не просто различаются “как попало”, а согласно строгому правилу: они кратны друг другу. Если взять ноту “C0” (“до” нулевой октавы) и “C1” (“до” первой октавы), то их частоты будут различаться ровно в два раза.

Тут стоит вспомнить, что звуки, частоты которых различаются в кратное количество раз, приятны нашему слуху. Бинго! Мы только что поняли небольшую частичку музыкальной теории.

Если же мы хотим рассчитать частоту нот не из смежных октав, а допустим ноту “C0” и ноту “C3” из третьей октавы, то мы должны использовать более “хитрую” формулу. Дело в том, что каждая следующая октава повышает частоту выбранной ноты ровно в два раза.

Формула вычисления частоты выбранной ноты из нулевой октавы в произвольной октаве представлена ниже (5.7.)

$$f * 2^n \quad (5.7)$$

Где “f” – это частота выбранной ноты нулевой октавы, а “n” – то номер октавы, начиная с нуля.

Возьмём с вами для сравнения нулевую и четвёртую по счёту октаву – частоты нот представлены в таблице 5.5 (полная таблица октав представлена в Октавная система.)

№ октавы	Слогоное обозначение	Научное обозначение	Частота (Гц)
0	До	C0	16.352
	Ре	D0	18.354
	Ми	E0	20.602
	Фа	F0	21.827
	Соль	G0	24.500
	Ля	A0	27.500
	Си	B0 (H0)	30.868
4	До	C4	261.630
	Ре	D4	293.660
	Ми	E4	329.630
	Фа	F4	349.230
	Соль	G4	392.000
	Ля	A4	440.000
	Си	B4 (H4)	493.880

Видно, что нота “C0” имеет частоту 16.352 Гц. Если мы подставим это значение в формулу 5.7, то получим частоту ноты “C4” (см. формулу 5.8.)

$$f * 2^n = C0 * 2^4 = 16.352 * 2^4 = 261,632 \quad (5.8)$$

С полученными знаниями мы теперь можем запрограммировать простую мелодию, чем и займёмся в следующем разделе.

## 5.6 Программирование простых мелодий

Для того, чтобы запрограммировать мелодию, нам потребуется узнать ноты, из которых состоит данная мелодия, и их порядок. Как правило, эта информация записывается в виде нотной записи – но если вы ещё не умеете читать нотную запись, то можно найти мелодии в упрощенной записи, где используется буквенная (научная) нотация. К примеру, возьмём мелодию “Twinkle, Twinkle, Little Star”<sup>7</sup> – английскую колыбельную:

Рис. 5.6.8: “Twinkle, Twinkle, Little Star”



<sup>7</sup>[https://ru.wikipedia.org/wiki/Twinkle,\\_Twinkle,\\_Little\\_Star](https://ru.wikipedia.org/wiki/Twinkle,_Twinkle,_Little_Star)

Из таблицы 5.1, мы уже знаем, как различать нотные “закорючки”, чтобы понять их длительность, но мы пока не разбирали, как определить частоты нот по нотной записи. Поэтому ниже приводится полный список нот композиции в правильном порядке (см. ??), где ноты разбиты по строкам таким образом, чтобы каждая строка соответствовала одному такту.

№ такта	Ноты			
0	C4	C4	G4	G4
1	A4	A4	G4	
2	F4	F4	E4	E4
3	D4	D4	C4	
4	G4	G4	F4	F4
5	E4	E4	D4	
6	G4	G4	F4	F4
7	E4	E4	D4	
8	C4	C4	G4	G4
9	A4	A4	G4	

Используя наши знания про нотные “закорючки”, попробуем назначить длительность для каждой из нот, подписьав её в скобках (см. 5.6.)

№ такта	Ноты			
0	C4 ( $\frac{1}{4}$ )	C4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{4}$ )
1	A4 ( $\frac{1}{4}$ )	A4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{2}$ )	
2	F4 ( $\frac{1}{4}$ )	F4 ( $\frac{1}{4}$ )	E4 ( $\frac{1}{4}$ )	E4 ( $\frac{1}{4}$ )
3	D4 ( $\frac{1}{4}$ )	D4 ( $\frac{1}{4}$ )	C4 ( $\frac{1}{2}$ )	
4	G4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{4}$ )	F4 ( $\frac{1}{4}$ )	F4 ( $\frac{1}{4}$ )
5	E4 ( $\frac{1}{4}$ )	E4 ( $\frac{1}{4}$ )	D4 ( $\frac{1}{2}$ )	
6	G4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{4}$ )	F4 ( $\frac{1}{4}$ )	F4 ( $\frac{1}{4}$ )
7	E4 ( $\frac{1}{4}$ )	E4 ( $\frac{1}{4}$ )	D4 ( $\frac{1}{2}$ )	
8	C4 ( $\frac{1}{4}$ )	C4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{4}$ )
9	A4 ( $\frac{1}{4}$ )	A4 ( $\frac{1}{4}$ )	G4 ( $\frac{1}{2}$ )	

Для того, чтобы запрограммировать данную мелодию, удобно в начале программы объявить каждую ноту в виде константы. Каждая константа будет хранить частоту звука.<sup>8</sup>

Нам пока потребуются только ноты из четвёртой октавы. Константы разумно объявить до функции `Setup`, в самом верху программы.

```
const float c4 = 261.630;
const float d4 = 293.660;
const float e4 = 329.630;
const float f4 = 349.230;
const float g4 = 392.000;
const float a4 = 440.000;
const float b4 = 493.880;
```

<sup>8</sup>Имена констант обычно пишутся заглавными буквами, т.е. правильнее было бы именовать эти константы “C4”, “D4”, “E4” и т.д. Однако мы используем здесь буквы в нижнем регистре, чтобы избежать конфликтов имён с константами, которые уже есть в Arduino (к примеру, “A4”).

Не забывайте, что нам нужно ещё объявить константу, хранящую номер порта, куда подключен динамик – пусть это будет цифровой порт номер 8:

```
// Номер порта, куда подключен динамик:
const int SPEAKER_PIN = 8;
```

Также нам нужно знать темп мелодии – то есть, BPM. Для “Twinkle, Twinkle, Little Star” это параметр равен 120 ударов в минуту.

Как только мы объявили все необходимые константы и узнали BPM, то запрограммировать мелодию не составит труда. Напишем код нашей функции `loop`:

```
void loop() {
    const long BPM = 120;
    const long MINUTE = 60 * 1000000;
    const long T = (MINUTE / BPM) * 4;

    // 0-й такт.
    play_tone(SPEAKER_PIN, c4, T / 4);
    delay(100);
    play_tone(SPEAKER_PIN, c4, T / 4);
    delay(100);
    play_tone(SPEAKER_PIN, g4, T / 4);
    delay(100);
    play_tone(SPEAKER_PIN, g4, T / 4);
    delay(100);

    // 1-й такт.
    play_tone(SPEAKER_PIN, a4, T / 4);
    delay(100);
    play_tone(SPEAKER_PIN, a4, T / 4);
    delay(100);
    play_tone(SPEAKER_PIN, g4, T / 2);
    delay(100);

    // и так далее
}
```

Допишите необходимые части кода (`setup`, `play_tone` и т.д.) и загрузите его в Arduino. Если вы всё сделали правильно, то у вас должна заиграть мелодия. Прекрасно!

Для тех, кому не терпится послушать, как звучит мелодия, мы предлагаем взглянуть в приложение C, где представлен полный код программы.

Но данное решение является не совсем оптимальным с точки зрения количества кода, которое необходимо написать. Решением данной проблемы является использование **массивов**.

## 5.7 Массивы

*Массив* – это переменная, состоящая из группы других переменных одного типа. В массиве мы сможем хранить ноты нашей мелодии.

Визуально массив можно представить в виде последовательности “коробочек”, каждая из которых имеет порядковый номер (называемый *индексом*) и может хранить один элемент (5.7.)

c4	c4	g4	g4	a4	a4	g4
0	1	2	3	4	5	6

В приведённом примере массив хранит семь элементов, при этом номер (индекс) первого элемента равен нулю (“Настоящие программисты считают с нуля!”), а номер последнего равен шести. Если мы попытаемся взять несуществующий элемент (например, седьмой, или минус первый), то это приведёт к ошибке.

Нам нужно создать массив из нужного количества элементов, если быть точным, то 28 элементов, по количеству нот, используемых в нашей мелодии, и заполнить массив значениями. Всё так же, как и раньше – указываем тип переменной и её название, но чтобы указать, что это массив, после имени пишем квадратные скобки и в них количество элементов, из которых будет состоять массив:

```
float melody[28] = {
    c4, c4, g4, g4,
    a4, a4, g4,
    f4, f4, e4, e4,
    d4, d4, c4,
    g4, g4, f4, f4,
    e4, e4, d4,
    g4, g4, f4, f4,
    e4, e4, d4,
};
```

Для обращения к определённым элементам массива нужно написать имя массива и в квадратных скобках номер элемента. Например, если захотелось нам поменять значение нулевого элемента, то мы могли бы сделать это так:

```
melody[0] = g4;
```

Этот массив стоит объявить перед функцией `loop` в нашей программе. Внутри же `loop` мы можем пройтись по данному массиву в цикле и воспроизвести каждую из нот, использовав в качестве номера элемента счётчик цикла:

```
void loop() {
    const long BPM = 120;
    const long MINUTE = 60 * 1000000;
    const long T = (MINUTE / BPM) * 4;

    for (int note_idx = 0; note_idx < 28; note_idx++) {
        play_tone(SPEAKER_PIN, melody[note_idx], T / 4);
        delay(100);
    }
}
```

Заметьте, насколько сократилась запись нашей мелодии. Однако мы временно потеряли возможность задавать длительность для каждой ноты в отдельности.

Чтобы это исправить, можно создать дополнительный массив с длительностями нот – назовём его `melody_t`.

Каждый элемент нашего массива `melody_t` будет содержать длительность ноты из массива `melody` в виде знаменателя простой дроби, где в числителе у нас находится длина такта. Например, нота номер ноль (“C4”) из массива `melody` имеет в музыкальном произведении длительность  $\frac{1}{4}$ , следовательно её длительность в массива `melody_t` будет записана, как 4.

```
// Массив с нотами (их частотами.)
float melody[28] = {
    c4, c4, g4, g4,
    a4, a4, g4,
    f4, f4, e4, e4,
    d4, d4, c4,
    g4, g4, f4, f4,
    e4, e4, d4,
    g4, g4, f4, f4,
    e4, e4, d4,
};

// Массив с длительностями нот.
float melody_t[28] = {
    4, 4, 4, 4,
    4, 4, 2,
    4, 4, 4, 4,
    4, 4, 2,
    4, 4, 4, 4,
    4, 4, 2,
    4, 4, 4, 4,
    4, 4, 2,
};
```

После этого следует обновить наш код воспроизведения мелодии:

```
// ...

void loop() {
    const long BPM = 120;
    const long MINUTE = 60 * 10000000;
    const long T = (MINUTE / BPM) * 4;

    for (int note_idx = 0; note_idx < 28; note_idx++) {
        play_tone(SPEAKER_PIN,
                  melody[note_idx],
                  T / melody_t[note_idx]);
        delay(100);
    }
}
```

Обратите внимание на код `T / melody_t[note_idx]`, который рассчитывает как раз длительность ноты – в числителе простой дроби стоит `T`, а в знаменателе – `melody_t[note_idx]`.

$$\text{Длина ноты в микросекундах} = \frac{T}{\text{melody\_t}[note\_idx]} \quad (5.9)$$

При всём этом запись из двух массивов выглядит достаточно громоздко, так как нужно следить за тем, чтобы оба массива (нот и их длительностей) совпадали по размеру.

Чтобы решить эту проблему, мы можем использовать двумерные массивы, о которых пойдёт речь в следующем разделе.

А пока самые нетерпеливые из вас могут заглянуть в приложение D и увидеть полный код программы, воспроизводящей мелодию выше.

### 5.7.1 Двумерные массивы

Было бы круто разместить наши ноты таким образом, чтобы каждая нота лежала в ячейке массива вместе со своей длительностью. К счастью, у нас есть такая возможность – мы можем использовать *двумерные массивы*.

Схематическое изображение двумерного массива представлено в виде таблицы 5.7.1.

№ строки	№ столбца	
	0	1
0	c4	4
1	c4	4
2	g4	4
3	g4	4
4	a4	4
5	a4	4
6	g4	2

Каждая строка нашего массива должна содержать описание одной ноты. Столбец с номером ноль содержит частоту ноты, а столбец номер один содержит её длительность в виде знаменателя простой дроби, где в числителе у нас находится длина такта. Например, нота номер ноль (“C4”) имеет в музыкальном произведении длительность  $\frac{1}{4}$ , следовательно её длительность будет записана, как 4.

Записать программно мелодию в виде двумерного массива можно следующим образом:

```
float melody[28][2] = {
    {c4, 4}, {c4, 4}, {g4, 4}, {g4, 4},
    {a4, 4}, {a4, 4}, {g4, 2},
    {f4, 4}, {f4, 4}, {e4, 4}, {e4, 4},
    {d4, 4}, {d4, 4}, {c4, 2},
    {g4, 4}, {g4, 4}, {f4, 4}, {f4, 4},
    {e4, 4}, {e4, 4}, {d4, 2},
    {g4, 4}, {g4, 4}, {f4, 4}, {f4, 4},
    {e4, 4}, {e4, 4}, {d4, 2},
};
```

Как можно видеть, теперь каждый элемент массива – это по сути одномерный массив из двух элементов, записанный в фигурных скобках. Например, элемент номер ноль нашего массива `melody` содержит массив `{c4, 4}` – частота ноты и её длительность.

Теперь мы можем адаптировать код воспроизведения мелодии под наши задачи:

```
// ...

void loop() {
    const long BPM = 120;
    const long MINUTE = 60 * 1000000;
    const long T = (MINUTE / BPM) * 4;
```

```

for (int note_idx = 0; note_idx < 28; note_idx++) {
    play_tone(SPEAKER_PIN,
              melody[note_idx][0],
              T / melody[note_idx][1]);
    delay(100);
}
}

```

Используя двумерные массивы, можно кратко и ёмко описать мелодию, даже намного более сложную, чем “Twinkle, Twinkle, Little Star”.

На этом этапе нам необходимо разобрать, как же работает *нотный стан* (называемый также *нотоносцем*), на котором располагаются ноты – для того, чтобы уметь самостоятельно определять, где какая нота (частота) находится.

## 5.8 Нотный стан

Мы можем программировать простые мелодии, не зная нотной записи (например, используя готовые примеры из интернета) и для большинства популярных мелодий (вроде “Имперского марша” из “Звёздных войн”) найти ноты в научной нотации (или даже готовые программы для Arduino!) не составит труда. Но в какой-то момент мы можем столкнуться с ситуацией, когда для нашей любимой мелодии есть только ноты и более ничего. Поэтому, прежде чем двигаться дальше, неплохо бы остановиться на нотной записи.

Посмотрим ещё раз на мелодию “Twinkle, Twinkle, Little Star”.

Рис. 5.8.9: “Twinkle, Twinkle, Little Star”



Ранее мы проигнорировали расположение нот по оси “Y” и вместо этого использовали готовые буквенные обозначения. Теперь пришло время внимательно посмотреть на эти группы по пять линий и обозначения на них.

Начнём с линий – они называются *нотным станом* (его также называют “нотоносцем”, поскольку он “несёт” на себе ноты.)

Поверх нотного стана записываются ноты и другие обозначения. В самом начале линий пишется большая закорючка, называемая *ключом* – ключ определяет положение определённой ноты на нотном стане. Выше в мелодии “Twinkle, Twinkle, Little Star” используется только один из видов ключа – называемого *скрипичным ключом*. Скрипичный ключ обводит кружочком-завитком ту линию, на которой располагается нота “соль” четвёртой октавы (“G4”):

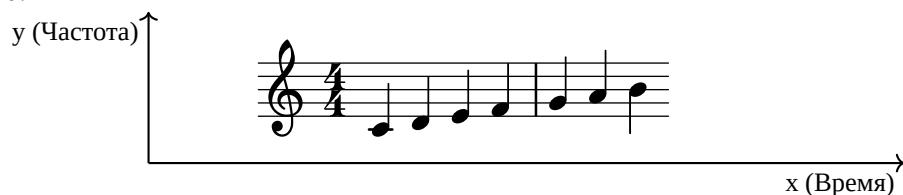
Рис. 5.8.10: Скрипичный ключ и нота “G4”.



