Разработка микропроцессорных систем

Нижегородский радиотехнический колледж
nntc.nnov.ru

2016-2021 гг.

Содержание

Введение

По нашему мнению, программирование является двоякой 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.

Данный курс позволяет людям, желающим освоить программирование, сделать первый шаг на пути в профессии программиста. В рамках курса "Программирование микроконтроллеров" предлагается изучить основы программирования на C, разрабатывая приложения для платформы Arduino (используются специально разработанные учебные макеты для упрощения взаимодействия с платформой.)

Платформа Arduino построена на базе микроконтроллера ATmega и является доступным решением как для обучения программированию, так и для разработки проектов. Arduino и необходимые для начала работы компоненты дёшевы и широко распространены, и могут быть куплены как в России (в частности, в Нижнем Новгороде), так и заказом через интернет с доставкой из-за границы (при этом, покупка оборудования не является обязательной для участников курса.)

По ходу данного курса вам предлагается сделать ряд небольших проектов: различные виды "бегущего огня", вывод информации на семисегментный индикатор, управление работой системы с помощью кнопок и потенциометров; в конце курса рассматривается тема воспроизведение музыки на Arduino (с основами музыкальной теории) и вывод звука на подключённый динамик.

Знания и навыки, полученные в рамках данного курса, могут быть применены при разработке программ на C/C++ под другие платформы (x86, x86_64, ARM и пр.)

Авторы

Данная книга/методическое пособие разработано Артёмом Вячеславовичем Попцовым <avp@nntc.nnov.ru>.

Кроме того, в разработке данного документа принимали участие следующие люди:

- 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.

Лицензия

Copyright © 2016, 2017, 2018, 2019 Артём Вячеславович Попцов <avp@nntc.nnov.ru>

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

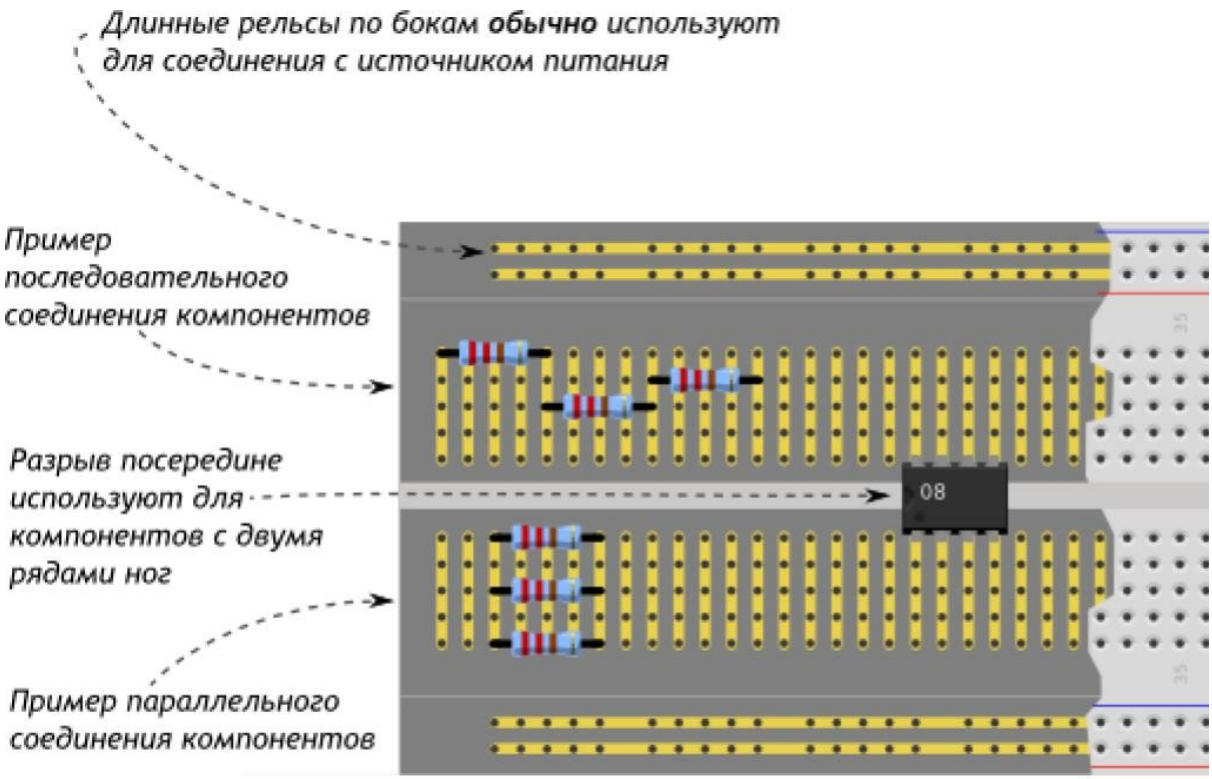
Данная работа распространяется на условиях лицензии «Attribution-NonCommercial-ShareAlike» («Атрибуция — Некоммерческое использование — На тех же условиях») 4.0 Всемирная (CC BY-NC-SA 4.0):

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.ru

Тема 1. Знакомство с набором разработки Arduino

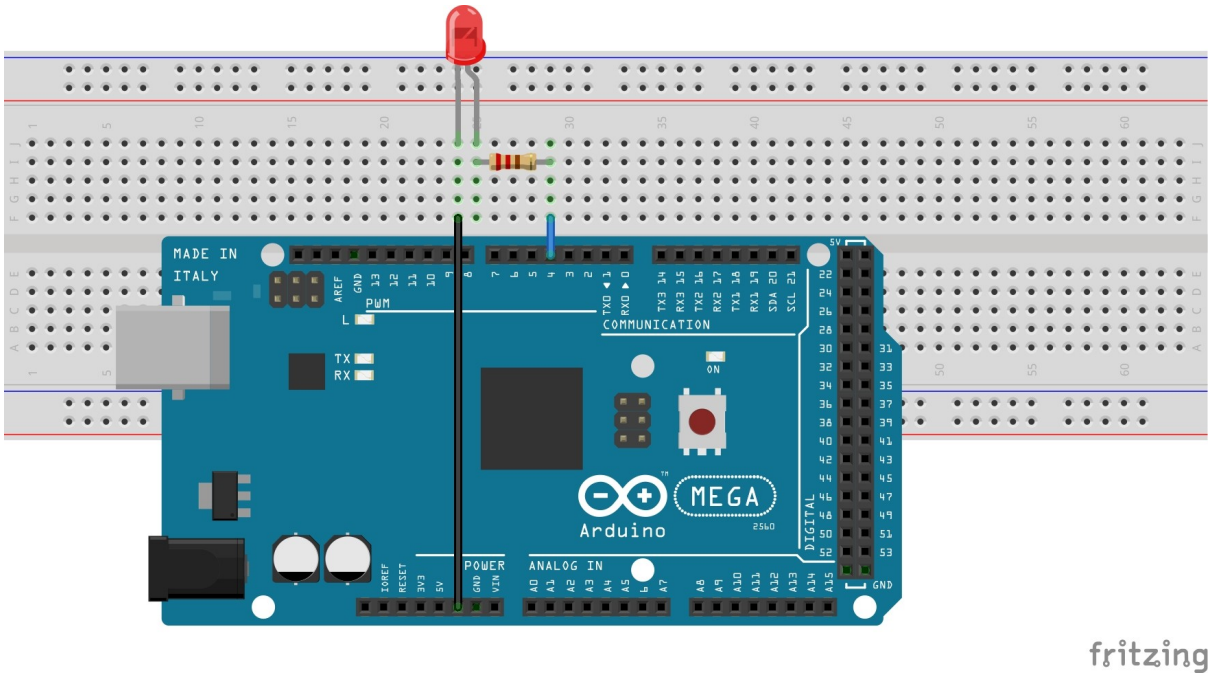
01.01. Работа с макетной платой

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.[1]



1 Больше про работу с макетной платой можно узнать на сайте http://arduino.ru/

Пример подключения светодиода к Arduino:



- Черный провод подключен к arduino и идёт на вывод gnd (минус)
- Синий провод подключен к arduino и идёт 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.

01.02. Подключение Arduino к компьютеру

Чтобы подключить ардуино 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 выберите

Arduino с которой вы сейчас работаете, затем в подменю “Порт” выберите порт, к которому подключена Arduino.

01.03. Знакомство со средой разработки Arduino[2]

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.

Ниже приведено описание кнопок в интерфейсе Arduino IDE.

Пиктограмма	**Название**	**Описание**
	Verify/Compile	Проверка программного кода на ошибки, компиляция.
	New	Создание нового скетча.

2 http://arduino.ru/Arduino_environment

	Open	Открытие меню доступа ко всем скетчам в блокноте. Открывается нажатием в текущем окне.
	Save	Сохранение скетча.
	Upload to I/O Board	Компилирует программный код и загружает его в устройство Arduino. Описание загрузки приведено ниже.
	Serial Monitor	Открытие мониторинга последовательной шины (Serial monitor).

01.04. Основы работы с мультиметром[3]

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.

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

3 http://sebeadmin.ru/kak-polzovatsa-multimetrom.html

Где цифрами обозначены:

1. Режим вольтметра постоянного тока.
2. Режим вольтметра переменного тока.
3. Режим амперметра постоянного тока.
4. Режим измерения электрического сопротивления.
5. Режим прозвонки (свето)диодов.
6. Выходы для подсоединения щупов.

01.05. 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 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.

01.05.01. Задачи

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

01.06 Переменные и память

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:

Условное отображение оперативной памяти:

Адрес ячейки	Значение ячейки
0000	15
0001	3
0002	0
…	0
…	0

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

Переменная имеет определённый тип и своё 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:**

result = a + b; //Если присваивать значение переменной после её объявления, то повторно писать её тип не нужно.

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 delayVal:

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

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

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

Можно, например, увеличивать delayVal на 100 при каждом выполнении loop():

```
void loop() {
    int delayVal = 100;
```

```
        digitalWrite(2, HIGH);
        delay(delayVal);
        digitalWrite(2, LOW);
        delay(delayVal);
        delayVal = delayVal + 100;
}
```

Кстати, строчку **“delayVal = delayVal + 100”** можно заменить на **delayVal += 100** 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 delayVal со значением 100 и потому задержка остаётся той же. Сейчас delayVal объявлена как локальная переменная внутри loop(), следует объявить её за пределами функции, чтобы она стала глобальной:

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

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

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

```
        if(delayVal < 600){
                delayVal += 100;
        }
}
```

Об условиях и других управляющих конструкциях - в следующей главе.

01.07. Управляющие конструкции языка C++

01.07.01. 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.

01.07.02. 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 (параметрический цикл).
- Цикл с предусловием.
- Цикл с постусловием.

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

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

```
//      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.

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
}
```

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.

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").

01.08. 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.

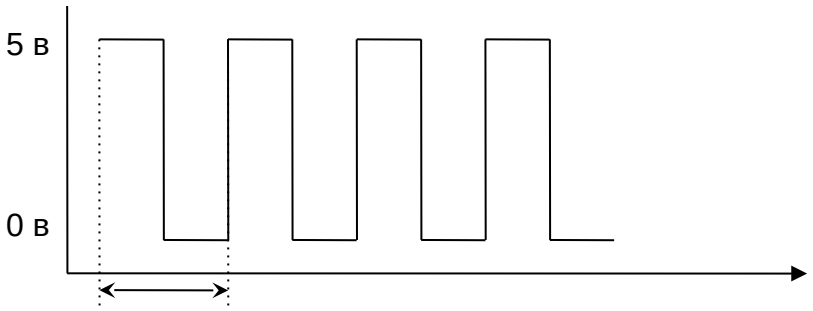
Тема 2. Знакомство с новыми компонентами и конструкциями языка

02.01. Широтно-импульсная модуляция

Широтно-импульсная модуляция, или 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 на порт 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.

02.02. 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 примерно следующую картину:



Длина периода (P)

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 возведение числа 10 в отрицательную степень. Ниже приведена таблица с указанием различных долей секунды:[4]

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

02.03. 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

4 Для полного списка кратных и дольных единиц см. статью "Секунда" в Википедии.

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).

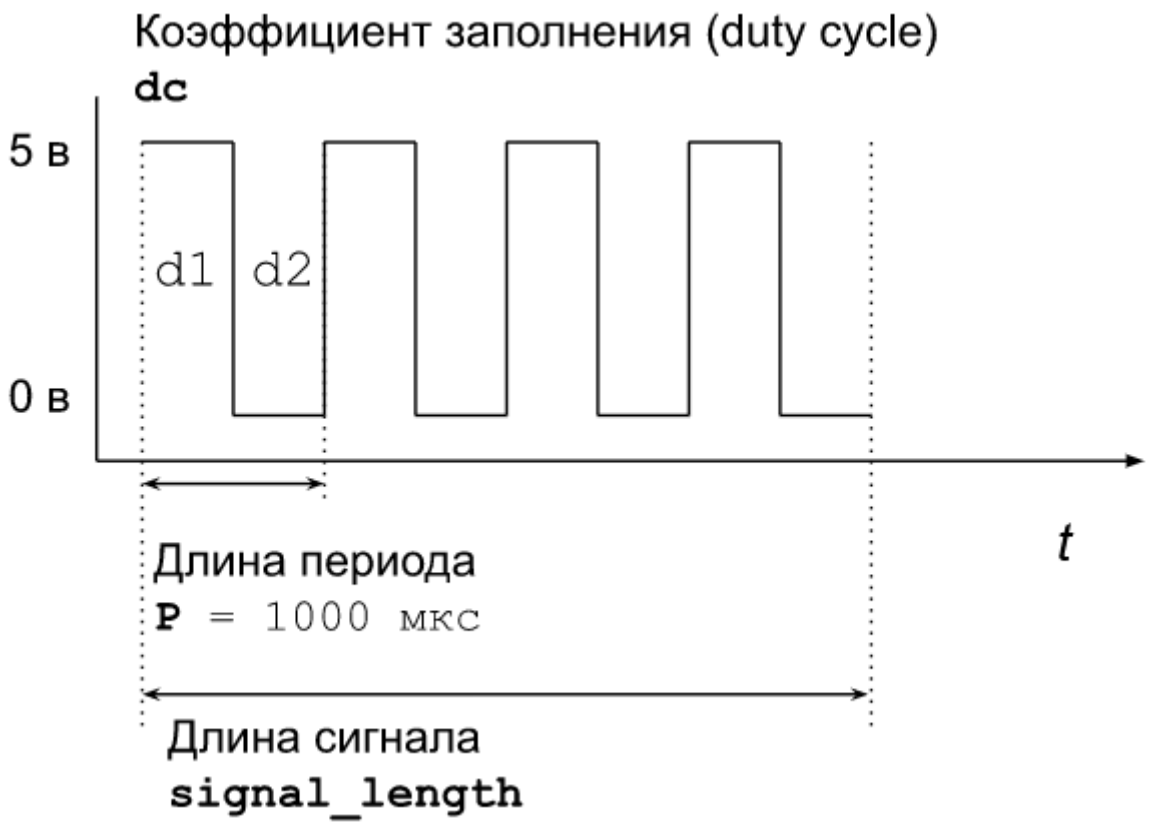
Отношение периода следования сигнала к длительности импульса называется *скважностью*. В англоязычной литературе величина, обратная скважности, называется *коэффициентом заполнения* (англ. *duty cycle*).

Мы будем использовать термин "коэффициент заполнения".

Пример: 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), 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.



В программе мы обозначим длину периода константой `P`, равной 1000 мкс. Значения задержек, заданных переменными `d1` и `d2`, необходимо вычислить на основе коэффициента заполнения, задаваемого параметром функции `duty_cycle` (который задаётся дробным значением -- к примеру, 0.5.)

Имея перед глазами этот график, нетрудно набросать словесное описание функции (назовём функцию `pwm`, 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 `pin`
2. Коэффициент заполнения, заданный дробным числом -- к примеру, 50% будет задано, как 0.5; назовём этот параметр `duty_cycle`
3. Длина ШИМ сигнала в микросекундах; назовём этот параметр `signal_length`

Запишем тоже самое на языке C++:

```
void pwm(int pin, float dc, long signal_length) {
    // 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. Первым делом нам необходимо задать константу P:

```
const int P = 1000; // мкс
```

Обратите внимание, что мы используем ключевое слово `const` для того, чтобы пометить `WAVE_LENGTH`, 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 `duty_cycle`, заданного при вызове функции:

```
int d1 = P * duty_cycle;
int d2 = P - d1;
```

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

```
int count = signal_length / P;
```

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(pin, HIGH);
      delayMicroseconds(d1);
      digitalWrite(pin, LOW);
      delayMicroseconds(d2);
}
```

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

02.03.01. 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.

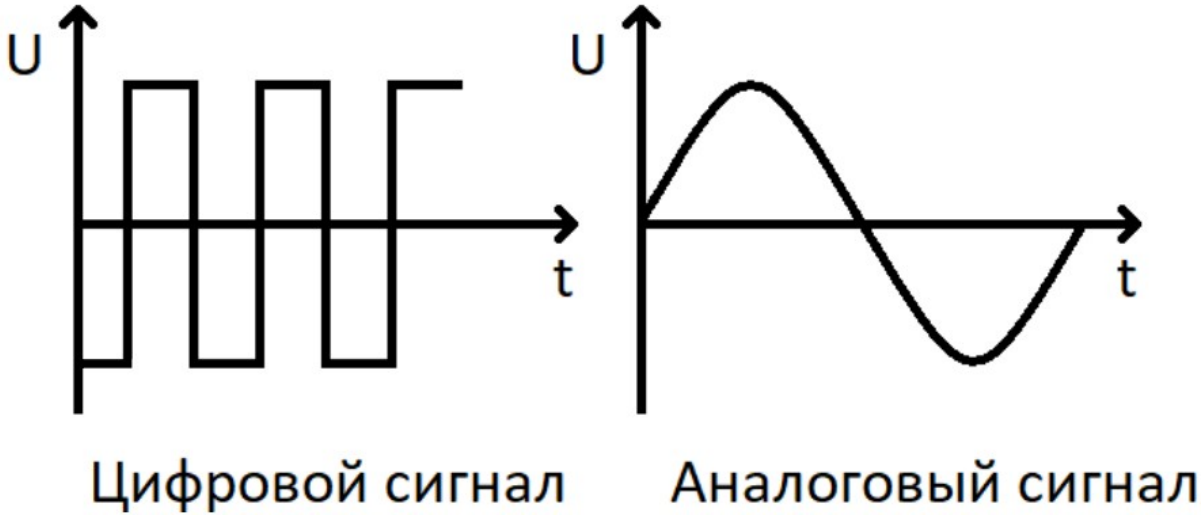
02.04. Использование потенциометра

Поставим себе задачу - вручную регулировать скорость “бегущего огня” в реальном времени. Микроконтроллер должен принимать извне некий сигнал, величину которого мы можем регулировать каким-либо элементом управления и использовать в качестве параметра:

Сигнал -> Элемент управления -> Arduino -> Параметр

Для считывания сигнала можно использовать аналоговые порты Arduino. 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.



Микроконтроллер измеряет напряжение на аналоговом порту и преобразовывает полученное значение в число от 0 до 1023, которое мы можем использовать для своих нужд.

В качестве элемента управления может служить такой компонент как потенциометр. Переменный резистор или потенциометр - это резистор, сопротивление которого, как понятно из названия, можно изменять. Используя потенциометр, мы можем влиять на поступающее напряжение на порт, тем самым регулируя нужный нам параметр.

Подключаем крайние выводы потенциометра к питанию и земле, а центральный - к аналоговому порту.

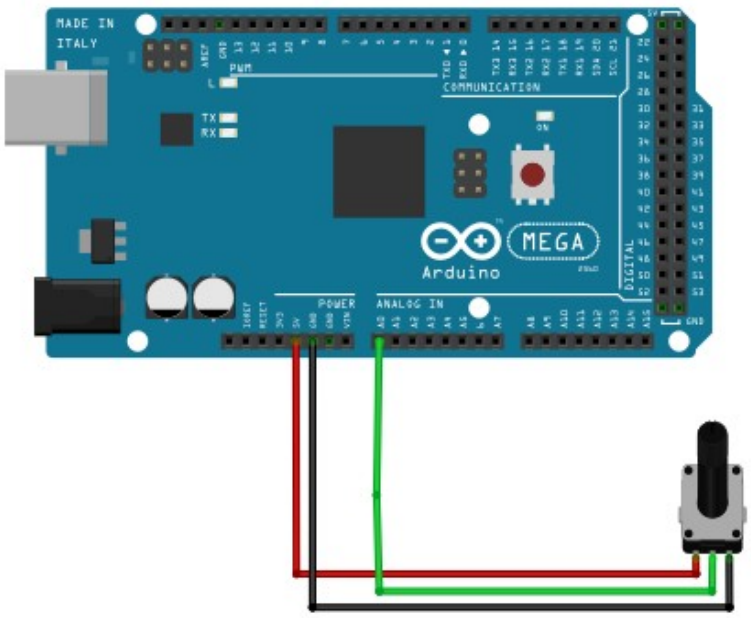


Рисунок - Подключение потенциометра

В коде будем использовать функцию analogRead(analogPin), где analogPin - номер аналогового порта. Эта функция возвращает результат вышеописанной операции - число от 0 до 1023.

```
int value = analogRead(0); //Получаем значение с аналогового порта 0
```

Осталась одна проблема - нам неизвестно значение, которое получил микроконтроллер. Для этого воспользуемся *последовательным портом*.

02.05. Последовательный порт[5]

02.05.01. Основы работы с Arduino через 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.

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

```
Serial.begin(9600);
```

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*.[6] Обычно данный параметр принимает одно из следующих значений: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 115200.

5 Использовались сайты: http://arduino-diy.com/arduino-processing-osnovi
http://arduino.ru/Reference/Serial

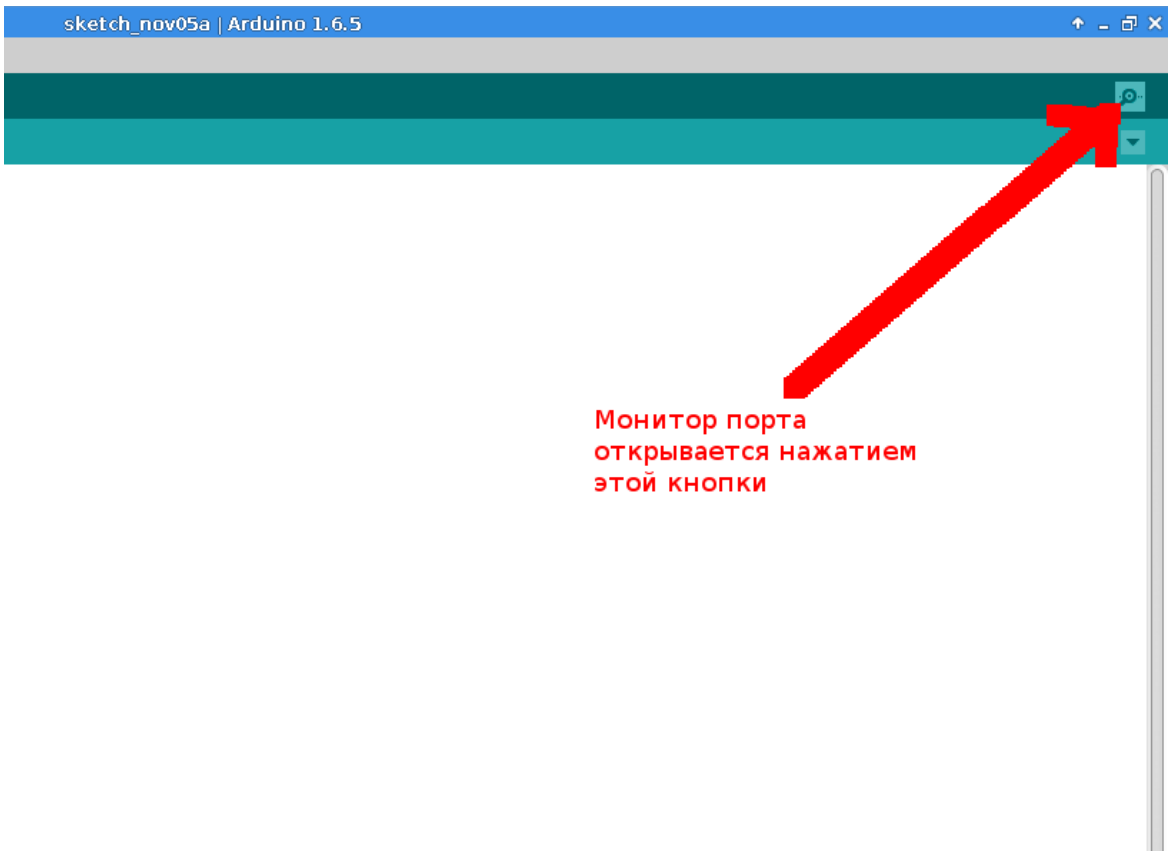
6 https://ru.wikipedia.org/wiki/%D0%91%D0%BE%D0%B4

02.05.02. Передача данных с Arduino на компьютер

Теперь попробуем передать данные с аналогового порта Arduino на компьютер. В качестве примера мы просто отправим строку "Hello, world!" по последовательному порту:

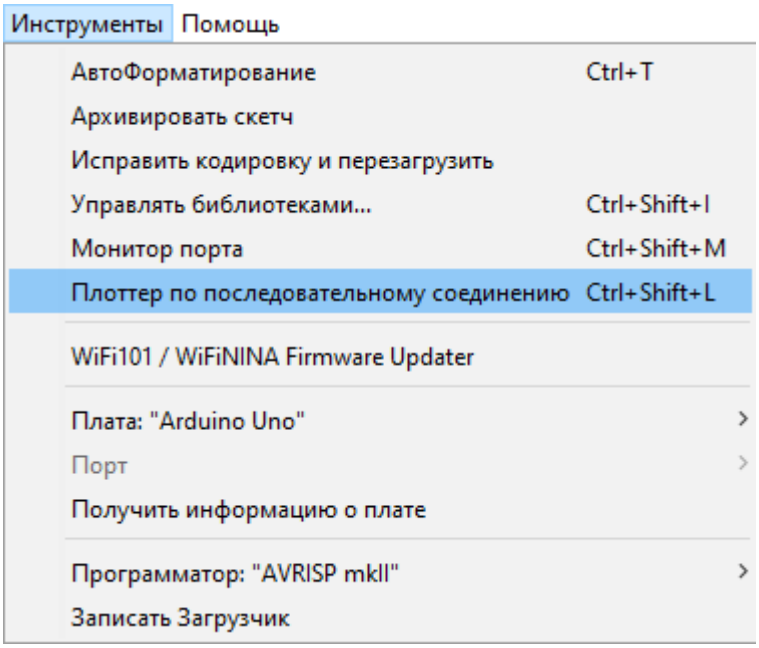
```
void setup() {
  Serial.begin(9600);  // устанавливаем скорость порта
}

  void loop() {

    int value = analogRead(0); // Получаем данные с аналогового порта

     // отправляем переменную value с помощью

    // последовательного порта:

     Serial.println(value);

     delay(100); // ждём 100 мс перед следующей отправкой

  }
```

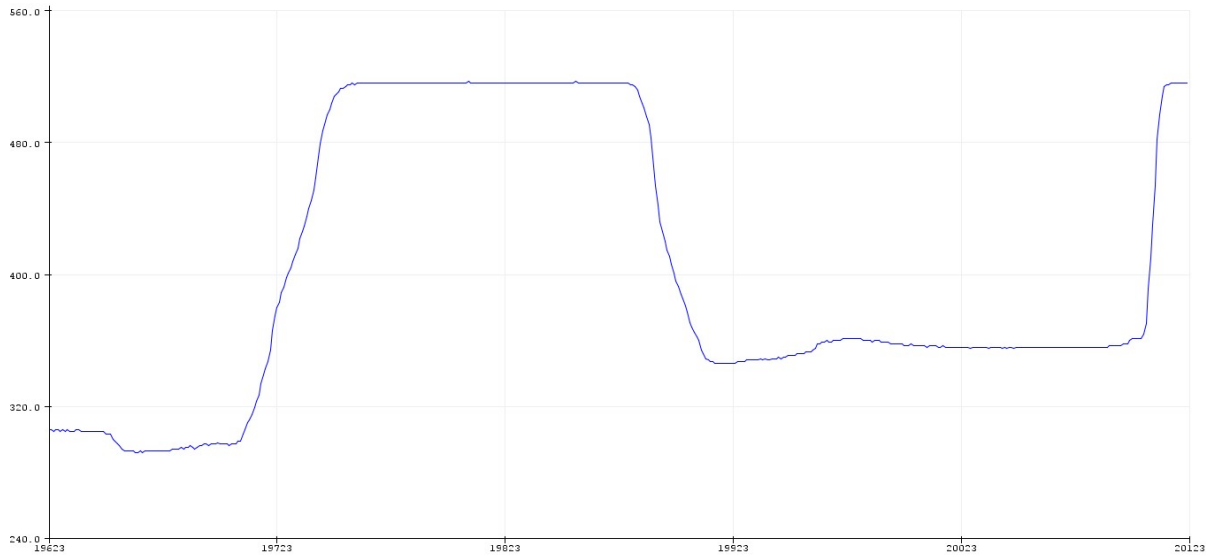
Результат выполнения программы можно увидеть, открыв *монитор порта* в Arduino IDE. Доступ к монитору порта можно получить, нажав кнопку в виде лупы в правом верхнем углу редактора кода:



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

Также можно использовать плоттер по последовательному соединению, который выводит графики поступающих данных. Чтобы его открыть, нужно перейти во вкладку *Инструменты - Плоттер по последовательному соединению:*





Плоттер по последовательному соединению

02.05.03. Использование монитора порта для отладки программ

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.

02.05.04. Сбор и обработка данных на стороне компьютера

Другой задачей, решаемой с помощью функций записи данных в последовательный порт, является сбор данных на стороне компьютера. Arduino IDE позволяет нам визуализировать данные через специальный плоттер (доступ к которому можно получить, выбрав в меню "Инструменты" пункт "Плоттер по последовательному соединению", либо нажав комбинацию клавиш Ctrl+Shift+L)

02.05.05. Передача данных с компьютера на Arduino [7]

Передавать данные с Arduino на компьютер мы уже научились. Теперь посмотрим на передачу данных в обратном направлении. Для того чтобы передать данные с компьютера на Arduino также необходимо выполнить настройку последовательного порта; кроме этого, нам потребуется задействовать несколько новых функций.

7 http://pashkevich.me/article/6.html

Serial.read()

Данная функция читает байт данных из поступивших на Arduino. То есть возвращает вам некое целое число, с которым вы вольны делать что вашей душе угодно.

Каждый вызов этого метода будет возвращать вам следующий байт данных из тех что поступили на Arduino.

Если возвращать нечего, то есть вы считали все что было, данная функция вернет -1

> Примечание: Если передаются именно байты, возникает проблема: -1 это 0xFF, т.е. 255. такой же байт, как и все остальные. поэтому нужно сперва вызывать функцию `available` .

Допустим мы отправили 1 байт данных на Arduino и использовали нижеприведенный участок кода:

```
int incomingByte;
void loop()
{
   if (Serial.available() > 0)
   {
      incomingByte = Serial.read();
   }
}
```

После того как вы считаете этот байт данных, он будет перемещен в вашу переменную а функция `Serial.available` снова будет возвращать НОЛЬ, пока не поступят новые данные.

То есть, когда вы считываете байт, показания счетчика принятых байт уменьшается и `Serial.available` будет показывать на 1 байт меньше.

Помните, данная функция возвращает только 1 байт данных, если например вы передали 4 символа каждый по 1 байту, вам потребуется 4 раза вызвать данную функцию чтобы прочитать эти символы и самостоятельно позаботится о том чтобы разместить их в массив символов либо воспользоваться функцией `Serial.readBytes`.

Serial.parseInt

Данная функция просматривает данные, поступившие на Arduino, и ищет среди них набор кодов (чисел) от 48 до 57, которые соответствуют символам чисел от 0 до 9 и преобразует все это в правильное целочисленное значение.

Таким образом если вы с монитора порта передадите "число" (на самом деле, строку) "72", данный метод увидит 2 последовательных байта 55 и 51, корректно преобразует его в число 72 и вернет его как правильное целочисленное значение.

Давайте напишем маленькую эхо-программу, которая покажет принцип работы данной функции и позволит вам узнать какому символу соответствует то или иное число.

```
int incomingInt = 0;
void setup()
{
   Serial.begin(9600);
   Serial.setTimeout(2000);
}
void loop()
{
   if (Serial.available() > 0)
   {
      incomingInt = Serial.parseInt();
      Serial.write(incomingInt);
   }
}
```

Данная программа будет работать так. Если в мониторе порта вы введете строку "72" то монитор порта отправит его на Arduino как два байта данных в виде чисел 55 и 51, функция `Serial.parseInt` подождет 2000 миллисекунд (как видите я поменял время ожидание с 1 секунды на 2 секунды, чтобы нагляднее показать кое какие аспекты) увидит эти два значения и преобразует их в одно целочисленное 72 и присвоит его переменной `incomingInt`, мы с помощью метода `Serial.write` передадим число 72 как есть обратно в монитор порта (почему именно этот метод нужен читайте далее) где монитор порта корректно преобразует число 72 в соответствующий символ и покажет нам символ "H" который соответствует коду 72.