Как можно видеть на рисунке выше, вторая линия снизу соответствует ноте “Соль” – поскольку она обведена скрипичным ключом. Следовательно, все ноты, которые “зацепились” за эту линию, будут нотой “Соль” четвёртой октавы (“G4.”)

Но ноты могут записываться не только на самих линиях, но и посерёдке между ними.

Схематически расположение нот можно представить в виде следующего графика:

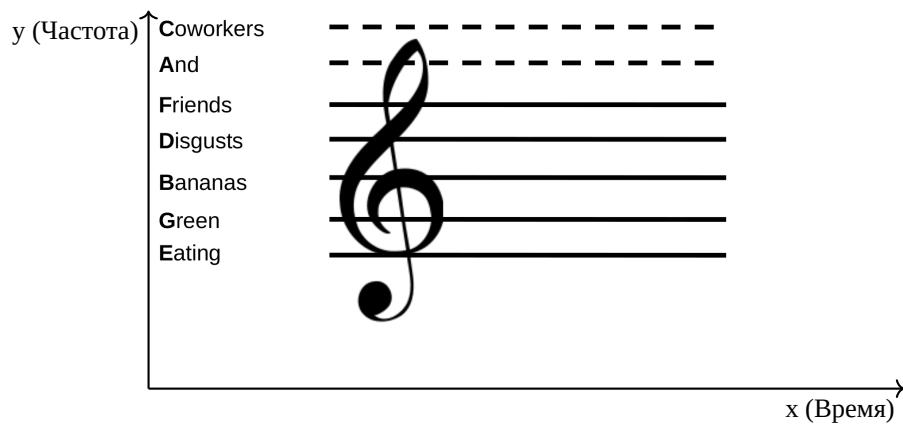


На данном “музыкальном” графике по порядку выстроены ноты “С”, “Д”, “Е”, “Ф”, “Г”, “А”, “В”. При движении вверх по оси “у”, частота звуков повышается, при движении вниз – понижается.

Следовательно, если мы будем двигаться выше по линейкам, то между второй снизу и третьей линейкой (она является средней на рисунке) будет находиться следующая нота после “Соль” (“G4”) – а именно нота “Ля” (“A4”). Третья же линейка (средняя) соответствует ноте “Си” (“B4”) и так далее. Если мы пойдём ещё выше, от над средней линейкой находится уже начало следующей, пятой октавы – то есть, нота “До” (“C5.”).

Если будем двигаться вниз от ноты “Соль” четвёртой октавы, то будем идти в обратную сторону: между первой снизу и второй линией находится нота “Фа” (“F4”), самая первая (нижняя) линия соответствует ноте “Ми” (“E4.”).

Для того, чтобы запомнить расположение нот на нотном стане, можно воспользоваться “запоминалками” – мнемониками. В интернете их можно найти великое множество. Вот наш вариант мнемоники для скрипичного ключа:



Если читать слова снизу вверх, то образуется фраза “Eating Green Bananas Disgusts Friends And Coworkers” (“Поедание зелёных бананов вызывает отвращение у друзей и коллег по работе”). Первая буква слова кодирует ноту в научной нотации. Вверху мы, помимо основных пяти линеек, подрисовали ещё две дополнительные линии, выделив их пунктиром. На нотном стане дополнительные линии сверху и снизу добавляются, если не хватает основных линий для записи композиции. Мнемоника кодирует только основные линии нотного стана, однако зная их, мы можем понять, какие ноты находятся между линиями.

## 5.9 Паузы в музыке

Есть ещё один момент, на котором мы до текущего момента не заостряли внимание — паузы в произведении. Правильные паузы также важны, как и сами ноты.

В нотной записи паузы отмечаются специальными значками (см. рисунок 5.9.11.)<sup>9</sup>

Рис. 5.9.11: Паузы в музыке.



Целая пауза равна по длине целой ноте, половинная — половине целой ноты и т.д. Иными словами, мы можем использовать подход, примененный нами для вычисления длительности нот, для расчета длительности пауз в произведении.

Обозначения пауз с их длительностями представлены в таблице 5.2.

Таблица 5.2: Некоторые возможные длительности пауз.

Начертание	Длительность	Название
—	$\frac{1}{1}$	“Целая”
—	$\frac{1}{2}$	“Половина”
{	$\frac{1}{4}$	“Четверть”
γ	$\frac{1}{8}$	“Восьмая”
γ	$\frac{1}{16}$	“Шестнадцатая”

Для реализации паузы в программе необходимо во-первых создать специальную ноту с нулевой частотой. Пауза в музыке называется “Покой” (или “Rest”

<sup>9</sup>В музыке существуют паузы, занимающие несколько тактов, либо очень короткие паузы – тридцать вторые, шестьдесят четвёртые и т.п. Используются они редко, поэтому мы их не будем разбирать здесь.

по-английски), поэтому для обозначения паузы в программе мы будем использовать заглавную букву “R”.

```
const float R = 0; // Пауза ("Rest")
```

Далее нам необходимо изменить функцию `play_tone` таким образом, чтобы она могла корректно “воспроизвести” паузы. Для этого нам необходимо добавить такое условие, чтобы, если частота ноты больше нуля, то функция выполняла тот код, который мы использовали раньше; иначе – чтобы выполнялась просто задержка.

```
// Функция воспроизведения звука указанной частоты.
void play_tone(int port, float f, long t) {
    if (f > 0) {
        const int T = 1000000 / f;
        int d = T / 2;
        int count = t / T;
        for (int i = 0; i < count; i++) {
            digitalWrite(port, HIGH);
            delayMicroseconds(d);
            digitalWrite(port, LOW);
            delayMicroseconds(d);
        }
    } else {
        delay(t / 1000); // Пауза
    }
}
```

Обратите внимание, что для создания задержки мы используем код `delay(t / 1000)` – делить `t` на 1000 необходимо по той причине, что время проигрывания ноты (`t`) задаётся в микросекундах, а функция `delay` принимает время ожидания в миллисекундах. Чтобы преобразовать микросекунды в миллисекунды, достаточно поделить количество микросекунд на 1000 (так как в каждой микросекунде содержится 1000 миллисекунд.) Почему же мы не могли использовать функцию `delayMicroseconds` для организации задержек (пауз) прямо в микросекундах, без преобразования? Ответ прост – `delayMicroseconds` не умеет долго ждать, и значения `t` для неё будут слишком большими; если мы попытаемся использовать `delayMicroseconds` с большими отрезками времени, то она не будет корректно их обрабатывать, и задержка получится неправильной.

Для наглядной демонстрации использования пауз возьмём другое музыкальное произведение – “Кабы не было зимы” из мультфильма “Простоквашино”.

В нотной записи на рисунке 5.9.12 представлена часть мелодии, которую мы попытаемся переложить на программный код. Полную версию мелодии можно увидеть на рисунке В.0.1.

Можно заметить две четвертные паузы (R), которые необходимо добавить в массив нот, чтобы произведение звучало, как в оригинале.

Можно также заметить двойные ноты, записанные одна над другой – это значит, что данные ноты должны играться одновременно. Для упрощения нашей задачи мы пока будем брать самую верхнюю ноту из группы.

Попробуем вписать ноты в массив и послушать, как они будут звучать. Чтобы не запутаться, разобьём ноты по группам, согласно тактам (по одной группе на

Рис. 5.9.12: Часть мелодии “Кабы не было зимы” из мультфильма “Простоквашино”.



строке) – а рядом с каждой группой в виде комментария напишем номер этого такта (например, такт номер ноль мы пометили /\* 0 \*/.)

```
float prostokvashino[28][2] = {
    /* 0 */ {b3, 8}, {b3, 8}, {b4, 8}, {f4, 16}, {a4, 8}, {g4, 8}, {e4, 4},
    /* 1 */ {d4, 8}, {d4, 8}, {d5, 8}, {c5, 16}, {c5, 8}, {b4, 8}, {R, 4},
    /* 2 */ {d5, 8}, {c5, 8}, {a4, 8}, {f4, 8}, {c5, 8}, {b4, 8}, {b4, 4},
    /* 3 */ {b3, 8}, {b3, 8}, {b4, 8}, {a4, 16}, {a4, 8}, {g4, 8}, {R, 4},
};
```

При воспроизведении мелодия будет похожей на оригинал, однако вы можете заметить некоторые “несоответствия”. Источников данных несогласий несколько. Например, мы не учитываем, что длительность некоторых нот, которые помечены точкой справа (вот так: “♪.”), больше стандартной.

## 5.10 Ноты с точками

В музыкальной нотации точка, которая ставится справа от ноты, увеличивает её длительность на половину от базовой.

Например, если у нас точка идёт после восьмушки (“♪.”), то следовательно к её длительности будет прибавляться половина от её длительности. Половинка от восьмушки – это шестнадцатая. Чтобы сложить простые дроби, которые у нас получились, необходимо привести их к общему знаменателю. И формула вычисления длительности будет следующая:

$$\text{♪.} = \text{♪} + \frac{1}{8} = \frac{1}{8} + \frac{1}{16} = \frac{2}{16} + \frac{1}{16} = \frac{3}{16} \quad (5.10)$$

Получившееся число  $\frac{3}{16}$  для нас неудобно, так как мы в программе подставляем в числитель дроби длительность одного такта, а тут у нас получается, что необходимо поставить длину трёх тактов. Чтобы избавиться от этой неудобной тройки в числителе, мы можем разделить числитель и знаменатель на 3.

$$\frac{3/3}{16/3} = \frac{1}{16/3} \quad (5.11)$$

Получившееся число  $16/3$  необходимо подставить в массив с нашими нотами. Например, третья нота нулевого такта – “B4” – как раз восьмая с точкой. В массиве её длительность надо исправить – вместо {b4, 8} написать {b4, 16.0}

/ 3.0}. Тоже самое необходимо сделать с другими нотами, возле которых стоит точка.

Для того, чтобы данный код уместился в книгу, нам пришлось разбить каждый такт на две строки, но по комментариям и отступам должно быть понятно, что происходит в коде.

```
float prostokvashino[28][2] = {
    /* 0 */ {b3, 8}, {b3, 8}, {b4, 16.0 / 3.0},
    /* */ {f4, 16}, {a4, 8}, {g4, 8}, {e4, 4},
    /* 1 */ {d4, 8}, {d4, 8}, {d5, 16.0 / 3.0},
    /* */ {c5, 16}, {c5, 8}, {b4, 8}, {R, 4},
    /* 2 */ {d5, 8}, {c5, 8}, {a4, 8},
    /* */ {f4, 8}, {c5, 8}, {b4, 8}, {b4, 4},
    /* 3 */ {b3, 8}, {b3, 8}, {b4, 16.0 / 3.0},
    /* */ {a4, 16}, {a4, 8}, {g4, 8}, {R, 4},
};
```

Теперь наша мелодия звучит ещё более похоже на оригинал.

Здесь стоит упомянуть, что в музыке встречаются ноты двумя точками справа, что даёт удлинение ноты на половину её длительности и на половину от половины – но подобные ситуации редки из-за неудобства расчёта необходимой длительности при чтении музыкантом нот с листа. То ли дело нам, программистам – прочитали не спеша, запрограммировали, а там пусть компьютер сам пыжится над нашим творением!

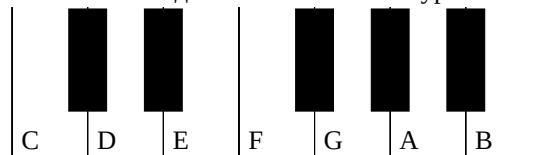
## 5.11 Полутона, диезы и bemоли

Мы с вами всю эту главу говорим, что нот в октаве семь, и суммарно звуков 63 штуки, если считать по всем октавам. Но это не совсем точно – на самом деле звуков в октаве не семь, а двенадцать!

Вот так сюрприз. Откуда берутся дополнительные пять звуков? Оказывается, что в музыке есть так называемые *полутона*, которые позволяют разнообразить музыку новыми звуками.

Нагляднее всего эти скрытные звуки легче всего увидеть на клавиатуре пианино.

Рис. 5.11.13: Одна октава на клавиатуре пианино.



Видно, что между парами нот “C”–“D”, “D”–“E”, “F”–“G”, “G”–“A”, “A”–“B” находятся чёрные клавиши. Если мы посчитаем, сколько всего клавиш в одной октаве, то получим двенадцать штук – двенадцать звуков.

Объясняется это тем, что между двумя соседними клавишами *расстояние в один полутон*, если представить частотный диапазон октавы как некий отрезок. Большинство пар нот имеют друг от друга достаточное расстояние, чтобы туда,

ровно посерёдке, добавить ещё одну клавишу (исключение составляют пары “B”–“C”, “E”–“F”.)

Каких-то особых закорючек для этих дополнительных звуков в музыке не применяется, однако есть “модификаторы” для основных семи нот, поднимающих или понижающих их частоту на полутон.

Для того, чтобы например получить частоту клавиши между парой “C”–“D”, можно поднять частоту “C” на пол-тона, либо понизить частоту “D” на те же полтона.

Модификатор, повышающий частоту ноты на пол-тона называется *диезом*, тогда как аналогичный модификатор, понижающий частоту на пол-тона, называется *бемолем*.

Ноты с модификатором “Диез” помечаются решёткой (“#”), стоящей перед нотой – например, на рис. 5.11.14 изображена “Фа Диез” четвёртой октавы.

Рис. 5.11.14: “Фа Диез” четвёртой октавы.



Чтобы высчитать частоту “F4#”, надо найти среднее арифметическое для частот ноты “F4” и следующей перед ней ноты “G4” (см. формулу 5.12.)

$$\frac{F4 + G4}{2} = F4\# \quad (5.12)$$

Программно вычислить “F4#” не составляет труда, как показано в коде ниже. Заметьте, что мы обозначили получившуюся частоту, как `f4s`, так как “Фа Диез” по-английски пишется, как “Fa Sharp”, и мы используем букву “s” из слова “sharp” после ноты для краткости.

```
const float f4 = 349.230;
const float g4 = 392.000;
const float f4s = (f4 + g4) / 2; // F4 Диез
```

Подобный подход работает и с другими нотами.

Кстати, для тех нот, после которых нет чёрной клавиши (“B”, “E”), диезом является просто следующая нота – например, “E4#” – это “F4”, а “B4#” не что иное, как “C5”.

Таким образом:

```
const float e4 = 329.630;
const float f4 = 349.230;
const float e4s = f4; // E4 Диез
```

Бемоли по логике работают схоже с диезами, с одним отличием – они **понижают** частоту ноты на половину тона. Обозначаются бемоли специальным символом “♭”, который ставится перед нотой, на которую “накладывается” модификатор “бемоль”.

Для примера возьмём “E4♭” (см. рисунок 5.11.15.)

Бемоль для неё – это частота ровно посерёдке между “E4” и предыдущей нотой от неё (“D4”), как показано на формуле 5.13.

Рис. 5.11.15: “Фа Диез” четвёртой октавы.



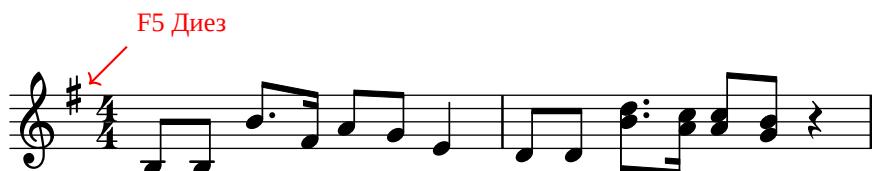
$$\frac{E4 + D4}{2} = E4\flat \quad (5.13)$$

Для обозначения bemолей в программном коде мы будем добавлять букву “f” к имени ноты, после её номера октавы, так как в английском ноты с bemолями называются “приплюснутыми” (“flat”) – например, “E4 Бемоль” будет называться “E4 Flat”.<sup>10</sup>

```
const float d4 = 293.660;
const float e4 = 329.630;
const float e4f = (e4 + d4) / 2; // E4 Бемоль
```

Все эти диезы и bemоли мы пока с вами рассматривали в вариантах, когда знак-модификатор пишется разу перед нотой, на которую “накладывается заклинание” – они называются знаками “по-месту”, или “случайными” (англ. *accidentals*). Но музыкантам неудобно писать подобные модификаторы перед каждой нотой, если подобных случаев в композиции много. Чтобы решить эту проблему, в музыке используются знаки диезов и bemолей, которые ставятся в начале нотного стана – в начале линий нотоносца. Влияние таких значков распространяется на все ноты подобные той, на которую наложено “заклинание”.

Вернёмся к мелодии “Кабы небыло зимы”. Если посмотреть на начало каждой строки, то можно увидеть решёнку на “F5” – это означает, что все ноты “F” будут диезами.



Зная это, мы можем соответствующим образом модифицировать код нашей мелодии.

```
const float f4s = (f4 + g4) / 2;

// ...

float prostokvashino[28][2] = {
    /* 0 */ {b3, 8}, {b3, 8}, {b4, 16.0 / 3.0},
    /* 1 */ {f4s, 16}, {a4, 8}, {g4, 8}, {e4, 4},
    /* 2 */ {d4, 8}, {d4, 8}, {d5, 16.0 / 3.0},
    /* 3 */ {c5, 16}, {c5, 8}, {b4, 8}, {R, 4},
    /* 4 */ {d5, 8}, {c5, 8}, {a4, 8},
```

<sup>10</sup> В музыке иногда встречаются двойные диезы и двойные bemоли, что означает необходимость брать ноту на два полутона выше или ниже – по сути, брать следующую или предыдущую ноту.

```

/* */ {f4s, 8},  {c5, 8},  {b4,          8},  {b4,  4},
/* 3 */ {b3,  8},  {b3,  8},  {b4, 16.0 / 3.0},
/* */ {a4, 16},  {a4,  8},  {g4,          8},  {R,   4},
};

// ...

```

## 5.12 Музыкальный размер

Вы могли видеть, что на нотном стане в самом начале, возле скрипичного (или басового) ключа часто написано  $\frac{4}{4}$  – что же это означает?

Пометка  $\frac{4}{4}$  (читается как “четыре четверти”) обозначает *музыкальный размер* произведения. С точки зрения кодирования мелодии в программе это не влияет ни на частоту нот, ни на их длительность. При этом, однако, данная пометка непрямую влияет на звучание произведения, и без её учёта все произведения будут звучать “плоско” и менее интересно.

Удивительный эффект музыкальный размер даёт благодаря *акцентированию* определённых нот.

Например, посмотрим ещё раз на “Twinkle, Twinkle, Little Star” (5.12.16.)

Рис. 5.12.16: “Twinkle, Twinkle, Little Star” в размере четыре четверти.



Поскольку композиция записана в музыкальном размере “четыре четверти”, то в один такт убирается ровно четыре четвёртных ноты, или суммарно единица (или одна целая нота.) Числитель данной дроби указывает, сколько частей – или, по-другому называемые *долей* – убирается в такт. Знаменатель дроби указывает, на какие именно доли делится такт. В размере “четыре четверти” такт делится на четыре части по одной четвертной ноте.

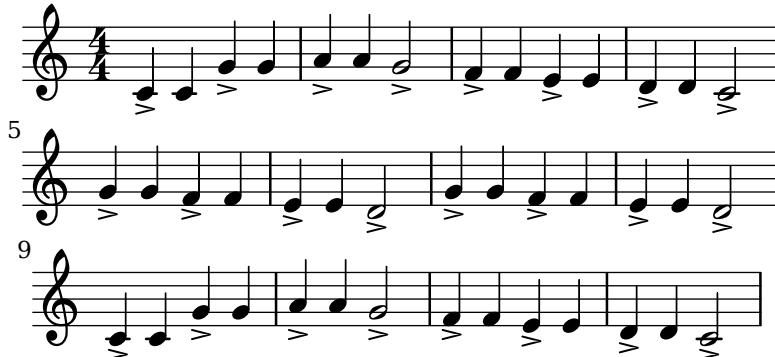
При игре музыкального произведения на каком-либо инструменте акцент идёт обычно на первую ноту из такта – на её *сильную долю*. Доли, которые не акцентированы, называются *слабыми долями*.

С точки зрения исполнения акцентированные ноты должны звучать громче, или каким-либо другим способом выделяться в общем звучании.

Акцент нот обозначается значком “>” над (или под) нотой. Если мы расставим значки, обозначающие акцент, то получим следующую запись (см. рис. 5.12.17.)

Музыкальный размер “четыре четверти” называется *сложным*, так как он получен слиянием двух более *простых* размеров, а именно “две четверти”.

Рис. 5.12.17: “Twinkle, Twinkle, Little Star”



Таким образом, в размере “четыре четверти” кроме сильной доли, появляется вторая доля, называемая *относительно сильной*.

Как можно видеть на рис. 5.12.17, первый (основной) акцент ставится на первую ноту в такте – в нашем случае, первую четверть. Второй, второстепенный, акцент ставится на третью ноту в такте, или же можно сказать, что на первую ноту второй половины такта (относительно сильную долю.) Основной акцент по определению более выраженный, чем второстепенный.

Если мы возьмём другой музыкальный размер – например, две четверти ( $\frac{2}{4}$ ), то произведение будет звучать по-другому, поскольку основной и единственный акцент будет на первую ноту каждого такта, и относительно слабая доля будет отсутствовать.

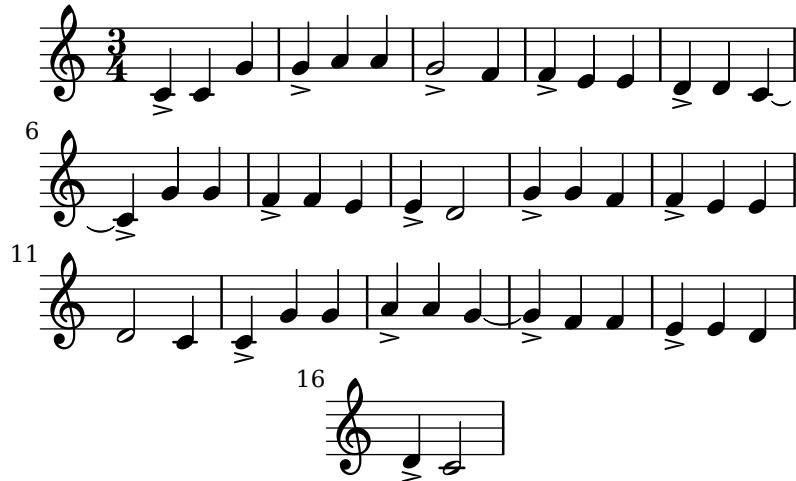
Рис. 5.12.18: “Twinkle, Twinkle, Little Star” в размере две четверти.

Музыкальный размер “две четверти” используется в таких стилях музыки, как например полька.

Если мы возьмём музыкальный размер “три четверти” (см. рис. 5.12.19), в такт будет убираться ровно три четвертных ноты. Таким образом, акцент будет идти на первую четверть из трёх в каждом такте. При этом, некоторые ноты длиной  $\frac{1}{2}$

как бы “разрезаются” тактовой чертой на две четверти.

Рис. 5.12.19: “Twinkle, Twinkle, Little Star” в размере три четверти



Такая запись в отношении “Twinkle, Twinkle, Little Star” выглядит противовесственно, и после таких экспериментов к вам в дверь может постучаться музыкальная инквизиция.

Тем не менее, если мы сыграем в таком размере композицию музыкальном инструменте, то она будет звучать вальсирующе, ведь музыкальный размер “две четверти” обычно используется для вальса.

Каким же образом мы можем выразить эти музыкальные нюансы в нашем программном коде и в реализации аппаратной части, чтобы они украсили наше музыкальное произведение? Изменение кода включает в себя несколько этапов.

Во-первых, самым простым для нас способом выделить какие-то определённые ноты является подключение дополнительного динамика с меньшей громкостью к Arduino. Ноты, которые должны звучать тише, будут отправляться на него. А те ноты, которые должны быть акценированными, будут отправляться на громкий динамик. Допустим, громкий динамик будет у нас на цифровом порту 2, а тихий динамик – на цифровом порту 3.

```
const int LOUD_SPEAKER_PIN = 2; // Громкий динамик.
const int QUIET_SPEAKER_PIN = 3; // Тихий динамик.

// ...

void setup() {
    pinMode(LOUD_SPEAKER_PIN, OUTPUT);
    pinMode(QUIET_SPEAKER_PIN, OUTPUT);
}
```

Во-вторых, двумерный массив нот должен теперь иметь не два столбца, а три, так как в третьем столбце мы как раз будем хранить громкость ноты. Исходя из параметра громкости, который на данный момент может иметь всего два уровня – 0 (тихо) и 1 (громко), мы будем выбирать динамик для воспроизведения ноты.

Для размера “четыре четверти” мы будем первую ноту из такта делать громче остальных.

```
// ...

float twinkle_twinkle_little_star[][][3] = {
    /* 0 */ {c4, 4, 1}, {c4, 4, 0}, {g4, 4, 0}, {g4, 4, 0},
    /* 1 */ {a4, 4, 1}, {a4, 4, 0}, {g4, 4, 0}, {g4, 4, 0},
    /* 2 */ {f4, 4, 1}, {f4, 4, 0}, {e4, 4, 0}, {e4, 4, 0},
    /* 3 */ {d4, 4, 1}, {d4, 4, 0}, {c4, 4, 0},
    /* 4 */ {g4, 4, 1}, {g4, 4, 0}, {f4, 4, 0}, {f4, 4, 0},
    /* 5 */ {e4, 4, 1}, {e4, 4, 0}, {d4, 4, 0}, {d4, 4, 0},
    /* 6 */ {g4, 4, 1}, {g4, 4, 0}, {f4, 4, 0}, {f4, 4, 0},
    /* 7 */ {e4, 4, 1}, {e4, 4, 0}, {d4, 4, 0},
};

// ...
```

Далее при воспроизведении музыки нам надо выбирать нужный динамик, в соответствии с громкостью (акцентом) ноты.

```
// ...

void loop() {
    const long BPM = 120;
    const long MINUTE = 60 * 1000000;
    const long T = (MINUTE / BPM) * 4;

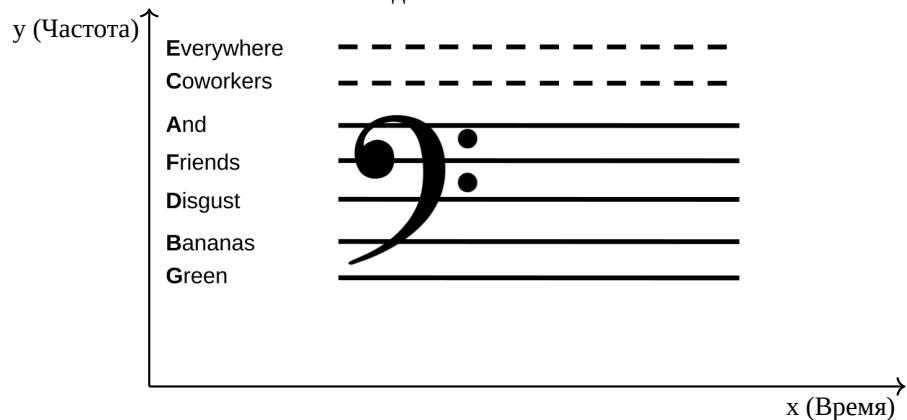
    for (int note_idx = 0; note_idx < 28; note_idx++) {
        if (melody[note_idx][2] == 1) {
            // Нота с акцентом
            play_tone(LOUD_SPEAKER_PIN,
                      melody[note_idx][0],
                      T / melody[note_idx][1]);
        } else {
            // Нота без акцента
            play_tone(QUIET_SPEAKER_PIN,
                      melody[note_idx][0],
                      T / melody[note_idx][1]);
        }
        delay(100);
    }
}
```

### 5.13 Басовый ключ

Кроме скрипичного ключа в музыке часто используется *басовый ключ*. Обычно басовый ключ используется для тех музыкальных инструментов, у которых частотный диапазон находится ниже, чем охватывает скрипичный ключ.

Мнемоника для басового ключа представлена на рис. 5.13.20.

Рис. 5.13.20: Мнемоника для запоминания нот басового ключа.



## 5.14 Создание музыкальной группы

Теперь, когда мы понимаем, как устроен ритм, и умеем читать ноты, то можно приступить к формированию музыкальной “группы” из двух или более Arduino.

Многие музыкальные композиции, которые мы могли видеть, включают в себя две или более музыкальные дорожки. Часто это партия для второй руки при игре на пианино, либо же несколько разных музыкальных инструментов (например, гитара и вокал.)

В этой ситуации самый простой для нас способ сыграть композицию в целом – это взять несколько микроконтроллеров Arduino и научить их играть совместно.

### 5.14.1 Выбор названия

В первую очередь, для создания музыкальной группы нам надо придумать ей хорошее название. Пусть в нашем случае названием будет “Second Hand Of Darkness” (сокращённо “SHOD”) – в переводе с английского “Вторая рука тьмы”. Такое название отлично подходит для рок-группы, да и просто звучит круто.

Далее это название надо правильно стилизовать, чтобы нас совсем приняли за “своего” в рядах профессиональных музыкантов.<sup>11</sup>



Рис. 5.14.21: Логотип нашей музыкальной группы.

Подготовительные этапы выполнены. Теперь надо выбрать музыкальную композицию и сыграть её.

### 5.14.2 Выбор дебютной композиции

Многие музыкальные группы начинают свой путь в “большую музыку” с исполнения каверов на известные композиции. Пусть и нашей первой композицией будет Arduino-кавер на композицию “Sweet Dreams” под авторством Marilyn Manson (см. рис. 5.14.22).<sup>12</sup>

Данная композиция имеет две основных музыкальных “дорожки” — первая является вокалом, вторая представляет собой гитарную партию.

### 5.14.3 Реализация мелодии

Для того, чтобы реализовать первую “дорожку” нашей мелодии, нам следует создать музыкальные проект для Arduino, как мы делали до этого. Воспроизведение звуков по-прежнему будем делать через нашу функцию `play_tone`.

Обе части композиции будут заданы в виде двумерных массивов. Первый массив будет играть басовую часть. Обратите внимание, что основная тема композиции повторяется.

<sup>11</sup> Для названия используется шрифт “Guttural”, распространяемый под лицензией Creative Commons Attribution: <https://www.fontspace.com/guttural-font-f8257>.

<sup>12</sup> За основу взята партитура <https://musescore.com/user/13484026/scores/5329623>, выполненная пользователем “Instrumental Rock”, лицензия CC-BY-NC (<https://creativecommons.org/licenses/by-nc/4.0/>).

```
float sweet_dreams_guitar[28][2] = {
    /* 01 */
    {c4, 8}, {c4, 8}, {g4, 8}, {c4, 8},
    {a4b, 8}, {c4, 8}, {g4, 8}, {c4, 8},
    /* 02 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 03 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 04 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 05 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 06 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 07 */
    {c4, 8}, {c4, 8}, {g4, 8}, {c4, 8},
    {a4b, 8}, {c4, 8}, {g4, 8}, {c4, 8},
    /* 08 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 09 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 10 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 11 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
    /* 12 */
    {a3b, 8}, {a3b, 8}, {e4b, 8}, {f4, 8},
    {g3, 8}, {g3, 8}, {d4, 8}, {e4b, 8}
};
```

Вторая часть мелодии (вокал) будет реализована аналогичным образом.

```
float sweet_dreams_vocals[28][2] = {
    /* 01 */
    {R, 1},
    /* 02 */
    {R, 1},
    /* 03 */
    {R, 4}, {e5b, 4}, {e5b, 4}, {c5, 8}, {e5b, 4},
    /* 04 */
    {e5b, 8.0 / 3.0}, {d5, 2},
```

```

/* 05 */
{e5b, 4}, {c5, 8}, {e5b, 8.0 / 3.0}, {c5, 4},
/* 06 */
{e5b, 8}, {f5, 8.0 / 3.0}, {d5, 8.0 / 3.0}, {c5, 8},
/* 07 */
{e5b, 8}, {e5b, 8}, {c5, 8}, {e5b, 8.0 / 3.0},
/* 08 */
{e5b, 8}, {e5b, 8.0 / 3.0}, {d5, 2},
/* 09 */
{e5b, 4}, {c5, 8}, {e5b, 4}, {c5, 8.0 / 3.0},
/* 10 */
{e5b, 8}, {e5b, 8}, {f5, 8}, {e5b, 8}, {d5, 8}, {d5, 8.0 / 3.0},
/* 11 */
{e5b, 8}, {e5b, 8}, {c5, 4}, {e5b, 4}, {c5, 8}, {e5b, 4},
/* 12 */
{c5, 8.0 / 3.0}, {R, 2}
};

```

При работе с двумя Arduino возникает проблема – как загрузить в каждую из них свою мелодию. Самый очевидный, однако не самый удобный вариант – это сделать отдельный проект для каждой из них, и загружать по-очереди. Неудобство тут в том, что придётся каждый раз следить, чтобы в каждую из Arduino был загружен свой проект.