Таким образом мы можем передавать числовые значения с компьютера на платформу Arduino и дальше использовать эти значения.

02.06. Управление ШИМ и "бегущим огнём" с компьютера

Теперь, когда мы умеем передавать данные через последовательный порт, можно подумать о том, чтобы добавить управление программой Arduino с компьютера.

Для начала попробуем управлять “бегущим огнём” с компьютера. Чтобы это реализовать нам необходимо настроить последовательный порт, затем завести переменную которая будет хранить переданные с компьютера значения. После подготовки мы можем начать видоизменять нашу программу “бегущего огня”. После изменений программа будет выглядеть примерно таким образом:

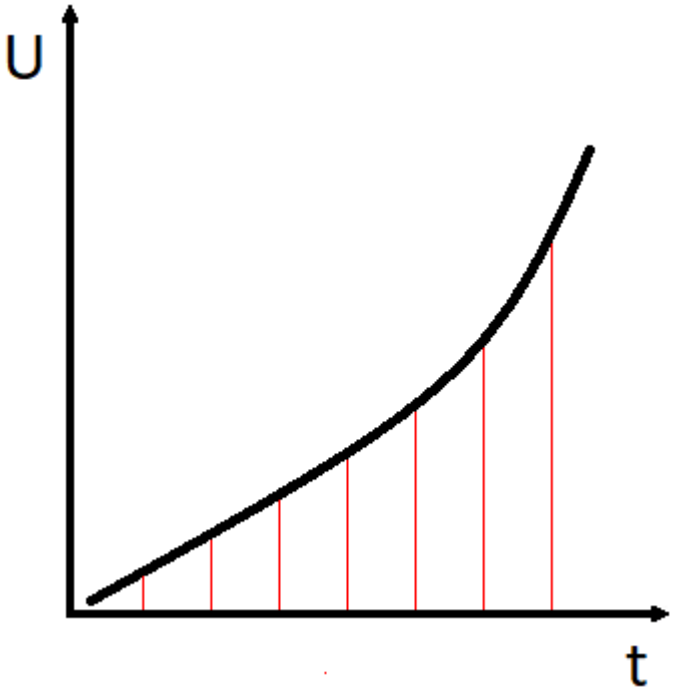
Возьмём уже знакомую нам программу, генерирующую мелодию. Что если позволить пользователю с компьютера выбирать, какую мелодию следует сыграть? Всё так же нам понадобится настроить последовательный порт и задать переменную хранящую значения, переданные с компьютера. Для того чтобы выбрать нужную мелодию нам необходимо: задать массив (саму мелодию), задать каждой мелодии (массиву) свое значение переменной, а затем как показано в прошлом примере - записать переданные данные с компьютера в переменную, затем проверять какое значение поступило с компьютера и подбирать нужную мелодию под это число.

Аналого-цифровой преобразователь

Чтобы не терялась нить повествования, в главе “Использование потенциометра” стоило умолчать о том, как Arduino проводит преобразование напряжения с аналогового порта в дискретное значение от 0 до 1023. Задавшись вопросом - “А почему именно 1023?”, мы сможем познакомиться с такой интересной штукой как АЦП. Аналого-Цифровой Преобразователь (АЦП) - это устройство, которое и проводит преобразование.

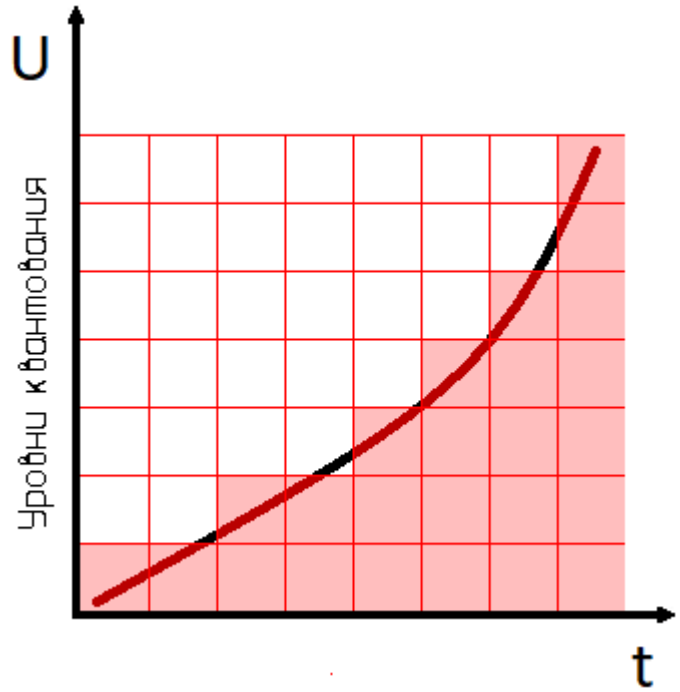
Преобразование проходит в три этапа:

1. Дискретизация. Выбираются значения из исходного аналогового сигнала через равные временные промежутки:

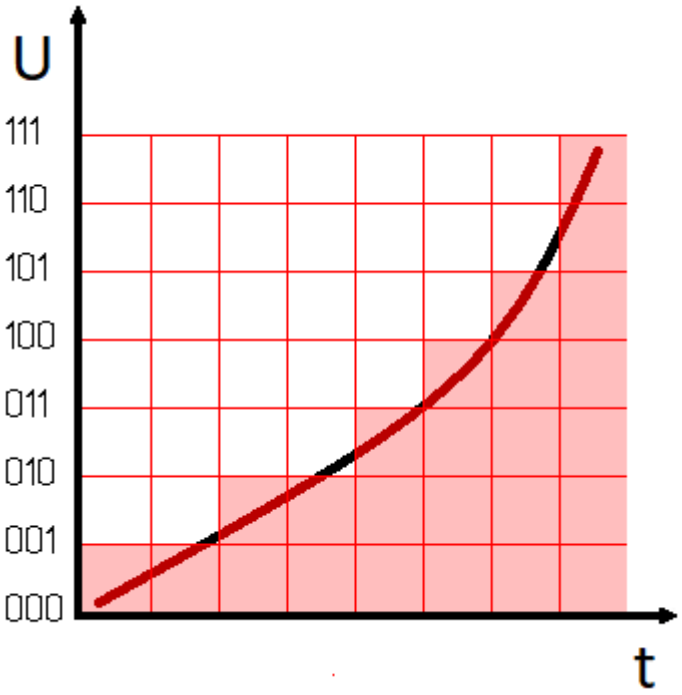


Характеристика, отражающая эти временные промежутки, называется *частота дискретизации*.

2. Квантование. Полученные значения заменяются ближайшим значением из набора фиксированных величин - уровней квантования:



3. Кодирование. Квантованным значениям присваивается цифровой код:



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.

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 видео- картах, 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.

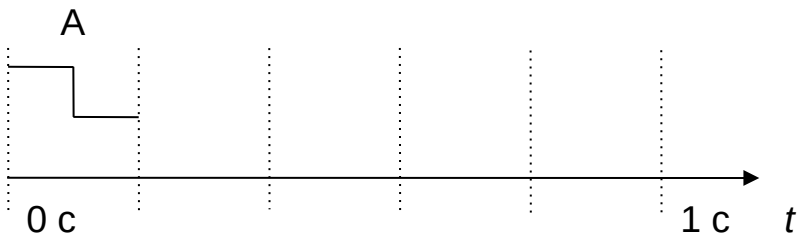
02.07. Генерация звука

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 поданный на цифровой порт колеблет мембрану динамика 1000 раз в секунду. Ниже приведена таблица некоторых кратных единиц частот в Герцах:

Название	**Величина**	**Пример**
Герц (Гц)	$1\,Гц, или\, 10^0\,Гц$	$100 \times 10^0\,Гц$ = 100 Гц
Килогерц (КГц)	$1000\,Гц, или\, 10^3\,Гц$	$100 \times 10^3\,Гц$ = 100 КГц
Мегагерц (МГц)	$1000000\,Гц, или\, 10^6\,Гц$	$100 \times 10^6\,Гц$ = 100 МГц
Гигагерц (ГГц)	$1000000000\,Гц, или\, 10^9\,Гц$	$100 \times 10^9\,Гц$ = 100 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:

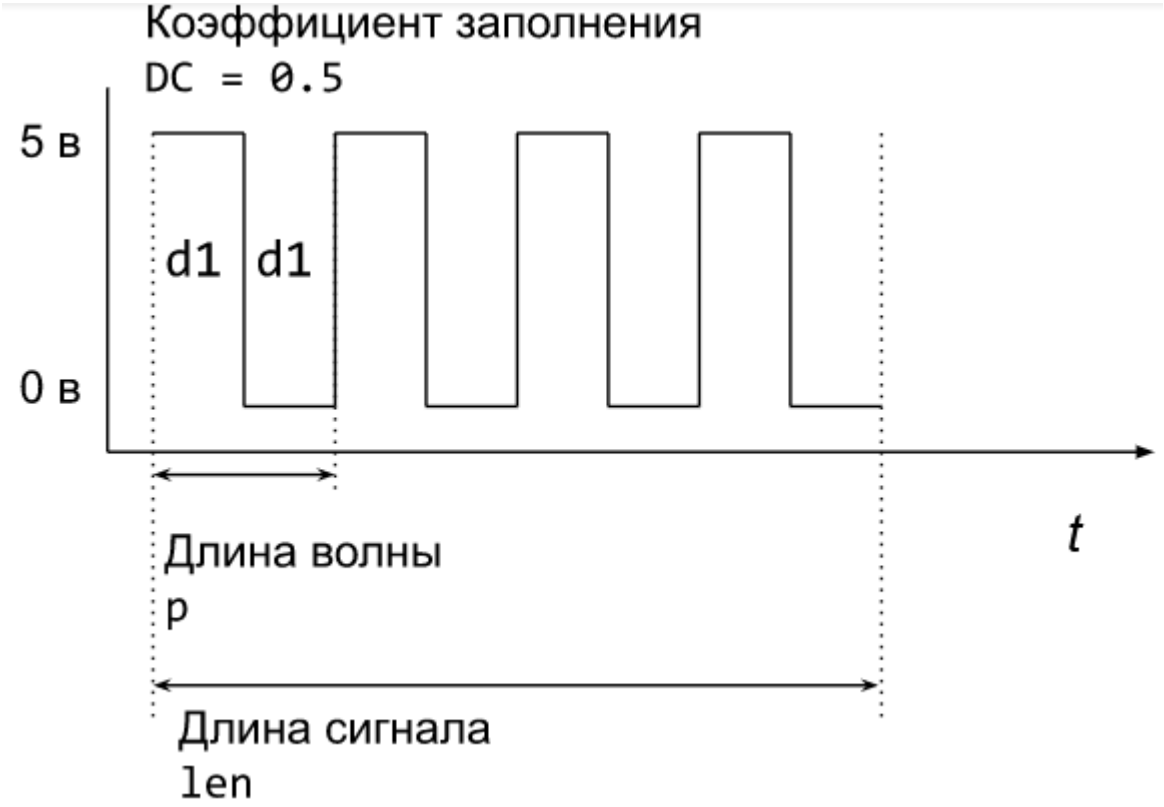


Если известно, что колебание A помещается 5 раз в 1 секунду, то говорят, что частота данного сигнала равна 5 Гц. Узнать период можно, разделив 1 секунду (заданную в микросекундах) на частоту (5 Гц):

$$\frac{1000000\,мкс}{5\,Гц} = 200000\,мкс$$

Получается, что длина волны равна 200000 мкс, или $200 \times 10^3\,мс$. 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 `play_tone` для генерации звука с нужной частотой на указанном порту.

Посмотрим, что данная функция должна принимать в качестве параметров:

1. Номер цифрового порта, к которому подключен динамик; назовём этот параметр `pin`
2. Частота f, измеряемая в Герцах.
3. Длина звукового сигнала; назовём этот параметр `len`

На языке C++ это будет выглядеть примерно так:

```
void play_tone(int pin, float f, long signal_length) {
        // тело функции
}
```

Теперь пришло время написать тело функции. Начнём с того, что зададим коэффициент заполнения в виде константы:

```
        const float DC = 0.5; // 50%
```

Теперь из частоты найдём период p:

```
long p = 1000000 / f;
```

Далее посчитаем длину задержки `d1`:

```
int d1 = p * DC;
```

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

```
int count = len / p;
```

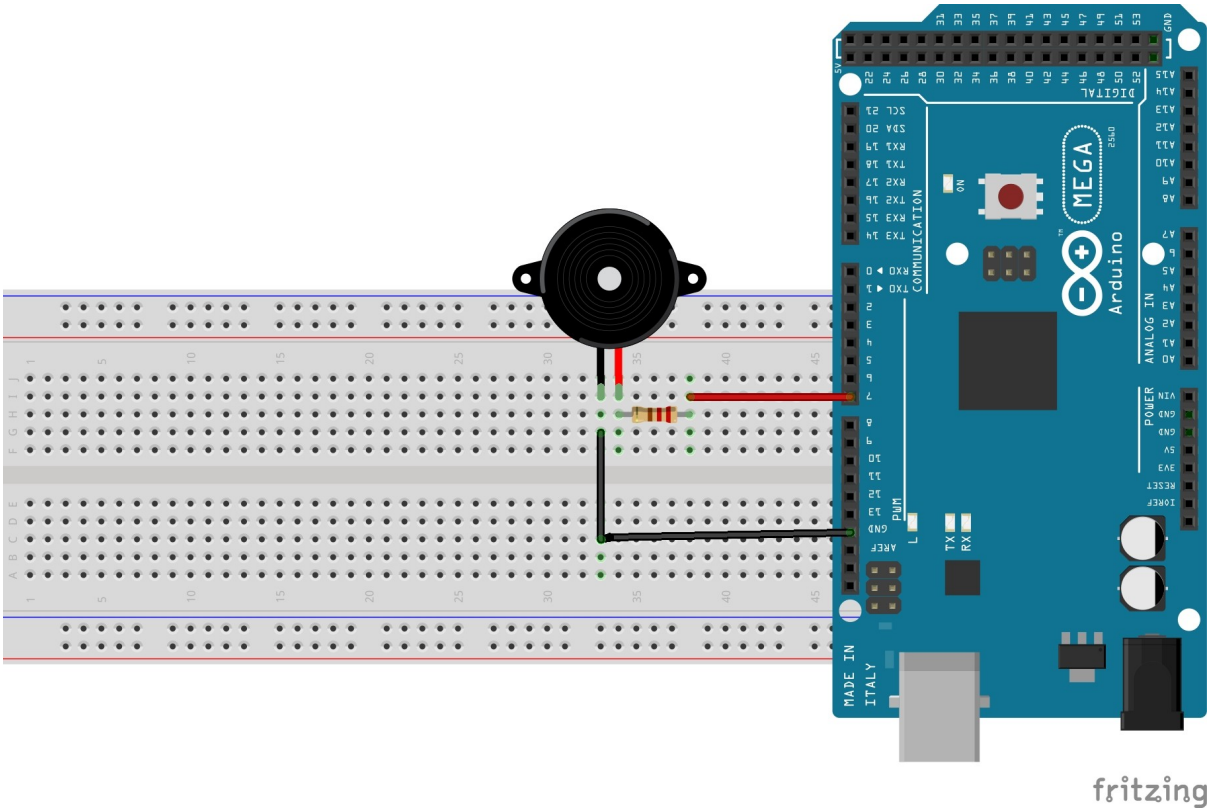
Почти всё готово. Осталось только написать цикл, который будет генерировать заданную волну нужное количество раз. Здесь отлично подойдёт цикл со счётчиком, `for`:

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

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

> Но прежде, небольшое отступление. Дело в том, что в большинстве случаев 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. Как вариант, вы можете реализовать вариант функции, которая оперирует не длиной волны, а частотой. Подумайте над этим в свободное время. И не бойтесь экспериментировать!

02.07.01. Подключение динамика[8]



- Красный провод идет, через резистор, на цифровой вход (в данном случае на 7-ой цифровой вход).
- Чёрный провод идёт на выход GND (на минус).

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.[9]

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?)

8 https://sites.google.com/site/vanyambauseslinux/arduino/kak-podklucit-k-arduino
9 См. статью "Пьезоэлектрический эффект" в Википедии для более подробного описания эффекта.

Соберём указанную выше 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.

02.07.02. 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.
3. Модифицируйте систему таким образом, чтобы частота сигнала зависела от положения ручки потенциометра.
4. Сделайте включение звукового сигнала по нажатию кнопки.

02.08. Синтез музыки и технологии

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.

02.08.01. Октавная система

Как вы, возможно, знаете, музыка строится из нот -- их всего семь: до, ре, ми, фа, соль, ля, си. Каждой ноте соответствует определённая частота. Мы уже встречали с вами частоту 261.63 Гц в предыдущей главе -- так вот, данная частота соответствует ноте "си". Ниже в таблице представлена таблица названий нот и их частот[10]:

Слоговое обозначение	**Научное обозначение**	**Частота, в Герцах**
до[1]	C4	261.63

10 Для полной октавной системы смотрите статью “Октавная система” в википедии (https://ru.wikipedia.org/wiki/Октавная_система)

ре[1]	D4	293.66
ми[1]	E4	329.63
фа[1]	F4	349.23
соль[1]	G4	392.00
ля[1]	A4	440.00
си[1]	H4 (B4)	493.88

Кроме слогового обозначения нот (от "до" до "си"), в центре можно заметить столбец с обозначением нот по *научной нотации* -- мы будем с вами использовать именно научную нотацию в дальнейшем. Теперь попробуйте найти по таблице ноту с частотой 349.23 Гц, которую мы использовали с вами в предыдущей главе -- какая это нота?

Хотя нот всего семь, диапазон частот, используемых в музыке, более обширен -- он простирается выше и ниже нашей таблицы: от 16.352 Гц до 7902.1 Гц. Как такое может быть? Дело в том, что данный диапазон частот разделён на 9 групп -- *октав* -- и в каждой октаве повторяются все те же семь нот, но уже с другими частотами.

Для различения нот разных октав обычно явно указывают, к какой октаве та или иная нота принадлежит (к примеру, нота "до" первой октавы.) В научной нотации октава, к которой принадлежит нота, определяется числом, стоящим после английской буквы (C, D, E, F, G, A, H или B). К примеру, F0 обозначает ноту "фа" самой низкой по частоте октавы, а нота F8 обозначает ноту "фа" самой высокой октавы. Иными словами, чем больше число после буквы, тем выше частота ноты.

У каждой октавы есть своё имя. Ноты C4-H4 в таблице, которую мы с вами рассмотрели выше, относятся к *первой октаве* -- она находится в центре частотного диапазона. Ноты C5-H5 относятся ко *второй октаве*, которая -- как можно догадаться -- идёт после первой. А вот октава, которая идёт *перед* первой, называется *малой октавой*, и ноты, которые в неё входят, именуются C3-H3 в научной нотации.

Одинаковые ноты в разных октавах звучат на слух очень похоже -- например, нота C4 звучит похоже на ноту C3, только выше по частоте. Если посмотреть на частоты этих двух нот, то можно заметить, что нота C4 ("до" первый октавы) в два раза выше по

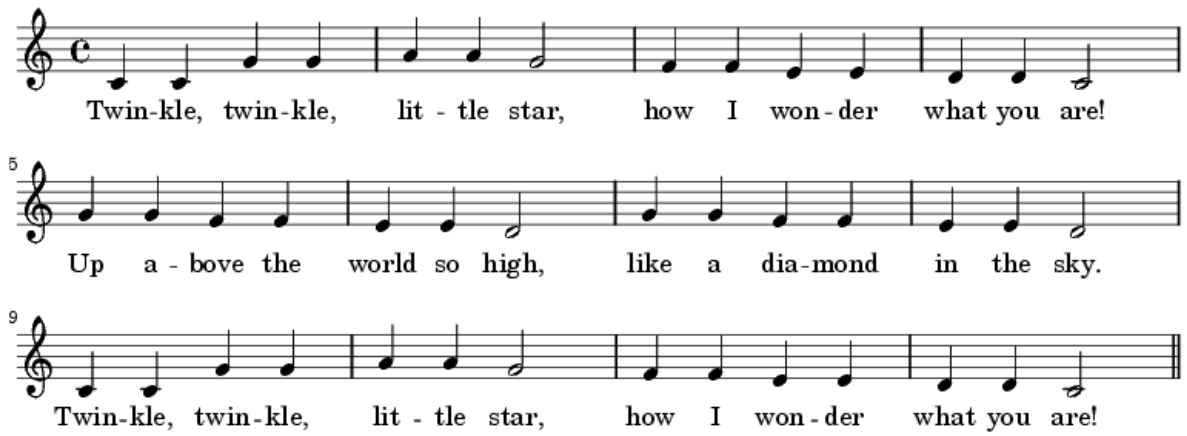
частоте, чем нота C3 ("до" малой октавы). Именно это свойство и лежит в основе октавной системы, используемый в музыке.

Если вам интересно покопать глубже этот вопрос, то рекомендуем ознакомиться со статьёй "Октавная система" в русскоязычной Википедии -- в этой статье можно посмотреть частоты нот для всех октав, что может пригодиться при программировании мелодий на Arduino.

02.08.02. 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"[11] -- английскую колыбельную:



Это относительно простая мелодия, однако без знания нотной записи прочитать её будет проблематично. Мы не будем сейчас вдаваться в изучение нотной грамоты; вместо этого ниже представлена запись этой мелодии в научной нотации:

```
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
C4 C4 G4 G4 A4 A4 G4
```

Для того, чтобы запрограммировать данную мелодию, удобно в начале программы объявить каждую ноту в виде константы. Каждая константа будет хранить длину волны, рассчитанную из частоты:[12]

11 https://ru.wikipedia.org/wiki/Twinkle,_Twinkle,_Little_Star

12 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

```
const long c4 = 1000000 / 261.63;
const long d4 = 1000000 / 293.66;
// И так далее
```

Как только мы объявили все необходимые константы (для "Twinkle, Twinkle, Little Star" нам потребуются только ноты из первой октавы), то запрограммировать мелодию не составит труда. Однако, как мы уже говорили, большинство задач может быть решено несколькими способами, и некоторые способы лучше (можно сказать, *красивее*), чем другие.

Первое решение, которое может прийти на ум -- это вызвать нашу функцию `play_tone` для каждой ноты, в нужном нам порядке:

```
const long NOTE_LENGTH = 1000000; // 1 секунда
void loop() {
    play_tone(SPEAKER_PIN, c4, NOTE_LENGTH);
    play_tone(SPEAKER_PIN, c4, NOTE_LENGTH);
    play_tone(SPEAKER_PIN, g4, NOTE_LENGTH);
    play_tone(SPEAKER_PIN, g4, NOTE_LENGTH);
    // и так далее
}
```

Но данное решение слишком громоздко - представьте, сколько раз нужно будет вызвать прописывать функцию `play_tone`, если мелодия будет состоять из большего количества нот! По сути, нам придётся делать один вызов функции на каждую ноту.

Вероятно есть более красивое решение данной задачи. Если мы посмотрим на пример кода выше, то увидим, что все вызовы функции `play_tone` похожи, за исключением второго параметра -- ноты. Мы уже сталкивались с подобной ситуацией, когда делали "бегущий огонь" -- вы наверняка помните, как была решена проблема с повторяющимся кодом? Мы использовали циклы!

Как только вы освоитесь с циклами, то сможете быстро выявлять в коде места, которые можно переписать с использованием циклов, и таким образом, сделать ваши

избежать конфликтов имён с константами, которые уже есть в Arduino (к примеру, A4.)

программы короче. И красивее. Не забывайте, что программы должны быть написаны, в первую очередь, для чтения людьми, и только во вторую очередь -- для исполнения компьютером.[13]

В данном случае, для использования цикла нам потребуется организовать наши данные - ноты мелодии в подходящую структуру, которую удобно использовать с циклами. И здесь как нельзя лучше подходят *массивы*.

13 *"programs must be written for people to read, and only incidentally for machines to execute."* -- Abelson & Sussman, *SICP,* preface to the first edition:
https://mitpress.mit.edu/sicp/front/node3.html

02.08.03. Использование массива для программирования мелодии

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

```
long melody[28];
```

Массив, содержащий 28 переменных типа `long` объявлен, но сейчас он пуст, нужно заполнить его нотами нашей мелодии:

```
long melody[28] = {
      c4, c4, g4, g4, a4, a4, g4,
      // и так далее
};
```

Для обращения к определённым элементам массива нужно написать имя массива и в квадратных скобках номер элемента. Например, захотелось нам поменять значение первого элемента, сделаем мы это так:

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

Обратите внимание, что нумерация начинается с нуля, т.е. первый элемент имеет индекс 0, а второй - 1, и так далее.

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:

```
const long NOTE_LENGTH = 1000000; // 1 секунда
void loop() {
      for (int note_idx = 0; note_idx < 28; ++note_idx) {
            play_tone(SPEAKER_PIN, melody[note_idx], NOTE_LENGTH);
      }
```

```
}
```

В следующей главе мы посмотрим, как сделать нашу мелодию более благозвучной.

02.08.04. Задачи

1. Добавьте паузы между воспроизведением нот.
2. Попробуйте сделать управление скоростью воспроизведения мелодии с использованием потенциометра.
3. Сделайте "светомузыку" -- модифицируйте программу и схему на макетной плате таким образом, чтобы на каждую из семи нот загорался свой светодиод. После воспроизведения ноты светодиод должен гаснуть.

02.09. Основы нотной грамоты

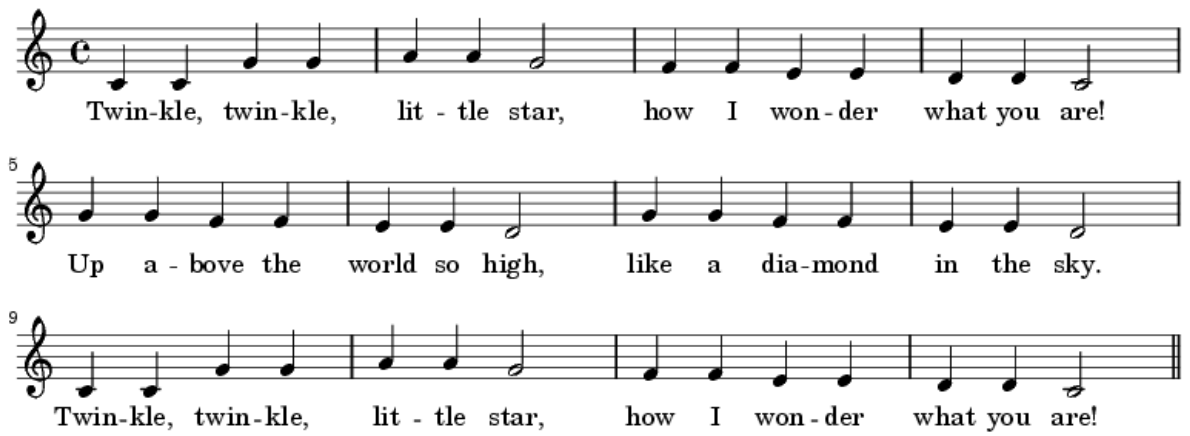
В предыдущих главах мы запрограммировали Arduino для воспроизведения простой мелодии. Тем не менее, с точки зрения музыкальной теории, наша мелодия ещё далека от совершенства. Чтобы сделать мелодию более благозвучной, нам потребуется усовершенствовать нашу программу (и попутно ещё немного разобраться в нотной грамоте.)

02.09.01. Нотная запись

Мы можем программировать простые мелодии, не зная нотной записи -- 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 нотной записи.

> Вы можете пока пропустить данный раздел, но мы рекомендуем ознакомиться с ним в любом случае, если вы не знакомы с нотной грамотой. Информация, данная здесь, может помочь вам при программировании мелодий.

Посмотрим ещё раз на мелодию "Twinkle, Twinkle, Little Star":



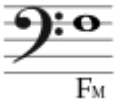
Ранее мы проигнорировали все эти закорючки, называемые нотами, и сразу перешли к использованию научной нотации (поскольку она проще.) Но что же всё-таки означают эти линии и странные обозначения на них?

Начнём с линий -- они называются *нотным станом* (его также называют "нотоносцем", поскольку он "несёт" на себе ноты.) Поверх нотного стана записываются ноты, паузы 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 ноту "соль" (G) на нотном стане:

Как можно видеть на рисунке выше, вторая линия снизу соответствует ноте "соль" -- поскольку она обведена скрипичным ключом. 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" (G) -- а именно нота "ля" (A). Третья же линейка (средняя) соответствует ноте "си" (H или B) и так далее. Если будем двигаться вниз от ноты "соль", то будем идти в обратную сторону: между первой снизу и второй линией находится нота "фа" (F), самая первая (нижняя) линия соответствует ноте "ми" (E).[14]

Кроме скрипичного ключа часто используется так называемый *басовый ключ*, который обводит на нотном стане ноту "фа" (F) малой октавы:

14 Попробуйте запрограммировать ноту "ми" первой октавы на Arduino и пропеть её, стараясь попадать под частоту проигрываемой ноты: "Ми-ми-ми-ми..." И кто после этого будет говорить, что изучение программирования и музыкальной теории является скучным занятием?..



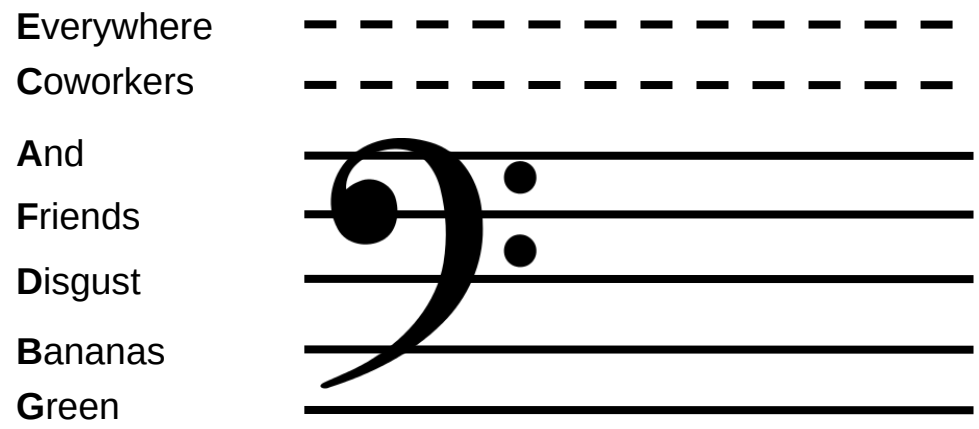
Вот она, нота "фа"! На фортепиано обычно басовый ключ ставится перед партией для левой руки, а скрипичный ключ -- перед партией для правой руки.

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

Если читать слова снизу вверх, то образуется фраза "**E**ating **G**reen **B**ananas **D**isgusts **F**riends **A**nd **C**oworkers" ("Поедание зелёных бананов вызывает отвращение у друзей и коллег по работе"). Первая буква слова кодирует ноту в научной нотации (не забывайте, что нота "B" также может обозначаться "H".) Вверху мы, помимо основных пяти линеек, подрисовали ещё две дополнительных линии, выделив их пунктиром. На нотном стане дополнительные линии сверху и снизу добавляются, если не хватает основных линий для записи композиции.

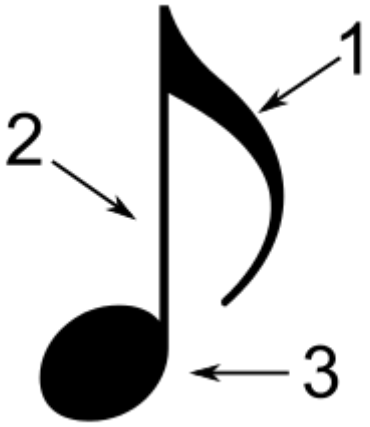
Мнемоника кодирует только основные линии нотного стана, однако зная их, мы можем понять, какие ноты находятся между линиями.

Тоже самое, только для басового ключа:



Если прочитать снизу вверх, то получится фраза "**G**reen **B**ananas **D**isgust **F**riends **A**nd **C**oworkers **E**verywhere" ("Зелёные бананы вызывают отвращение у друзей и коллег по работе везде".)

Теперь поговорим про сами ноты (те самые закорючки, которые нарисованы поверх линеек.) По положению жирной точки (или кружочка) закорючки можно узнать, какая это нота. К слову, жирная точка называется *головкой ноты*:

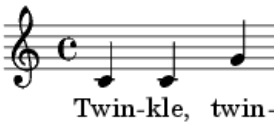


На изображении выше цифрами указаны:

1. Флажок
2. Штиль
3. Головка

Теперь, если мы посмотрим на ноты для мелодии "Twinkle, Twinkle, Little Star", то третья нота (считая слева от верхнего левого угла нотной записи) -- это "соль" первой

октавы (G4), поскольку её головка находится как раз на линии, обведённой скрипичным ключом:



Таким образом по положению нот на нотном стане можно понять, какие ноты надо играть. Кроме того, графическое изображение нот даёт нам ещё одну очень важную информацию -- как долго конкретная нота должна звучать.

02.09.02. Длительности нот

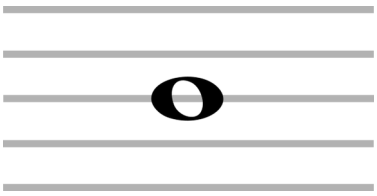
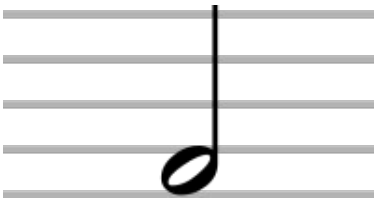
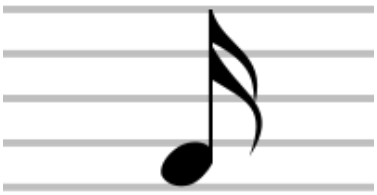
В музыкальной теории у каждой ноты есть своя длительность. Длительность нот не задаётся в секундах, миллисекундах или микросекундах -- вместо этого, длительность нот представляет собой некоторое отношение одной ноты к другой. Есть целые ноты, половинки, четвертинки, восьмушки и т.д.

> Отношения между нотами строятся довольно просто. Допустим, целая нота равна по длительности 4-м секундам. Тогда половинная нота будет длиться половину времени от целой ноты, т.е. 2 секунды, четвертная -- 1 секунду и т.д.

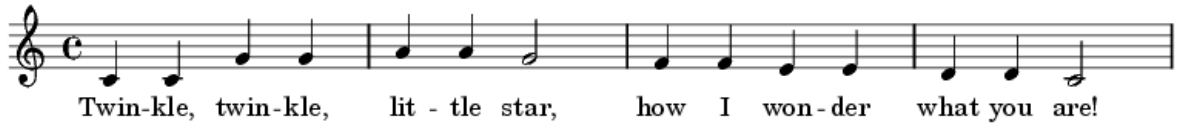
В нотной записи длительности нот можно узнать по их начертанию -- задумывались ли вы когда-нибудь, почему некоторые ноты пишутся с "флажком", а другие без? Или почему некоторые ноты закрашены полностью, а другие представляют из себя кружочки? Так вот, по этим различиям можно определить длительность ноты.

Ниже в таблице приведены некоторые распространённые длительности нот с их обозначениями.[15]

15 См статью "Длительность (музыка)" в Википедии для описания всех длительностей.

Обозначение	Название	Длительность
	Целая	1
	Половинная	$\frac{1}{2}$
	Четвертная	$\frac{1}{4}$
	Восьмая	$\frac{1}{8}$
	Шестнадцатая	$\frac{1}{16}$

Для более "живого" исполнения наших мелодий следует правильно рассчитывать длительности для каждой отдельной ноты. В нотной записи нашей мелодии можно увидеть два вида нот: половинные (с незакрашенными головками) и четвертинные (с закрашенными головками):



Видно, что каждая седьмая нота является половинной.

Доработаем нашу программу, чтобы учитывать длительности нот. Первое, что мы должны сделать -- задать длительность целой ноты, от которой мы будем в дальнейшем высчитывать половинки и четвертинки. Поменяем значение константы `NOTE_LENGTH` (или объявим её, если ещё не сделали это):

```
const long NOTE_LENGTH = 4000000; // 4 секунды в микросекундах
```

Далее, после массива `melody` (который мы рассматривали в разделе "Использование числового массива для программирования мелодии"), объявим ещё один массив -- такого же размера, как и `melody` -- и назовём его `length`:

```
long length[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,
}
```

Данный массив хранит отношение длительности каждой из нот к длительности целой ноты `NOTE_LENGTH`. При этом, элемент массива `length` с индексом N задаёт длительность ноты из массива `melody` с индексом N.

Например, элемент массива `length[0]` задаёт длительность ноты `melody[0]` в ¼ целой ноты (четвертинку.)

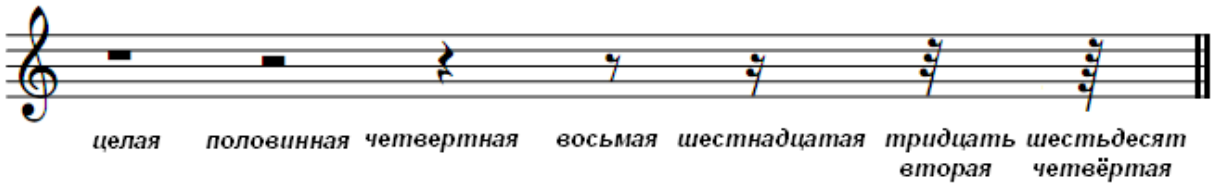
Теперь, при воспроизведении каждой ноты мелодии мы можем рассчитать её длину, поделив длину целой ноты `NOTE_LENGTH` на значение из массива `length`:

```
void loop() {
```

```
      for (int note_idx = 0; note_idx < 28; ++note_idx) {
            play_tone(SPEAKER_PIN, melody[note_idx],
                        NOTE_LENGTH / length[note_idx]);
      }
}
```

02.09.03. Паузы между нотами

Есть ещё один момент, на котором мы до текущего момента не заостряли внимание -- паузы в произведении. Правильные паузы также важны, как и сами ноты -- с ними произведение звучит живее. В нотной записи паузы отмечаются специальными значками:



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

Хотя в произведении "Twinkle, Twinkle, Little Star" явно пауз не выделено, каждая нота должна заканчиваться перед тем, как начнётся следующая. Если мы не будем учитывать это, то одинаковые ноты, следующие друг за другом, будут сливаться в одну длинную ноту и произведение будет звучать странно. Чтобы это исправить, попробуйте добавить небольшие паузы (порядка 100 мс) между нотами. Это можно сделать с помощью обычных задержек.

Однако во многих произведениях вы можете встретить явно указанные паузы между нотами. Как с ними быть? Идеальным вариантом была бы возможность добавления пауз к массиву нот. Этого можно достичь, добавив к ноту специальную константу, определяющую паузу. Назовём эту константу R (от англ. *Rest* -- покой.) Данная константа, как и другие ноты, должна иметь какое-то значение -- длину волны -- чтобы функция `play_tone` могла "воспроизвести" паузу. Во время паузы же нам

необходима тишина. Этого можно достичь, задав для паузы очень большую длину волны -- например, половину секунды:

```
const long R = 1000000 / 2; // пауза
```

Таким образом, мы задали частоту для паузы в 2 Гц, что для человека не будет слышно. С другой стороны, если мы добавим эту константу в массив нот и зададим для неё длительность в массиве `length` (как для обычной ноты), то мы получим паузу нужной длины.

02.09.04. Задачи

1. Попробуйте добавить небольшие паузы (порядка 100 мс) между нотами при воспроизведении "Twinkle, Twinkle, Little Star", добавить задержку в цикл, воспроизводящий ноты. Поменяйте значение задержки таким образом, чтобы произведение лучше звучало (на ваш взгляд -- или, лучше сказать, слух.)
2. Добавьте в программу поддержку добавления пауз нужной длины между нотами, как было описано в разделе выше.
3. Измените код таким образом, чтобы подключённый потенциометр влиял не только на длительность нот, но и пауз.
4. Найдите пример мелодии с явно указанными паузами и запрограммируйте эту мелодию.

02.10 ЖК-дисплей

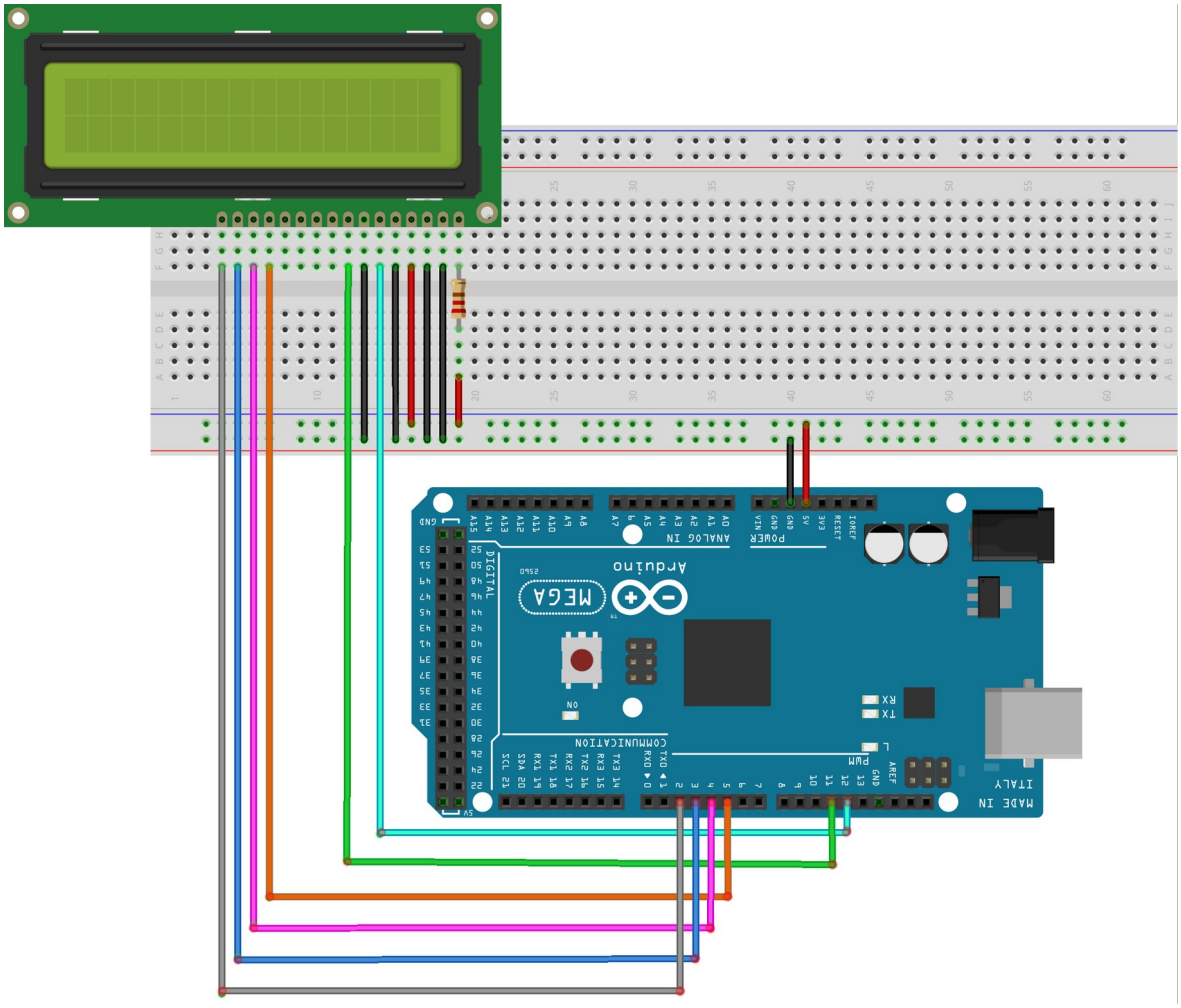
Жидкокристаллический дисплей (ЖК-дисплей) позволяет выводить информацию разного рода, что может быть полезно в проектах, где требуется взаимодействие с пользователем.

Один из распространённых видов дисплеев -- 16 столбцов на 2 строки. Это значит, что на него можно вывести две строки текста, в каждой по 16 символов. Хотя дисплей оперирует символами, среди них встречаются не только буквы, но и различные изображения. Кроме того, мы можем создавать собственные символы и использовать их для отрисовки чего-либо (например, создания анимации.)

Описание дисплея: http://www.melt.aha.ru/pdf/mt-16s2h.pdf

Список символов: https://static.chipdip.ru/lib/243/DOC000243344.pdf

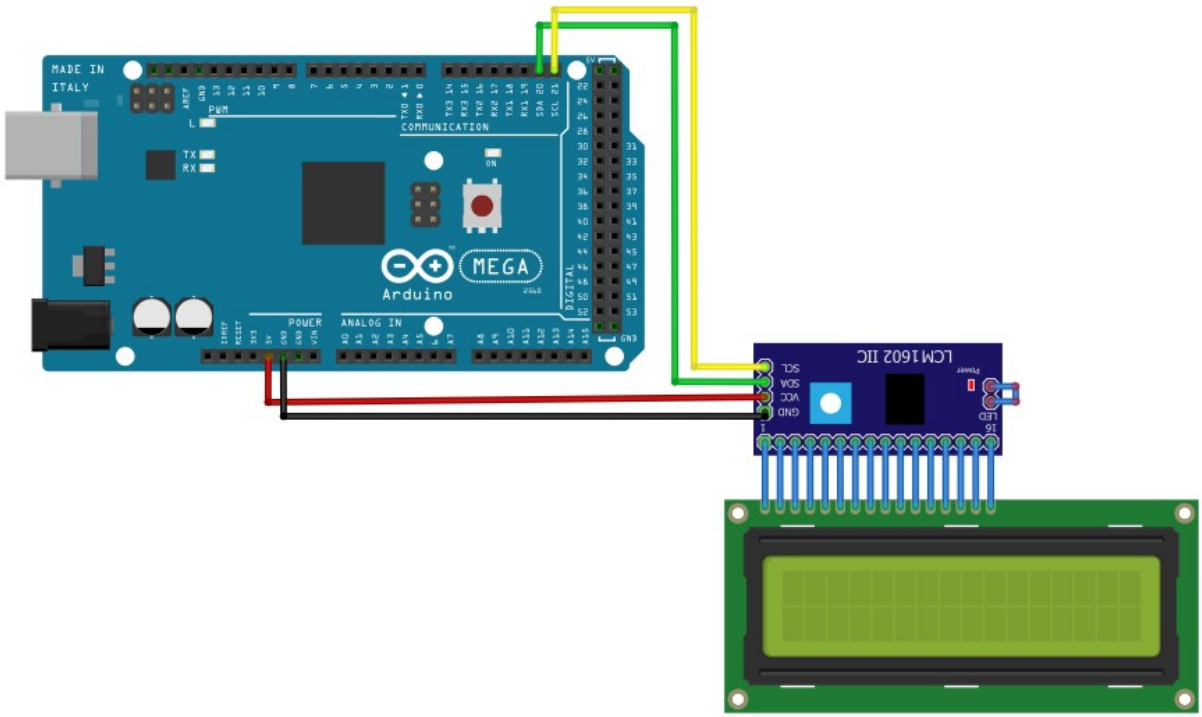
Ниже представлена схема параллельного подключения ЖК-дисплея к платформе Arduino Mega 2560.



- Чёрные провода подключены к GND, т.е к земле (минусу);

- Красные провода подключены к 5V (к плюсу). Обратите внимание, что одни из них подключен через резистор -- это подключение светодиодной подсветки дисплея. Резистор должен быть не меньше, чем 200 Ом;
- Голубой провод подключен к 12-у цифровому порту arduino;
- Зелёный провод подключен к 11-у цифровому порту arduino;
- Оранжевый провод подключен к 5-у цифровому порту arduino;
- Розовый провод подключен к 4-у цифровому порту arduino;
- Синий провод подключен к 3-у цифровому порту arduino;
- Серый провод подключен ко 2-у цифровому порту arduino;

При наличии I2C-модуля можно сократить количество подключаемых проводов до 4-х. Схема подключения с использованием I2C:



- VCC подключён к 5 В порту
- GND к земле
- SCL к 21 порту
- SDA к 20 порту

02.10.01 Вывод информации на дисплей

Перед тем, как начать работать с ЖК-дисплеем, его нужно настроить (точнее настроить Arduino на работу с ним). В случае параллельного подключения нужно использовать библиотеку `LiquidCrystal.h`. В начале программы пишем её подключение:

```
#include <LiquidCrystal.h>
```

Затем, создаём переменную lcd, которая будет в себе хранить объект, позволяющий работать с дисплеем:

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
```

В функции `setup` мы должны написать следующее:

```
lcd.begin(16, 2);
```

Где 16 - количество столбцов, а 2 - количество строк на дисплее.
При подключении при помощи I2C используется библиотека LiquidCrystal_I2C.h.
Подключение библиотеки:

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

Создание объекта для обращения к дисплею:

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

Аргументы функции:

- 0x27 адрес устройства для I2C
- 16, 2 - размеры дисплея

Аналог функции begin, системная функция инициализации дисплея

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

Подсветка дисплея

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

Чтобы вывести на ЖК-дисплей нужную нам информацию, нам необходимо установить курсор в желаемую позицию (можно привести аналогию с текстовым редактором -- чтобы вставить текст в документ, мы сначала должны установить курсор в нужное

место, затем добавить новый текст), для этого в функции `loop` мы можем использовать функцию `setCursor`:

```
lcd.setCursor (1, 0);
```

где 1 - номер столбца, а 0 - номер строки.

Теперь для вывода информации на дисплей в нашей программе нужно написать:

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

где в скобках пишется всё то, что мы хотели увидеть на ЖК-дисплее. Строки символов пишутся внутри двойных кавычек (“ ”), а переменные и константы -- без кавычек.

Для очистки экрана можно воспользоваться командой

```
lcd.clear();
```

02.10.02 Вывод кириллицы

Каждому символу соответствует определенный код и при компиляции программы, если строка содержит кириллицу, она будет конвертирована в коды по таблице utf-8, ср-1251 или в зависимости от настроек компилятора. Экран, в свою очередь, ожидает увидеть данные в своей кодировке. Так, например, букве “Я” соответствует код B1 в шестнадцатеричной системе. Чтобы передать на экран строку, например, “Яndex”, необходимо в явном виде с помощью последовательности \x## встроить в строку код символа. В случае с буквой “я”, данная последовательность будет выглядеть так: \xB1, а вывести строку “Яndex” на дисплей можно таким образом:

```
Lcd.println (“\xB1ndex”);
```

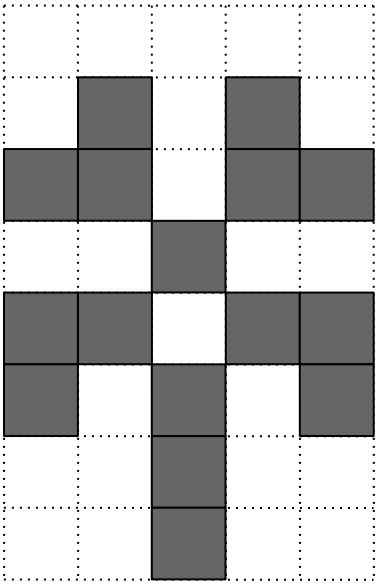
02.10.03 Примеры символов для отрисовки

ЖК-дисплей имеет свою память, в которой хранится таблица символов. В этой таблице из общего числа ячеек 8 выделено для создания пользовательских, нестандартных символов.

Если в программе необходимо использовать нестандартные символы можно создать свои собственные!

Для этого вам потребуется написать массив, к примеру нужно нарисовать “цветок/пальму”, изображенную на рисунке:

К примеру, мы захотели описать новый символ в виде пальмы:



Для этого нужно завести переменную типа char для хранения кода нового символа и массив, в котором указывается, как должна выглядеть наша пальма:

```
const char TREE = 0;
byte tree[8] = {
   0b00000,
   0b01010,
   0b11011,
   0b00100,
   0b11011,
   0b01110,
   0b00100,
   0b00100
};
```

Если внимательно посмотреть на массив, то можно увидеть очертание пальмы.

За тем внутри функции setup, после вызова функции begin (или init), создаём наш новый символ и присваиваем номер:

```
lcd.createChar(TREE, tree);
```

Где **TREE** -- номер нового символа в кодовой таблице, **tree** -- имя массива, где нарисован символ.

Для того чтобы вывести новый символ необходимо написать:

```
lcd.print( TREE );
```

Дополнительные символы можно нарисовать, используя следующий сервис:
https://omerk.github.io/lcdchargen/

Теперь немного о новых для нас типах данных.
byte это тот же int, но с меньшим диапазоном значений - 0..255.
Переменная типа char рассчитана на хранение одного символа. Имеет диапазон значений от -126 до 127.
Объявление char:

```
char chr = 'k';
```

Так мы объявили переменную chr со значением в виде символа 'k'. Обратите внимание, что присваиваемое значение указывается в одинарных кавычках.

Если посмотреть на то, как мы выводили текст на дисплей, то можно заметить, что текст указывается в двойных кавычках:

```
lcd.print("Hello");
```

Указанное в двойных кавычках сообщение - это ещё один новый тип данных - String. Такие переменные нужны для хранения строк текста.

```
String message = "Hello!";
lcd.print(message);
```

Интересная особенность String в том, что это, на самом деле, массив переменных типа char, а это значит, что мы можем обращаться к любому символу строки по его индексу в массиве:

```
String message = “Hello!”;
        char FirstCharacter = message[0]; //FirstCharacter = ‘H’
        //…
    if(FirstCharacter == ‘H’){
    //…
    }
```

Теперь, зная это, мы можем сделать, например, красивый эффект вывода строки по буквам.

Для этого напишем функцию, принимающую строку и выводящую её по буквам:

```
void Message(String msg) {
//Проходимся циклом по всей строке с помощью функции length(),
//возвращающей длину строки
  for (int i = 0; i < msg.length(); i++) {
    lcd.setCursor(i, 0); //Перемещаем курсор
    lcd.print(msg[i]);   //Печатаем один символ
    delay(100);
  }
  delay(500);
//Стираем выведенное сообщение справа налево
  for (int i = msg.length(); i >= 0; i--) {
    lcd.setCursor(i, 0);
    lcd.print(" ");
    delay(100);
  }
}
```

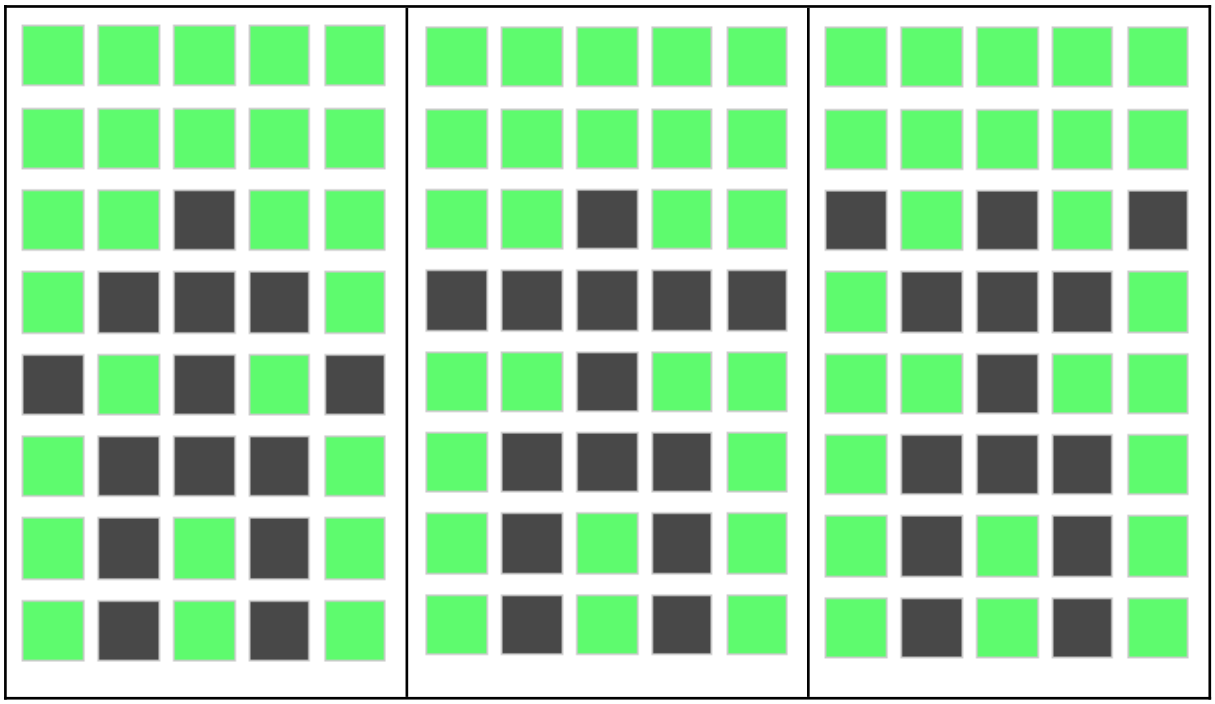
Теперь эту функцию можно вызвать, например, в setup:

```
Message(“Hello there”);
```

02.10.04. Анимация

Для создания анимации необходимо во-первых иметь нарисованные "кадры" и во-вторых, сменять их по прошествии времени.

Например, пусть у нас будут следующие кадры анимации:



Если мы будем сменять эти кадры по-очереди, то будет казаться, что человечек делает зарядку.

Создадим три кадра анимации в виде новых символов:

```
const char MAN1 = 0;
byte man1Img[8] = {
  //Код символа…
};
const char MAN2 = 1;
byte man2Img[8] = {
  //Код символа…
};
const char MAN3 = 2;
```

```
byte man3Img[8] = {
  //Код символа…
};
```

Для более удобного доступа к кадрам анимации, их все можно хранить в массиве:

```
char ManAnimation[4] = {MAN1,MAN2,MAN3,MAN2};
```

Также добавим счётчик кадров анимации:

```
byte ManAnimationCounter = 0;
```

В setup() пропишем создание новых символов

```
  lcd.createChar(MAN1,man1Img);
  lcd.createChar(MAN2,man2Img);
  lcd.createChar(MAN3,man3Img);
```

Затем, в loop() выбираем нужный символ для отрисовки с помощью счётчика кадров

```
  lcd.print(ManAnimation[ManAnimationCounter]);
  ManAnimationCounter++;
  if(ManAnimationCounter > 3){
    ManAnimationCounter = 0;
  }
```

02.11. Немного о заблуждениях

Обратите внимание на переменную счётчика кадров из прошлой главы, а именно, что тип этой переменной - byte, а не int:

```
byte ManAnimationCounter = 0;
```

Планировалось, что это сэкономит память микроконтроллера, ведь int занимает 2 байта, а byte - только один.

Микроконтроллер ATmega2560, который использует наша плата, имеет 8192 байта оперативной памяти, и выходит, что один байт экономии погоды не сделает. В этом ничего плохого нет, но в каком-нибудь большом проекте подобное может иногда сбивать с толку при попытке понять, как работает программа, потому что byte принято использовать при передаче данных по сети, или, например, чтения файлов.

При программировании микроконтроллеров, в условиях ограниченных ресурсов, абсолютно нормально уменьшать объем потребляемой памяти, но не стоит пытаться оптимизировать программу за счёт замены типов отдельных переменных на другие. В этом есть смысл, если у вас есть какой-нибудь двумерный массив размером 50x50, в котором значения ячеек не превышают диапазон byte, а используется, например, int. В этом случае замена int на byte обеспечит значительную экономию. Вывод напрашивается сам собой - byte и другие типы, меньшие чем int, нужно использовать только там, где они необходимы. Также стоит обращать внимание на наличие повторяющегося кода. Например, одна функция прописана множество раз, когда можно обойтись одним циклом, который будет вызывать эту функцию, что тоже может уменьшить вес программы.

Иногда проскальзывает такое мнение, что чем короче имя переменной, тем меньше занимает места программа, или, вообще, программа быстрее выполняется. Но такие заявления в корне неверны. Дело в том, что имя переменной - это понятие абстрактное, это уровень языка, а не машинного кода. После компиляции от переменной остаётся только адрес на ячейку памяти, в которой хранится её значение. Главное при именовании переменных - это осмысленность, нужно, чтобы название отражало предназначение переменной.

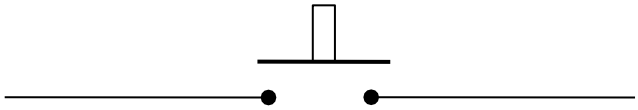
02.12. Прерывания

02.12.01. Подключение кнопки

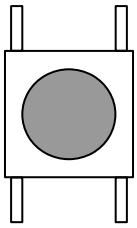
Кнопки позволяют организовать взаимодействие с пользователем, позволяя влиять на работу микропроцессорной системы: с помощью них можно переключать режимы работы системы, выбирать пункты из какого-либо экранного меню, управлять роботом или же виртуальным персонажем.

Кнопки бывают различных типов; мы рассмотрим так называемые *тактовые кнопки*, которые "не запоминают" состояние нажатия --- при нажатии на такую кнопку контакт замыкается, при отпускании кнопки он размыкается.

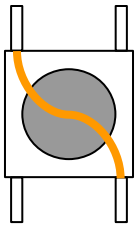
На электрических схемах тактовая кнопка обозначается следующим образом:



Схематичное изображение одной из распространённых конструкций кнопок представлено ниже:



Как можно заметить, данная кнопка имеет 4 вывода, хотя для "обычной" кнопки достаточно 2. "Лишние" контакты на данной кнопке нужны для того, чтобы одновременно замыкать две цепи одним нажатием кнопки. Контакты замыкаются попарно крест-накрест -- например, когда кнопка нажата, то левый верхний контакт замыкается с правым нижним:



Изучение работы с кнопкой лучше начать с её подключения -- пример подключения кнопки к платформе Arduino Mega 2560 приведён ниже. Для подключения кнопки можно использовать любой порт Arduino (даже аналоговый), но мы для начала рекомендуем использовать цифровые порты 18 и 19. Почему именно их? Для этого есть две причины: во-первых, эти порты, как правило, в простых проектах ничем не

заняты; во-вторых, использование этих портов немного упростит использование кнопки в дальнейшем.

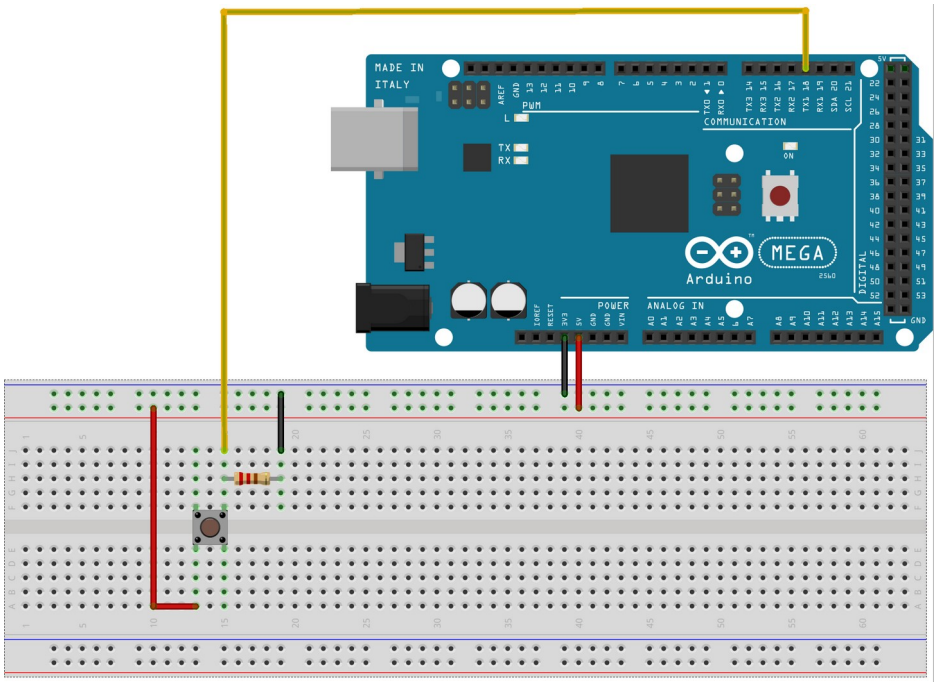


Схема подключения кнопки.

- Желтый провод идёт с 18ой ноги Arduino и попадает на левую верхнюю ногу кнопки **и** через резистор идёт на минус
- Чёрные провода соединяются с выходим **GND** (минусом)
- Красные провода соединяются с выходом **5V** (плюсом)

Примечание: не забывайте самое главное - резистор.
Данное включение резистора в цепь называется подтяжкой к земле. Такой резистор обеспечивает исправную работу кнопки.

Как только кнопка подключена, можно перейти к программированию обработки нажатия кнопки.

02.12.02. Считывание и обработка состояния кнопки

Есть как минимум два способа работы с кнопкой: простой, но во многих случаях неудобный, и более сложный (но и более универсальный и удобный) способ.

Начнём с простого. Цифровые порты могут работать в двух режимах: на вывод информации (`OUTPUT`) и на ввод данных (`INPUT`). Как вы помните, для работы со

светодиодами и динамиком мы использовали режим OUTPUT -- в этом режиме микроконтроллер может выводить сигнал на порт, управляя таким образом чем-либо в реальном мире. Теперь же нам потребуется настроить порт, к которому подключена кнопка, на режим INPUT -- для считывания сигнала от кнопки. Используя режим INPUT, можно считывать сигнал не только от кнопки, но и от других внешних устройств.

Проверять кнопку на нажатие можно считывая напряжение на порту используя функцию digitalRead(pin), которая возвращает 1 или 0 в зависимости от напряжения:

```
int button1 = digitalRead(buttonPin);
if(button1 == 1){
        //Кнопка нажата
}
```

Или же, можно использовать прерывания.

02.12.03. Прерывания

Бывают моменты, когда при работе программы она должна реагировать на определенное событие извне. В таких случаях платформа Arduino при определенных событиях на какой-то момент “отвлекается” от выполнения основной программы и приступает к выполнению действий (сценария) которые должны быть выполнены при условии возникновения определенного события (прерывания).

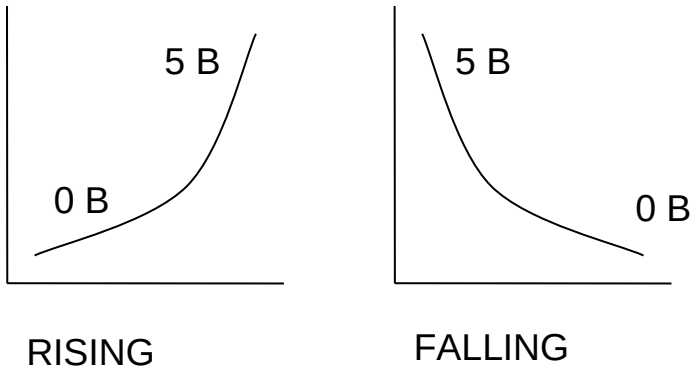
Чтобы реализовать обработку прерываний на платформе Arduino нужно в функции `void setup` вызвать функцию - `attachInterrupt`.

Пример:

```
attachInterrupt(digitalPinToInterrupt(2), handleButton, FALLING);
```

`digitalPinToInterrupt()` - здесь мы задаем порт на котором будет прерывание,
`handleButton` - обработчик прерывания, это функция которая будет работать только при прерывании (при событии, на которое должна реагировать программа),
`FALLING` - событие за счёт которого срабатывает работает прерывание.

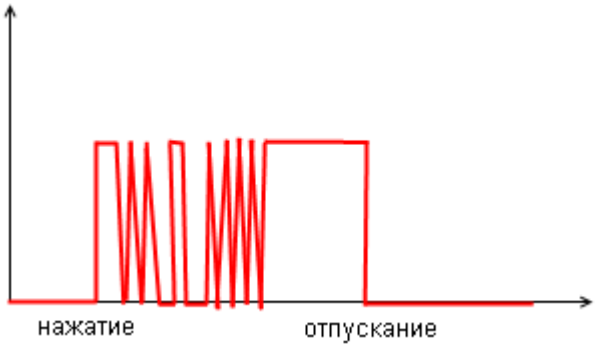
Примеры событий:



Теперь, если всё сделано правильно, обработчик прерывания (та функция, которая выполняется при прерывании) будет срабатывать в нужный нам момент (событие).

02.12.04. Дребезг контактов

Возможно вы могли заметить, что после единичного нажатия кнопка передавала сигнал нажатия дважды или якобы не нажималась вовсе, такое явление называется *дребезгом*. Дребезг возникает в результате нестабильного переключения. В этом случае вместо четкого переключения появляются случайные неконтролируемые многократные замыкания и размыкания контактов. Все это происходит в короткий момент переключения, длятся такие замыкания приблизительно 40—100 миллисекунд.



Даже этого времени хватает, чтобы Arduino успела среагировать на эти хаотичные замыкания.

Существует два способа борьбы с дребезгом:

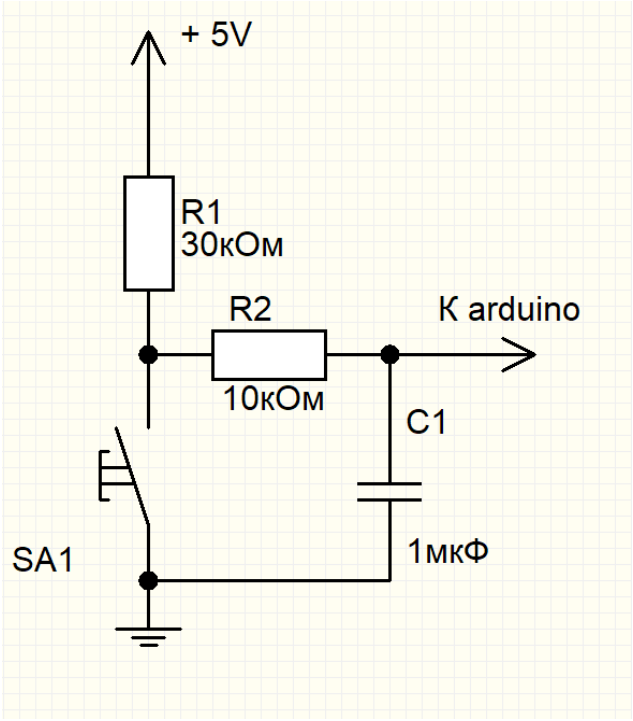
1. Аппаратное решение
2. Программное решение

02.12.05. Аппаратное решение

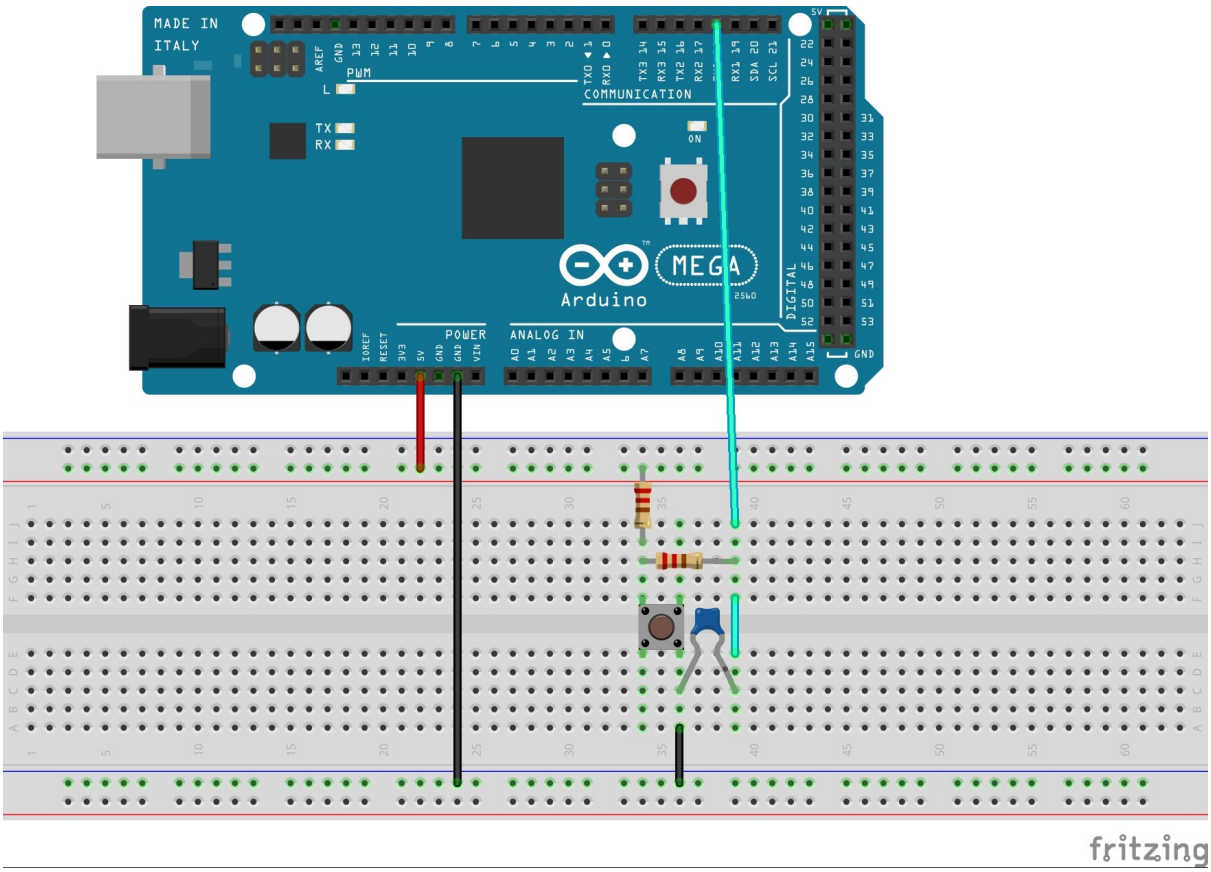
Рассмотрим сначала аппаратное решение. В данном способе устранение дребезга достигается включением дополнительных деталей в схему подключения кнопки (например, на макетной плате).

Работа дополнительных деталей заключается в том, что колебания (дребезг) сглаживаются собранным на макетной плате фильтром.

Электрическая схема простейшего RC-фильтра :



Подключение к Arduino:



- Голубой провод идёт с 18ого входа arduino и идёт на резистор номиналом в 10 кОм и на конденсатор, как указано в электрической схеме сверху.
- Чёрные провода соединяются с выходом gnd (минусом)
- Красные провода соединяются с выходом 5V (плюсом)

02.12.06. Программное решение

Теперь рассмотрим программное решение проблемы дребезга. В данном методе для избежания дребезга контактов программно отключается реакция Arduino на повторное нажатие кнопки в течении определённого времени. Таким образом, когда на Arduino с нажатием кнопки появится первое колебание, она перестанет воспринимать какие-либо колебания на этом выводе в течении определённого времени.

02.12.07. Опрос кнопки

Пример использования кнопки для управления работы светодиода **без** использования прерывания приведён ниже.

В функции void `setup` мы должны установить режимы работы выводов:

```
      pinMode(13, OUTPUT);
      pinMode(2, INPUT);
```

Затем нам понадобятся переменные -- чтобы хранить состояние кнопки:

```
      byte button = 0;    // Состояние кнопки
      byte oldbutton = 1; // Последнее состояние кнопки, для
                          // исключения ложных переключений
```

Теперь напишем в функцию `loop` проверку и реализацию управления светодиодом:

```
if (digitalRead(2) == HIGH) { // Кнопка нажата?
     delay(100); //Защита от дребезга
    //Проверка, что состояние кнопки изменилось
    if (button == oldbutton) {
        button = ! button;
    }
} else {
    oldbutton = button;
}
digitalWrite(13, button);  //Переключаем светодиод
```

02.12.08. Прерывания

Пример программного решения дребезга кнопки используя прерывания:

```
const int BUTTON_PIN = 18;
void setup() {
    // Установка прерывания
    pinMode(BUTTON_PIN, INPUT_PULLUP); // button
    attachInterrupt(digitalPinToInterrupt(BUTTON_PIN),
                    butISR, RISING);
}
```

```
// инициализация переменных
bool is_button_disabled = false;
unsigned long button_timer = 0;
int is_led_enabled = false;
```

Создаём функцию-обработчик прерываний butISR которая будет принимать только 1 колебание от нажатой кнопки:

```
void butISR(){
    // Не реагируем на кнопку, если не прошло 500 мс
    // с прошлого нажатия
    if (! is_button_disabled) {
        is_button_disabled = true;
        button_timer = millis();
        is_led_enabled = ! is_led_enabled;
  }
  check_button_timeout();
}
```

Также можно написать функцию которая будет проверять время с прошлого нажатия и активацию кнопки:

```
void check_button_timeout() {
    if (is_button_disabled) {
        if ((millis() - button_timer) > 500) {
            is_button_disabled = false;
            button_timer = 0;
        }
    }
}
```

Тема 3. Создание простых 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 - помощь 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.

03.01. Игра 1

Геймплей нашей первой игры будет заключаться в следующем:
Игрок появляется в помещении, заполненном случайным образом врагами и стенами, также, должен быть ключ, взяв который, игрок может выйти из помещения и получить сообщение о выигрыше.

03.01.01. Создание игровой карты

Создадим игровую карту. Хранить её будем в двумерном массиве. Двумерный массив представляет собой матрицу, состоящую из строк и столбцов.
Объявление двумерного массива 20 на 4, играющего роль игровой карты, выглядит следующим образом:

```
char game_map[20][4];
```

Попробуем разместить на нашей игровой карте, например, стену:

```
game_map[1][2] = 255;
```

Так мы заполнили ячейку [1 ; 2] массива игровой карты кодом символа, похожего на стену, из кодовой таблицы ЖК-дисплея.

Теперь напишем функцию отрисовки карты:

```
void RenderMap(){
  for (int y = 0; y < 4; y++) {     //Проходимся по массиву
    for (int x = 0; x < 20; x++) {
      lcd.setCursor(x,y);           //В соответствующей точке
      lcd.print(game_map[x][y]);    //Печатаем символ из массива
    }
  }
}
```

Вызывать эту функцию будем в setup(). Если выполним написанный код, то мы увидим, что на игровом поле появился символ, который мы указывали! Код этого символа можно записать как константу:

```
const char WALL 255;
```

Теперь мы можем заполнить ячейку карты уже таким образом:

```
game_map[1][2] = WALL;
```

Мы можем заполнить игровую карту вручную при объявлении:

```
game_map[20][4] = {0,0,255,255}, //1 строка
                  {0,0,255,0},   //2 строка
                  {255,0,0,0},   //3 строка
                  //и так далее
```

Но, это долго и неудобно. Проще автоматизировать этот процесс. Будем заполнять игровую карту случайным образом, в этом нам поможет генератор псевдослучайных чисел - функция random(min,max), где min и max - нижняя и верхняя граница случайных значений. Пример:

```
int randValue = random(0,15);
//randValue получит случайное значение от нуля до 14, а не 15.
```

Проходимся по всему массиву:

```
for (int y = 0; y < 4; y++) {
  for (int x = 0; x < 20; x++) {
    if (random(0, 10) < 2) { // Если рандом возвращает число < 2,
      game_map[x][y] = WALL;// То заполняем клетку символом стены
    } else {                    //Иначе
      game_map[x][y] = ' '; //Заполняем пробелом
//Обратите внимание! Массив состоит из переменных char, значит
//пробел пишем в одинарных кавычках!
    }
  }
 }
```

Можно оформить этот код в отдельную функцию GenerateMap():

```
void GenerateMap(){
  //…
}
```

Прекрасно, стены расположились в случайном порядке! Но это, как вы уже могли заметить, при повторных запусках карта не меняется. Причина - принцип работы random(). Случайное число генерируется на основе параметра, называемого зерном. Если зерно не изменять, то мы будем получать одинаковый набор чисел. Используя функцию randomSeed() мы можем решить эту проблему:

```
void setup(){
  //…
  randomSeed(analogRead(0));
  GenerateMap();
}
```

Из-за шума на аналоговом порту, считываемое значение всегда будет разным, благодаря чему, зерно будет меняться и мы будем получать разные последовательности случайных чисел. Теперь при каждом запуске нас встречает новая карта!

03.01.02. Добавление игрока

Напишем функцию для опроса кнопок:

```
        int buttonUP;
        int buttonDOWN;
        int buttonLEFT;
        int buttonRIGHT;

        void InputController(){
          buttonUP = digitalRead(BUTTON_UP);
          buttonDOWN = digitalRead(BUTTON_DOWN);
          buttonLEFT = digitalRead(BUTTON_LEFT);
          buttonRIGHT = digitalRead(BUTTON_RIGHT);
}
```

Приступим к добавлению игрока. Объявим переменные для координат игрока.

```
        int playerX, playerY;  //Текущие координаты игрока
        int prevplayerX,prevplayerY; //Предыдущие координаты
```

Для чего нужны предыдущие координаты - скоро станет ясно.

Следующая в очереди - функция движения игрока:

```
void PlayerController(){
  prevplayerX = playerX; //Присваиваем предыдущим координатам текущие
  prevplayerY = playerY; //до того, как изменятся текущие
  int dirX = 0; //Направление по X
  int dirY = 0; // Направление по Y
  if(buttonUP == LOW){ //Нажата кнопка вверх - направление по Y
меняется
    dirY = -1; // Дальше - по аналогии
  } else if(buttonDOWN == LOW){
    dirY = 1;
  } else {
    dirY = 0;
  }
  if(buttonLEFT == LOW){
```

```
    dirX = -1;
  } else if(buttonRIGHT == LOW){
    dirX = 1;
  } else {
    dirX = 0;
  }
//Если текущая позиция игрока + направление не заходит за границы,
//то изменяем координаты
  if(playerX + dirX >= 0 && playerX + dirX < 20){
    playerX += dirX;
  }
  if(playerY + dirY >= 0 && playerY + dirY < 4){
    playerY += dirY;
  }
}
```

Оператор “&&” - это логическое “И”. Условие истинно, только в том случае, если все выражения будут истинными.

Напишем функцию отрисовки персонажа:

```
          void RenderPlayer(){
            lcd.setCursor(playerX,playerY); //Установка курсора
            lcd.print(PLAYER); //Печать символа игрока
          //Проверяем, совпадают ли предыдущие координаты с текущими
            if(prevplayerX != playerX || prevplayerY != playerY){
              lcd.setCursor(prevplayerX,prevplayerY);
              lcd.print(" ");//Предыдущую позицию игрока закрашиваем
                            // пробелом
            }
}
```

Мы могли бы просто рисовать игрока по его координатам, но, т.к. мы не перерисовываем карту каждую итерацию, символ персонажа в предыдущей точке не будет стираться, т.е. при движении будет печататься множество символов персонажа. В этом блоке мы проверяем, совпадают, ли предыдущие координаты с текущими:

```
if(prevplayerX != playerX || prevplayerY != playerY)
```

Оператор “||”обозначает логическое “ИЛИ”. Это значит, что условие истинно, если хотя бы одно выражение является истинным:
(prevplayerX НЕ РАВНО playerX) ИЛИ (prevplayerY НЕ РАВНО player)

Если текущие координаты не совпадают с предыдущими, значит, что персонаж переместился и нужно закрасить пробелом его предыдущую позицию.

Теперь, написанные функции вызываем в цикле loop():

```
void loop() {
  InputController();
  RenderPlayer();
  PlayerController();
  delay(100);
}
```

Посмотрите на то, как аккуратно выглядит наш код. Если бы мы не оформляли код для опроса кнопок, отрисовки игрока и его движений, в отдельные функции, а писали всё подряд в loop(), читать код было бы труднее.

03.01.03. Коллизии персонажа

Итак, если запустим программу и посмотрим на результат. Персонаж перемещается по карте, но он совсем не обращает внимания на стены. Займёмся коллизиями. Давайте подумаем, в чём суть коллизий? Это запрет на перемещение, при условии, что если не пуста клетка, в которую персонаж пытается переместиться.
Будем модифицировать PlayerController() - в условие проверки на заход за границы экрана добавим ещё одно выражение:

```
if(playerX+dirX >= 0 && playerX + dirX < 20 &&
  game_map[playerX][playerY + dirY] == ‘ ’) //Новое выражение
//Можно писать в одну строку, но это не очень красиво
```

Рассмотрим это выражение:

game_map[playerX][playerY + dirY] == ‘ ’

Персонаж пытается переместиться на клетку вверх(или вниз), поэтому мы проверяем, что находится в точке [playerX , playerY + dirY], если это пробел, то блок условия выполнится, и персонаж переместится. Точно так же с перемещением по X.

Функция должна выглядеть следующим образом:

```
void PlayerController(){
  prevplayerX = playerX;
  prevplayerY = playerY;
  int dirX = 0;
  int dirY = 0;
  if(buttonUP == LOW){
    dirY = -1;
  } else if(buttonDOWN == LOW){
    dirY = 1;
  } else {
    dirY = 0;
  }
  if(buttonLEFT == LOW){
    dirX = -1;
  } else if(buttonRIGHT == LOW){
    dirX = 1;
  } else {
    dirX = 0;
  }
  if(playerX + dirX >= 0 && playerX + dirX < 20 && game_map[playerX
+ dirX][playerY] == ' '){
    playerX += dirX;
  }



  if(playerY + dirY >= 0 && playerY + dirY < 4 && game_map[playerX]
```

```
      [playerY + dirY] == ' '){
         playerY += dirY;
       }
      }
```

03.01.04. Новая генерация карты

Коллизии работают отлично! Но, как уже можно было заметить, иногда стены блокируют проход в другую часть карты. Нужно модифицировать функцию генерации карты GenerateMap(). Попробуем размещать на карте фиксированное количество стен длиной в 2 клетки. Первый вариант генерации можно удалить.
Сначала пройдёмся по карте и разместим везде пробелы:

```
      for (int y = 0; y < 4; y++) {
       for (int x = 0; x < 20; x++) {
          game_map[x][y] = ' ';
       }
  }
```

Дальше - цикл размещения вертикальных стен:

```
       for(int n = 0 ; n < 3; n++){
         randomSeed(analogRead(8)); //Изменяем зерно рандома
         int randX = random(1,20); //Случайные координаты стены
         int randY = random(0,3);
         game_map[randX][randY] = WALL; //Размещаем первую клетку
         game_map[randX][randY + 1] = WALL; //Рядом с ней вторую
   }
```

Цикл размещения горизонтальных стен:

```
for(int n = 0; n < 3; n++){
  randomSeed(analogRead(8)); //Изменяем зерно рандома
```

```
    int randX = random(1,20); //Случайные координаты клетки стены
    int randY = random(0,4);
//Проверяем, нет ли по выбранным координатам других стен
if(game_map[randX][randY] == WALL || game_map[randX+1][randY]== WALL)
{
     if(n > 0) n--;
// Если стены есть-уменьшаем счётчик цикла на единицу
//Затем командой continue сразу переходим к следующей итерации цикла,
//пропустив весь код ниже.
      continue;
    }
    game_map[randX][randY] = WALL;
    game_map[randX + 1][randY] = WALL;
  }
```

В нынешнем виде, генерация карты выглядит приемлемо.

03.01.05. Ключ и дверь

Добавим ключ и дверь. Разместим их случайным образом. Но хранить их в масиве карты не будем. Объявим вне функции переменные:

```
int ExitX,ExitY; // Координаты выхода
int KeyX,KeyY; //Координаты ключа
```

Перейдём в конец функции GenerateMap() и добавим следующее:

```
    randomSeed(analogRead(8));
    ExitX = random(1,20); //Координаты выхода
    ExitY = random(0,4);
    randomSeed(analogRead(8));
    KeyX = random(1,20); //Координаты ключа
    KeyY = random(0,4);
    game_map[ExitX][ExitY] = WALL;//На месте выхода - стена
                                  //Подберём ключ - эту стену убираем
    while(game_map[KeyX][KeyY] != ' '){ //Пока по координатам ключа
```

```
    randomSeed(analogRead(8)); // не будет пробела, меняем координаты
    KeyX = random(1,20);
    KeyY = random(0,4);
  }
```

Напишем новую функцию - RenderEntities(), где будем отрисовывать ключ, выход и в будущем противников, т.е. сущности:

```
void RenderEntites(){
  lcd.setCursor(KeyX,KeyY);
  lcd.print((char)KEY);
  lcd.setCursor(ExitX,ExitY);
  lcd.print((char)EXIT);
}
```

И нужно не забыть добавить её в loop().

Теперь научим персонажа поднимать ключ. В PlayerController() просто сравниваем координаты ключа и персонажа. Если совпадают, то - ключ взят. Но, как же нам описать наличие или отсутствие ключа у персонажа? Здесь на помощь к нам приходит новый тип данных - bool. Переменная типа bool используется как переключатель, имеет два состояния - true или false.
Объявим вне функции переменную для того, чтобы знать, взят ключ или нет:

```
bool IsKeyTaken = false; // Если не присваивать значение, то будет false
```

Итак, в PlayerController() проверяем совпадение координат, меняем IsKeyTaken на true и можно вывести сообщение:

```
 if(playerX == KeyX && playerY == KeyY && IsKeyTaken == false){
 game_map[ExitX][ExitY] = ' '; //Освобождаем путь к выходу
   IsKeyTaken = true;
   Message("Key is taken!");
 }
```

Попробуйте сами придумать, как реализовать функцию вывода сообщения.

После того, как ключ взят, он не должен отрисовываться. Отредактируем RenderEntities():

```
void RenderEntites(){
  if(IsKeyTaken == false){
      lcd.setCursor(KeyX,KeyY);
      lcd.print(KEY);
  }
    lcd.setCursor(ExitX,ExitY);
    lcd.print(EXIT);
}
```

03.01.06. Победа

В PlayerController() допишем проверку на совпадение координат персонажа и выхода. Если есть ключ и координаты совпадают, то выводим сообщение о победе, затем нужно начать игру заново. Для этого вызываем функции генерации и отрисовки карты, перемещаем персонажа в начальную точку и “убираем” ключ:

```
  if(playerX == ExitX && playerY == ExitY && IsKeyTaken){
    Message("Victory!");
    playerX = 0;
    playerY = 0;
    GenerateMap();
    RenderMap();
    IsKeyTaken = false;
  }
```

Можно оформить отдельную функцию - GameRestart(), а в качестве параметра - текст сообщения, которое будет показываться при вызове этой функции.

03.01.07. Добавление противников

Наконец-то у персонажа появилась какая-то цель. Но даётся она ему слишком уж легко. Приступаем к добавлению противников. Пусть их на карте будет трое. Заведём для хранения их координат два массива - первый для координаты по X, второй - для Y:

```
int EnemiesX[3];
int EnemiesY[3];
```

Разместим противников на карте. В GenerateMap() задаём случайные координаты, затем в цикле будем их менять, если в выбранной точке будет стена:

```
for (int i = 0; i < 3; i++) {
  randomSeed(analogRead(8));
  EnemiesX[i] = random(1, 19);
  EnemiesY[i] = random(0, 4);
  while (game_map[EnemiesX[i]][EnemiesY[i]] != ' ') {
    EnemiesX[i] = random(1, 19);
    EnemiesY[i] = random(0, 4);
  }
```

Теперь противников нужно отрисовать. Перейдём в функцию RenderEntities() и добавим следующее:

```
for (int i = 0; i < 3; i++) {
  lcd.setCursor(EnemiesX[i], EnemiesY[i]);
  lcd.print(ENEMY); // По выбранным координатам рисуем символ врага
}
```

Противники появились на карте. Но нужно их как-то оживить. Научим их двигаться. Объявим массивы для хранения направлений движения противников.

```
int EnemiesDirsX[3];
```

```
int EnemiesDirsY[3];
```

Работает это просто - суть в том, чтобы к координатам противников прибавлять значение их направления, положительное или отрицательное.
Напишем функцию EnemyController():

```
void EnemyController() {
  for (int i = 0; i < 3; i++) {
    EnemiesX[i] += EnemiesDirsX[i];
    EnemiesY[i] += EnemiesDirsY[i];
  }
}
```

Массив с направлениями движений можно задать вручную, но зачем нам это делать вручную, когда можно написать код, который будет сам каждый раз задавать случайное значение?
Перейдём в GenerateMap(). Объявим переменную, которой будет присваиваться значение от 0 до 3, затем, в зависимости от присвоенного значения, будут выбираться направления движений. Добавим следующий код:

```
//Значение, в зависимости от которого будет происходить выбор направления
int randomDir = random(0, 4);
//Просто перебираем все варианты
    if(randomDir == 0){
      EnemiesDirsX[i] = 1; //Движение по горизонтали
      EnemiesDirsY[i] = 0;
    } else if(randomDir == 1){
      EnemiesDirsX[i] = 0; //По вертикали
      EnemiesDirsY[i] = 1;
    } else if(randomDir == 2){
      EnemiesDirsX[i] = -1;
      EnemiesDirsY[i] = 0;
    } else if(randomDir == 3){
      EnemiesDirsX[i] = 0;
```

```
    EnemiesDirsY[i] = -1;
  }
}
```

Отлично! Противники научились двигаться! Вот только двигаются они слишком быстро, да через стены, а ещё нужно модифицировать код отрисовки противников. Нужно бы это исправить, но пока посмотрим снова на код для выбора направления движений.
Конструкцию if-else можно написать слегка иначе, если познакомимся с новым условным оператором switch.

03.01.08. Оператор switch

Структура выглядит следующим образом:

```
switch(Переменная){
  case 0: //Если указанная вначале переменная равна 0, то
    // Выполняется действие внутри блока case:
    //...
   break; // Выход из блока switch
   case 230: // Если указанная вначале переменная равна 230, то
    // Выполняется действие внутри блока case:
    //...
   break; // Выход из блока switch
}
```

Перепишем выбор направления движений:

```
 int randomDir = random(0, 4);
//Начало блока. В скобках - переменная, которая будет сравниваться с
//указанными значениями в блоках case
 switch (randomDir) {
```

```
    case 0: //Аналог условия (randomDir == 0)
      EnemiesDirsX[i] = 1;
      EnemiesDirsY[i] = 0;
      break; // Выход из блока switch.
    case 1: //Аналог условия (randomDir == 1)
      EnemiesDirsX[i] = 0;
      EnemiesDirsY[i] = 1;
      break;
    case 2: //Аналог условия (randomDir == 2)
      EnemiesDirsX[i] = -1;
      EnemiesDirsY[i] = 0;
      break;
    case 3: // Аналог условия (randomDir == 3)
      EnemiesDirsX[i] = 0;
      EnemiesDirsY[i] = -1;
      break;
  }
```

switch позиционирует себя как удобная и более производительная замена конструкции if-else. Но в нашем случае, здесь может быть максимум четыре операции сравнения, прироста производительности по сравнению с if-else не будет, поэтому здесь использование switch это личное дело каждого, кому-то может больше нравится синтаксис switch, кому-то больше if-else. Прирост производительности будет в том случае, если операций сравнения будет большое количество, например в цикле на сотни и тысячи итераций. Но пытаться оптимизировать долго выполняющийся код путём замены if-else на switch глупо. Если алгоритм стал выполняться слишком долго, значит нужно перерабатывать сам алгоритм, а не тратить время на подобные мелочи. Также существует такое мнение, что если вы используете большое количество условий для ветвления алгоритма, значит вы что-то делаете неправильно.

03.01.09. Отрисовка противников

Вернёмся к нашей игре. Модифицируем код отрисовки противников:

```
  for (int i = 0; i < 3; i++) {
    //Рисуем врага по его координатам
    lcd.setCursor(EnemiesX[i], EnemiesY[i]);
    lcd.print(ENEMY);
//Клетку, расположенную в противоположной движению стороне,
закрасим
//стеной, если на карте в том месте действительно стена, иначе -
//закрашиваем пробелом
    lcd.setCursor(EnemiesX[i] - EnemiesDirsX[i],
                  EnemiesY[i] - EnemiesDirsY[i]);
    if (game_map[EnemiesX[i] - EnemiesDirsX[i]]
              [EnemiesY[i] - EnemiesDirsY[i]] == WALL) {
      lcd.print(WALL);
    } else {
      lcd.print(' ');
    }
  }
```

03.01.10. Коллизии противников

Научим врагов видеть препятствия. Модифицируем EnemyController() - будем менять направление движения врагов на противоположное, если следующая клетка не пустая или враг стоит на границах экрана:

```
 for (int i = 0; i < 3; i++) {
//Если враг на границе экрана или следующая клетка не пустая
      if (EnemiesX[i] == 0 || EnemiesX[i] == 19 ||
      game_map[EnemiesX[i] + EnemiesDirsX[i]][EnemiesY[i]] != ' ') {
        EnemiesDirsX[i] *= -1; //меняем направление на противоположное
      }
      //По вертикали аналогично
      if (EnemiesY[i] == 0 || EnemiesY[i] == 3 || game_map[EnemiesX[i]]
[EnemiesY[i] + EnemiesDirsY[i]] != ' ') {
```

```
      EnemiesDirsY[i] *= -1;
    }
    EnemiesX[i] += EnemiesDirsX[i];
    EnemiesY[i] += EnemiesDirsY[i];
  }
```

03.01.11. Ограничение скорости противников

Враги бегают из стороны в сторону, что достаточно неплохо, но их скорость уж слишком высока. Заставим их бегать потише. Заведём переменную-таймер, если она будет равна нулю, то разрешаем врагам бежать, если больше нуля - то запрещаем:

```
int EnemiesMoveTimer; //Переменная-таймер
void EnemyController() {
  if (EnemiesMoveTimer == 0) {//Если таймер равен нулю, то двигаем врага
    EnemiesMoveTimer = 3; //Задаём таймеру значение отсчёта
    for (int i = 0; i < 3; i++) {
     //...
     //Код движений врага
     //...
    }
  }
  EnemiesMoveTimer--; //При каждом выполнении этой функции,
}                     // значение таймера уменьшается
```

Аналогичным образом можно замедлить и персонажа, если кажется, что он перемещается слишком быстро. Или же, можно ограничить перемещение персонажа одной клеткой по одному нажатию клавиши - завести переменную-переключатель bool, добавить её в условие перемещения персонажа, при выполнении условия - изменять состояние этого переключателя, что будет означать запрет перемещения. А разрешать тогда, когда ни одна кнопка не будет нажата.

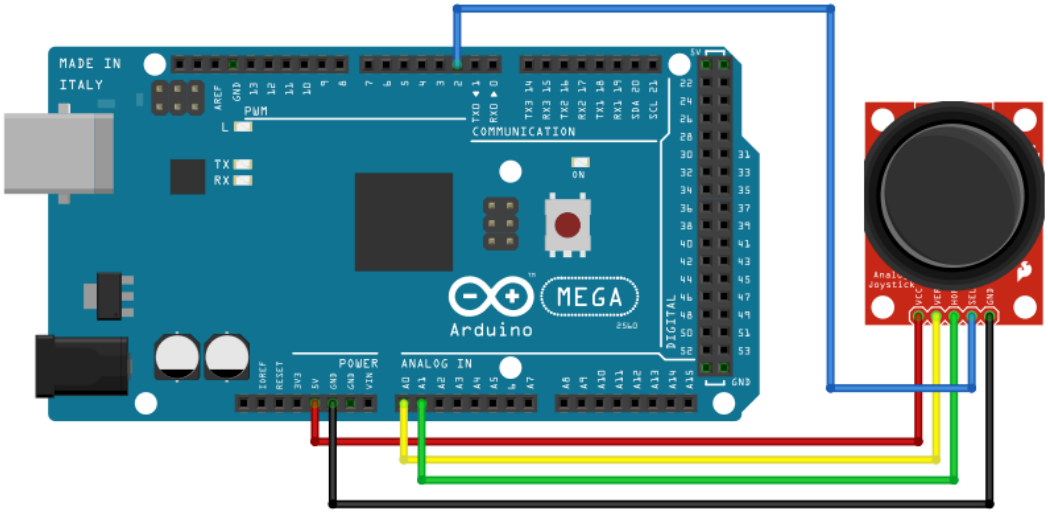
03.01.12. Проигрыш

Врагов-то мы оживили, но сейчас они для нашего персонажа безобидны. Сделаем так, чтобы при столкновении персонажа с противником появлялось сообщение о проигрыше и начиналась новая игра. В PlayerController циклом пройдёмся по врагам и сравним позиции игрока и врагов:

```
for(int i = 0; i < 3; i++){
  if(playerX == EnemiesX[i] && playerY == EnemiesY[i]){
    //Начинаем игру заново
  }
}
```

Можно попробовать изменить способ управления персонажем - использовать не кнопки, а джойстик.

03.01.13. Подключение джойстика::



- Красный и чёрный провода - питание и земля;
- Жёлтый и зелёный - вертикальная и горизонтальная оси, подключены к аналоговым портам;
- Синий - кнопка джойстика, подключена к цифровому порту.

Просто считываем показания с аналоговых портов, к которым подсоединены выводы джойстика:

```
joyX = analogRead(0);
joyY = analogRead(1);
```

В PlayerController(), в условии проверки нажатия клавиши можно добавить новое выражение:

```
if (buttonUP == LOW || joyY > 700) {
  dirY = -1;
 } else if (buttonDOWN == LOW || joyY < 300) {
  dirY = 1;
 } else {
  dirY = 0;
 }
//Для оси X аналогично
```

Суть в том, что превышение порогового значения будет означать движение в соответствующем направлении.

На этом этапе наша игра имеет хоть и простой, но законченный вид. Читатель может сам придумывать новые идеи и обогащать игру новыми механиками.

03.01.14 Объектно-ориентированное программирование

В нашей игре, для хранения координат противников, мы заводили два отдельных массива. Для такой игры этого будет вполне достаточно. Но что, если мы заходим добавить больше параметров для каждого противника? Например, очки здоровья, скорости, отображаемый символ и тому подобное. В таком случае придётся объявлять ещё несколько массивов для каждого параметра. Это не очень-то и удобно. Но программисты решили, что нужно что-то ещё придумать, и придумали такую вещь, как ООП.

Объектно-ориентированное программирование (ООП) - это подход к написанию программ, при котором программа состоит из отдельных объектов, которые взаимодействуют друг с другом. Оперируя совокупностью параметров и функций, как объектами, мы можем повысить читаемость кода, а также упростить масштабирование.

Попробуем описать сущность противника, используя элементы ООП:

- Описание объекта противника, его шаблона со всеми необходимыми параметрами:

```
Объект Противник
{
        КоординатаX
        КоординатаY
        НаправлениеX
        НаправлениеY
        Функция Ходить
        Функция Отображать
}
```

- Объявление описанного объекта и взаимодействие с ним:

```
Создать Противник
Противник.КоординатаX = 10
Противник.Ходить
Противник.Отображать
…
```

Ещё пример:

```
Создать массив[10] Противники
Противники[0].КоординатаX = 10
Противники[0].Ходить
Противники[1].КоординатаX = 11
Противники[1].Ходить
…
```

Довольно псевдокода. На практике такой объект называется классом (class):

```
class Enemy {
        int X;
        int Y;
```

```
        void Move(){
            //...
        }
};
```

Описав класс противника один раз в начале программы (вне функций setup и loop), можно использовать его сколько угодно раз. Объявляем переменную типа Enemy:

```
Enemy enemy;
```

Обращение к параметрам созданного класса осуществляется через точку:

```
enemy.X = 10;
```

Но, при попытке скомпилировать этот код, мы получим ошибку. Есть такое понятие, как модификаторы доступа. Сейчас описанные переменные и функция являются приватными. Это значит, что обратиться к ним извне класса невозможно. Для этого нужно объявить их как публичные, делается это так:

```
class Enemy {
    public:
        int X; //Присваивать значение не обязательно
        int Y;
        void Move(int X,int Y){
            //...
        }
};
```

Правило хорошего тона - описывать классы в виде некой закрытой структуры. Представим автомобиль. Как он работает изнутри мы не знаем, но зато мы можем управлять им, используя руль и педали. Так и здесь, если нам не нужно, чтобы можно было извне изменять, например, координаты противника, то объявим их как приватные переменные, а функцию перемещения оставим публичной:

```
class Enemy {
```

```
        private:
            int X;
            int Y;
        public:
            void Move(){
                //...
            }
};
```

Таким образом, влиять на координаты противника мы сможем только через функцию Move, в которой, например, может быть описана логика проверки захода координат за границы экрана, а не задавать абсолютно любое значение, которое может привести к неправильному поведению программы.

Но, что делать, если нужно задавать какое-то начальное положение противника, а доступа к координатам у нас нет? Можно объявить ещё одну публичную функцию, но для такого рода ситуаций имеется конструктор класса. Конструктор - это функция, которая вызывается в момент создания класса, т.е. вызов конструктора будет на этой строчке:

```
Enemy enemy;
```

Объявление конструктора:

```
class Enemy {
        private:
            int X;
            int Y;
        public:
            Enemy(int X_,int Y_){  //Конструктор
                X = X_;
                Y = Y_;
            }
            void Move(){
                X++;
                Y--;
            }
```

```
};
```

Конструктор имеет точно такое же имя, как и имя класса.
Объявление модифицированного класса:

```
Enemy enemy(10,2);
```

Аргументы, которые требует конструктор, пишутся в скобках после названия переменной класса.

Если мы захотим создать целую кучу противников, то мы без проблем сделаем это!

```
const Kucha = 3;
Enemy enemies[Kucha] = { Enemy(10,1), Enemy(10,2),Enemy(10,3)};
for(int i = 0; i < Kucha; i++)
{
        Enemies[i].Move();
}
```

Теперь попробуйте самостоятельно внедрить в нашу игру элементы ООП:

- Описать класс игрока, в котором будут храниться все его параметры и функции;
- Описать класс противников в котором будут храниться все его параметры и функции;
- Инициализировать массив противников;

03.02 Игра 2

Геймплей нашей второй игры будет заключаться в следующем:

- Персонаж расположен в левой части экрана и может перемещаться вертикально;
- Справа налево, в сторону персонажа, двигаются препятствия, которые персонаж должен избегать;

- Среди препятствий, ведущих к проигрышу, иногда появляются монеты, которые персонаж должен собирать;
- При сборе монет, увеличивается счётчик очков;
- При столкновении с препятствием, читаются данные из памяти, о лучшем результате, если текущее количество очков больше записанного значения, то новое загружается в память.

Для описания объектов будем использовать элементы ООП.
Для начала добавим функцию опроса кнопок:

```
void InputController() {
  buttonUP = digitalRead(BUTTON_UP);
  buttonDOWN = digitalRead(BUTTON_DOWN);
}
```

03.02.01. Описание класса персонажа

Приступим к описанию класса персонажа. Персонаж будет передвигаться только по вертикали, значит объявляем переменные:

- для координаты по Y;
- предыдущую координату по Y;
- Направление движения (вверх или вниз).

```
class Player {
  public:
    int prevY;
    int Y = 0;
    int dirY = 0;
};
```

Добавим две функции:

- Controller - для описания логики перемещения персонажа;
- Render - его отображение на экране.

```
  void Controller() {
    prevY = Y; // Предыдущая координата
    if (buttonUP == LO) { // Кнопка вверх нажата
      dirY = -1; //Направление вверх
    } else if (buttonDOWN == LOW) { //Кнопка вниз нажата
      dirY = 1; //Направление вниз
    } else {
      dirY = 0; //Стоять на месте
    }
    Y += dirY; // Перемещение, согласно направлению
    if (Y < 0) { //Проверка на заход за границы
      Y = 0;
    } else if (Y > 3) {
      Y = 3;
    }
  }

  void Render() {
    lcd.setCursor(0, Y);
    lcd.print(PLAYER);  //Вывод на экран символа игрока
    if (prevY != Y) { //Закрашивание пробелом предыдущей позиции
      lcd.setCursor(0, prevY);
      lcd.print(' ');
    }
  }
};
```

Создаём переменную Player вне loop(), причём после описания класса игрока:

```
Player player;
```

Затем, в loop() вызываем функции Controller() и Render():

```
void loop() {
  InputController();
  player.Controller();
  player.Render();
}
```

Готово, персонаж двигается и перемещается корректно. Приступим к описанию класса объекта препятствия.

03.02.02. Описание класса препятствия

Объявим с модификатором private переменные для координат по X,Y, предыдущие координаты и таймер движения - для замедления перемещения препятствий.

```
class Obstacle {
  private:
    int prevX = 20;
    int prevY = 1;
    int MoveTimer = 0;
    int X = 20;
    int Y = 1;
}
```

Функции Controller() и Render() для описания логики перемещения и отображения на экране стоит объявлять как публичные:

```
public:
    void Controller() {
      randomSeed(analogRead(8)); //зерно рандомайзера
      if (MoveTimer == 0) { //Если MoveTimer = 0, то перемещаем объект
        prevX = X; //Предыдущие координаты
        prevY = Y;
        if (X >= 0) { //Если X > 0, то передвигаем влево
          X--;
        }             //Если MoveTimer = 0, то теперь задаём ему
        MoveTimer = 3;  //начальное значение
```

```
    } else {
      MoveTimer--; // Иначе - отнимаем единицу
    }
  }
  void Render() { //Функция отображения на экране
    if (MoveTimer == 0) { //Отображать, только если MoveTimer = 0
      if (X >= 0 && X < 20) { //Если за границы не заходит, то
        lcd.setCursor(X, Y); //выводим на экран символ препятствия
        lcd.print(OBSTACLE);
      }
      //Если одна из предыдущих координат не равна текущим и
      // если предыдущая координата по X не заходит за границы
      if ((prevX != X || prevY != Y) && prevX >= 0 && prevX < 20) {
        lcd.setCursor(prevX, prevY);
        lcd.print(' ');
      }
    }
  }
```

Объявляем вне loop() массив препятствий:

```
const int ObstaclesCount = 14;
Obstacle obstacles[ObstaclesCount];
```

В loop() проходимся по массиву и вызываем функции этих объектов:

```
for (int i = 0; i < ObstaclesCount; i++) {
  obstacles[i].Controller();
  obstacles[i].Render();
}
```

03.02.03. Добавление функции начальной позиции класса препятствия

Препятствия движутся в сторону игрока, но они не возвращаются в правую часть. Добавим функцию StartPosition(), которая будет перемещать препятствия в правую часть.

- Объявим переменную prevStartObstacleX для хранения начальной координаты предыдущего препятствия
- Y-координате задаём случайное значение
- X-координату присваиваем значение prevStartObstacleX и прибавляем случайное число
- Если prevStartObstacleX стало слишком большим, то задаём X минимальное значение 20
- После - prevStartObstacleX присваиваем значение новой X-координаты, а переменные, которые хранят предыдущие координаты препятствия, должны быть равны новому значению X-координаты.

```
void StartPosition() {
  Y = random(0, 4);
  X = prevStartObstacleX + random(1, 4);
  if (prevStartObstacleX >= 80) {
    X = 20;
  }
  prevStartObstacleX = X;
  prevX = X;
  prevY = Y;
}
```

Вернёмся в функцию Controller(), там, где мы уменьшаем координату по X:

```
if (X >= 0) { //Если X > 0, то передвигаем влево
        X--;
}
```

Добавим конструкцию else и там вызываем StartPosition():

```
if (X >= 0) {
```

```
      X--;
    } else { //Если X меньше нуля, то вызываем StartPosition()
      StartPosition();
    }
```

По делу, нужно бы вызывать эту функцию и при создании массива obstacles. Сделаем это в конструкторе класса Obstacle, под модификатором public:

```
Obstacle() {
  StartPosition();
}
```

03.02.04. Столкновение с персонажем

Настал черёд функции столкновения с персонажем. Проверку на столкновение будем делать из класса препятствия. Напишем функцию PlayerCollision(). Рассуждать будем так:

- Функция принимает в качестве параметра Y-координату игрока.
- Если X препятствия = 0 И Y препятствия = Y-координате игрока, то запускаем процедуру проигрыша.

```
void PlayerCollision(int PlayerY) {
  if (X == 0 && PlayerY == Y) {
     //Проигрыш
  }
}
```

03.02.05. Проигрыш

Вроде бы, всё правильно, но есть одно НО. В случае проигрыша, желательно переставить все объекты на начальную позицию, а доступа к массиву препятствий изнутри класса препятствия нет. Можно просто переключать глобальную bool-переменную, отвечающую за проигрыш, или же можно поступить несколько иначе. Обратите внимание, мы условились, что void - это начало объявления функции. На самом деле, перед именем функции указывается тип возвращаемого значения. Посмотрите на функцию digitalRead:

```
buttonUP = digitalRead(BUTTON_UP);
```

Переменной, отведённой для хранения состояния кнопки, присваивается какое-то значение, которое функция отправляет на выход. Обобщённое описание этой функции в системной библиотеке выглядит примерно так:

```
int digitalRead(int Pin){
        //...
        int pinState = //...
        return pinState;
}
```

int в начале - это тип выходного значения или, как правильнее, тип возвращаемого значения. pinState - переменная, в которую записывается состояние порта, затем, с помощью ключевого слова return, эту переменную отправляют на выход этой функции. Именно тип этой переменной и нужно указывать при объявлении функции.

Давайте поступим так же. Пусть наша функция PlayerCollision будет возвращать переменную типа bool, значение которой будет зависеть от выполнения условия столкновения. Значит, bool - тип возвращаемого значения, который мы укажем перед именем:

```
bool PlayerCollision(int PlayerY) {
     if (X == 0 && PlayerY == Y) {
        //Проигрыш
      }
    }
```

Если столкновение произошло, то возвращаем переменную типа bool со значение true, иначе - возвращаем false:

```
bool PlayerCollision(int PlayerY) {
  if (X == 0 && PlayerY == Y) {
     return true;
  }
  return false;
}
```

По сути, эта запись является более короткой версией такой записи:

```
bool PlayerCollision(int PlayerY) {
  bool IsCollision = false;
  if (X == 0 && PlayerY == Y) {
     IsCollision = true;
     return IsCollision;
  }
  return IsCollision;
}
```

Лучше использовать первый вариант, он короче и понятнее.

Использовать эту функцию будем следующим образом:

В loop(), в цикле, в котором проходимся по массиву препятствий, добавляем условие:

```
for (int i = 0; i < ObstaclesCount; i++) {
  obstacles[i].Controller();
  obstacles[i].Render();
  //Если PlayerCollision возвращает true, значит перезапускаем игру
  if (obstacles[i].PlayerCollision(player.Y) == true) {
     //Функция запуска новой игры
     GameRestart();
  }
}
```

03.02.06. Функция перезапуска игры

Лучше бы добавить отдельную функцию для перезапуска игры - GameRestart

```
void GameRestart() {
  for (int i = 0; i < ObstaclesCount; i++) {
    obstacles[i].StartPosition();
  }
  Speed = 100;
  Message("You lose coins!", "Try again");
```

```
  Score = 0;
}
```

Отлично! Столкновения добавлены. Теперь попробуем добавить монеты в игру. Для начала добавим счётчик очков Score.

03.02.07. Добавление монет

Пусть с некоторой вероятностью, при размещении в начальной позиции, препятствие будет становится монетой. Допишем в класс Obstacle переменную:

```
bool IsCoin = false;
```

Эта переменная будет определять, чем является этот объект. В StartPosition пропишем следующее:

```
IsCoin = false;
    if (random(0, 10) < 3) {
      IsCoin = true;
    }
```

По умолчанию, при вызове StartPosition, IsCoin будет равна false, но если рандомайзер возвращает значение меньше 3, то препятствие будет считаться монетой. Модифицируем PlayerCollision. Добавим условие. Если IsCoin = false, то проигрыш, если true, то увеличиваем счётчик очков:

```
bool PlayerCollision(int PlayerY) {
  if (X == 0 && PlayerY == Y) {
    if (IsCoin == true) { //Если монета
      Score++; //Увеличиваем количество очков.
      StartPosition(); // Перемещаем в начальную позицию
      Message("Coin! Score:" + String(Score));
      return false;//Возвращаем false, иначе будет засчитан проигрыш
    } else { //Если не монета, то проигрыш
      Message("You lose coins!", "Try again");
      Score = 0;
```

```
      Message("Score:" + String(Score));
      return true;
    }
  }
  return false;
}
```

Также нужно модифицировать функцию отображения препятствия - Render.

```
void Render() {
  if (MoveTimer == 0) {
    if (X >= 0 && X < 20) {
      lcd.setCursor(X, Y);
      if (IsCoin == true) { // Если монета, то выводим один символ
        lcd.print(COIN);
      } else { //Иначе - другой
        lcd.print(OBSTACLE);
      }
    }
    if ((prevX != X || prevY != Y) && prevX >= 0 && prevX < 20) {
      lcd.setCursor(prevX, prevY);
      lcd.print(' ');
    }
  }
}
```

03.02.08. Изменение скорости игры

Добавим усложнение игры, зависящее от времени. Будем уменьшать задержку в конце loop(). Заведём переменную Speed, отвечающую за эту задержку, переменную SpeedTimer, затем напишем функцию GameSpeed, вне каких-либо классов.

```
void GameSpeed() {
```

```
  if (SpeedTimer == 0) { //Если таймер на нуле, то уменьшаем задержку
    Message("Game is", "getting Harder!");
    if (Speed > 60) { //не будем уменьшать Speed ниже 50.
      Speed -= 10;
    }
    SpeedTimer = 100; //Начальное значение таймера
  } else {
    SpeedTimer--; //Уменьшаем таймер.
  }
}
```

Вызываем эту функцию внутри loop:

```
GameSpeed();
delay(Speed);
```

Также, при проигрыше, в GameRestart, стоит присваивать Speed начальное значение.

03.02.09. Запись в память

Приступим к выполнению последнего пункта - записи очков в память. Для этого предусмотрена EEPROM-память.

EEPROM (англ. Electrically Erasable Programmable Read-Only Memory – электрически стираемое перепрограммируемое постоянное запоминающее устройство. Это энергонезависимая память, к которой имеется доступ из выполняющейся программы. EEPROM предназначена для записи пользовательских данных.

Для работы с EEPROM нужно подключить библиотеку <EEPROM.h>. Пользоваться ей очень просто. Есть несколько функций для чтения и записи данных с немного отличающимся функционалом, но мы будем использовать функции put и get.

- EEPROM.put(адрес, переменная) - записывает в память по указанному адресу значение переменной.
- EEPROM.get(адрес, переменная) - читает данные из памяти по указанному адресу и записывает их в указанную переменную.

Модифицируем GameRestart, в итоге функция должна иметь следующий вид:

```
void GameRestart() {
  for (int i = 0; i < ObstaclesCount; i++) {
    obstacles[i].StartPosition();
  }
  Speed = 100;
  int prevScore = 0; // Здесь храним значение, прочитанное из памяти
  EEPROM.get(0, prevScore); //читаем значение по адресу 0
  if (prevScore < Score) { //Если новый счёт больше старого, то
    EEPROM.put(0, Score); // записываем в память по адресу 0 Score
  }
  Message("You lose coins!", "Try again",2000);
     Message("Previous Score - " + String(prevScore), "New Score - " +
String(Score),5000);
  Score = 0;
}
```

Теперь, при проигрыше, мы будем видеть лучший результат.

Пусть эта игра получилась достаточно простой, зато мы познакомились с новыми, важными моментами языка.

03.03 Игра 3

Данная игра будет сложнее и масштабнее предыдущих. В ходе разработки мы реализуем большой игровой мир размером 80x60 ячеек, даруем персонажу возможность копать препятствия, добавим инвентарь, и ещё многое (не многое) другое. Использовать элементы ООП здесь не будем, т.к. вы, скорее всего, ещё недостаточно с ним освоились, эту игру можно написать и без этого.

Давайте начнём с несколько иных задач. В нашей игре мы будем полностью задействовать память ЖК-дисплея для пользовательских символов. Хранить их всех в

одном файле с кодом не очень-то и удобно. Есть отличное решение этой проблемы - написать свою библиотеку.
Создадим новый пустой файл под названием “Characters.ino” и сохраним его в одной папке с проектом. В нашей игре препятствия на карте будут в виде камней. Создадим символ для камня и пропишем его в этот пустой файл:

```
const char STONE = 3;
byte StoneChar[8] = {
 0b00000,
 0b00000,
 0b00110,
 0b01111,
 0b11111,
 0b11111,
 0b11110,
 0b01110
};
```

Сохраним файл. Теперь, нужно создать ещё один файл - “Characters.h” - его название должно совпадать с предыдущим, но расширение - “.h”. Сохраним в той же папке. В этом файле указываются переменные, к которым будет доступ извне этой библиотеки. Т.е в этом файле прописываем следующее:

```
extern const char STONE;
extern byte StoneChar[8];
```

Так мы указали, что переменные с этими именами будут доступны извне библиотеки. Сохраняем файл. Переходим к основному файлу. Прописываем в начале подключение нашей библиотеки:

```
#include "Characters.h"
```

Теперь мы сможем добавлять новые символы, не заставляя сильно разрастаться файлу с основной программой.

03.03.01. Создание игрового мира

Сначала разберёмся с созданием игрового мира. Объявим массив типа char нужного размера:

```
const int MAP_W = 80;
const int MAP_H = 64;
char game_map[MAP_W][MAP_H];
```

Препятствия на карте будут в виде камней. Напишем функцию по их размещению:

```
void GenerateMap() {
  randomSeed(analogRead(8));
  //Счётчик итераций высокой вероятности возникновения камней
  int HighStoneChanceCounter = 0;
  //Переключатель для смены вероятности
  bool CanChangeStoneChance = true;
  for (int i = 0; i < MAP_W; i++) { //Проходимся по всей карте по X и Y
    for (int j = 0; j < MAP_H; j++) {
      if (i % 10 == 4 && j % 10 == 2) { //Условие повышения вероятности
        if (CanChangeStoneChance) { //Если разрешена смена вероятности
          //То задаём значение для счётчика итераций для высокой
          //вероятности возникновения камней
          HighStoneChanceCounter = random(2, 5);
          CanChangeStoneChance = false; //Запрещаем смену вероятности
        }
      }  else if(HighStoneChanceCounter == 0){
        CanChangeStoneChance = true; // Разрешаем смену вероятности
      }

      int StoneChance = 5; //Вероятность возникновения камня
      if (HighStoneChanceCounter > 0) { //Счёт высокой вероятности > 0
        StoneChance = 16; //Задаём высокую вероятность
        HighStoneChanceCounter--;
      }
      if (random(StoneChance) >= 4) { // Условие возникновения камня
```

```
        game_map[i][j] = STONE; //Размещаем символ камня
      } else {
        game_map[i][j] = SPACE; //Иначе - пустота
      }
    }
  }
}
```

Проходимся циклом по всему массиву, некоторое условие будет истинно, то увеличиваем вероятность возникновения в ячейке камня. Причём высокая вероятность будет сохранятся на протяжении нескольких итераций цикла, что позволит размещать на карте скопления камней.
Повышать вероятность возникновения камня будем тогда, когда будет истинным выражение:

```
if (i % 10 == 4 && j % 10 == 2)
```

Оператор “%” обозначает деление с остатком, т.е.вы этом выражении происходит деление с остатком координат по X и Y на 10.

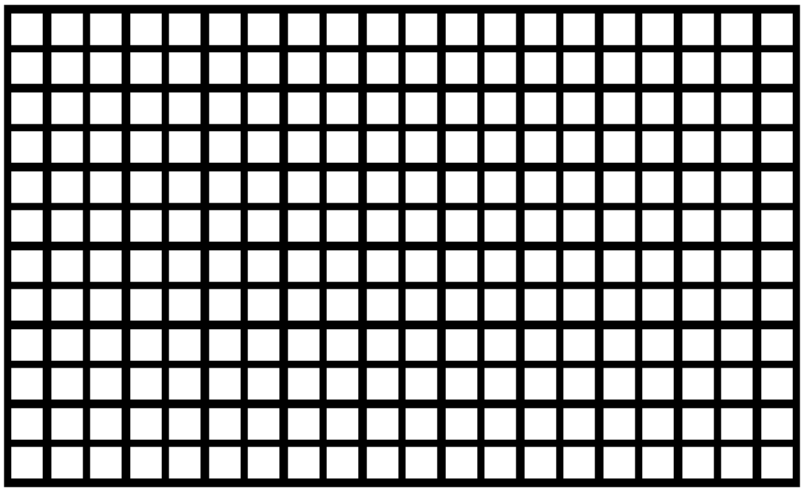
03.01.02. Функция опроса элементов управления

Напишем функцию для опроса элементов управления, нам она скоро понадобится:

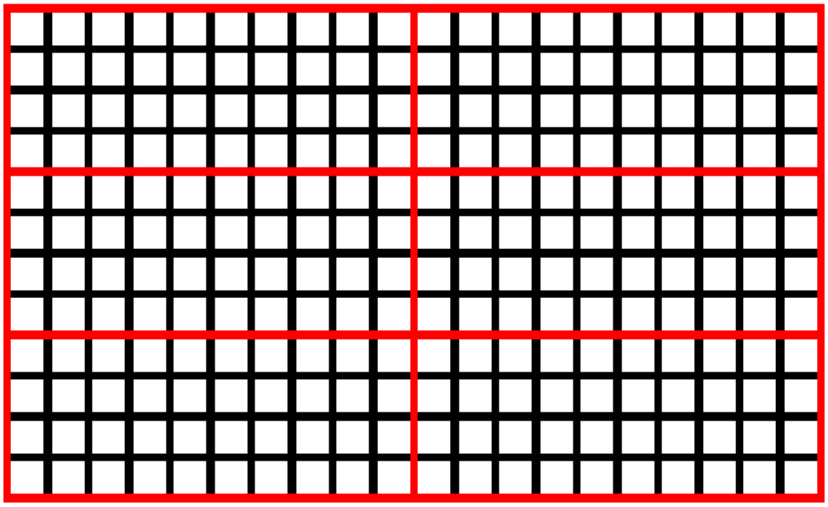
```
void InputController() {
  rb = digitalRead(RIGHTBUTTON);
  lb = digitalRead(LEFTBUTTON);
  ub = digitalRead(UPBUTTON);
  db = digitalRead(DOWNBUTTON);
  joyX = analogRead(0);
  joyY = analogRead(1);
}
```

03.03.03. Функция отрисовки карты

Теперь самое интересное - функция отрисовки карты. Рассуждаем так:
Это - игровая карта



Так мы её поделили на несколько частей, равных размерам экрана



Будем определять в каком из таких прямоугольников находится игрок и отрисовывать именно эту часть.

Напишем две функции, отвечающие за определение такого прямоугольника:

```
// map_player_x и map_player_y - глобальные координаты игрока, тип int
int GetMapBoundX() {
  return map_player_x / 20 * 20;
}
int GetMapBoundY() {
  return map_player_y / 4 * 4;
}
```

Эти две функции возвращают нам координаты верхнего левого угла прямоугольника: первая координату по X, вторая - по Y. Работают они так: Например, позиция игрока

по X равна 24. Ширина экрана - 20 символов. Делим позицию игрока на ширину экрана, и должно получится значение 1.2, но так как обе переменные имеют тип int, результат округляется до 1. Затем, умножается на размер экрана, и, получается, что координата по X прямоугольника, в котором находится игрок, равна 20. Точно так же и с Y координатой.
Можно ещё добавить перегрузки для этих функций. Перегруженные функции - это функции с одинаковыми именами и отличающиеся аргументами:

```
int GetMapBoundX(int x) {
  return x / 20 * 20;
}
int GetMapBoundY(int y) {
  return y / 4 * 4;
}
```

Если первые функции давали результат для позиции игрока, то здесь мы сможем указать позицию любого другого объекта.

Идём дальше. Функция отрисовки карты:

```
void RenderMap(bool ShowMsg) {
  lcd.clear(); //Сначала всё очищаем
  int boundX = GetMapBoundX(); //Получаем координаты участка карты
  int boundY = GetMapBoundY();
    // Вывод координат текущего участка карты (Необязательно)
if(ShowMsg){
  lcd.setCursor(6, 0);
  lcd.print("Location:");
  lcd.setCursor(7, 2);
  lcd.print(boundX);
  lcd.setCursor(10, 2);
  lcd.print("-");
  lcd.setCursor(12, 2);
  lcd.print(boundY);
  delay(700);
```

```
  lcd.clear();
}
   //
  for (int i = 0; i < 20; i++) {
    for (int j = 0; j < 4; j++) {
      lcd.setCursor(i, j); //Задаём курсору локальные координаты экрана
      //Если в эта ячейка карты не пустая, то выводим её содержимое
      if (game_map[boundX + i][boundY + j] != SPACE) {
        lcd.print(game_map[boundX + i][boundY + j]);
      }
    }
  }
}
```

Вызываем написанные ранее функции для определения нужного участка карты. Затем, проходимся по этому участку, прибавляя к его координатам локальные координаты экрана и выводим содержимое ячеек в соответствующем месте экрана.

03.03.04. Добавление игрока

Добавим игрока. Будем использовать четыре пары переменных:

```
int player_x = 0, player_y = 48;                    // (1)
int prev_player_x, prev_player_y;                    // (2)
int xdir = 0, ydir = 0;                             // (3)
int map_player_x = player_x, map_player_y = player_y; // (4)
```

1 - Локальные координаты игрока;

2 - Предыдущие локальные координаты игрока;

3 - Направление движение игрока;

4 - Глобальные координаты игрока на карте.