Мы можем предложить более удобный способ – использовать один проект для обеих Arduino, при этом по значению на одном из цифровых портов определять, какая из мелодий должна воспроизводиться. Ставя перемычку (провод), замыкающую порт SWITCH\_PORT на землю, мы говорим Arduino воспроизводить гитарную партию. Если же перемычки нет, то тогда воспроизводится локальная партия. Таким образом, мы при любом раскладе получим воспроизведение разных дорожек мелодии на разных Arduino.

```

// Порт, куда подключен динамик.
const int SPEAKER = 2;
// Порт-переключатель мелодии.
const int SWITCH_PORT = 10;

void setup() {
    pinMode(SPEAKER, OUTPUT);
    pinMode(SWITCH_PORT, INPUT_PULLUP);
}

void loop() {
    const int BPM = 93;
    const long MINUTE = 600000000;
    const long T = (MINUTE / BPM) * 4;

    // Исходя из состояния порта SWITCH_PORT решаем,
    // какую из мелодий играть.
    if (digitalRead(SWITCH_PORT) == LOW) {
        int count = sizeof(sweet_dreams_guitar)

```

```

        / sizeof(sweet_dreams_guitar[0]));
for (int i = 0; i < count; i++) {
    play_tone(SPEAKER,
              sweet_dreams_guitar[i][0],
              T / sweet_dreams_guitar[i][1]);
    delay(10);
}
} else {
    int count = sizeof(sweet_dreams_vocals)
               / sizeof(sweet_dreams_vocals[0]));
    for (int i = 0; i < count; i++) {
        play_tone(SPEAKER,
                  sweet_dreams_vocals[i][0],
                  T / sweet_dreams_vocals[i][1]);
        delay(10);
    }
}
}

```

#### 5.14.4 Синхронизация мелодий

Для запуска мелодий одновременно можно сразу на двух Arduino нажать кнопки сброса (“Reset”) и одновременно их отпустить.

Несмотря на то, что мы получили одновременное воспроизведение обеих частей мелодии, со временем вы можете заметить, что они рассинхронизируются – ноты одной части начинают играть невпопад с другой частью.

Этому неприятному эффекту есть несколько причин. Начнём их разбирать по мере значимости их влияния на мелодию.

Первая причина в рассинхронизации кроется вот в этих строчках:

```

for (int i = 0; i < count; i++) {
    play_tone(SPEAKER,
              sweet_dreams_vocals[i][0],
              T / sweet_dreams_vocals[i][1]);

    delay(10); // ----- Проблема здесь.
}

```

Представим, что у нас воспроизводятся первые два такта мелодии (см. 5.14.23.)

Здесь видно, что в вокальной партии у нас две целых паузы – по сути, две глухие ноты, тишина. Во второй же у нас суммарно за эти два такта воспроизводится  $8 + 8 = 16$  нот. После каждого воспроизведения ноты (т.е. после каждого выполнения `play_tone`) у нас явно прописана задержка в некоторое количество миллисекунд. Таким образом, в первом такте у нас суммарная задержка будет:

$$2 * 10\text{мс} = 20\text{мс} \quad (5.14)$$

Во гитарной же партии общая задержка будет составлять:

$$16 * 10\text{мс} = 160\text{мс} \quad (5.15)$$

Получается разница в 140 мс между мелодиями, и данная погрешность будет накапливаться со временем – чем длиннее мелодия и больше различаются длительности нот в двух её частях, тем более заметна будет рассинхронизация.

Чтобы решить данную проблему, мы должны избавиться от вызова фиксированного `delay` и перейти к добавлению задержки между нотами, вычисляемой на основе длины отдельно взятой ноты. Удобнее всего это сделать внутри функции `play_tone`.

```
// Функция воспроизведения звука указанной частоты.
void play_tone(int port, float f, long t) {
    if (f > 0) {
        const int T = 1000000 / f;
        int d = T / 2;
        int count = t / T;
        for (int i = 0; i < count; i++) {
            digitalWrite(port, HIGH);
            delayMicroseconds(d);
            digitalWrite(port, LOW);
            delayMicroseconds(d);
        }
    } else {
        delay(t / 1000); // Пауза
    }
    // Задержка после ноты составляет примерно 10%
    // от её длины:
    delayMicroseconds(t * 0.10);
}
```

После этих изменений в `play_tone` надо поменять и код воспроизведения мелодии, убрав `delay`:

```
// ...

void loop() {
    const int BPM = 93;
    const long MINUTE = 60000000;
    const long T = (MINUTE / BPM) * 4;

    if (digitalRead(SWITCH_PORT) == LOW) {
        int count = sizeof(sweet_dreams_guitar)
                    / sizeof(sweet_dreams_guitar[0]);
        for (int i = 0; i < count; i++) {
            play_tone(SPEAKER,
                      sweet_dreams_guitar[i][0],
                      T / sweet_dreams_guitar[i][1]);
        }
    } else {
        int count = sizeof(sweet_dreams_vocals)
                    / sizeof(sweet_dreams_vocals[0]);
        for (int i = 0; i < count; i++) {
            play_tone(SPEAKER,
```

```
    sweet_dreams_vocals[i][0],  
    T / sweet_dreams_vocals[i][1]);  
}  
}  
}
```

Второй источник накопления ошибки – это погрешность, возникающая при вычислении длины периода на основе частоты в `play_tone`. Возьмём к примеру ноту С4 – её частота составляет 261.63 Гц. Код, рассчитывающий период колебания из частоты у нас в `play_tone` выглядит так:

```
const int T = 1000000 / f;
```

Если мы подставим в эту формулу частоту ноты C4 и посчитаем период колебания, то получим:

$$1000000 \text{мкс} / 261.63 \text{Гц} = 3822.191644689 \text{мкс} \quad (5.16)$$

Как видно из формулы, длина одного колебания для С4 не является целым числом, при этом дробная часть будет отброшена (поскольку мы сохраним период в тип данных `int`) и мы останемся со значением 3822 мкс.

Чтобы узнать количество повторений цикла, создающего колебания мембранны динамика, мы делим длительность звука  $t$  на период  $T$ :

```
int count = t / T;
```

Пусть длина нашей ноты будет 1с (1000000 мкс.) Подставим в данную формулу значение 3822 мкс и посмотрим, что получится:

$$1000000 \text{мкс} / 3822 \Gamma_{\text{Ц}} = 261.643118786 \quad (5.17)$$

Опять же, дробная часть отбрасывается, и мы получаем значение 261 – именно это количество раз повториться наш цикл. Каждое повторение цикла имеет длительность примерно в 3822 мкс, если не учитывать накладные расходы на вызов функций.

Таким образом, повторив цикл колебания мембранны динамика 261 раз, мы получим следующее общее время работы цикла:

$$261_{\text{MKC}} * 3822_{\text{MKC}} = 997542_{\text{MKC}}$$

Ошибка составляет:

$$1000000 \text{мкС} - 997542 \text{мкС} \equiv 2458 \text{мкС}$$

Или примерно 2.5 мс. Это довольно небольшое значение ошибки, и роль его может быть незначительна. К тому же, сам вызов функций `digitalWrite` и `delayMicroseconds` тоже занимает некоторое время. Если же мы всё-таки хотим скомпенсировать данную ошибку, то после цикла в `play_tone` нам необходимо добавить дополнительную задержку, рассчитанную по формуле:

$$t = (T * \text{count}) \quad (5.20)$$

Код с компенсацией ошибки будет выглядеть так:

```
// Функция воспроизведения звука указанной частоты.
void play_tone(int port, float f, long t) {
    if (f > 0) {
        const int T = 1000000 / f;
        int d = T / 2;
        int count = t / T;
        for (int i = 0; i < count; i++) {
            digitalWrite(port, HIGH);
            delayMicroseconds(d);
            digitalWrite(port, LOW);
            delayMicroseconds(d);
        }
        // Компенсация ошибки:
        delayMicroseconds(t - (T * count));
    } else {
        delay(t / 1000); // Пауза
    }
    delayMicroseconds(t * 0.10);
}
```

= 93

Вокал

Гитара

3

6

9

12

Рис. 5.14.22: Marilyn Manson, Trent Reznor, “Sweet Dreams”.

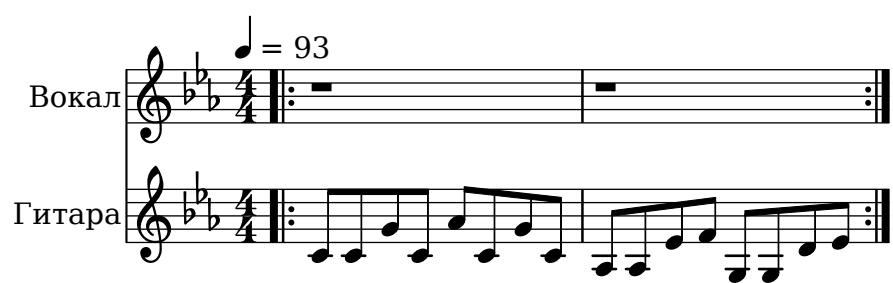


Рис. 5.14.23: Marilyn Manson, Trent Reznor, “Sweet Dreams” (отрывок.)



## Глава 6

# Разработка игр

Компьютерные игры – очень увлекательное занятие, завлекающее на многие часы. Но что, если попробовать создать свою собственную игру? На самом деле, создание небольших игр это не так уж и сложно, даже при наличии базовых навыков программирования. Разрабатывать игры может быть не менее увлекательным занятием, чем играть в них!

Цель этой главы – помочь в создании нескольких игр, в первую очередь ради закрепления изученного материала. Преимущество такого подхода в том, что читателю не составит труда самостоятельно придумывать множество игровых задач, реализуя которые, он может закрепить известные ему конструкции языка, а также, сталкиваясь с трудностями, может возникнуть необходимость в изучении нового материала, что позволяет расти навыкам программирования читателя.

### 6.1 ЖК-дисплей

Для вывода информации мы будем использовать *жидкокристаллический дисплей* (ЖК-дисплей), подключаемый к Arduino. Принцип работы данных дисплеев аналогичен обычным ЖК-дисплеям, которые выводят информацию на вашем компьютере. Более конкретно мы будем использовать *текстовый ЖК-дисплей*, который предназначен для вывода текстовых символов.

На текстовых ЖК-дисплеях, экран поделён на клетки, внутри каждой из которых можно отрисовать один символ (букву, знак препинания, цифру, или просто какую-либо картинку.) Между клетками обычно находится расстояние в один пиксель. Подобные дисплеи плохо подходят для отрисовки произвольных изображений, тем не менее, некоторый простор для творчества у нас имеется.

Несмотря на свойства дисплея, которые на первый взгляд кажутся слишком ограничивающими для наших задач, используя наше мастерство и творческий подход, мы можем добиться достаточно интересных результатов в плане разработки игр.

Рабочая область дисплея 16x2 (16 столбцов, 2 строки) схематически выглядит, как показано на рис. 6.1.1.



Рис. 6.1.1: Схематическое изображение дисплея.

### 6.1.1 Подключение дисплея

Для упрощения нашей работы мы будем использовать ЖК-дисплей с интерфейсом передачи данных, который называется “I2C” (читается “ай-ти-си”).) Не вдаваясь в подробности можно сказать, что данный вариант подключения требует всего 4 провода: питание, земля и две линии передачи данных.

Сам модуль I2C часто уже припаян к дисплею, но может идти и отдельно. В случае отдельного модуля, подключение выглядит, как на рис. 6.1.2.

### 6.1.2 Вывод текста

Согласно традиции, первая программа, которую мы выведем – это “Привет, Мир!”. Однако поскольку вывод русского текста на большинстве дисплеев организован сложнее, нежели чем латиницы, то мы будем выводить “Hello, World!”.

Для этого нам необходимо, во-первых, установить библиотеку для работы с дисплеем; во-вторых, настроить дисплей на нужный режим работы, и только после этого мы сможем вывести текст.

Начнём со скачивания библиотеки. Проще всего это сделать через менеджер библиотек, доступный из меню “Инструменты” (“Tools”). Существуют несколько библиотек, которые могут обеспечить работу ЖК-дисплея по протоколу I2C. Мы можем взять библиотеку “Liquid Crystal I2C” версии 1.1.1 под авторством Марко Шварца (Marco Schwartz).

Установив библиотеку, мы сможем использовать её функционал.

Первым делом в коде нам необходимо подключить библиотеку “LiquidCrystal\_I2C.h” – это делается через специальную команду `#include`.

```
#include <LiquidCrystal_I2C.h>
```

В глобальной области кода (до функции `setup` и `loop`) необходимо создать специальную переменную, через которую мы будем работать с дисплеем. Эта переменная отличается от того, что мы видели ранее – в данной переменной хранится не число, а ссылка более сложный *объект*<sup>1</sup>, который располагается где-то в памяти при работе программы внутри микроконтроллера.

Назовём переменную `lcd`, по сокращению LCD – “Liquid Crystal Display”:

```
LiquidCrystal_I2C lcd(0x27, 16, 2);
```

Далее в функции `setup` необходимо выполнить настройку дисплея.

```
lcd.init();
lcd.backlight();
```

<sup>1</sup> Термин “объект” относится к методологии программирования, которая называется *объектно-ориентированное программирование* (сокращённо “ООП”.) Мы пока не будем заострять наше внимание на теме ООП, но любознательный читатель может получить общее представление о данной методологии, например, из соответствующей статьи в Wikipedia.

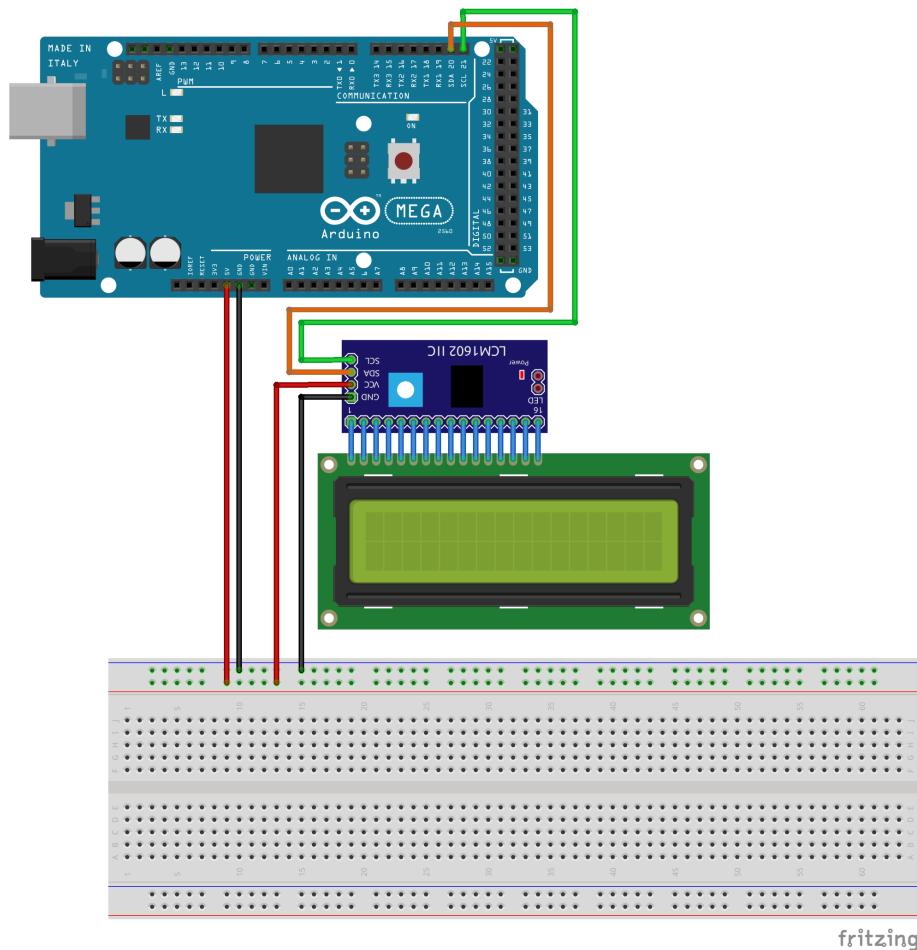


Рис. 6.1.2: Схема подключения ЖК-дисплея 16x2 по интерфейсу I2C к Arduino Mega 2560.

После этого уже в функции `loop` мы можем вывести текст. Вывод текста, как правило, делается в два этапа: во-первых, необходимо установить место, в которое будет “впечатан” текст. Для этого существует специальная функция `setCursor`.

```
lcd.setCursor(0, 0);
```

Первый параметр `setCursor` задаёт позицию курсора по оси X, второй параметр задаёт позицию по оси Y.

После этого наконец-то мы можем вывести текст.

```
lcd.print("Hello, world!");
```

Полностью код будет выглядеть примерно так, как показано ниже.

```
#include <LiquidCrystal_I2C.h>
```

```
LiquidCrystal_I2C lcd(0x27, 16, 2);

void setup() {
    lcd.init();
    lcd.backlight();
}

void loop() {
    lcd.setCursor(0, 0);
    lcd.print("Hello, World!");
}
```

При необходимости очистить экран можно “затереть” текст, впечатав поверх пробелы, или же воспользоваться функцией `clear`, которая позволяет очистить экран полностью.

### **Задачи**

1. Выровняйте тест по середине дисплея.
2. Сделайте так, чтобы текст мигал.
3. Разработайте “Бегущую строку”, где текст будет перемещаться по экрану справа налево.

## **6.2 Выбор жанра и разработка сюжета**

Разработка игры, как и любого другого серьёзного проекта, как правило, начинается с проектирования – выбора жанра, разработки сюжета.

В плане жанра у нас очень широкий выбор опций: здесь и ролевые игры (Role-Playing Game, “RPG”), и игры-“стрелялки” (First-Person Shooter, “FPS”), платформеры, аркады и т.п.

Предпочтение одних жанров игр другим – очень субъективная вещь, так что сразу оговоримся, что наши решения в рамках данной книги ни в коем случае не должны убеждать вас, что они являются единственными верными. Вы можете использовать навыки, полученные в рамках данной практики, для разработки игр других жанров.

В качестве жанра мы выберем игру-платформер, поскольку это перекликается со многими старыми играми.

Одна из игр, которую мы будем брать, как источник вдохновения – “Dangerous Dave in the Haunted Mansion” 1991-го года, разработанную компанией “id Software”.

Сюжет нашей игры будет разворачиваться в некотором имении, которое кишит различными агрессивными монстрами. Наша задача будет пройти через ряд уровней, чтобы добраться до заветной двери, которая должна вывести нас из здания.

## **6.3 Отображение игрового персонажа**

В большинстве жанров компьютерных игр существует некоторый персонаж, который представляет нас в игре и которым мы можем управлять.

Для начала мы можем в качестве изображения персонажа взять какой-нибудь символ – допустим, символ собачки “@”.

Зададим “изображение” в виде символьной константы где-то в начале программы:

```
const char PLAYER = '@';
```

Далее нам необходимо задать позицию персонажа в двумерном пространстве нашей игровой “карты”:

```
int player_x = 0;  
int player_y = 0;
```

После этого мы можем отобразить персонажа на экране. Полный код программы будет выглядеть примерно так, как показано ниже.

```
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);

const char PLAYER = '@';

int player_x = 0;
int player_y = 0;

void setup() {
    lcd.init();
    lcd.backlight();
}

void loop() {
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);
}
```

После загрузки данной программы в Arduino, на экране на нулевой строке в первом столбце должен появиться символ “@”.

Поскольку мы задали позицию персонажа в виде переменных, то меняя эти переменные мы можем менять позицию персонажа на экране. Для изменения позиции в играх обычно используются кнопки и/или манипулятор вида “джойстик”. В следующем разделе мы как раз посмотрим, как подключить кнопки и сделать обработку их нажатий.

## 6.4 Реализация управления

Для управления игровым персонажем нам потребуется некоторое устройство ввода информации в наш компьютер (микроконтроллер). Мы сделаем своё “устройство ввода”, состоящее из нескольких кнопок, расположенных на макетной плате.

### 6.4.1 Подключение кнопки

Попробуем подключить кнопку к Arduino. Кнопка по своей сути не более чем замыкатель двух контактов. Самым простым аналогом кнопки являются два разъединённых провода (разомкнутая электрическая цепь), которые можно соединить между собой, или опять разъединить. Само собой, такой способ управления устройством неудобен (и даже небезопасен, при больших напряжениях и токах в электрической цепи), поэтому кнопки обычно представлены некими закрытыми устройствами, которые имеют некий способ замыкать цепь без необходимости брать в руки концы проводов.

Кнопки по принципу работы бывают разные. Самый простой их вид – *тактовые кнопки*.

“Тактовыми” называются кнопки, которые не “запоминают” своё состояние, и сразу же после нажатия возвращаются в исходное состояние (как правило, разомкнутое.)

Другой вид кнопок, с которыми мы сталкиваемся в быту – это те, которые “запоминают” своё состояние; такие кнопки обычно называются “переключателями”.

На рис. 6.4.3 можно видеть один из возможных вариантов подключения кнопки. Можно подумать, что такой вариант достаточен – когда кнопка нажата, то цепь замыкается и ток идёт от 5V до цифрового порта номер 2. И действительно, в этом случае с порта можно считывать значение 1 (HIGH).

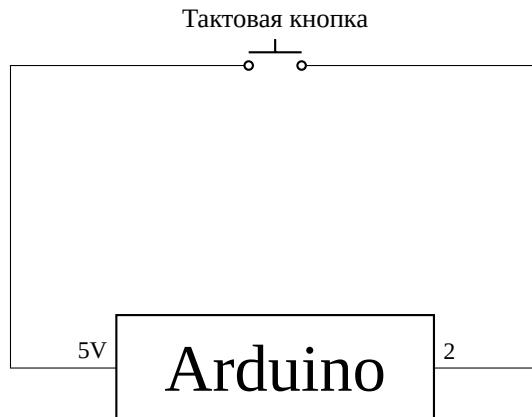


Рис. 6.4.3: Возможная схема подключения кнопки к Arduino.

Пример простейшей программы,читывающей значение с кнопки, представлен ниже.

```
void setup() {
    pinMode(2, INPUT);
}

void loop() {
    // Считываем значение с кнопки и сохраняем в новую переменную "value".
    int value = digitalRead(2);
    // Далее эту переменную можно использовать в коде, чтобы определить
```

```
// состояние кнопки: если в переменной находится значение 1,
// то кнопка нажата, если 0 -- то не нажата.
}
```

Чтобы наглядно получить представление о том, как значение на кнопке будет меняться при нажатии, мы можем добавить вывод данных на компьютер.

```
void setup() {
    pinMode(2, INPUT);
    Serial.begin(9600);
}

void loop() {
    // Считываем значение с кнопки и сохраняем в новую переменную "value".
    int value = digitalRead(2);
    // Далее эту переменную можно использовать в коде, чтобы определить
    // состояние кнопки: если в переменной находится значение 1,
    // то кнопка нажата, если 0 -- то не нажата.

    // Выводим значение с кнопки впорт.
    Serial.println(value);
}
```

К сожалению, наблюдение за значением на кнопке скорее всего нам покажет, что даже когда кнопка не нажата, на ней иногда “проскакивает” значение 1. Как такое может быть? Объяснение простое – как мы видели в главе “Белый шум”, вокруг нас присутствует электромагнитный фон, который улавливается схемами и приводит к тому, что иногда значение на цифровых портах переходит рубеж, после которого Arduino считает это значение единицей.

Решить эту проблему можно как минимум двумя способами. Первый способ заключается в том, чтобы поставить специальный подтягивающий резистор, который будет подтягивать значение на порту, куда подключена кнопка, к земле (GND.)

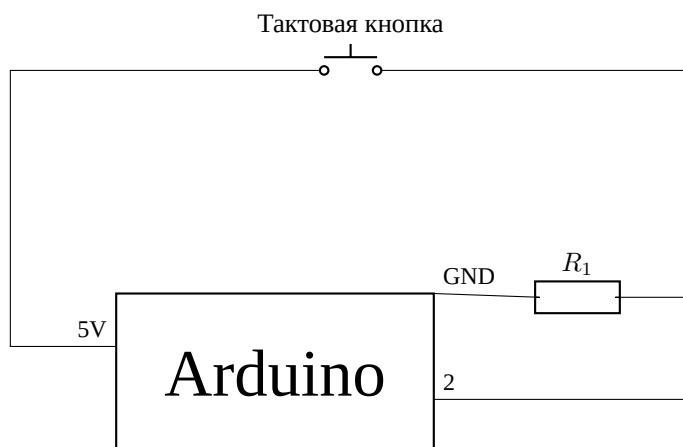


Рис. 6.4.4: Схема подключения кнопки к Arduino с подтягивающим резистором.

Второй способ подключения кнопки – использование встроенного в Arduino подтягивающего резистора, который подключен на 5V. В этом случае схема подключения значительно упрощается и становится похожа на то, что было показано на рис. 6.4.3, но с одним отличием: цифровой порт должен при нажатии на кнопку замыкаться не на 5V, а на GND (см. рис. 6.4.5.)



Рис. 6.4.5: Схема подключения кнопки к Arduino при использовании встроенного подтягивающего резистора к 5V.

Настройка порта в этом случае должна производиться на режим INPUT\_PULLUP. В остальном же код нашего примера остаётся без изменений.

```
void setup() {
    pinMode(2, INPUT_PULLUP);
    Serial.begin(9600);
}

void loop() {
    // Считываем значение с кнопки и сохраняем в новую переменную "value".
    int value = digitalRead(2);
    // Далее эту переменную можно использовать в коде, чтобы определить
    // состояние кнопки: если в переменной находится значение 1,
    // то кнопка нажата, если 0 -- то не нажата.

    // Выводим значение с кнопки впорт.
    Serial.println(value);
}
```

При таком использовании работа с кнопкой “инвертируется” – когда она нажата, то значение на порту будет “0” (LOW), а когда не нажата, то “1” (HIGH.)

#### 6.4.2 Обработка нажатий

Для начала зададим номер порта, куда подключена наша (пока единственная) кнопка, в виде именованной константы BUTTON\_R (“R” от слова “Right”, “Вправо”.)

```
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);

// Кнопки управления.
const char BUTTON_R = 2; // Кнопка ``Вправо``

const char PLAYER = '@';

int player_x = 0;
int player_y = 0;
```

Далее в `setup` необходимо настроить режим работы порта для кнопки через `pinMode`, как мы делали это в предыдущих примерах.

```
void setup() {
    lcd.init();
    lcd.backlight();

    // Настройка кнопок управления.
    pinMode(BUTTON_R, INPUT_PULLUP);
}
```

После этого следует определиться, как мы будем обрабатывать нажатия кнопок. Сейчас самым простым для нас способом обработки нажатий является поочерёдный опрос кнопок, про другие способы мы с вами поговорим позднее.

Для обработки нажатий мы будем в `loop` проверять значение на порту, куда подключена кнопка, используя уже знакомый нам `digitalRead`, и если значение будет `LOW`, то менять позицию персонажа.

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        player_x++;
    }
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);
}
```

Таким образом, при нажатии кнопки “ВПРАВО” мы будем получать смещение персонажа вправо.

В общем виде, код нашей игры на текущий момент выглядит так:

```
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27, 16, 2);

// Кнопки управления.
const char BUTTON_R = 2; // Кнопка ``Вправо``

const char PLAYER = '@';
```

```

int player_x = 0;
int player_y = 0;

void setup() {
    lcd.init();
    lcd.backlight();

    // Настройка кнопок управления.
    pinMode(BUTTON_R, INPUT_PULLUP);
}

void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        player_x++;
    }
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);
}

```

Здесь, однако же, кроется проблема – при единичном (на наш взгляд) нажатии кнопки, игрок может пробежать до конца карты, или даже скрыться за её пределами. Это возникает, так как короткое (единичное) нажатие на кнопку с точки зрения человека занимает огромное время с точки зрения компьютера. За то время, пока мы держим палец на кнопке, перед тем, как её отпустить, функция `loop` успевает сработать несколько раз; следовательно, несколько раз будет считано и обработано состояние кнопки.

Для борьбы с этой проблемой необходимо добавить короткую задержку (например, в 100мс) в конце `loop`:

```

void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        player_x++;
    }
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);

    // Задержка, чтобы избежать слишком быстрого считывания
    // нажатий на кнопки.
    delay(100);
}

```

Точно таким же образом мы можем добавить кнопку движения влево, слегка дополнив код нашего примера. Поскольку исходный код нашей игры будет дальше расти, для краткости будем сокращать уже написанные части, акцентируя внимание на том, что поменялось.

```

// ... Код подключения библиотеки и создания
//      переменной "lcd". ...

// Кнопки управления.
const char BUTTON_R = 2;           // Кнопка "Вправо"

```

```

const char BUTTON_L = 3;           // Кнопка "Влево"

const char PLAYER = '@';

// ... Код задания переменных позиции игрока. ...

void setup() {
    // ... Код настройки дисплея. ...

    // Настройка кнопок управления.
    pinMode(BUTTON_R, INPUT_PULLUP);
    pinMode(BUTTON_L, INPUT_PULLUP);
}

void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        player_x++;
    }
    if (digitalRead(BUTTON_L) == LOW) {
        player_x--;
    }
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);

    // Задержка, чтобы избежать слишком быстрого считывания
    // нажатий на кнопки.
    delay(100);
}

```

Запустите эту программу, попробуйте походить влево-вправо. В скором времени вы заметите, что есть серьёзная проблема – игрок может выйти за пределы дисплея (“уйти за карту”.)

### 6.4.3 Ограничение движения игрока

Чтобы исправить вышеописанную проблему, необходимо добавить проверки позиции игрока прежде, чем позволить ему сдвинуться влево или вправо. Эти проверки нужны при обработке нажатий кнопок.

Для движения вправо нам нужно смотреть, не выйдет ли игрок при следующем “шаге” (перемещении на клетку) за карту. Поскольку мы знаем, что размер нашего дисплея – 16 столбцов, то мы можем ограничить движение вправо 15-й клеткой.

```

// ... Код, который мы писали до этого ...

void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // Делаем ограничение движения игрока вправо.
        // Если игрок уже на 15-й клетке, то дальше двигаться
        // ему нельзя.
    }
}

```

```

    if (player_x < 15) {
        player_x++;
    }
}
if (digitalRead(BUTTON_L) == LOW) {
    player_x--;
}
lcd.setCursor(player_x, player_y);
lcd.print(PLAYER);

// Задержка, чтобы избежать слишком быстрого считывания
// нажатий на кнопки.
delay(100);
}

```

С движением влево нужна подобная же проверка, с разницей в том, что крайняя левая точка у нас – это нулевая клетка.