```
void PlayerControl() {
  xdir = 0;
  ydir = 0;
```

```
if (joyX < 300) { // Выбор направления в зависимости от джойстика
  xdir = -1;
  ydir = 0;
}
if (joyX > 700) {
  xdir = 1;
  ydir = 0;
}
if (joyY < 300) {
  xdir = 0;
  ydir = 1;
}
if (joyY > 700) {
  xdir = 0;
  ydir = -1;
}
map_player_x += xdir;// Прибавление к координатам направления
player_x += xdir;    //К локальным и
map_player_y += ydir;//И к глобальным координатам
player_y += ydir;
if (player_x < 0) { //Если локальные координаты вышли за границы
  player_x = 19;    // экрана, то переносим игрока на экране в
  RenderMap();  //противоположную сторону и
}                   // вызываем метод отрисовки
if (player_x > 19) { //Игрок пошёл вправо - перемещаем налево
  player_x = 0;
  RenderMap(); //Вызываем отрисовку карты
}
if (player_y < 0) { //То же самое для Y
  player_y = 3;
  RenderMap();
}
if (player_y > 3) {
  player_y = 0;
  RenderMap();
```

```
  }
if (map_player_x < 0) { // Проверка глобальных координат на
    map_player_x = 0;   // Заход за границы карты
    player_x = 0;
  }
  if (map_player_x > MAP_W - 1) {
    map_player_x = MAP_W - 1;
    player_x = 19;
  }
  if (map_player_y < 0) {
    map_player_y = 0;
    player_y = 0;
  }
  if (map_player_y > MAP_H - 1) {
    map_player_y = MAP_H - 1;
    player_y = 3;
  }
}
```

Теперь функция отрисовки персонажа:

```
void RenderPlayer() {
  //Если предыдущие координаты не совпадают с нынешними
  if (prev_player_x != player_x || prev_player_y != player_y) {
    lcd.setCursor(prev_player_x, prev_player_y); //Устанавливаем курсор
    // Печатаем содержимое ячейки
    lcd.print(game_map[GetMapBoundX() + prev_player_x]
                      [GetMapBoundY() + prev_player_y]);
  }
  lcd.setCursor(player_x, player_y); //Выводим игрока в текущей позиции
  lcd.print(PLAYER);
}
```

Готово. Вызываем в setup() GenerateMap() и RenderMap(), в loop():

```
  InputController(); //Опрос элементов управления
 PlayerControl(); //Функции логики игрока и противника
 RenderPlayer();
 prev_player_x = player_x;
 prev_player_y = player_y;
```

А теперь наслаждаемся исследованием большого открытого мира.
Мир, наполненный одними только камнями, выглядит скучновато. Добавим небольшое разнообразие. Пусть при генерации карты, на месте камня, будет 50% вероятность размещения не камня, а его маленьких осколков. Сначала нарисуем символ таких осколков, затем в GenerateMap(), в условие возникновения камня, напишем следующее:

```
if (random(StoneChance) >= 4) { // Условие возникновения камня
    if (random(2) == 0) { // 50% вероятность
      game_map[i][j] = STONEFRAGMENTS; // Так будут осколки камня
    } else {
      game_map[i][j] = STONE; // А так - целый камень
    }
} else { //...
```

Вот, уже стало немножко красивее. Но надо бы запретить ходить сквозь большие камни. Давайте будем перемещать персонажа только тогда, когда ячейка, в которую идёт персонаж, будет содержать символ
из списка разрешённых.
Модифицируем PlayerController():

03.03.05. Столкновение игрока с препятствиями

```
//Массив элементов, по которым можно ходить
char UnCollideChars[] = { SPACE, STONEFRAGMENTS};
//Проходимся циклом по этому массиву
 for (int i = 0; i < 2; i++) {
 //Если в содержимое нужной ячейки равно символу из списка и
 //направление не равно 0
  if (game_map[map_player_x][map_player_y + ydir] == UnCollideChars[i]
```

```
      && ydir != 0){
    map_player_y += ydir; //То перемещаем
    player_y += ydir;
    break; //Выходим из цикла
   }
   // То же самое по X
   if (game_map[map_player_x + xdir][map_player_y] == UnCollideChars[i]
      && xdir != 0){
    map_player_x += xdir;
    player_x += xdir;
    break; //Выходим из цикла
   }
 }
```

Столкновения сделали. Но, смотрите, если мы добавим в массив разрешённых символов новые (а мы добавим новые), то придётся увеличивать и количество итераций цикла. Вместо того, чтобы сделать это вручную, автоматизируем это действие следующим образом:

```
for (int i=0; i < sizeof(UnCollideChars) / sizeof(UnCollideChars[0]); i++)
```

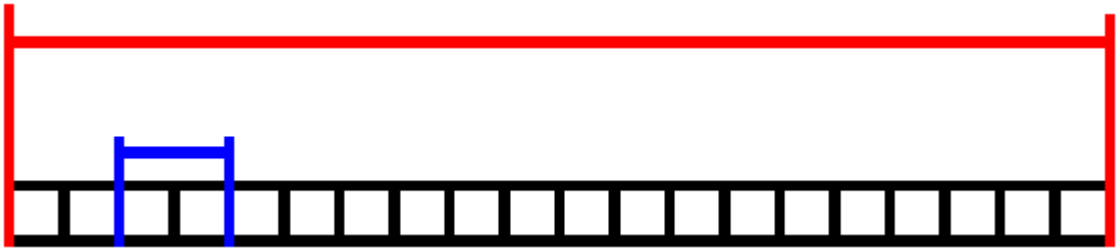
Оператор sizeof возвращает занимаемый объём памяти объекта. Например, попытаемся узнать размер переменной типа int:

```
int a = 0;
int b = sizeof(a);
```

Переменная b будет равна 2, т.к. размер int - 2 байта. Таким образом, в выражении

```
sizeof(UnCollideChars) / sizeof(UnCollideChars[0]),
```

сначала вычисляется размер, занимаемый массивом, затем это значение делится на размер одного из его элементов и, в результате, получаем количество элементов массива:



Красным выделен размер массива, синим - размер одного из его элементов.

03.03.06. Функция копания

Вернёмся к нашим камням. Камни - они зачем? Они только и существуют ради того, чтобы их кто-нибудь расколол на части. Так добавим же систему копания!
Сначала создадим два новых символа - символ кирки - PICKAXE и символ куска железа - IRON.

Функция Dig():
Принимает позицию камня, определяет на каком участке карты он находится. Если камень в том же участке, что и игрок, то выставляем курсор на указанную позицию камня и выводим на том месте символ кирки. Далее, с 50% вероятностью размещаем на карте либо кусок железа, либо каменные осколки. Затем, через 300 мс, снова проверяем участок карты игрока и камня, если они совпадают, то выводим либо каменные осколки, либо железа, в зависимости от значения рандома.

```
void Dig(int x, int y) {
   int pickaxeBoundX = GetMapBoundX() + x; // Определим в
   int pickaxeBoundY = GetMapBoundY() + y;
   if (GetMapBoundX(pickaxeBoundX) == GetMapBoundX() &&
GetMapBoundY(pickaxeBoundY) == GetMapBoundY()) {
     lcd.setCursor(x, y);
     lcd.print(PICKAXE);
   }
   int r = random(0, 2);
   if (r == 0) {
     game_map[GetMapBoundX() + x][GetMapBoundY() + y] =
STONEFRAGMENTS;
   } else {
```

```
    game_map[GetMapBoundX() + x][GetMapBoundY() + y] = IRON;
  }
  delay(300);
  if (GetMapBoundX(pickaxeBoundX) == GetMapBoundX() &&
GetMapBoundY(pickaxeBoundY) == GetMapBoundY()) {
    lcd.setCursor(x, y);
    delay(100);
    if (r > 0) {
      lcd.print(IRON);
    } else {
      lcd.print(STONEFRAGMENTS);
    }
  }
}
```

Добавим в массив символов, по которым можно ходить, IRON.
Вызывать эту функцию будем из PlayerController(). Добавим туда такую конструкцию.

```
        if (game_map[map_player_x + xdir][map_player_y + ydir] == STONE &&
            map_player_x + xdir >= 0 && map_player_x < MAP_W &&
            map_player_y + ydir >= 0 && map_player_y < MAP_H) {
          ActionValue++;
          if (ActionValue >= 2) {
            Dig(player_x + xdir, player_y + ydir);
            ActionValue = 0;
          }
        }
```

Большое и страшное условие - это проверка на наличие камня в ячейке, в которую пытается переместиться персонаж, а также проверка на заход за границы карты.
ActionValue - это задержка для копания, без неё Dig будет вызываться мгновенно.
В аргументы Dig прописываем локальные координаты + направление движения.
И обнуляем ActionValue.
Проверяем работоспособность. Вроде, выглядит неплохо.

03.03.07. Инвентарь

И вот, мы подошли к тому, чтобы научить персонажа заносить предметы в свой инвентарь.

Объявим два массива:

```
char Items[16];
int ItemValues[16];
```

Items - для хранения предметов
ItemValues - для хранения характеристики прочности инструментов

Напишем функцию- RenderInventory, которая будет выводить окно инвентаря

Новые переменные:

- InventoryCursorPos - позиция инвентарного курсора.
- INVENTORYCURSOR - символ инвентарного курсора

Проходимся по массиву предметов и, если текущий элемент не пуст, то выводим символ предмета.

```
void RenderInventory() {
  lcd.clear();
  lcd.setCursor(InventoryCursorPos, 1);
  lcd.print(INVENTORYCURSOR);
  for (int i = 0; i < sizeof(Items) / sizeof(Items[0]); i++) {
    lcd.setCursor(i, 0);
    if (Items[i] != NULL) {
      lcd.print(Items[i]);
    }
  }
}
```

Напишем несколько вспомогательных функций для инвентаря:

FreeSpaceInInv - проверяет, есть ли в инвентаре пустое место, если да, то возвращает true, иначе - false

```
bool FreeSpaceInInv() {
  for (int i = 0; i < 16; i++) {
    if (Items[i] == NULL) {
      return true;
    }
  }
  return false;
}
```

AddItem добавляет предмет в инвентарь

```
void AddItem(char item) {
  if (item != 0) {
    for (int i = 0; i < 16; i++) {
      if (Items[i] == NULL) {
        Items[i] = item;
        ItemValues[i] = 100;
        return;
      }
    }
  }
}
```

GetItem Возвращает символ, который находится на карте по указанным координатам.

```
char GetItem(int x, int y) {
      char item = game_map[x][y]; //Берём символ с карты
      game_map[x][y] = SPACE;     //Убираем с карты
      return item;                //Возвращаем символ
}
```

DropItem выбрасывает предмет из инвентаря, на который указывает курсор, если на позиции игрока пустое место, иначе будет выведено сообщение.

Новая переменная: CanPickUpItems - будет нужна для разрешения заноса в инвентарь предметов

```
void DropItem() {
  if (game_map[map_player_x][map_player_y] == SPACE) {
    //Размещаем на карте выбранный символ
    game_map[map_player_x][map_player_y] = Items[InventoryCursorPos];
    //Обнуляем характеристику предмета
    ItemValues[InventoryCursorPos] = 0;
    Items[InventoryCursorPos] = NULL; //Убираем предмет из инвентаря
    CanPickUpItems = false; //Запрещаем поднимать предметы
  } else {
    Message("No free space", "");
  }
}
```

CheckItem Проверяет инвентарь на наличие предмета и возвращает true, если он есть

```
bool CheckItem(char item) {
  for (int i = 0; i < 16; i++) {
    if (Items[i] == item && ItemValues[i] > 0) {
      return true;
    }
  }
  return false;
}
```

Message - функция, выводящая сообщение с каким либо выбором

```
int Message(String msg1, String msg2) {
  // Выводим текст
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print(msg1);
  lcd.setCursor(0, 1);
  lcd.print(msg2);
  delay(500);
  //В зависимости от нажатой кнопки, функция возвращает разное
значение
  bool Finished = false;
  while (!Finished) {
    InputController();
    if (rb == 0) {
      return 1;
    } else if (lb == 0) {
      return 0;
    } else if (ub == 0) {
      return 2;
    } else if (db == 0) {
      return 3;
    }
  }
}
```

Наконец, функция управления инвентарём:

Новая переменная: InventoryIsOpen - отвечает за открытие и закрытие инвентаря.

```
void InventoryControl() {
  if (rb == 0 && InventoryCursorPos < 15) { //Движение курсора
    InventoryCursorPos++;
  } else if (lb == 0 && InventoryCursorPos > 0) {
```

```
    InventoryCursorPos--;
  }
  //Если выбранный предмет - кирка или меч, то выводим сообщение
  //со значением его характеристики
  if (ub == 0 && (Items[InventoryCursorPos] == PICKAXE ||
Items[InventoryCursorPos] == SWORD)) {
    Message("Durability:", ItemValues[InventoryCursorPos] + String("/100"));
  }
 // Если выбранный предмет - не пустота
  if (ub == 0 && Items[InventoryCursorPos] != NULL) {
    delay(200);
    //Если Message возвращает 1, что соответствует нажатой правой
кнопке
    if (Message("Drop?", "RB - YES LB - NO") == 1) {
      delay(500);
      //Выбрасываем предмет
      DropItem();
    }
  }
  //Выход из инвентаря
  if (db == 0) {
    lcd.clear();
    InventoryIsOpen = false;
    RenderMap(false);
  }
}
```

Модифицируем loop():

```
    if (!InventoryIsOpen) {
      PlayerControl();
      RenderPlayer();
      prev_player_x = player_x;
```

```
    prev_player_y = player_y;
    delay(150);
  } else {
    RenderInventory();
    InventoryControl();
    delay(150);
  }
```

В зависимости от состояния InventoryIsOpen будут вызываться функции либо инвентаря, либо персонажа.

Перейдём в PlayerControl и сделаем несколько изменений.

Добавим код открытия инвентаря:

```
    if (db == 0) {
      lcd.clear();
      InventoryIsOpen = true;
      lcd.setCursor(3, 0);
      lcd.print("INVENTORY");
      delay(700);
    }
```

Добавим код занесения в инвентарь предмета:

```
    //Сначала проверяем, разрешённый ли это предмет
     if (game_map[map_player_x][map_player_y] == IRON ||
    game_map[map_player_x][map_player_y] == PICKAXE ||
    game_map[map_player_x][map_player_y] == SWORD) {
      //Если в инвентаре есть пустое место и
      //поднимать предметы разрешено, то
      if (FreeSpaceInInv() && CanPickUpItems) {
        //Добавляем предмет в инвентарь
        AddItem(GetItem(map_player_x, map_player_y);
        delay(200);
      }
    }
```

Разрешать поднимать предметы будем тогда, когда персонаж сдвинулся с места, т.е. когда xdir или ydir не равны 0:

```
if (xdir != 0 || ydir != 0) {
  CanPickUpItems = true;
 }
```

Напишем новую функцию, которая будет уменьшать характеристику прочности кирки:

```
void PickaxeBreak() {
  for (int i = 0; i < 16; i++) {
    //Если кирка есть
    if (Items[i] == PICKAXE && ItemValues[i] > 0) {
      //Отнимаем значение прочности кирки
      ItemValues[i] -= random(2, 14);

      //Если прочность упала до нуля
      if (ItemValues[i] <= 0) {
        //Выводим сообщение
        Message("Pickaxe is", "broken");
        //Убираем кирку из инвентаря
        Items[i] = NULL;
        delay(1000);
        RenderMap(false); //Не выводим координаты участка карты
      }
      return; //Здесь return выполняет функцию выхода из этой функции
    }
  }
}
```

Модифицируем Dig

```
void Dig(int x, int y) {
    if (CheckItem(PICKAXE)) {
     //Код копания
     PickaxeBreak();
    }
 }
```

Если есть в инвентаре кирка, значит копать можно, и отнимаем прочность кирки.

03.03.08. Плавильня

Добавим в игру новый объект - плавильню. Используя плавильню, мы сможем изготавливать кирку и меч за 3 единицы железа. Плавильни будут разбросаны по карте случайным образом. Добавим для неё символ - FURNACE, затем в GenerateMap будем размещать плавильни на карте:

```
for (int i = 0; i < 5; i++) {
  randomSeed(analogRead(8));
  int rY = random(2, 63);
  randomSeed(analogRead(9));
  int rX = random(16, 80);
  game_map[rX][rY] = FURNACE;
}
```

Нужно написать функцию взаимодействия с плавильней:

```
void CreateMenu() {
 // Подсчёт количества железа в инвентаре
  int ironcount = 0;
  int i = 0;
  for (i = 0; i < 16; i++) {
    if (Items[i] == IRON) {
      ironcount++;
    }
  }
  // Если его больше 3
  if (ironcount >= 3) {
    //Вызываем диалоговое окно, в котором выбираем, что изготовить
    int answer = Message("Create:", String(PICKAXE) + ":LB" + " " +
String(SWORD) + ":RB");
    if (answer == 0) {
```

```
    AddItem(PICKAXE); //Добавление предмета в инвентарь
    Message("PICKAXE CREATED", "");
   } else if (answer == 1) {
    AddItem(SWORD);
    Message("SWORD CREATED", "");
   //Другими кнопками можно выйти из этого меню
   } else if (answer == 2 || answer == 3) {
    RenderMap(false);
    return;
   }
   //Удаление из инвентаря трёх единиц железа
   ironcount = 0;
   for (i; i > 0; i--) {
    if (Items[i] == IRON && ironcount < 3) {
     Items[i] = NULL;
     ironcount++;
    }
   }
   delay(1000);
  }
  RenderMap(false);
}
```

В PlayerControl добавим следующий код:

```
        if (game_map[map_player_x + xdir][map_player_y + ydir] == FURNACE)
        {
           ActionValue++;
           if (ActionValue >= 2) {
             CreateMenu();
             ActionValue = 0;
           }
         }
```

Если персонаж наткнулся на плавильно, то вызываем меню взаимодействия с ней.

Вы наверняка обратили внимание на такой предмет, как меч. Добавили его отнюдь не случайно. За нашим персонажем будет гоняться враг, от которого меч сможет защитить.

03.03.09. Враг

Очередь создания врага. Враг в какой-то момент времени будет набегать (~~как домики набигают~~) на игрока и наносить урон, если у него не будет в инвентаре меча, который с каждой атакой будет терять прочность, затем враг убегает и,через какое то время, снова прибежит.
Напишем функцию EnemyControl. Выглядит достаточно большой, но сложного в ней ничего нет.
Новые переменные:

- hp - очки здоровья игрока
- CurrentAttackTimer - таймер атаки врага, который запускает атаку, дойдя до значения AttackTimer
- AttackTimer конечное значение таймера
- IsAttack - состояние врага атака/не атака
- enemy_x, enemy_y, prev_enemy_x, prev_enemy_y - координаты врага, текущие и предыдущие соответственно
- enemy_target_x и enemy_target_y - точка, к которой должен бежать враг
- CanEnemyMove - ограничение движения врага
- state - состояние игры, позже станет понятно для чего оно

```
void EnemyControl() {
                        //Если таймер меньше максимального значения
  if (CurrentAttackTimer < AttackTimer) {
    CurrentAttackTimer++;    //увеличиваем таймер
  } else {                  //Иначе
    IsAttack = true;        //Запускаем атаку
    CurrentAttackTimer = 0;  //обнуляем таймер
    randomSeed(analogRead(8));
    AttackTimer = random(50, 200);//Определяем время для следующей
атаки
  }
  if (IsAttack) {             //Если атака началась
```

```
  enemy_target_x = player_x;  //Цель врага - координаты игрока
  enemy_target_y = player_y;
 } else { //Если враг не атакует
  //выбор цели врага
  int dir = random(0, 2) - 1;
  if (dir == 0) {
    dir = -1;
  }
  enemy_target_x = 10 + 11 * dir; //Задаём точку
  enemy_target_y = 2 + 3 * dir;
 }
 //Движение врага к цели, если движение разрешено
 if (enemy_x < enemy_target_x && CanEnemyMove) {
  enemy_x++;
 } else if (enemy_x > enemy_target_x && CanEnemyMove) {
  enemy_x--;
 }
 if (enemy_y < enemy_target_y && CanEnemyMove) {
  enemy_y++;
 } else if (enemy_y > enemy_target_y && CanEnemyMove) {
  enemy_y--;
 }
 //Столкновение врага с целью
 if (enemy_y == enemy_target_y && enemy_x == enemy_target_x) {
  enemy_y = prev_enemy_y; //Перемещаем врага в предыдущую точку
  enemy_x = prev_enemy_x;
  enemy_target_x = 25; //Задаём другую цель врагу, чтобы после атаки
  enemy_target_y = -5; //он бежал туда (можно поставить любую точку)
  if (IsAttack) { //Если враг столкнулся с цель и идёт атака
    if (CheckItem(SWORD)) {  //Если в инвентаре игрока есть меч
      //lcd.createChar(SWORD, SwordChar);
      lcd.setCursor(enemy_x, enemy_y); //Рисуем на месте врага меч
      lcd.print(SWORD);
      delay(300); //Делаем задержку
      SwordBreak(); //Новая функция - потеря прочности меча, напишем позже
```

```
      Message("You defended", ""); // Сообщение об отражении атаки
      delay(100);
      RenderMap(false); //Отрисовка карты
     } else { //Если меча у игрока нет
      hp -= random(10, 30); //Отнимаем очки здоровья
      delay(300);
      //Выводим сообщение об атаке и количество очков здоровья
      Message("You're attacked", "Health: " + String(hp));
      if (hp <= 0) {  //Если здоровье игрока на нуле
        hp = 0;
        state = 2; //Меняем состояние игры
        delay(2000);
      } else { //Если здоровье не на нуле
         RenderMap(false); //Отрисовываем карту
       }
     }
    }
    IsAttack = false; //Враг больше не атакует
  }
  //Если враг столкнулся с камнем, то камень рушится
  if (game_map[GetMapBoundX() + enemy_x][GetMapBoundY() + enemy_y] ==
STONE) {
    game_map[GetMapBoundX() + enemy_x][GetMapBoundY() + enemy_y] =
STONEFRAGMENTS;
  }
  //Если расстояние до игрока больше 10, то каждую итерацию разрешаем
  // или запрещаем движение врага, для его замедления, в результате
  // получается, что сначала враг будто крадётся, затем резко набегает
  if (abs(enemy_x - player_x) > 5) {
    // функция abs возвращает абсолютное значение числа, т.е. его
    // значение без минуса
    CanEnemyMove = !CanEnemyMove; //Постоянно запрещаем и разрешаем
  }
}
```

Теперь функция отрисовки врага - RenderEnemy, работает почти так же, как и отрисовка игрока:

```
void RenderEnemy() {
  // Если враг не за границами экрана, то рисуем его
  if (enemy_x >= 0 && enemy_x <= 15 && enemy_y >= 0 && enemy_y < 4) {
    lcd.setCursor(enemy_x, enemy_y);
    lcd.print(ENEMY1);
  }
//Если предыдущая позиция врага не за границами экрана
if ((prev_enemy_x != enemy_x || prev_enemy_y != enemy_y) &&
(prev_enemy_x >= 0 && prev_enemy_x <= 15 && prev_enemy_y >= 0 &&
prev_enemy_y <= 3) ) {
    lcd.setCursor(prev_enemy_x, prev_enemy_y);
    lcd.print(game_map[GetMapBoundX() + prev_enemy_x][GetMapBoundY() +
prev_enemy_y]);
  }
}
```

Осталась функция потери прочности меча - SwordBreak, работает так же, как и в случе с киркой:

```
void SwordBreak() {
  for (int i = 0; i < 16; i++) {
    if (Items[i] == SWORD && ItemValues[i] > 0) { //Если нашёлся меч
      ItemValues[i] -= random(10, 30); //Отнимаем его прочность
      if (ItemValues[i] <= 0) { //Если прочность на нуле
        Message("Sword is", "broken"); //Выводим сообщение
        Items[i] = NULL; //Убираем меч из инвентаря
        delay(1000);
        RenderMap(false);
      }
      return; //Выход из функции, если все операции с мечом совершили
    }
```

```
 }
}
```

Вызываем EnemyControl и RenderEnemy функции в loop там же, где и функции игрока:

```
        PlayerControl(); //Функции игрока
        RenderPlayer();
        prev_player_x = player_x;
        prev_player_y = player_y;
        EnemyControl(); //Функции врага
        RenderEnemy();
        prev_enemy_x = enemy_x;//Установление предыдущей позиции
    врага
        prev_enemy_y = enemy_y;
```

Отлично, разобрались с этим наконец то. Можно уже сейчас запустить и проверить, как враг атакует нашего персонажа. Но сейчас даже после обнуления очков здоровья игра не начнётся заново. Займёмся этим.
Переменная state отвечает за игровое состояние. У нас их три - сам игровой процесс, состояние победы и поражения. Модифицируем loop:

```
void loop() {
  InputController(); //Опрос элементов управления
  if (state == 0) {  // Состояние игрового процесса
    if (!InventoryIsOpen) {
      PlayerControl(); //Функции логики игрока и противника
      RenderPlayer();
      prev_player_x = player_x;
      prev_player_y = player_y;
      EnemyControl();
      RenderEnemy();
      prev_enemy_x = enemy_x;
      prev_enemy_y = enemy_y;
      delay(150);
```

```
    } else {
      RenderInventory(); //Функции логики работы игрового инвентаря
      InventoryControl();
      delay(150);
    }
  } else if (state == 1) { //Состояние победы
    Message("Congratulations!", "You win!");
    // Здесь функция перезапуска игры
  } else if (state == 2) { //Состояние поражения
    Message("YOU ARE DEAD", "Try again");
    //Здесь тоже перезапуск
  }
}
```

03.03.10. Перезапуск игры

Сейчас нужно дописать функцию перезапуска игры. Мы можем прописать присвоение всем нужным параметрам значения, соответствующие началу игры, что довольно муторно, или же, можем поступить иначе. Есть в C++ такая штука, как указатели. В рамках этого курса вдаваться в магию указателей мы не будем, но здесь воспользуемся этим механизмом языка.
Пропишем следующую строку в начале программы:

```
void(* resetFunc) (void) = 0;
```

Такой записью мы объявили указатель resetFunc на функцию, которая имеет тип возвращаемого значения void, и имеющая нулевой адрес в памяти.
Вызвав функцию, на которую указывает этот указатель следующим способом:

```
resetFunc();
```

Мы добьёмся того, что программа микроконтроллера будет выполнятся с самого начала, тем самым мы получим перезагрузку игры. Нужно добавить эту запись туда, где мы ходим перезагружать игру.

Перезагрузки мы добились, отлично, но есть недоделка. Игрок не имеет конечной цели, зато может проиграть, непонятно за что. Давайте придумаем цель игры. Пусть на карте случайным образом будет размещаться некий выход, который игрок нужно найти. Но, просто найти его будет недостаточно, нужно будет принести туда 30 единиц железа, после чего произойдёт смена игрового состояния на победу. Добавим новый символ для выхода - EXIT.

В GenerateMap() размещаем выход случайным образом:

```
randomSeed(analogRead(8));
int rY = random(6, 62);
randomSeed(analogRead(9));
int rX = random(MAP_W - 40, MAP_W - 20);
game_map[rX][rY] = EXIT;
game_map[rX - 1][rY] = '('; //Для красоты рядышком с выходом
game_map[rX + 1][rY] = ')'; //можно разместить скобки
```

Функция ExitFunc, которая будет вызываться при попытки подойти к выходу:

- IronCountToWin - количество железа, которое нужно занести
- TotalIronCount - количестве железа, которое игрок занёс в выход

```
void ExitFunc() {
  for (int i = 0; i < 16; i++) { //Если есть в инвентаре железо
    if (Items[i] == IRON) {
      TotalIronCount++; //Прибавляем счётчик
      Items[i] = NULL; //Отнимаем у игрока железо
    }
  }
  if (TotalIronCount < IronCountToWin) { //Если железа не хватает
    Message(String(IronCountToWin - TotalIronCount) + " pieces left", "");
    delay(300);
    RenderMap(false);
  } else { //Если его хватило, то меняем игровое состояние
    state = 1;
  }
}
```

Перейдём в PlayerControl. Вызов функции взаимодействия с выходом будет таким же, как и в случае с плавильней:

```
if (game_map[map_player_x + xdir][map_player_y + ydir] == EXIT) {
  ActionValue++; //Счётчик действия, задержка
  if (ActionValue >= 2) {
    ExitFunc();
    ActionValue = 0;
  }
}
```

Настала пора финального теста игры. Сейчас всё должно быть в порядке.

Проделана достаточно большая работа, не так ли? Но это ещё далеко не всё. Программную часть привели в порядок, но как насчёт того, чтобы привести в порядок ещё и часть аппаратную? Перейдём к следующей теме нашего курса - создание игровой консоли!

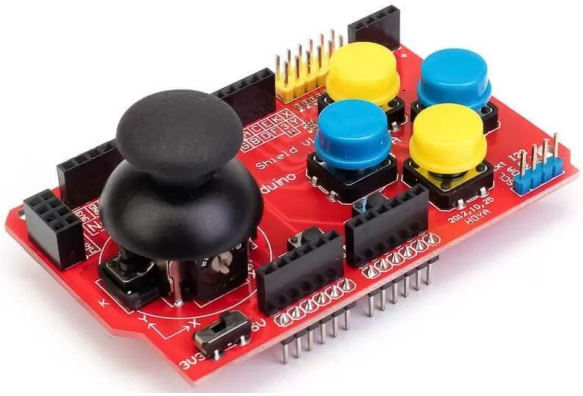
Тема 4. Разработка игровой консоли

Наша новая цель - разработать полноценное устройство в виде игровой консоли.

04.01.01 Выбор компонентов

Для начала, подготовим все необходимые компоненты:

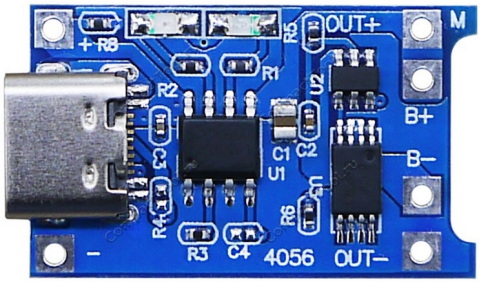
- Дисплей стоит использовать размером 20x4 с впаянным модулем I2C.
- Для элементов управления - плата расширения (Shield) бренда Funduino.



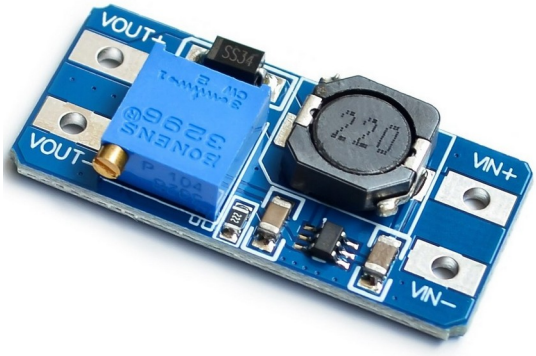
- Литий-ионный аккумулятор, в нашем примере будем использовать форм-фактор 18650

- Держатель для аккумулятора

- Контроллер зарядки TP4056



- Повышающий преобразователь MT3608



- Кнопка питания

Теперь обо всём по порядку. Плата расширения или, по английски - Shield, это модуль для Arduino, который можно устанавливать без пайки, просто вставив его контакты в пины Arduino. Для нашего устройства этот модуль хорошо подходит. Он имеет джойстик, 4 кнопки и 2 дополнительных, поменьше, но их мы задействовать не будем.

Мы делаем портативную игровую консоль, по определению она должна быть независима от источника питания. Поэтому нужно заполучить аккумулятор и отсек под него.

Для зарядки аккумулятора используется специальная плата - контроллер зарядки. Она выравнивает ток до приемлемого для аккумулятора уровня. Внешнее питание будет подаваться на аккумулятор через этот контроллер.

Если напрямую подавать напряжение с аккумулятора на Arduino, то ничего не выйдет по той причине, что рабочее напряжение Arduino - 5 В, а аккумулятор выдаёт меньшее значение. Поэтому необходим такой компонент как повышающий преобразователь. Он повышает напряжение с аккумулятора до нужного уровня и подаёт на Arduino.

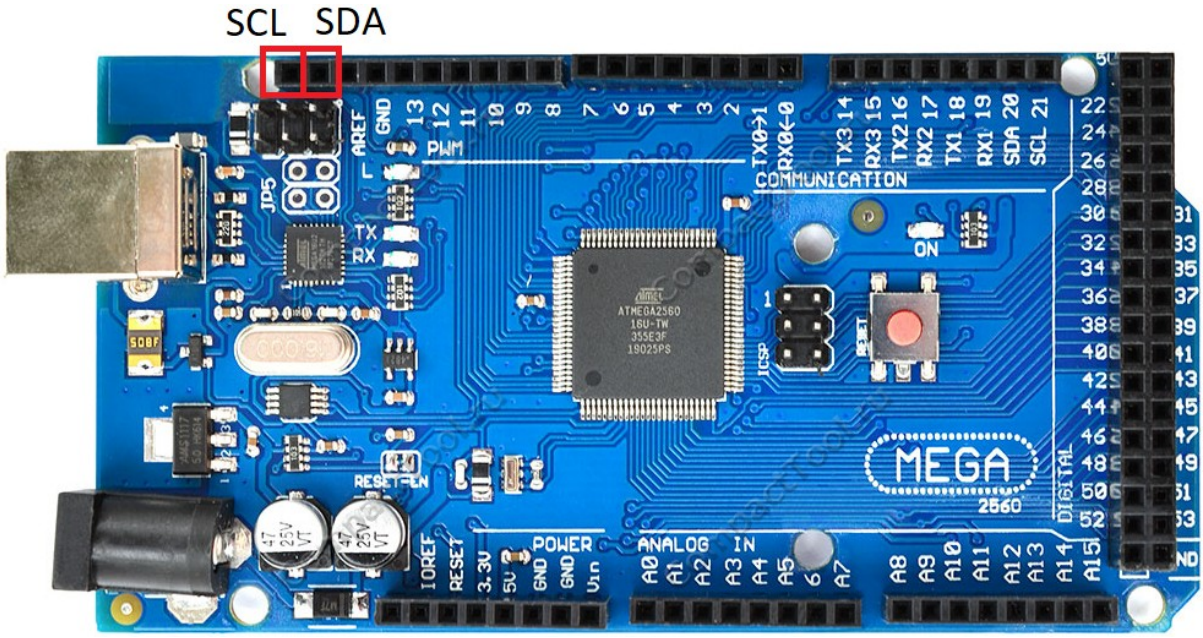
Также необходима кнопка питания, которая будет разъединять контакты с аккумулятором. Обратите внимание, что обычную кнопку, которая идёт в наборах

Arduino использовать нельзя, нужно использовать кнопку-переключатель, которая после нажатия сохраняет своё состояние. В интернете её можно найти по запросу “Power button switch” или просто “Кнопка питания”.

04.01.02 Сборка устройства

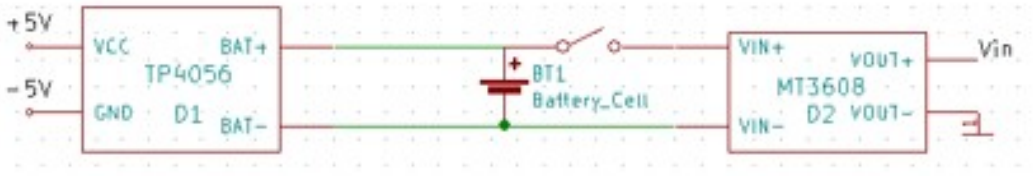
Первым делом, вставляем плату с джойстиком на плату. Джойстик должен быть со стороны USB-порта, а в пины с 0 по 7 - должны быть вставлены контакты.

Теперь нужно припаять дисплей. У I2C модуля 4 контакта - SCL, SDA, Vcc, GND.
На Arduino Mega SCL и SDA - самые левые пины, относительно цифровых портов:

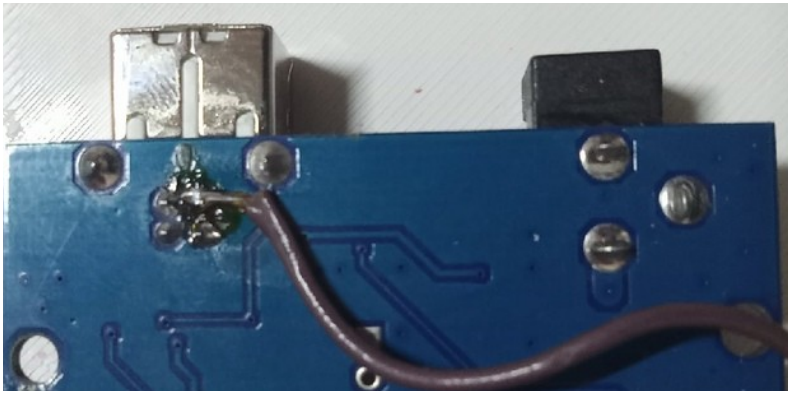


Либо же, 20 и 21 пины.

Припаиваться к Arduino следует снизу. Припаиваемся к соответствующим контактам I2C модуля дисплея. GND на I2C модуле припаиваем к GND на плате. Vcc - к 5 вольтовому пину. 5 В выдаёт линия цифровых портов в правой части платы, пронумерованных от 22 до 53. Проверяем работоспособность дисплея.
На случай надобности, можно припаять пьезодинамик к 12 цифровому пину и к земле.
Но не стоит трогать пины до 8, они задействованы кнопками платы управления.
Схема цепи, отвечающей за питание, представлена на рисунке:



На контроллере заряда имеется USB-порт, но мы его не задействуем, будем заряжать аккумулятор через порт на плате Arduino.



На фото провод припаян к “-” USB-порта. Можно припаять к любому GND-пину. Этот провод ведём к “-” на контроллере зарядки, “+” контроллера зарядки - к “+” USB-порта или к любому 5 вольтовому пину. “B+” и “B-” контроллера зарядки припаиваем к выводам держателя аккумулятора снизу, соответствующим “+” и “-”. “+” аккумулятора припаиваем к кнопке, а кнопку к “Vin+” повышающего преобразователя, “- ” аккумулятора - к “Vin-”. Перед тем, как припаивать повышающий преобразователь к Arduino, следует проверить его выходное напряжение, используя мультиметр.

Вращая выделенную на рисунке ручку переменного резистора, можно менять выходное напряжение.

После того, как выходное напряжение стало достигать 5В, припаиваем “Vout+” к “Vin” Arduino, “Vout-” повышающего преобразователя к земле.

На этом пайка закончена. Теперь Arduino может работать как от внешнего источника питания, так и от аккумулятора.

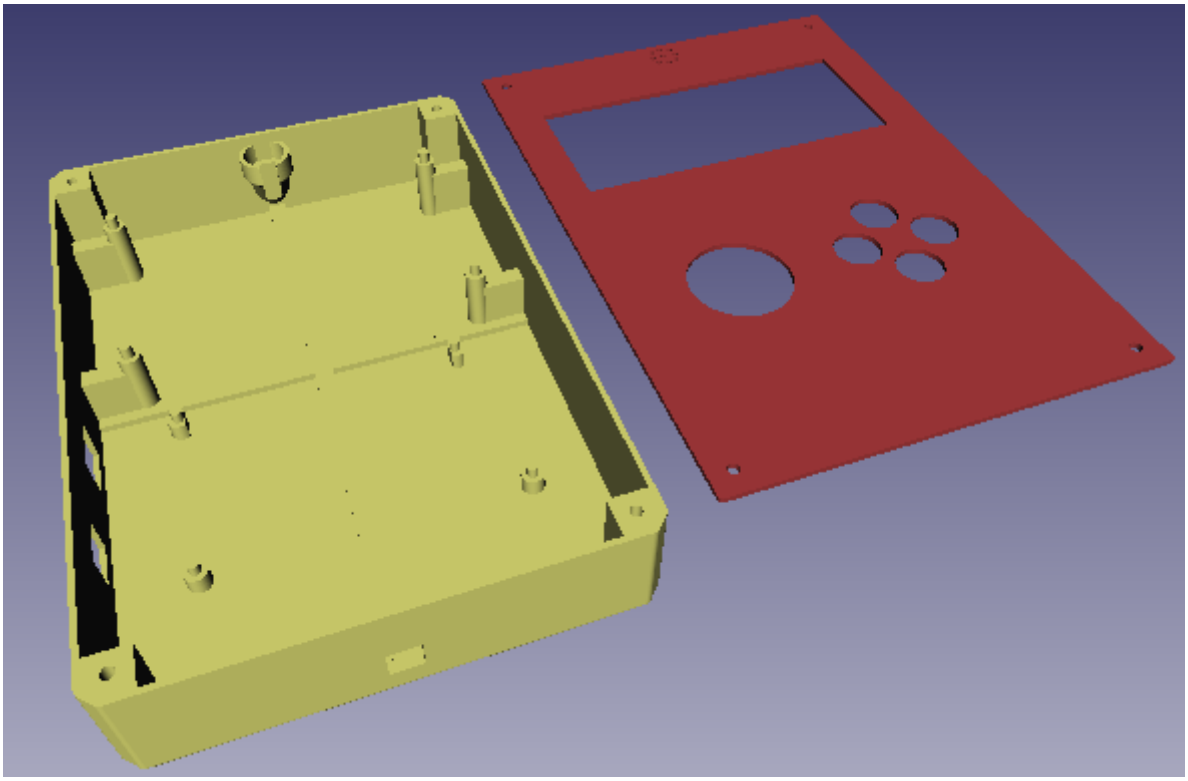
Внимание! Нельзя подавать питание одновременно и с аккумулятора, и на USB-порт!

Повышающий преобразователь и контроллер заряда можно зафиксировать на стенках держателя аккумулятора, используя термоклей.

Теперь перейдём к разработке корпуса.

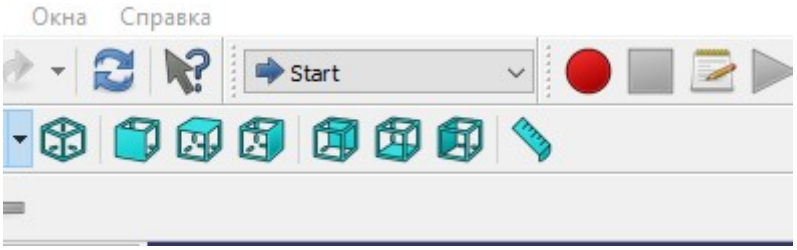
04.02 Разработка корпуса

Корпус будем разрабатывать в среде параметрического моделирования FreeCAD, и печатать на 3D-принтере.
Готовый вариант корпуса выглядит следующим образом:

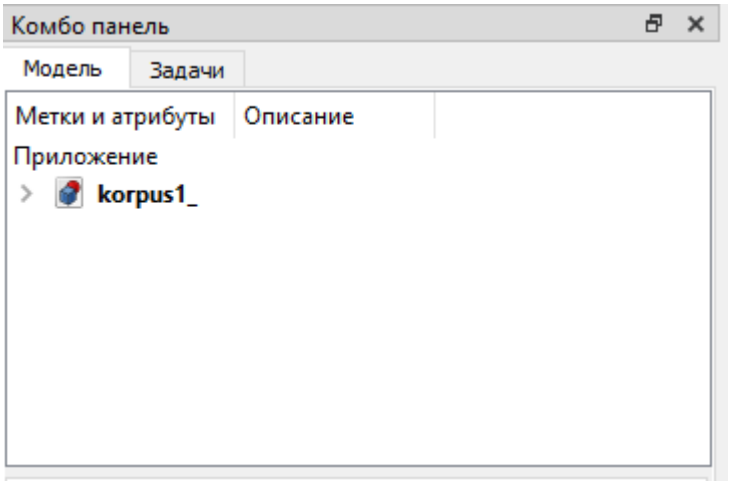


Он будет состоять из нижней части, на которой предусмотрены стойки для крепления платы и дисплея, стойка для пьезодинамика, отверстия для портов и кнопки, и верхней части - крышки с отверстиями для кнопок с джойстиком и вырезом для дисплея.

Итак, начинаем. Первым делом - создать новый проект. Нас встречает окно пустого проекта. Сначала смоделируем саму простую деталь - крышку. Во FreeCAD есть несколько режимов работы, которые называют верстаками (Workbench). При запуске выбран верстак “Start”.



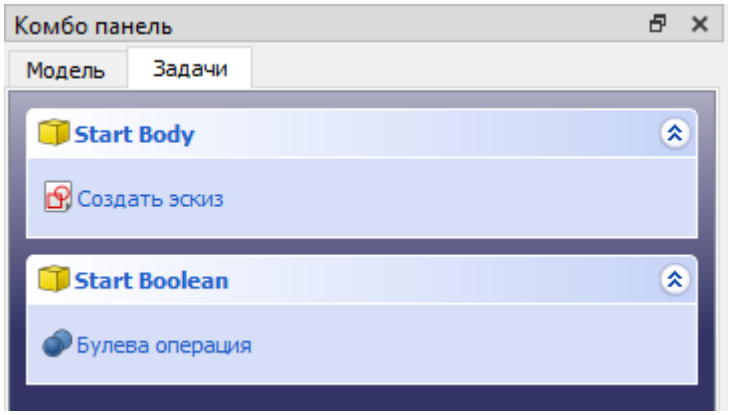
Для дальнейшей работы меняем его на Part Design. В левой части экрана есть окно - комбо-панель.



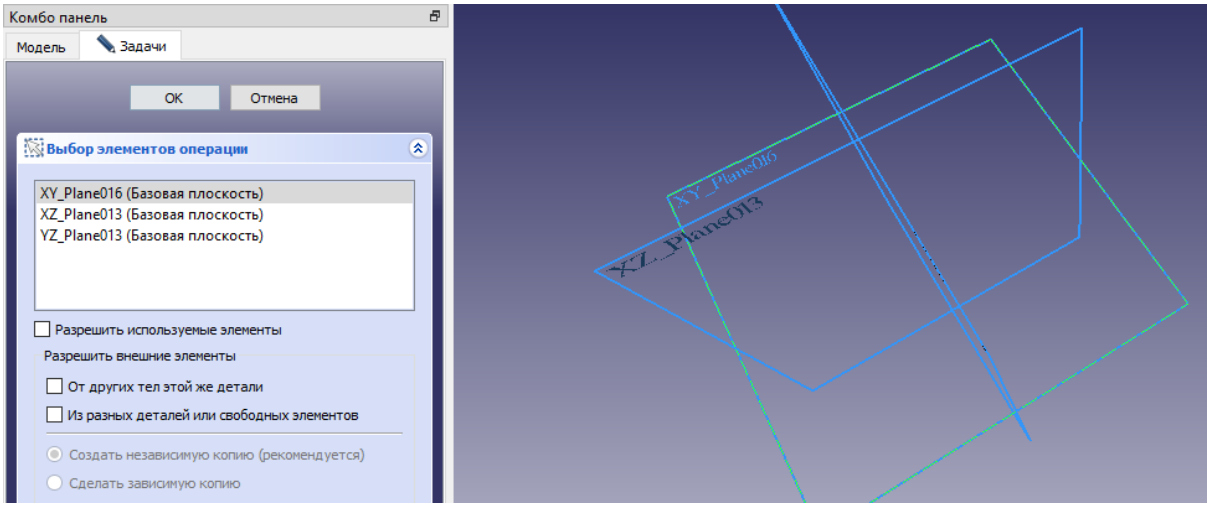
Находясь в верстаке Part Design, открываем на комбо-панели вкладку “задачи” и нажимаем “Создать тело”.

04.02.01 Заготовка крышки

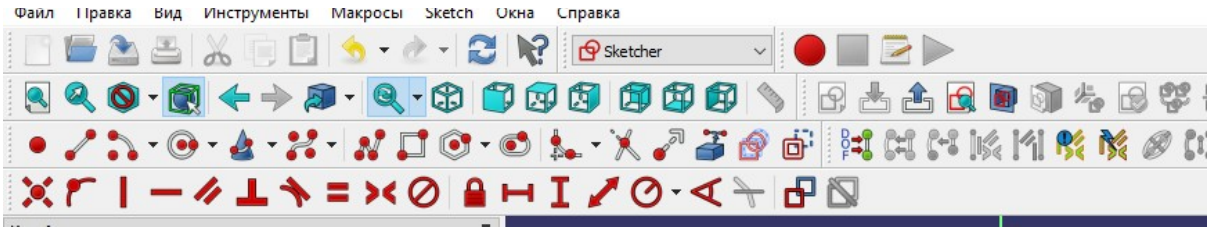
Перейдите на предыдущую вкладку в комбо-панел. Обратите внимание, что появился новый объект. Это наша будущая крышка.
Затем, во вкладке задачи, нажимаем “Создать эскиз”



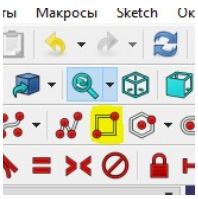
Далее - нужно выбрать плоскость, в которой будет располагаться эскиз. Выбираем XY. Выбранная плоскость выделяется другим цветом для наглядности.



Автоматически произошёл переход в другой верстак - Sketcher, предназначенный для создания чертежей.

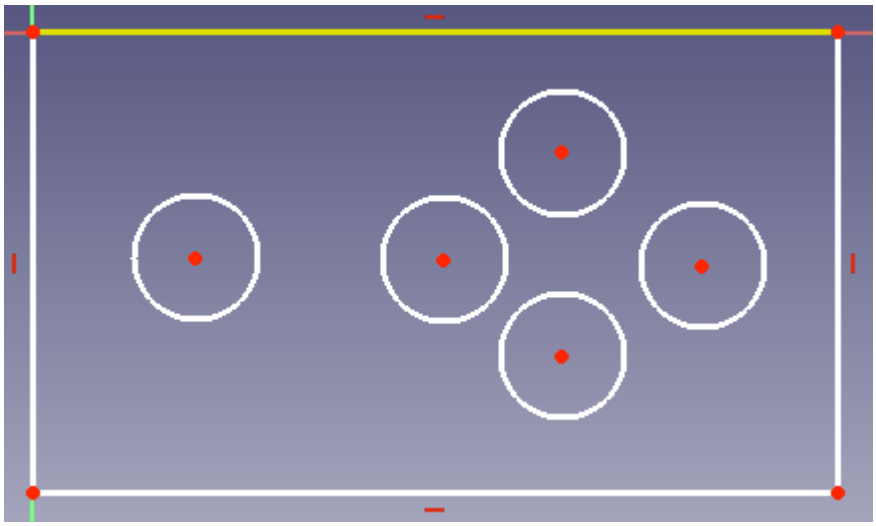


Суть создания чертежей во FreeCAD - это начертить эскиз, не задумываясь о размерах, затем задавать эти размеры инструментами ограничений.
Сначала начертим прямоугольник произвольного размера, используя данный инструмент

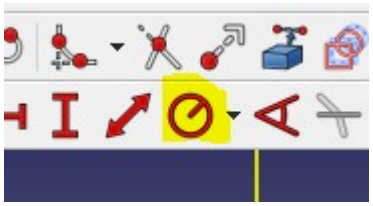


Чертим отверстия для кнопок и джойстика, используя инструмент создания окружности

Должно получится что то вроде этого:



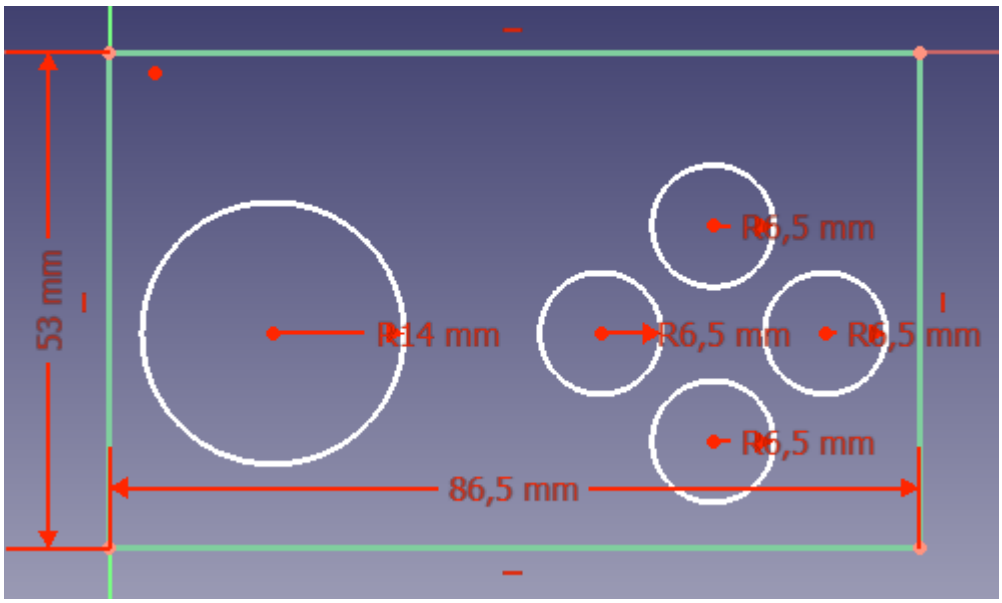
Теперь перейдём к инструментам ограничений. Этот инструмент позволяет установить радиус окружности.



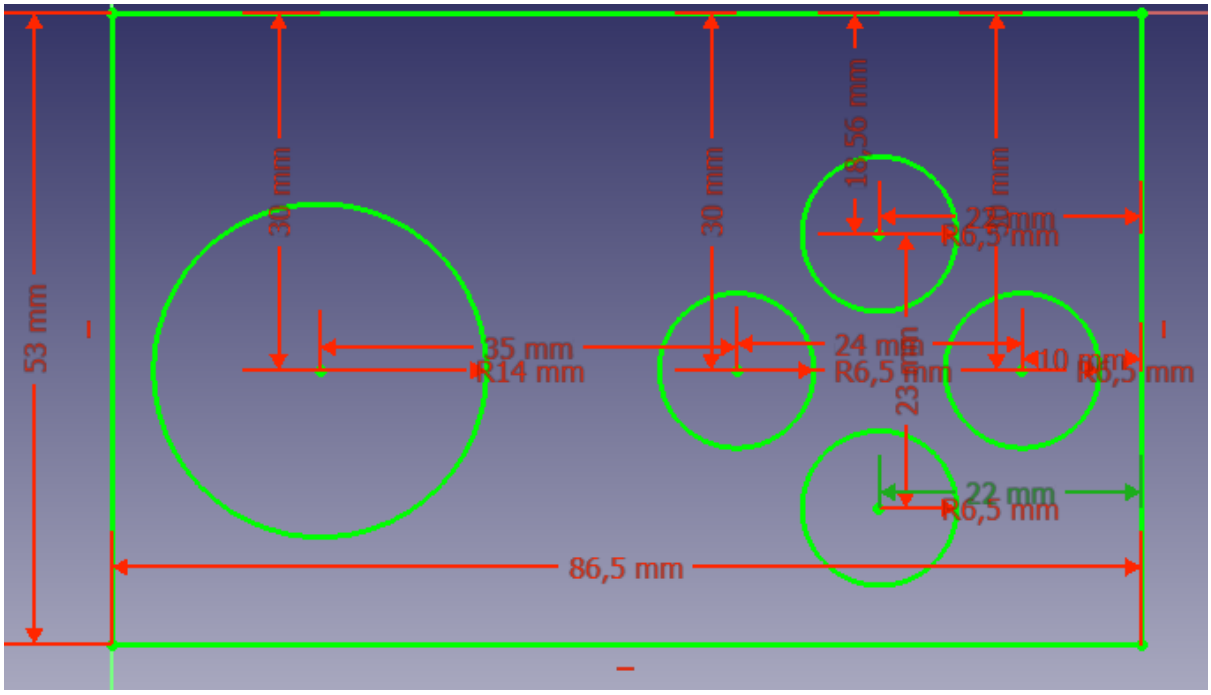