```

// ... Код, который мы писали до этого ...

void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // Делаем ограничение движения игрока вправо.
        // Если игрок уже на 15-й клетке, то дальше двигаться
        // ему нельзя.
        if (player_x < 15) {
            player_x++;
        }
    }
    if (digitalRead(BUTTON_L) == LOW) {
        // Делаем ограничение движения игрока влево.
        // Если игрок уже на нулевой клетке, то левее
        // двигаться ему уже некуда.
        if (player_x > 0) {
            player_x--;
        }
    }
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);

    // Задержка, чтобы избежать слишком быстрого считывания
    // нажатий на кнопки.
    delay(100);
}

```

Таким образом мы обработали оба случая, когда игрок мог выйти за нашу игровую карту.

## 6.5 Реализация игровой карты

В играх вокруг игрока обычно разворачивается некоторое действие – не-игровые персонажи (Non-Playable Characters, сокращённо “NPC”) ревностно патрулируют территорию карты; появляются и исчезают различные объекты, с которыми можно взаимодействовать; наконец, повсюду располагаются различные *статические* объекты, которые могут преграждать путь игроку, или же являться просто элементами декора.

Простым способом расположить отдельный объект на карте является задание ему координат на карте и отрисовка – так же, как мы делаем это с игровым персонажем. Однако при создании двух переменных (для координаты по оси X и по оси Y) на каждый объект, количество переменных будет расти очень быстро. Только представьте, что карта размером 20x4 может потенциально хранить 80 объектов, что даёт суммарно 160 переменных для адресации каждого из них! Для решения этой проблемы мы должны прибегнуть к уже известным нам массивам, с одной оговоркой – мы должны будем использовать *двумерные массивы*.

Схематическое изображение двумерного массива можно увидеть на рис. 6.5.6.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0																				
1																				
2																				
3																				

Рис. 6.5.6: Схематическое изображение массива 4x20 (4 строки, 20 столбцов.)

Можно также представить двумерный массив 4x20, как одномерный массив из 4-х элементов, каждая ячейка которого содержит ссылку на ещё один одномерный массив длиной 20 ячеек, как показано на рис. 6.5.7.

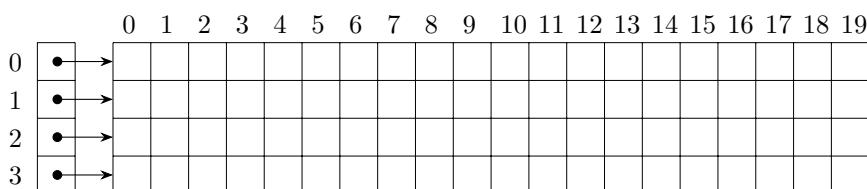


Рис. 6.5.7: Схематическое представление массива 4x20 в виде одномерного массива из 4-х элементов со ссылками на одномерные массивы по 20 элементов.

Для наших задач реализации игры нам потребуется задать массив, хранящий тип данных `char`. Размер нашего массива – нашей игровой карты – мы зададим с помощью именованных констант, где-нибудь в глобальной области кода, выше `setup`.

```
const int MAP_W = 20; // Ширина карты.
const int MAP_H = 4; // Высота карты.
```

Далее создадим сам двумерный массив, который станет нашей игровой картой.

```
char game_map[MAP_H][MAP_W];
```

Обратите внимание, что при объявлении массива используется две пары квадратных скобок. В первой паре скобок задаётся высота массива (количество строк), во второй – ширина массива (количество столбцов.)

Далее нам необходимо объявить функцию отображения карты на экране – назовём её `map_show`.

```
void map_show() {
    for (int y = 0; y < MAP_H; y++) {
        for (int x = 0; x < MAP_W; x++) {
            lcd.setCursor(x, y);
            lcd.print( game_map[y][x] );
        }
    }
}
```

Как мы видим, тело функции состоит из двух вложенных друг в друга циклов `for`. Первый из циклов, идущий по `y`, перебирает строки массива. Второй цикл по `x` перебирает столбцы. Поскольку цикл по `x` вложен в цикл по `y`, то на каждое изменение `y` происходит полный проход цикла по `x`.

Во вложенном цикле у дисплея вызывается функция `setCursor` для указания позиции вывода символа на экран. Далее символ печатается в указанную клетку через `print`.

Поскольку массив `game_map` имеет тип `char`, то символы, хранящиеся в нём, автоматически выводятся на дисплей в правильном виде – а именно, как графические символы, а не их коды в таблице символов.

Функцию `map_show` необходимо вызывать в `loop` для того, чтобы карта отобразилась на экране.

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // ...
    }
    if (digitalRead(BUTTON_L) == LOW) {
        // ...
    }

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // Отрисовка игрока поверх карты.
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);

    // Задержка, чтобы избежать слишком быстрого считывания
    // нажатий на кнопки.
    delay(100);
}
```

Обратите внимание, что отрисовка карты выполняется перед отрисовкой игрока. Если поменять этот порядок, то тогда игрок не будет виден на экране, ведь его изображение будет перетёerto отрисовкой карты.

После загрузки и запуска данной программы вы можете обнаружить, что дисплей наполнился случайными символами. Это произошло потому, что мы не инициализировали массив `game_map` значениями, и там сейчас во всех ячейках `mouse` – то есть, какие-то неожиданные значения.

Чтобы исправить эту ситуацию, нам необходимо создать функцию генерации карты, которая будет заполнять карту чем-то вменяемым. Для начала заполним карту пробелами, которые будут у нас символизировать пустое место на карте.

В глобальной области перед `setup` зададим специальную константу для обозначения пустого места.

```
const char SPACE = ' ';      // Пустое пространство.
```

Теперь мы можем использовать константу `SPACE` для задания пустого места. Перейдём к описанию функции генерации карты.

```
void map_generate() {
    for (int y = 0; y < MAP_H; y++) {
        for (int x = 0; x < MAP_W; x++) {
            game_map[y][x] = SPACE;
        }
    }
}
```

Как можно видеть, функция `map_generate` не сильно отличается от функции `map_show`.

Функцию `map_generate` необходимо вызывать один раз внутри функции `setup`.

```
void setup() {
    lcd.init();
    lcd.backlight();

    // Настройка кнопок управления.
    pinMode(BUTTON_R, INPUT_PULLUP);
    pinMode(BUTTON_L, INPUT_PULLUP);

    // Вызов функции генерации карты.
    map_generate();
}
```

После загрузки новой прошивки в Arduino, мы должны увидеть экран, где отображается только игрок, поскольку карта у нас “пустая” (заполнена пробелами.)

Теперь мы получили возможность размещать на карте объекты. Начнём с добавления нового игрового объекта – пусть это будет “стена”, т.е. непроходимый участок карты. Зададим стену в виде символа “#”.

```
const char SPACE = ' ';      // Пустое пространство.
const char WALL = '#';      // Стена.
```

После этого мы можем разместить этот объект на карте, модифицировав функцию `map_generate`. Добавим несколько стен.

```
void map_generate() {
    for (int y = 0; y < MAP_H; y++) {
        for (int x = 0; x < MAP_W; x++) {
            game_map[y][x] = SPACE;
        }
    }

    // Добавляем вручную на карту объекты.
    game_map[0][10] = WALL;
    game_map[1][10] = WALL;
    game_map[2][10] = WALL;
}
```

После загрузки программы в Arduino, мы увидим, что объекты появились на экране. Таким образом мы можем разместить и другие объекты. Однако возникает другая проблема – как игроку взаимодействовать с ними? В текущей реализации игрок свободно проходит сквозь стены, как привидение. Решение этой проблемы как раз описано в следующем разделе.

### Задачи

1. Объявите дополнительные игровые объекты в виде символьных констант, как мы это делали со `SPACE` и `WALL` – например, можно добавить “дверь” на следующий уровень, или же особый вид стен.
2. Добавьте дополнительные объекты на карту, для создания желаемого игрового мира.

#### 6.5.1 Взаимодействие игрока с объектами

Возможно, вам доводилось работать в графических редакторах вроде GNU Image Manipulation Program (GIMP) – подобные “продвинутые” редакторы позволяют создавать многослойные изображения. На отдельно взятом слое могут быть прозрачные области, сквозь которые виден нижележащий слой. Таким образом можно создавать сложные композиции. Тем не менее, при экспорте изображения в растровый графический формат (например, PNG) все эти слои объединяются в один слой, снизу вверх – то есть, каждый следующий слой накладывается на предыдущий с учётом прозрачности.

Мы можем представить себе игровой мир, как подобный набор “слоёв”. На данный момент у нас два слоя – самый нижний (назовём его нулевым) хранит объекты нашего мира (это как раз массив `game_map`). Слой выше (назовём его первым) хранит игрока.

Для того, чтобы игрок на первом слое мог взаимодействовать с игровыми объектами на нулевом, нам необходимо сделать сопоставление позиций игрока с содержащимся нулевого слоя.

Для примера попробуем сделать так, чтобы игрок не мог проходить сквозь стены. Если игрок пытается идти вправо на клетку, и там – стена, тогда мы не

меняем позицию игрока. Если же справа от игрока находится пустая клетка, то меняем позицию. Пример того, как это можно реализовать в коде, показан ниже.

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        if (player_x < 15) {
            // Проверяем, что находится справа от игрока.
            if (game_map[player_y][player_x + 1] != WALL) {
                // Если справа не стена, то тогда позволяем
                // игроку сместиться на одну клетку вправа.
                player_x++;
            }
        }
    }

    // ...
}
```

Здесь следует обратить особое внимание на данный код.

```
if (game_map[player_y][player_x + 1] != WALL) {
    // ...
}
```

Как мы помним, `game_map` хранит игровую карту. Для того, чтобы узнать, что находится на клетке карты справа от игрока, мы подставляем в качестве номера строки массива значение `player_y`, а в качестве номера столбца подставляем `player_x + 1` (прибавление единицы как раз позволяет от нас взять следующую клетку.) Затем полученное из массива значение сравнивается с константой `WALL` – мы проверяем, что значение не равно (“!=”) данной константе.

Поскольку мы будем подобные сравнения использовать достаточно часто, имеет смысл сделать специальные функции для проверки клеток карты на наличие определённых объектов. Например, мы можем создать вспомогательную функцию подобного рода:

```
// Функция, возвращающая 1 (true) в случае, если
// на клетке карты по координатам x, y находится стена.
// В противном случае функция возвращает 0 (false).
bool is_wall(int x, int y) {
    return game_map[y][x] == WALL;
}
```



**Примечание:** В языке C++ значение 0 означает “ложь” (“false”), тогда как любое ненулевое значение означает истину (например, числа 1, 42, -41 и т.п. являются значением “истина” с точки зрения языка.)

Используя новую функцию `is_wall` код внутри `loop` можно будет свести к следующему виду:

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
```

```

if (player_x < 15) {
    // Проверяем, что находится справа от игрока.
    if (!is_wall(player_x + 1, player_y)) {
        // Если справа не стена, то тогда позволяем
        // игроку сместиться на одну клетку вправа.
        player_x++;
    }
}

// ...
}

```

Обратите внимание на эту часть кода:

```

if (!is_wall(player_x + 1, player_y)) {
    // ...
}

```

Используется знак “!”, который означает логическое отрицание (“НЕ”), что инвертирует ответ, возвращённый функцией `is_wall`.

Точно также, как мы делали с движением вправо, надо обработать столкновение со стеной при движении влево. В итоге, код `loop` будет выглядеть следующим образом:

```

void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        if (player_x < 15) {
            if (!is_wall(player_x + 1, player_y)) {
                player_x++;
            }
        }
    }

    if (digitalRead(BUTTON_L) == LOW) {
        if (player_x > 0) {
            if (!is_wall(player_x - 1, player_y)) {
                player_x--;
            }
        }
    }

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // Отрисовка игрока поверх карты.
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);

    // Задержка, чтобы избежать слишком быстрого считывания
    // нажатий на кнопки.
}

```

```
    delay(100);
}
```

### 6.5.2 Сбор предметов с карты

Реализация сбора предметов с игровой карты реализуется аналогично взаимодействию со “стенами”. Начнём с определения символа, который будет обозначать собираемый объект – пусть это будут некие “аптечки” (“Health Points”, “HP”.)

```
const char SPACE = ' ';
const char WALL = '#';
const char HP = '*';
```

Далее создадим функцию для проверки наличия аптечки на указанной ячейке карты – по аналогии с тем, как мы делали со стенами.

```
// Функция, возвращающая 1 (true) в случае, если
// на клетке карты по координатам x, y находится аптечка.
// В противном случае функция возвращает 0 (false).
bool is_hp(int x, int y) {
    return game_map[y][x] == HP;
}
```

Используя данную функцию, мы можем сделать дополнительную проверку в `loop` на столкновение с аптечкой. Когда аптечка взята, то мы можем убрать объект с карты, просто вписав поверх него пустое пространство (“SPACE”).

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // ...
    }

    if (digitalRead(BUTTON_L) == LOW) {
        // ...
    }

    if (is_hp(player_x, player_y)) {
        game_map[player_y][player_x] = SPACE;
    }

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // ...
}
```

### 6.5.3 Улучшение отрисовки карты

Как мы сейчас можем видеть, изображение персонажа на экране дисплея мигает. Это связано с тем, что отрисовка персонажа у нас происходит отдельно от отрисовки карты – сначала рисуется карта, потом уже персонаж.

```
void loop() {
    // ...

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // Отрисовка игрока поверх карты.
    lcd.setCursor(player_x, player_y);
    lcd.print(PLAYER);

    // ...
}
```

Можно сказать, что наш игровой мир сейчас двухслойный. Представьте какой-нибудь графический редактор, с поддержкой слоёв. Мы можем рисовать на каждом отдельно взятом слое, а также включать и выключать отображение любого из слоёв. Вышестоящие слои при этом перекрывают нижестоящие; если на вышестоящим слое есть области прозрачности, то сквозь эти области можно увидеть нижележащие слои. Но когда мы экспортствуем изображение в растровое представление (например, в формат PNG), то все слои объединяются в одно двухмерное изображение.

Чтобы решить проблему с “миганием” персонажа, нам необходимо интегрировать отрисовку персонажа в метод отрисовки карты на экране, чтобы объединять все слои в один. Смотрите, на той клетке, где стоит персонаж, содержимое карты всё равно не видно – таким образом, нам нет смысла его отображать. Когда при отрисовке карты в функции `map_show` мы рисуем клетку, на которой стоит персонаж, мы просто не рисуем карту под ним.

Добавим функции `map_show` два параметра, в которые мы будем передавать позицию персонажа при вызове функции.

```
void map_show(int px, int py) {
    for (int y = 0; y < MAP_H; y++) {
        for (int x = 0; x < MAP_W; x++) {
            lcd.setCursor(x, y);
            if ((y == py) && (x == px)) {
                lcd.print(PLAYER);
            } else {
                lcd.print(game_map[y][x]);
            }
        }
    }
}
```

Внутри функции мы добавили проверку – если у нас на текущей позиции карты стоит персонаж, то мы рисуем его, а не содержимое карты.

Вызов функции будет выглядеть так.

```
void loop() {
    // ...

    // Вызов функции отрисовки карты на экране дисплея.
```

```

map_show(player_x, player_y);

// ВАЖНО: Отрисовка игрока поверх карты больше не нужна.
// lcd.setCursor(player_x, player_y);
// lcd.print(PLAYER);

// ...
}

```

#### 6.5.4 Вспомогательные функции для работы с картой

Для повышения удобства работы с картой мы сделаем ряд вспомогательных функций, которые позволяют нам устанавливать “объект” (символ) на указанную клетку карты или получать объект с карты по его координатам, очищать клетку карты и “двигать” объекты на карте.

Начнём с функции установления объекта на клетку карты:

```

// Функция размещения объекта на карте.
void map_set(int x, int y, char object) {
    game_map[y][x] = object;
}

```

Функция получения объекта с карты реализуется так:

```

// Функция получения объекта с карты.
char map_get(int x, int y) {
    return game_map[y][x];
}

```

Функция очистки клетки карты базируется на основе функции размещения объекта:

```

// Функция очистки клетки карты.
void map_clear(int x, int y) {
    map_set(x, y, SPACE);
}

```

Функция перемещения объекта реализует следующий простой алгоритм:

1. Скопировать объект из исходной клетки карты ( $x_1, y_1$ ) в целевую ( $x_2, y_2$ ).
2. Очистить исходную ( $x_1, y_1$ ) клетку карты.

Реализация подобной функции может быть сделана так:

```

// Функция перемещения объекта на карте.
void map_move(int x1, int y1, int x2, int y2) {
    map_set(x2, y2, map_get(x1, y1));
    map_clear(x1, y1);
}

```

## 6.6 Добавление звуков

Редкая компьютерная игра обходится без специальных звуковых эффектов. Добавим тоже в нашу игру немного звука.

Пусть динамик для вывода звука у нас будет подключен к восьмому порту – укажем это в коде с помощью именованной константы до функции `setup`:

```
const byte SPEAKER_PIN = 8; // Порт динамика
```

В `setup` теперь настроим порт динамика на режим вывода.

```
void setup() {
    lcd.init();
    lcd.backlight();

    // Настройка кнопок управления.
    pinMode(BUTTON_R, INPUT_PULLUP);
    pinMode(BUTTON_L, INPUT_PULLUP);

    // Настройка порта динамика.
    pinMode(SPEAKER_PIN, OUTPUT);

    // Вызов функции генерации карты.
    map_generate();
}
```

Теперь перенесём функцию `play_tone` из главы “Звук” в исходный код нашей игры, и добавим константы для нот.

```
// Четвёртая (в научной нотации, считая с нуля) октава.
const float c4 = 261.630;
const float d4 = 293.660;
const float e4 = 329.630;
const float f4 = 349.230;
const float g4 = 392.000;
const float a4 = 440.000;
const float b4 = 493.880;

// Функция воспроизведения звука указанной частоты.
void play_tone(int port, float f, long t) {
    const long T = 1000000 / f;
    long d = T / 2;
    int count = t / T;
    for (int i = 0; i < count; i++) {
        digitalWrite(port, HIGH);
        delayMicroseconds(d);
        digitalWrite(port, LOW);
        delayMicroseconds(d);
    }
}
```

Начнём со звука сбора предметов. В предыдущих главах мы как раз реализовали обработку столкновения игрока с предметами на карте. Теперь мы можем добавить звук при взятии “аптечки” с карты.

Вы можете придумать свой звук “по вкусу”; мы же возьмём ноты e4 и g4 с разными длительностями, что даст (на наш взгляд) интересный эффект.

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // ...
    }

    if (digitalRead(BUTTON_L) == LOW) {
        // ...
    }

    if (is_hp(player_x, player_y)) {
        game_map[player_y][player_x] = SPACE;

        // Воспроизведение звука взятия предмета.
        play_tone(SPEAKER_PIN, e4, 125000);
        play_tone(SPEAKER_PIN, g4, 250000);
    }

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // ...
}
```

Точно так же можно сделать озвучку для других событий игры (например, для выигрыша и проигрыша) – но об этом речь пойдёт позже.

### 6.6.1 Задачи

- Попробуйте реализовать звуковой эффект столкновения персонажа со стеной – для этого можно взять какой-нибудь звук низкой частоты (не обязательно музыкальную ноту) и при попытке игрока “войти в стену” воспроизводить его.

## 6.7 Графика

### 6.7.1 Использование готовых символов

Компьютеры рассматривают символы также, как и всё остальное – а именно в виде чисел. Каждому символу соответствует некий код, который компьютер запоминает и обрабатывает, если мы просим сохранить символ в память или же вывести его на экран.



**Примечание:** Одной из простейших форм кодирования символов является таблица ASCII (American Standard Code for Information Interchange), которая описывает символы с кодами от нуля до 127 включительно. В современных операционных системах для обеспечения поддержки различных языков вместо ASCII используется кодировка Unicode, в которой размер таблицы намного больше и включает в себя символы большинства народов мира. Тем не менее, Unicode сохраняет обратную совместимость с ASCII и первые 128 символов в Unicode повторяют символы из таблицы ASCII.

В текстовых ЖК-дисплеях есть встроенная таблица символов, согласно которой дисплей отображает символы. Набор этих символов различается в зависимости от модели дисплея, поэтому рекомендуется поискать таблицу символов именно для вашего дисплея.

Например, во многих дисплеях символ с кодом 255 (0xFF в шестнадцатиричной системе) обозначает полностью закрашенный блок. Для того, чтобы использовать данный символ в проекте, удобно завести для него константу, обозначающую данный игровой “объект”:

```
const char WALL = 0xFF; // Код символа
```

Обратите внимание, что код символа указывается без кавычек, так как мы явно указываем номер символа в таблице, а не его графическое представление.

После того, как мы создали константу для символа, мы можем выводить его в коде программы следующим образом:

```
void loop() {
    lcd.setCursor(0, 0);
    lcd.print(WALL);
}
```

### 6.7.2 Создание собственных символов

Несмотря на то, что дисплей в нашем проекте используется текстовый, и он не даёт нам возможности рисовать произвольные рисунки в любом месте экрана, нам доступна возможность создавать собственные символы. Эти символы затем можно отображать в клетках дисплея.

Размер клетки на нашем дисплее составляет 8x5 пикселей (8 строк на 5 столбцов.) Схематически это можно визуализировать, как показано на рис. 6.7.8.

На дисплее 4 строки на 20 столбцов всего 80 таких клеток, показанных на рис. 6.7.8.

Каждый пиксель описывается одним битом. Единица означает, что пиксель закрашен, ноль означает, что пиксель не закрашен. Например, строчка, показанная на рис. 6.7.9, может быть описана, как 0b01010.

	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					
7					

Рис. 6.7.8: Схематическое изображение клетки, в которой отрисовывается символ.

**Примечание:** Обратите внимание, что числа, начинающиеся с префикса `0b` (символ нуля и английской буквы “b”), записаны напрямую в двоичном виде. Не стоит забывать, что компьютер *все* данные, которые мы сохраняем в памяти, хранит в двоичном виде; задавая префикс перед числом, мы лишь меняем формат записи числа, а не формат хранения. Из других префиксов можно назвать `0x` (ноль-икс), позволяющий указывать шестнадцатиричные числа (например `0xFF`) и префикс `0` (ноль), позволяющий указывать восьмеричные числа.



0	1	0	1	0

Рис. 6.7.9: Кодирование пикселей в одной строке символа.

Одна строка символа кодируется одним байтом информации (восемью битами) из которых фактически используется только пять младших бит. Всего таких строк 8 штук, и для их хранения нам нужен массив на восемь байт.

Пример отрисовки человечка:

```
byte player_image[8] = {
    0b01110,
    0b01110,
    0b00100,
    0b01110,
    0b10101,
    0b01110,
    0b01010,
    0b01010
};
```

Визуальное представление данного символа:

Можно рисовать подобные изображения на тетрадном листе в клетку, и потом вручную переводить это в программный код. Само собой, программисты уже ав-

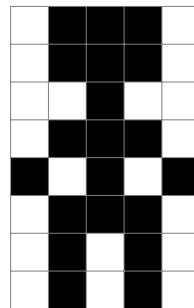


Рис. 6.7.10: Изображение игрока.

томатизировали этот процесс, и есть готовые online-ресурсы для рисования символов и автоматической генерации программного кода. Вот один из них: <https://omerk.github.io/lcdchargen/>

После того, как символ нарисован, его надо прописать в память дисплея, чтобы мы могли его нарисовать. Из встроенной таблицы символов нам обычно доступны коды символов от 0 до 7 включительно, что даёт возможность записать 8 собственных символов в память. Остальные символы во встроенной таблице переписать нельзя.

Для того, чтобы прописать символ в память, нам нужно использовать специальную функцию `createChar`, в которую мы должны передать код символа и желаемое его представление на дисплее, заданное через массив. Для задания кода символа удобно использовать константы типа `char`, поэтому мы создадим константу `PLAYER` для нашего игрока:

```
const char PLAYER = 0; // Код символа
// Визуальное представление символа.
byte player_image[8] = {
    0b01110,
    0b01110,
    0b00100,
    0b01110,
    0b10101,
    0b01110,
    0b01010,
    0b01010
};
```

Затем в `setup` пропишем символ в память дисплея:

```
void setup() {
    lcd.init();
    lcd.backlight();

    // Запись символа в память дисплея:
    lcd.createChar(PLAYER, player_image);
}
```

После этого мы можем вывести символ на ЖК-дисплей просто вызвав функцию `print`, передав в неё нашу константу `PLAYER`:

```
void loop() {
    lcd.setCursor(0, 0);
    lcd.print(PLAYER);
}
```

### 6.7.3 Задачи

- Добавьте символы для не-игрового персонажа (NPC) и для некоторого собираемого предмета.

## 6.8 Анимация

В данном разделе мы сделаем так, чтобы игрок поворачивался в ту сторону, в которую он идёт. Для этого нам необходимо сделать три варианта “модели” персонажа: первая, где персонаж стоит на месте; вторая, где персонаж смотрит вправо и третья, где персонаж смотрит влево.

Чтобы съэкономить драгоценные ячейки под собственные в памяти дисплея, мы будем использовать один и тот же код символа, меняя при этом отрисовку этого символа.

### 6.8.1 Добавление необходимых символов

У персонажа нашей игры будет три состояния: перемещение влево (“LEFT”), бездействие (“IDLE”), перемещение вправо (“RIGHT”). Для каждого из этих состояний необходимо сделать “текстуру”, которая будет применяться (см. рис. 6.8.11.)

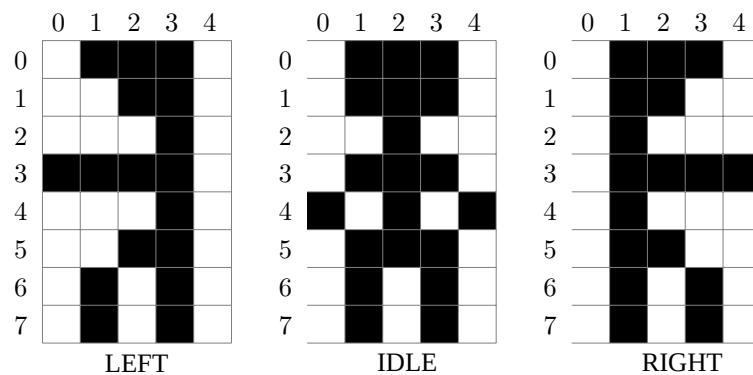


Рис. 6.8.11: Три варианта изображения игрока: смотрящий влево (“LEFT”), бездействующий (“IDLE”) и смотрящий вправо (“RIGHT.”).

### 6.8.2 Реализация анимации

Для того, чтобы реализовать анимацию поворота игрока в сторону направления движения, нам необходимо добавить вызов `createChar` непосредственно в код функции `loop`.

Первым делом после вызова `map_show` мы добавим перезапись символа игрока для того, чтобы вернуть его в бездействующее положение.

```
void loop() {
    map_show();
    lcd.createChar(PLAYER, player_idle_image);
    // ...
}
```

Затем мы должны перезаписывать символ игрока каждый раз, когда мы видим, что нажата кнопка вправа или влево.

```
void loop() {
    map_show();
    lcd.createChar(PLAYER, player_idle_image);

    if (digitalRead(BUTTON_R) == LOW) {
        lcd.createChar(PLAYER, player_right_image);
        player_x++;
    }
    if (digitalRead(BUTTON_L) == LOW) {
        lcd.createChar(PLAYER, player_left_image);
        player_x--;
    }
}
```

## 6.9 Игровая логика

В большинстве игр обычно есть определённые условия выигрыша и проигрыша, по которым программа определяет исход игрового процесса. В данном разделе мы научимся реализовывать игровую логику, делать подсчёт очков, добавлять дополнительные условия для выигрыша и проигрыша – в общем, изучим всё то, что сделает нашу игру интересной.

### 6.9.1 Условие выигрыша

В нашем игровом процессе выигрыш будет засчитываться тогда, когда игрок войдёт в некоторую дверь. Дверь, как и всё остальное в игре, будет представлена специальным символом – это может быть встроенный символ, или же нарисованный нами. Пусть для начала дверь будет обозначаться символом “D”:

```
const char SPACE = ' ';      // Пустое пространство.
const char WALL = '#';      // Стена.
const char HP = '*';        // Аптечка.
const char DOOR = 'D';      // Дверь.
```

Создадим функцию `is_door` для проверки наличия двери в клетке:

```
// Функция, возвращающая 1 (true) в случае, если
// на клетке карты по координатам x,y находится дверь.
// В противном случае функция возвращает 0 (false).
bool is_door(int x, int y) {
    return game_map[y][x] == DOOR;
}
```

Используя данную функцию, мы можем реализовать проверку, вошёл ли игрок в дверь. Когда игрок входит в дверь, то вызывается функция `win`:

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // ...
    }

    if (digitalRead(BUTTON_L) == LOW) {
        // ...
    }

    if (is_hp(player_x, player_y)) {
        // ...
    }

    if (is_door(player_x, player_y)) {
        win();
    }

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // ...
}
```

Функция `win` может быть реализована следующим образом:

```
void win() {
    lcd.setCursor(0, 0);
    lcd.print("Y O U W I N!");
    player_x = 0;
    player_y = 0;
    generate_map();
    delay(1000);
}
```

Как можно видеть, при выигрыше мы отображаем надпись “Y O U W I N”, затем происходит “сброс” игры в исходное состояние путём зануления координат персонажа и повторной генерации карты.

## 6.9.2 Условие проигрыша

Проигрыш в игре можно сделать, например, при столкновении игрока с монстром. Обозначим монстра в виде константы:

```
const char SPACE = ' ';      // Пустое пространство.
const char WALL = '#';       // Стена.
const char HP = '*';         // Аптечка.
const char DOOR = 'D';        // Дверь.
const char MONSTER = 'M';     // Монстр.
```

Далее добавим функцию проверки наличия монстра в клетке карты:

```
// Функция, возвращающая 1 (true) в случае, если
// на клетке карты по координатам x,y находится дверь.
// В противном случае функция возвращает 0 (false).
bool is_monster(int x, int y) {
    return game_map[y][x] == MONSTER;
}
```

В функцию `loop` добавим проверку столкновения игрока с монстром:

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // ...
    }

    if (digitalRead(BUTTON_L) == LOW) {
        // ...
    }

    if (is_hp(player_x, player_y)) {
        // ...
    }

    if (is_door(player_x, player_y)) {
        win();
    }

    if (is_monster(player_x, player_y)) {
        game_over();
    }

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // ...
}
```

Простейшая реализация функции `game_over` не сильно отличается от функции выигрыша `win`:

```
void game_over() {
    lcd.setCursor(0, 0);
    lcd.print("G A M E   O V E R");
    player_x = 0;
    player_y = 0;
    generate_map();
    delay(1000);
}
```

Поскольку и в функции `game_over`, и в функции `win` используется “сброс” игры в исходное состояние, то мы можем вынести этот набор операций в отдельную функцию `reset_game`:

```
// Функция "сброса" игры в исходное состояние.
void reset_game() {
    player_x = 0;
    player_y = 0;
    generate_map();
}

void win() {
    lcd.setCursor(0, 0);
    lcd.print("Y O U   W I N!");
    reset_game();
    delay(1000);
}

void game_over() {
    lcd.setCursor(0, 0);
    lcd.print("G A M E   O V E R");
    reset_game();
    delay(1000);
}
```

### 6.9.3 Дополнительные квесты

Обычно в играх прохождение уровня усложнено тем, что нужно решить дополнительные квесты – например, для выигрыша нужно взять ключ, который открывает дверь. Мы можем это достаточно легко реализовать, добавив дополнительный объект “КЛЮЧ” (KEY):

```
const char SPACE = ' ';
// Пустое пространство.

const char WALL = '#';
// Стена.

const char HP = '*';
// Аптечка.

const char DOOR = 'D';
// Дверь.

// Добавили эту строку:
const char KEY = 'K';
// Ключ.
```

В дверь можно будет зайти только при наличии ключа – без него она “закрыта”. Эта простая логика реализуется проверкой взятия ключа и изменением двери с закрытой на открытую. Таким образом нам требуется теперь два вида дверей – закрытый вариант (CLOSED\_DOOR) и открытый вариант (OPEN\_DOOR):

```
const char SPACE = ' ';
// Пустое пространство.

const char WALL = '#';
// Стена.

const char HP = '*';
// Аптечка.

// Изменили эти строки:
const char CLOSED_DOOR = 'd';
// Закрытая дверь.

const char OPEN_DOOR = 'D';
// Открытая дверь.

const char KEY = 'K';
// Ключ.
```

Важно заметить, что символы для каждого вида двери должны отличаться, чтобы компьютер также смог их различить в проверках, которые потребуются далее.

Пусть позиция двери у нас будет задана переменными `door_x` и `door_y`. Мы можем доработать функцию `map_generate`, добавив закрытую дверь (`CLOSED_DOOR`) на карту по указанным координатам:

```
// Позиция двери на карте:
byte door_x = 19;
byte door_y = 3;

void map_generate() {
    for (int y = 0; y < MAP_H; y++) {
        for (int x = 0; x < MAP_W; x++) {
            game_map[y][x] = SPACE;
        }
    }

    // Добавляем вручную на карту объекты.
    game_map[0][10] = WALL;
    game_map[1][10] = WALL;
    game_map[2][10] = WALL;
    game_map[door_y][door_x] = CLOSED_DOOR;
}
```

Добавим также функцию проверки наличия ключа в указанной клетке карты:

```
// Функция, возвращающая 1 (true) в случае,
// если на клетке карты по координатам x,y находится ключ.
// В противном случае функция возвращает 0 (false).
bool is_key(int x, int y) {
    return game_map[y][x] == KEY;
}
```

Когда игрок “наступает” на ключ, то эта закрытая дверь будет заменяться на открытую. Для работы с картой мы можем воспользоваться функциями `map_set` и `map_clear`, описанными в разделе 6.5.4:

```
void loop() {
    // ... Здесь идёт обработка нажатий клавиш...

    if (is_key(player_x, player_y)) {
        // "Открытие" двери ключом.
        map_set(door_x, door_y, OPEN_DOOR);

        // Очистка клетки карты.
        map_clear(player_x, player_y);
    }

    // ...
}
```

```
// Вызов функции отрисовки карты на экране дисплея.
map_show();

// ...
}
```

Соответственно нам теперь требуется доработка проверки выигрыша, т.к. вйти можно только в открытую дверь. Заменим функцию `is_door` на `is_open_door` и обновим проверку:

```
// Функция, возвращающая 1 (true) в случае, если
// на клетке карты по координатам x,y находится дверь.
// В противном случае функция возвращает 0 (false).
bool is_open_door(int x, int y) {
    return game_map[y][x] == OPEN_DOOR;
}
```

Обновим также код проверки выигрыша в функции `loop`:

```
void loop() {
    // ... Здесь идёт обработка нажатий клавиш...

    if (is_key(player_x, player_y)) {
        // "Открытие" двери ключом.
        map_set(door_x, door_y, OPEN_DOOR);

        // Очистка клетки карты.
        map_clear(player_x, player_y);
    }

    if (is_open_door(player_x, player_y)) {
        win();
    }

    // ...

    // Вызов функции отрисовки карты на экране дисплея.
    map_show();

    // ...
}
```

#### 6.9.4 Подсчёт очков

Для подсчёта очков (или собранных предметов) достаточно завести где-нибудь вверху программы глобальную переменную, которая будет меняться при возникновении события. Пусть переменная называется “`score`”, что в переводе с английского как раз и значит “счёт”:

```
int score = 0;
```

Изначально счёт равен нулю.

Теперь, когда игрок берёт какой-либо предмет с карты, можно увеличивать значение счётчика. Пусть в нашем случае будет подсчитываться количество собранных очков здоровья (Health Points, “HP”):

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // ...
    }

    if (digitalRead(BUTTON_L) == LOW) {
        // ...
    }

    if (is_hp(player_x, player_y)) {
        score++; // Увеличение счётчика.
        // ...
    }

    // ...

    map_show();
}

// ...
```

Имея в распоряжении такой счётчик, достаточно легко построить на этом дальнейшую логику игры. Например, можно сделать так, чтобы при 10 собранных предметах происходил выигрыш:

```
void loop() {
    if (digitalRead(BUTTON_R) == LOW) {
        // ...
    }

    if (digitalRead(BUTTON_L) == LOW) {
        // ...
    }

    if (is_hp(player_x, player_y)) {
        score++; // Увеличение счётчика.
        // ...
    }

    // Если собрано 10 предметов, то...
    if (score == 10) {
        win(); // ... засчитывается выигрыш.
    }

    // ...
```

```
map_show();
// ...
}
```

### 6.9.5 Отображение жизней персонажа с помощью светодиодов

Одним из интересных вариантов отображения жизней персонажа (или набранных очков) является использование подключенных к микроконтроллеру светодиодов – это экономит место на дисплее (не нужно занимать его часть отображением информации.)

Для того, чтобы реализовать данную возможность, необходимо подключить несколько светодиодов к микроконтроллеру, как мы это делали в главе 2.

Пусть в нашем случае будет три светодиода для отображения жизней персонажа. Светодиоды желательно подключить в последовательно расположенные свободные цифровые порты – например, с 22-го по 24-й. Начальный номер порта разумно задать глобальной константой, также как и максимальное количество жизней, чтобы было можно потом на них ссылаться:

```
// Начальный номер порта последовательности светодиодов
// для отображения жизней:
const byte HP_LED_START = 22;
// Максимальное количество жизней:
const byte HP_MAX        = 3;
```

В `setup` необходимо настроить эти порты на режим `OUTPUT`:

```
void setup() {
// ...
for (byte idx = 0; idx < HP_MAX; idx++) {
    pinMode(idx + HP_LED_START, OUTPUT);
}
}
```

Обратите внимание, как мы используем константы `HP_LED_START`, прибавляя его к счётчику цикла для того, чтобы получить номер порта светодиода – этот же трюк мы будем использовать далее.

Количество жизней пусть будет задано глобальной переменной `hp`, можно задать её сразу после константы `HP_LED_START`:

```
// Начальный номер порта последовательности светодиодов
// для отображения жизней:
const byte HP_LED_START = 22;
// Максимальное количество жизней:
const byte HP_MAX        = 3;

byte hp = 3; // Количество жизней персонажа.
```

Нам необходима функция, которая будет обновлять светодиоды, чтобы жизни персонажа отображались на них в реальном времени:

```
void show_hp() {
    for (byte idx = 0; idx < HP_MAX; idx++) {
        if (idx < hp) {
            // Выключаем светодиоды, порядковый номер которых меньше,
            // чем текущее количество жизней.
            digitalWrite(HP_LED__START + idx, HIGH);
        } else {
            // Выключаем остальные светодиоды.
            digitalWrite(HP_LED_START + idx, LOW);
        }
    }
}
```

Затем эту функцию необходимо вызвать где-нибудь в `loop`, чтобы количество жизней персонажа отображалось в реальном времени:

```
void loop() {
    show_hp();
    // ...
}
```

# Предметный указатель

- Музыка  
Басовый ключ, 86  
Бемоль, 80  
Диез, 80  
Диссонанс, 65  
Консонанс, 65  
Музыкальный размер, 82  
Нотный стан (нотоносец), 74  
Октачная система, 65  
Пауза, 76  
Ритм, 57  
    Такт, 57  
Удары в минуту (Beats Per Minute, BPM), 60  
Скрипичный ключ, 74  
Частота звука, 53
- Программирование  
Алгоритм, 23  
Константа, 33  
Массив, 69  
    Двумерный массив, 73, 109  
Переменная, 31  
    Глобальная переменная, 129  
Таблица символов, 119  
Условие, 33, 107  
Цикл, 34  
    Цикл с постусловием (do..while), 35  
    Цикл с предусловием (while), 35  
    Цикл со счётчиком (for), 35
- Разработка игр  
Анимация, 123  
Вывод текста, 98  
Графика, 119  
    Создание символов, 120  
    Стандартные символы, 119  
Дисплей, 97  
Жанр и сюжет, 100  
Звук, 118  
Игровая карта, 109, 115
- Библиотека функций, 117  
Игровая логика, 124  
Выигрыш, 124  
Квесты, 127  
Подсчёт отчков, 129  
Проигрыш, 125  
Игровой персонаж, 100  
Обработка столкновений с объектами, 112  
Ограничение движения игрока по карте, 107  
Подключение кнопки, 102  
Обработка нажатий, 104  
Сбор предметов с карты, 115  
Управление, 101
- Электроника  
ЖК-дисплей, 98  
Мультиметр, 26  
Напряжение, 9  
Разность потенциалов, 11  
Сопротивление, 12  
    Параллельное подключение резисторов, 15  
Последовательное подключение резисторов, 14  
Электрическая цепь, 10



# Словарь терминов

**BPM** Beats Per Minute – удары в минуту. 60

**LCD** Liquid Crystal Display – Жидкокристаллический дисплей. 98

**NPC** Non-Playable Character – Не-игровой персонаж. 123

**ГГц** Гигагерц ( $10^9$  Гц). 53

**Гц** Герц. 53

**КГц** Килогерц ( $10^3$  Гц). 53

**МГц** Мегагерц ( $10^6$  Гц). 53

**ШИМ** Широтно-Импульсная Модуляция. 47, 48, 54



## Приложение А

### Октачная система

№ октавы	Слоговое обозначение	Научное обозначение	Частота (Гц)
0	До	C0	16.352
	Ре	D0	18.354
	Ми	E0	20.602
	Фа	F0	21.827
	Соль	G0	24.500
	Ля	A0	27.500
	Си	B0 (H0)	30.868
1	До	C1	32.703
	Ре	D1	36.708
	Ми	E1	41.203
	Фа	F1	43.654
	Соль	G1	48.999
	Ля	A1	55.000
	Си	B1 (H1)	61.735
2	До	C2	65.406
	Ре	D2	73.416
	Ми	E2	82.407
	Фа	F2	87.307
	Соль	G2	97.999
	Ля	A2	110.000
	Си	B2 (H2)	123.470
3	До	C3	130.810
	Ре	D3	146.830
	Ми	E3	164.810
	Фа	F3	174.610
	Соль	G3	196.000
	Ля	A3	220.000
	Си	B3 (H3)	246.940
4	До	C4	261.630
	Ре	D4	293.660
	Ми	E4	329.630
	Фа	F4	349.230
	Соль	G4	392.000

	Ля A4	440.000
	Си B4 (H4)	493.880
	До C5	523.250
	Ре D5	587.320
	Ми E5	659.260
5	Фа F5	698.460
	Соль G5	783.990
	Ля A5	880.000
	Си B5 (H5)	987.770
	До C6	1046.500
	Ре D6	1174.700
	Ми E6	1318.500
6	Фа F6	1396.900
	Соль G6	1568.000
	Ля A6	1760.000
	Си B6 (H6)	1975.500
	До C7	2093.000
	Ре D7	2349.300
	Ми E7	2637.000
7	Фа F7	2793.800
	Соль G7	3136.000
	Ля A7	3520.000
	Си B7 (H7)	3951.100
	До C8	4186.000
	Ре D8	4698.600
	Ми E8	5274.000
8	Фа F8	5587.700
	Соль G8	6271.900
	Ля A8	7040.000
	Си B8 (H8)	7902.100

Таблица А.1: Октаавная система

## Приложение В

### Ноты мелодии “Кабы небыло зимы”

Рис. В.0.1: Мелодия “Кабы небыло зимы” из мультфильма “Простоквашино”. Автор: Евгений Крылатов.





## Приложение С

# Мелодия “Twinkle Twinkle Little Star” (версия 1.0)

Данный пример показывает полный вариант программирования мелодии “Twinkle Twinkle, Little Star”. Как можно видеть, пример получился достаточно длинным, отчасти потому, что в нём не используются массивы.

```
// Ноты:
const float c4 = 261.630;
const float d4 = 293.660;
const float e4 = 329.630;
const float f4 = 349.230;
const float g4 = 392.000;
const float a4 = 440.000;
const float b4 = 493.880;

// Номер порта, куда подключен динамик:
const int SPEAKER_PIN = 8;

// Функция воспроизведения звука на порту PORT
// с частотой F и длительностью T.
void play_tone(int port, float f, long t) {
    const long T = 1000000 / f;
    long d = T / 2;
    int count = t / T;
    for (int i = 0; i < count; i++) {
        digitalWrite(port, HIGH);
        delayMicroseconds(d);
        digitalWrite(port, LOW);
        delayMicroseconds(d);
    }
}

void setup() {
    pinMode(SPEAKER_PIN, OUTPUT);
}
```

```

void loop() {
    const long BPM = 120; // Удары в минуту.
    const long MINUTE = 60000000; // 1 минута в микросекундах.
    const long T = (MINUTE / BPM) * 4; // Длина целой ноты.
    const byte D = 100; // Задержка между нотами.

    // 0-й такт.
    play_tone(SPEAKER_PIN, c4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, c4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, g4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, g4, T / 4);
    delay(D);

    // 1-й такт.
    play_tone(SPEAKER_PIN, a4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, a4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, g4, T / 2);
    delay(D);

    // 2-й такт.
    play_tone(SPEAKER_PIN, f4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, f4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, e4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, e4, T / 4);
    delay(D);

    // 3-й такт.
    play_tone(SPEAKER_PIN, d4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, d4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, c4, T / 2);
    delay(D);

    // 4-й такт.
    play_tone(SPEAKER_PIN, g4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, g4, T / 4);
    delay(D);
    play_tone(SPEAKER_PIN, f4, T / 4);
    delay(D);
}

```

```
play_tone(SPEAKER_PIN, f4, T / 4);
delay(D);

// 5-й такт.
play_tone(SPEAKER_PIN, e4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, e4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, d4, T / 2);
delay(D);

// 6-й такт.
play_tone(SPEAKER_PIN, g4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, g4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, f4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, f4, T / 4);
delay(D);

// 7-й такт.
play_tone(SPEAKER_PIN, e4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, e4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, d4, T / 2);
delay(D);

// 8-й такт.
play_tone(SPEAKER_PIN, c4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, c4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, g4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, g4, T / 4);
delay(D);

// 9-й такт.
play_tone(SPEAKER_PIN, a4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, a4, T / 4);
delay(D);
play_tone(SPEAKER_PIN, g4, T / 2);
delay(D);
}
```

144ПРИЛОЖЕНИЕ С. МЕЛОДИЯ “TWINKLE TWINKLE LITTLE STAR” (ВЕРСИЯ 1.0)

## Приложение D

# Мелодия “Twinkle Twinkle Little Star” (версия 2.0)

В данном примере мы видим полный вариант программирования мелодии “Twinkle Twinkle, Little Star”, написанный с использованием одномерных массивов. Если сравнить с кодом в приложении C, то видно, что программа стала значительно короче; однако её можно сократить и даже упростить в некотором роде ещё – если использовать двумерные массивы.

```
// Ноты:  
const float c4 = 261.630;  
const float d4 = 293.660;  
const float e4 = 329.630;  
const float f4 = 349.230;  
const float g4 = 392.000;  
const float a4 = 440.000;  
const float b4 = 493.880;  
  
// Номер порта, куда подключен динамик:  
const int SPEAKER_PIN = 8;  
  
// Функция воспроизведения звука на порту PORT  
// с частотой F и длительностью T.  
void play_tone(int port, float f, long t) {  
    const long T = 1000000 / f;  
    long d = T / 2;  
    int count = t / T;  
    for (int i = 0; i < count; i++) {  
        digitalWrite(port, HIGH);  
        delayMicroseconds(d);  
        digitalWrite(port, LOW);  
        delayMicroseconds(d);  
    }  
}  
  
void setup() {
```

146ПРИЛОЖЕНИЕ D. МЕЛОДИЯ “TWINKLE TWINKLE LITTLE STAR” (ВЕРСИЯ 2.0)

```
pinMode(SPEAKER_PIN, OUTPUT);
}

// Массив с нотами (их частотами.)
float melody[28] = {
    c4, c4, g4, g4,
    a4, a4, g4,
    f4, f4, e4, e4,
    d4, d4, c4,
    g4, g4, f4, f4,
    e4, e4, d4,
    g4, g4, f4, f4,
    e4, e4, d4,
};

// Массив с длительностями нот.
float melody_t[28] = {
    4, 4, 4, 4,
    4, 4, 2,
    4, 4, 4, 4,
    4, 4, 2,
    4, 4, 4, 4,
    4, 4, 2,
    4, 4, 4, 4,
    4, 4, 2,
};

void loop() {
    const long BPM = 120; // Удары в минуту.
    const long MINUTE = 60000000; // 1 минута в микросекундах.
    const long T = (MINUTE / BPM) * 4; // Длина целой ноты.
    const byte D = 100; // Задержка между нотами.

    // Цикл воспроизведения мелодии из массива.
    for (int note_idx = 0; note_idx < 28; note_idx++) {
        play_tone(SPEAKER_PIN,
                  melody[note_idx],
                  T / melody_t[note_idx]);
        delay(D);
    }
}
```