Кликаем на окружность и задаём в появившемся окне нужный радиус.

Эти два инструмента ограничивают длину отрезка или расстояние между точками по горизонтали и вертикали.

Как и в случае с окружностями, аналогичным образом задаём размеры прямоугольнику. Пусть этот прямоугольник будет повторять размеры платы.

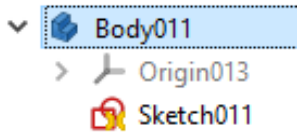


Теперь измеряем расстояния от границ платы до кнопок, расстояния между кнопками, расстояние джойстика от границ платы. Используя те же инструменты ограничения длин по горизонтали и вертикали задаём расстояние от центра окружностей до сторон прямоугольника.

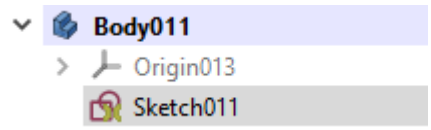


Все элементы стали зелёными, это значит, что в чертеже нет элементов без ограничений. После всех действий, в комбо-панели нажимаем кнопку “Закрыть”. Нас вернуло в верстак Part Design. Переходим в комбо-панель во вкладку “Модель”.

Выбираем эскиз внутри созданного тела. Если название тела написано нежирным шрифтом:

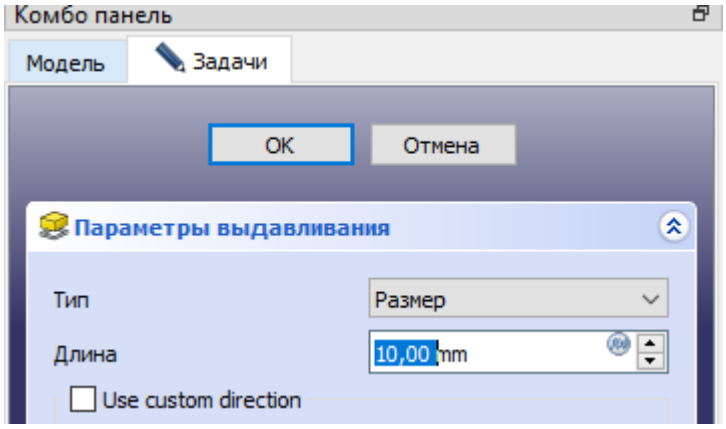


То нужно нажать два раза на него, чтобы оно стало таким:



Это особенности работы FreeCAD, таким образом мы указали, что это активное тело.

Теперь нажимаем на эскиз и ищем инструмент “Выдавить эскиз”

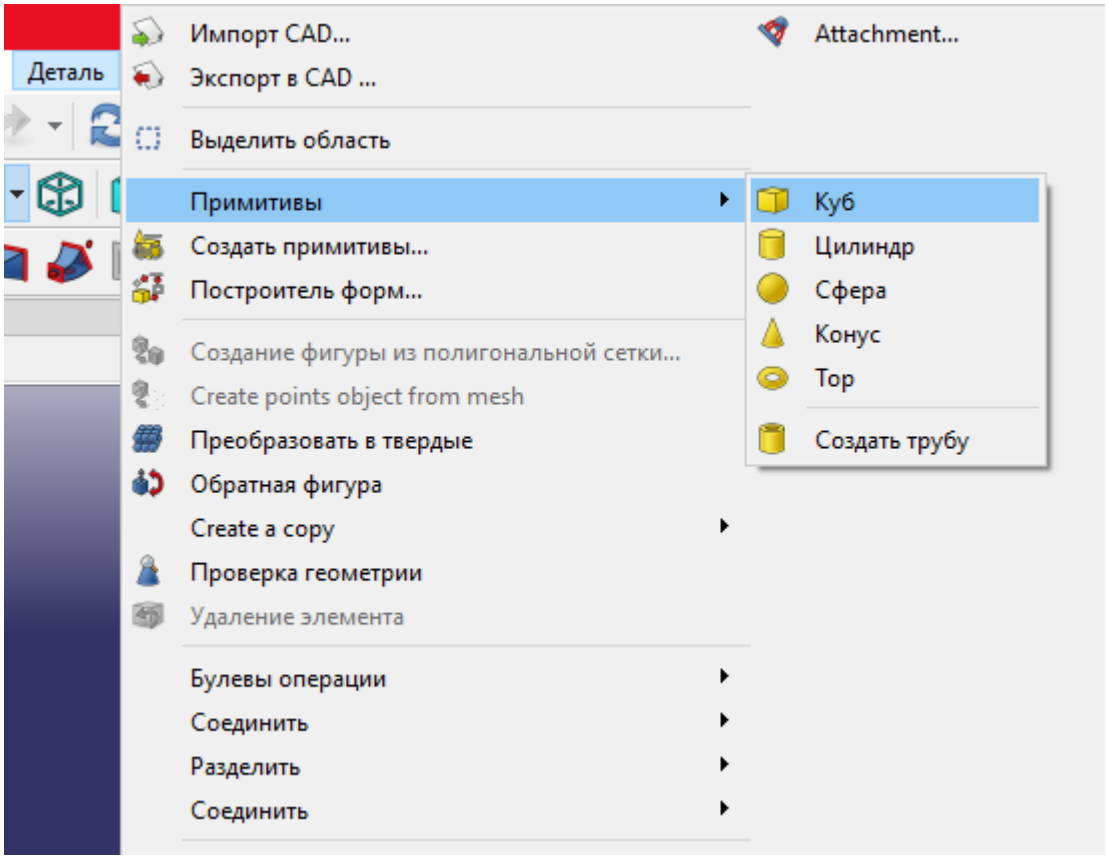


Выбираем необходимую толщину тела, 1.5 мм будет вполне достаточно. У нас появилась заготовка крышки.

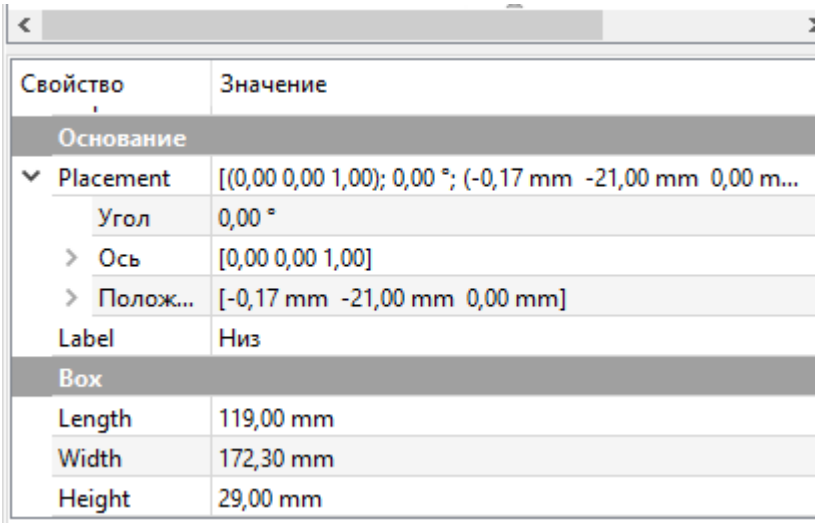
Пока что оставим крышку и начнём делать нижнюю часть. Выберем тело крышки и нажмём пробел - тело будет скрыто.

04.03. Нижняя часть

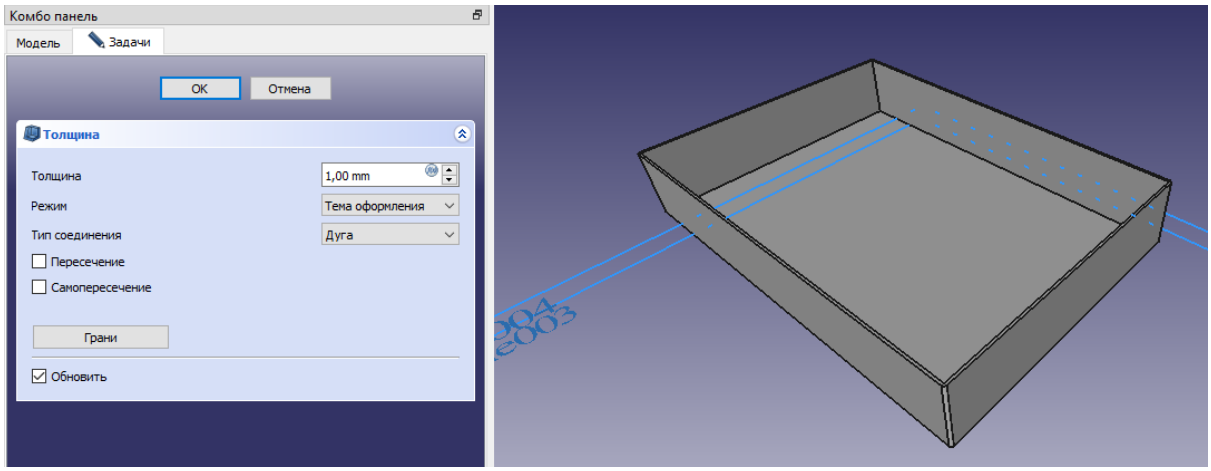
Выберем верстак “Part” и добавим Куб. Если какого-нибудь инструмента нет на виду, то его всегда можно найти в верхней панели, в списке всех инструментов.



Создав куб, задаём ему размеры будущего корпуса в комбо-панели, изменяя параметры Length,Width и Height.

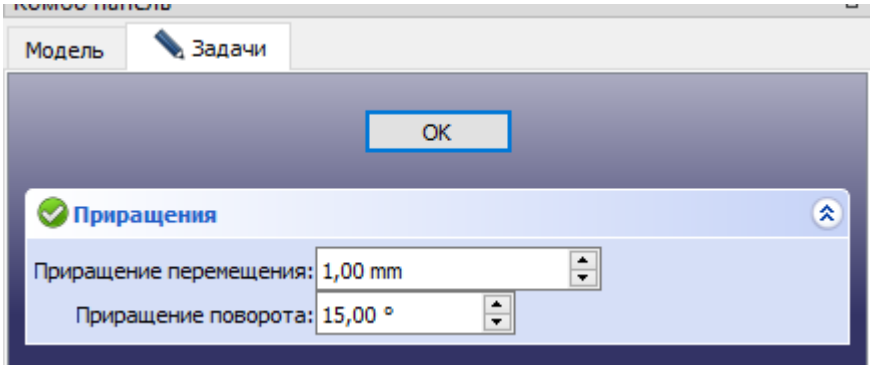


Выделяем верхнюю плоскость куба и ищем инструмент “Утилита смены толщины”



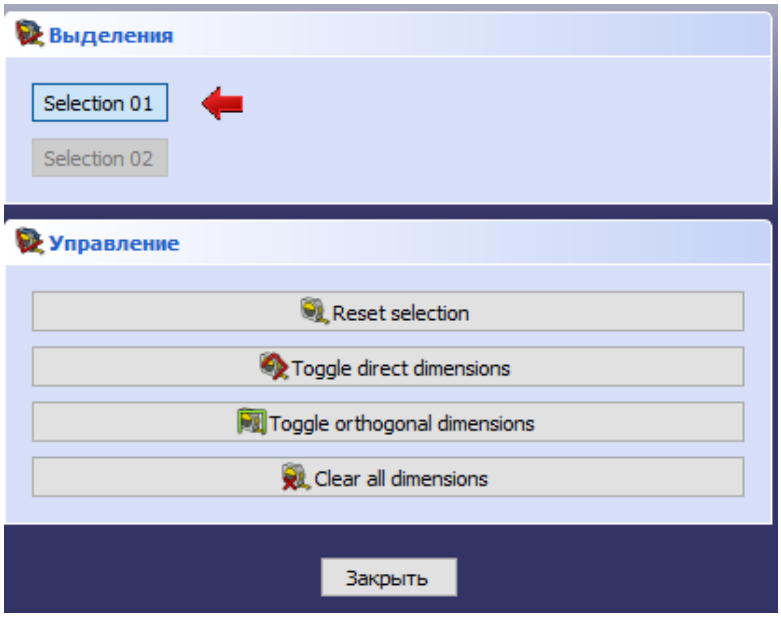
Задаём толщину 1.5 мм.

Добавим отверстия для кнопки питания и портов. Создадим куб, зададим нужные размеры. Правая кнопка мыши по кубу в комбо-панели -> Преобразовать. Приращение перемещения можно уменьшить



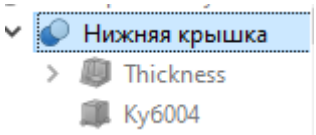
Появившимися стрелками перемещаем куб в место, где мы хотим сделать отверстие для кнопки питания. После можно измерить расстояния от граней куба до сторон корпуса, используя инструмент рулетку

Кликаем по грани или ребру, затем по другому элементу, до которого нужно измерить расстояние. Размеры появятся на экране. Если их нужно удалить, можно нажать “Clear All dimensions”



Формирование отверстия происходит путём булевой операции вычитания. Сначала нажимаем на нашу коробку, затем, с нажатым Ctrl, на куб. После, ищем булеву операцию “Обрезать две фигуры” :

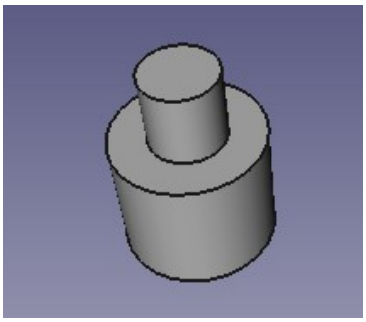
После нажатия, появилось отверстие. Теперь, в комбо-панели, в списке объектов, наш корпус представлен в виде группы из двух объектов под общим именем.



Нажимая пробел, можно, например, скрыть операцию вычитания, и показать, как выглядел объект до этой операции.

Таким же образом добавляем отверстия и для портов.

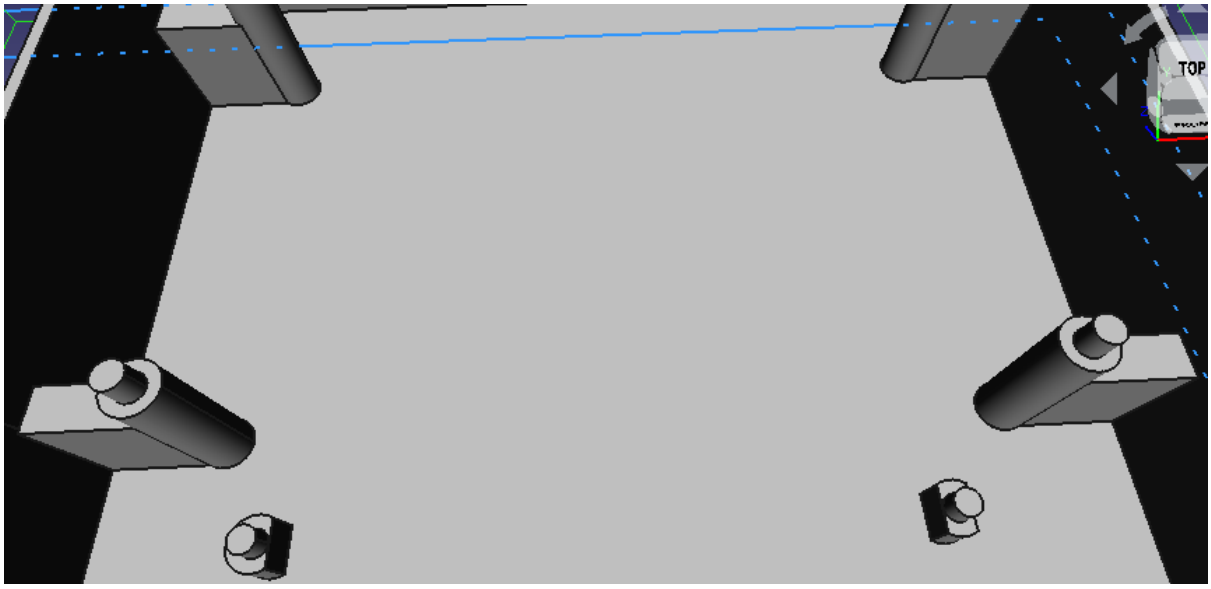
Сделаем стойки для крепления компонентов. Возьмём два цилиндра с различными радиусами и поместим их в одну точку:



Булева операция объединения объединяет под общим именем два разных объекта.

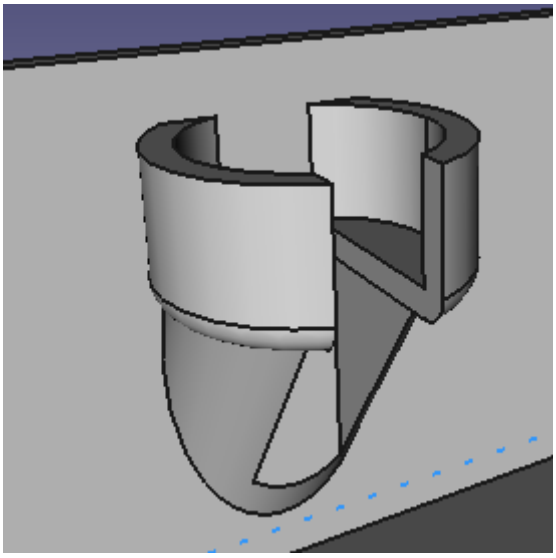
Выделим оба цилиндра и применим к ним эту операцию. Теперь размещаем стойки в нужных местах корпуса.

Стойки для ЖК-дисплея сделаем повыше. Для них также стоит добавить ребро жёсткости для большей надёжности:



Обратите внимание, стойки для платы немного обрезаны. Это сделано для того, чтобы контакты, расположенные рядом с монтажными отверстиями, не мешали насадить плату на стойки.

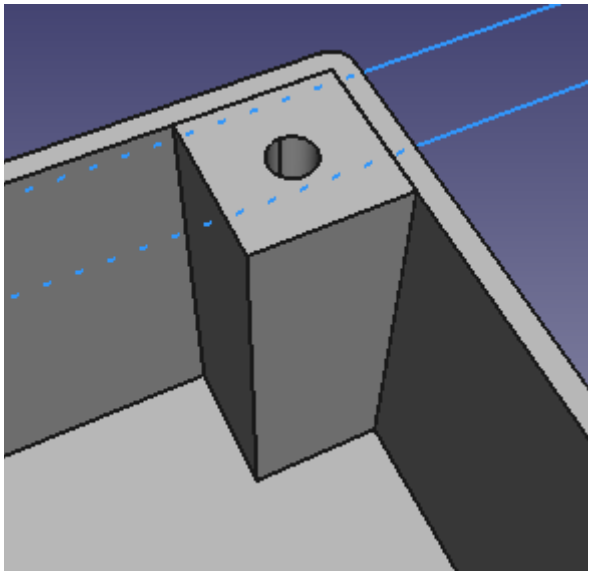
Это - держатель для пьезодинамика:



Чтобы получилась такая же модель, нужно:

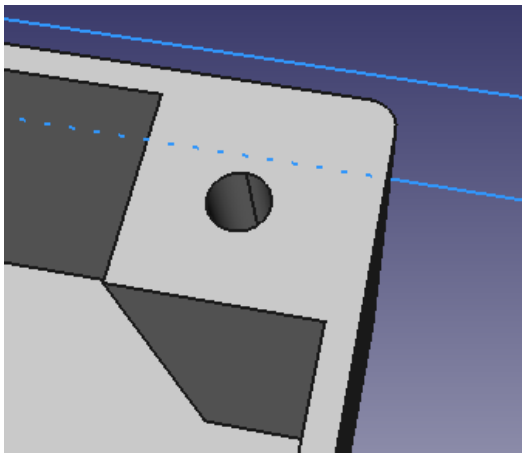
- Добавить цилиндр
- Вырезать цилиндром меньшего радиуса его внутренности
- Чтобы сделать подставку, нужно добавить цилиндр, и наклонить его в сторону держателя
- Объединить все элементы
- Добавить куб, деформировать его до нужного размера, вырезать из объединения прошлых операций этот куб.

По углам корпуса добавить крепления для винтов:

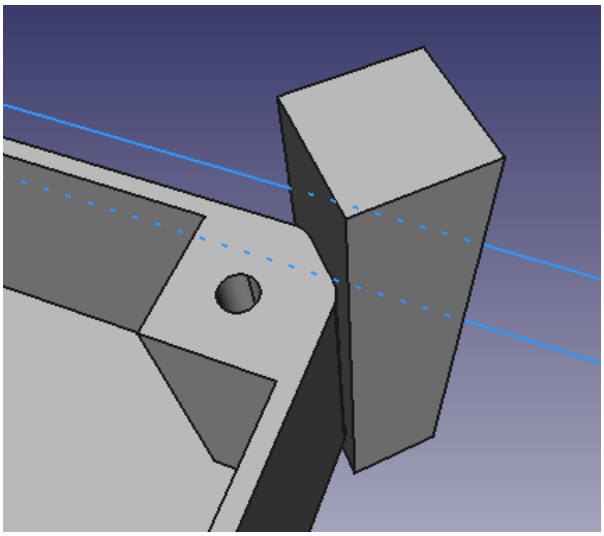


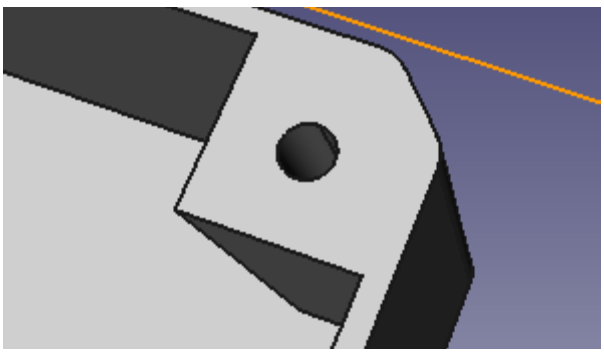
Затем, объединяем крепления, и все остальные отдельные объекты с корпусом. Переходим в верстак “OpenSCAD” и применяем ко всему корпусу инструмент “Create Refine Shape Feature:

Этот инструмент позволяет убрать лишние грани и рёбра, которые находятся внутри других объектов. В результате, получаем следующее:

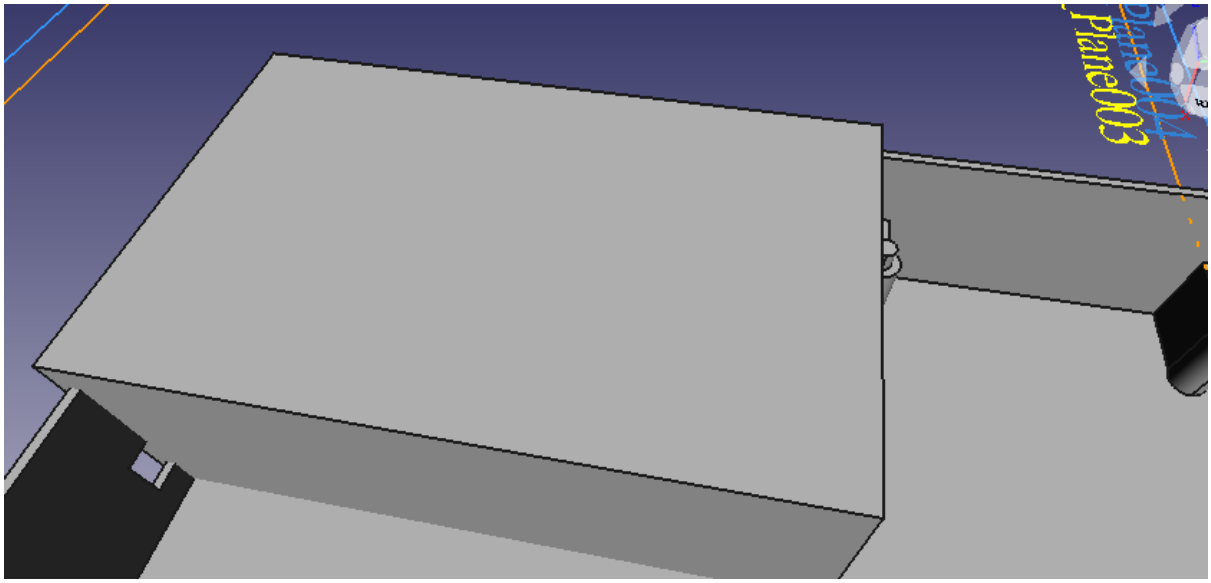


Добавим куб и обрежем угол корпуса:



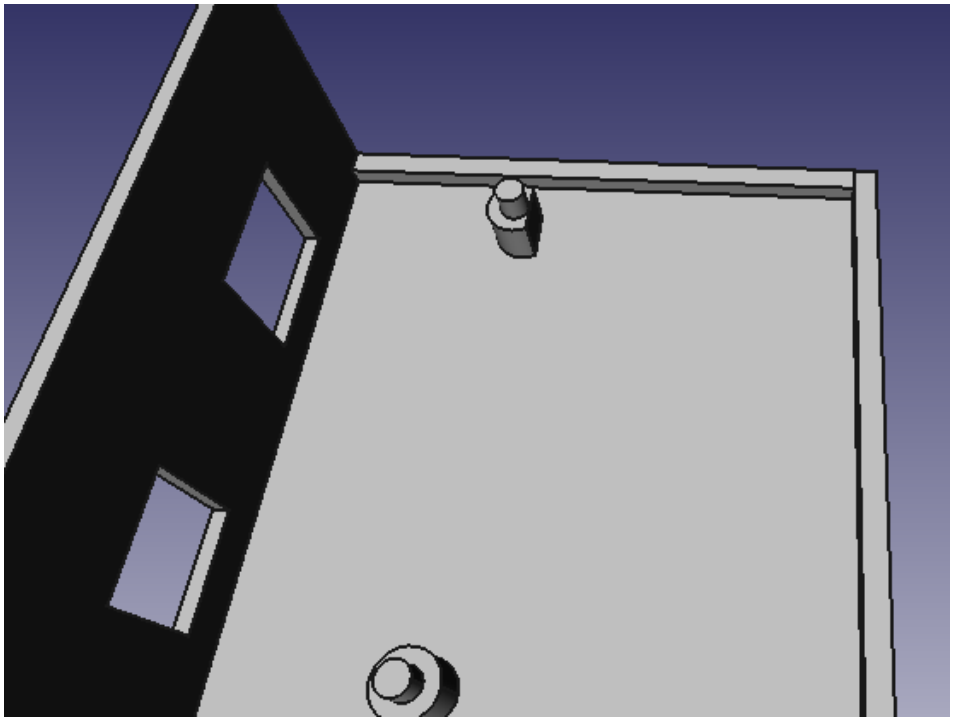


В силу ограничений 3D-принтера, не стоит печатать корпус целиком. Следует поделить его на четыре части и печатать отдельно, чтобы потом склеить. Добавим куб, присвоим его размерам значения, соответствующие четверти размеров корпуса:



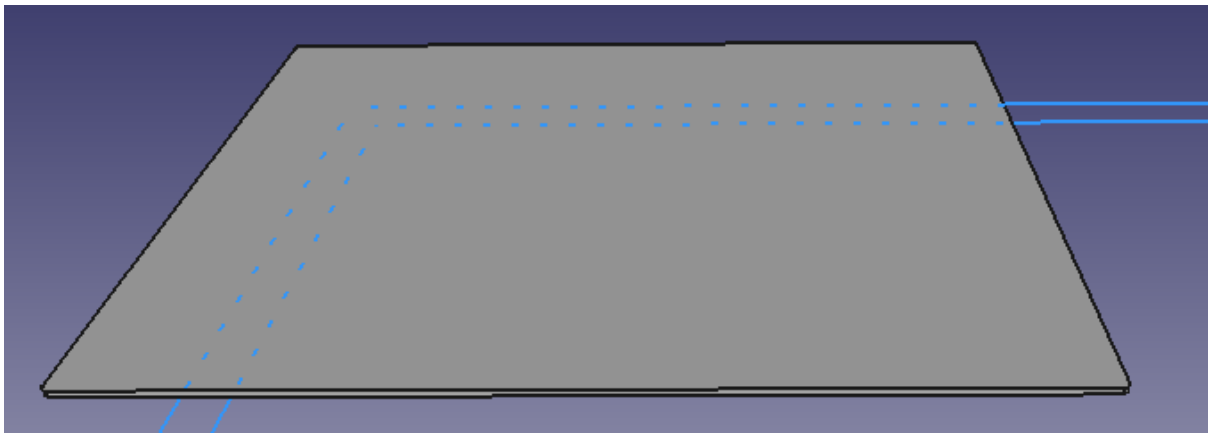
Создадим четыре копии всего корпуса. Затем, выделяем куб и одну из копий корпуса, применяем инструмент создания пересечения:

В результате, получаем четверть корпуса. Повторяем те же действия для создания оставшихся частей. Так как их будем склеивать, то, для больше площади соприкосновения, добавим утолщения на краях:



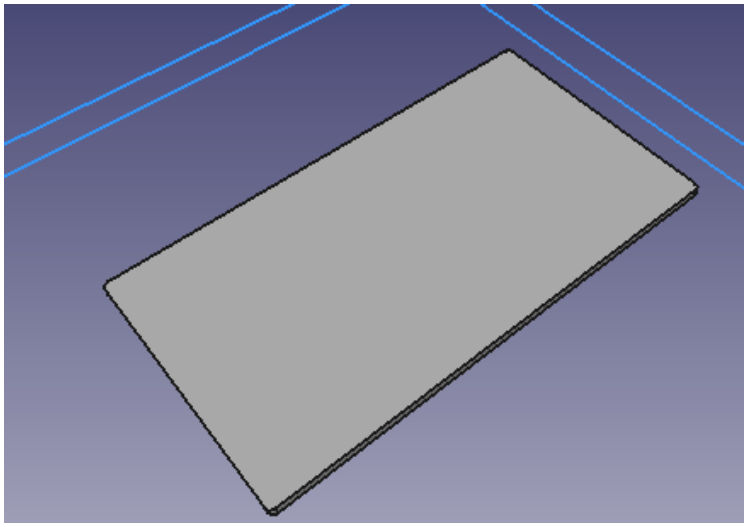
04.04. Крышка

Осталось доделать крышку. Создадим панель из куба нужного размера.

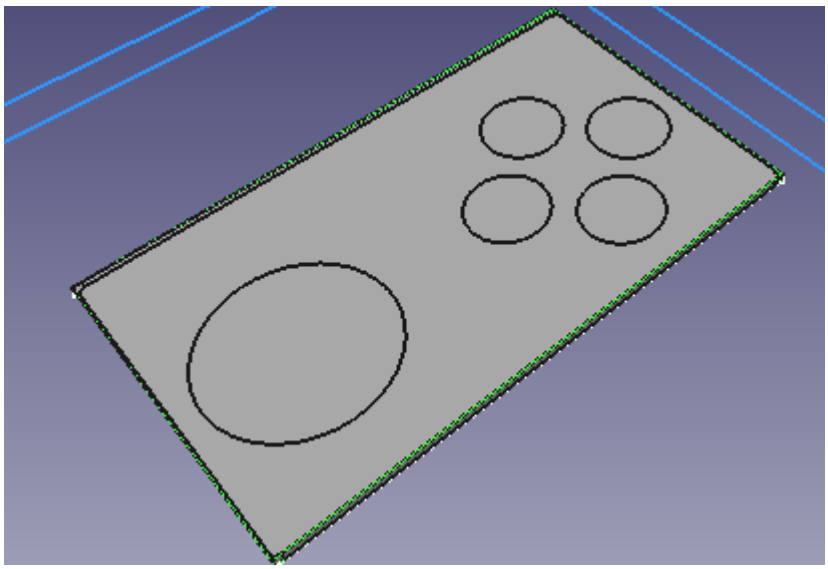


Теперь проведём следующие операции:

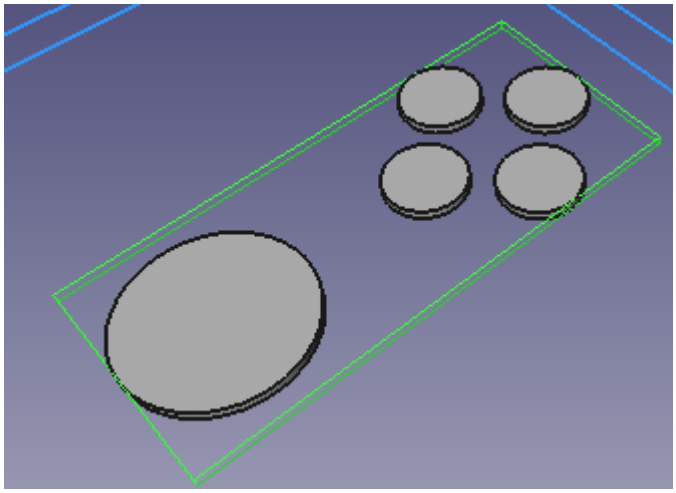
Создадим панель, повторяющую размеры заготовки крышки:



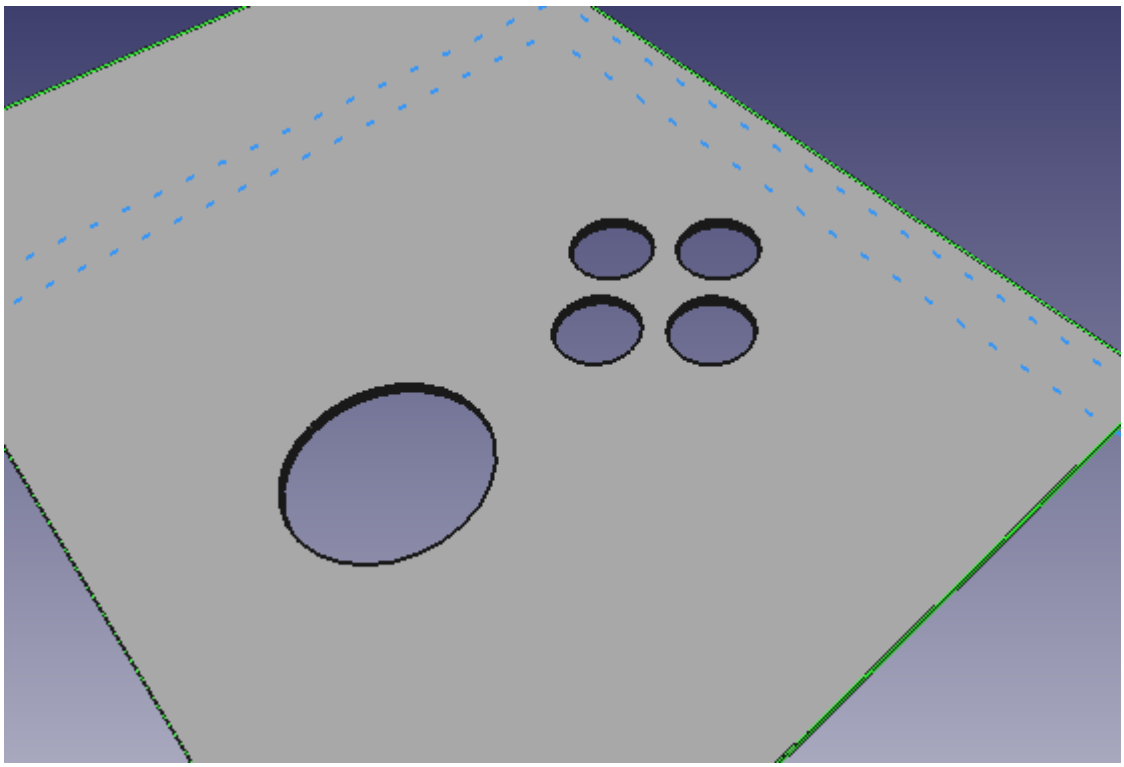
Разместим заготовку крышки и новую панель в одной точке.



Затем, вырезаем из панели заготовку крышки:

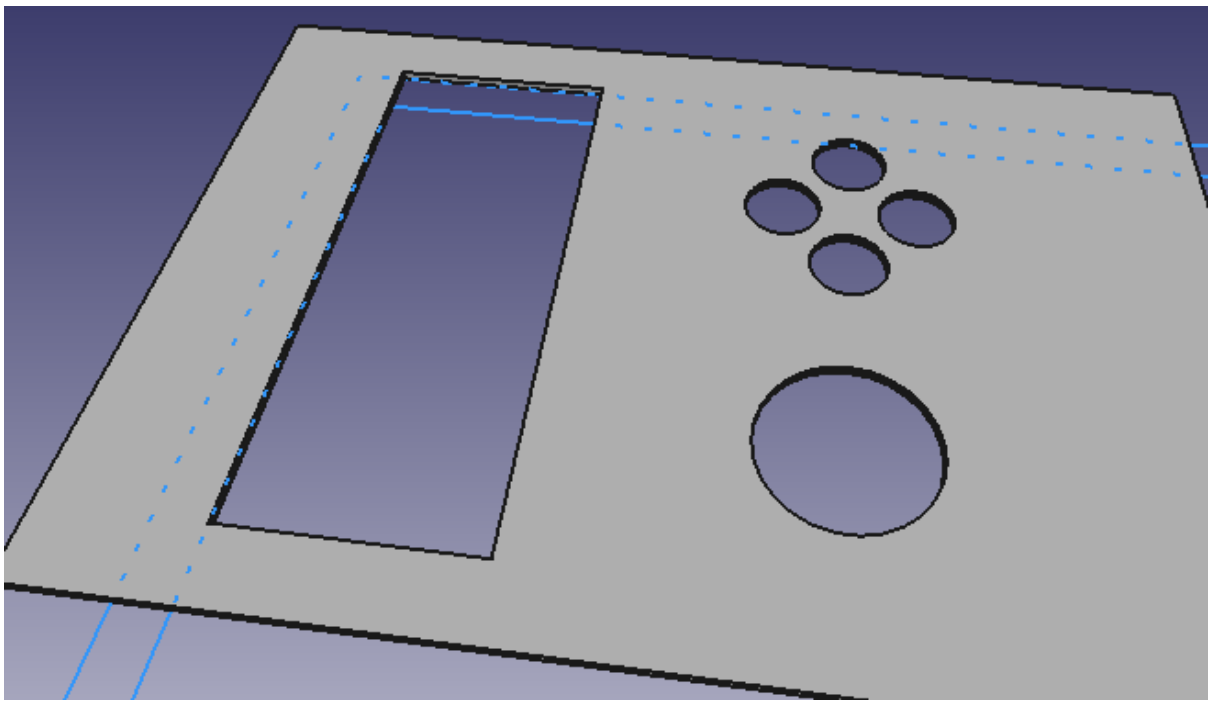


Вырезаем из большой панели объект из прошлой операции

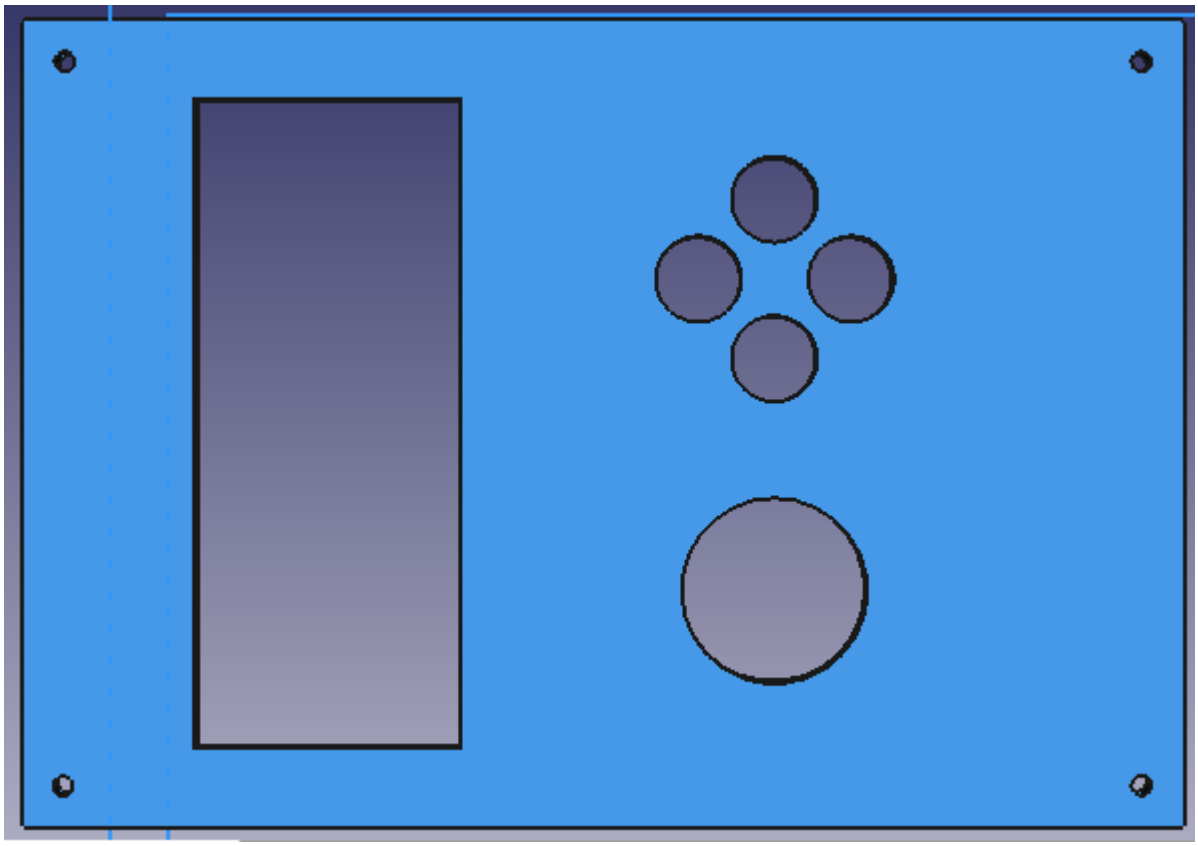


Сам по себе способ с созданием небольшой панели с отверстиями, с последующим переносом отверстий на большую панель не самый эффективный, но для обучения работе во FreeCAD - самое то.

Дальше делаем вырез для дисплея.

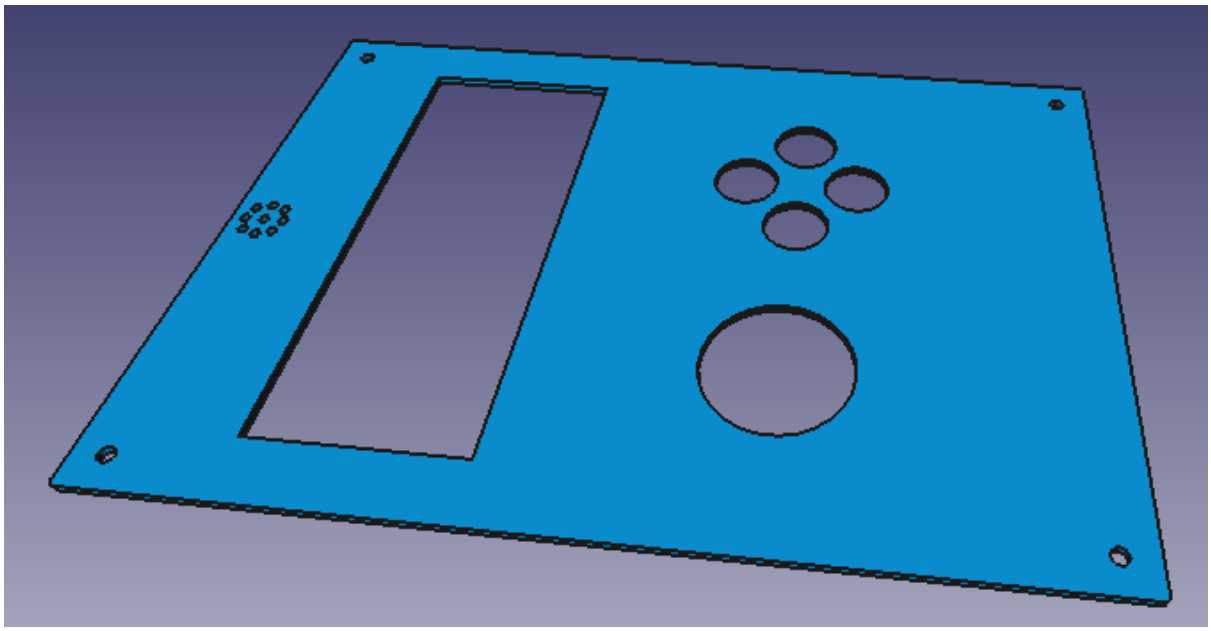


По краям делаем отверстия для винтов:



Сразу бросается в глаза цвет модели, не так ли? Удобно задавать различные цвета деталям, если их большое количество. В верстаке “Part”, в списке объектов ПКМ по нужному объекту -> “Внешний вид”.

Теперь осталось добавить небольшие отверстия для пьезодинамика, используя несколько небольших цилиндров:



Отлично! Корпус готов! Теперь, нужно сконвертировать модель в формат, подходящий для 3D-печати. Выбираем в списке нужный объект -> Файл -> Экспортировать,

выбираем формат файла - stl, указываем место сохранения файла. На этом разработка корпуса закончена.

Литература и интернет источники

Печатные издания

1. Керниган Б. У., Ритчи Д. М. Язык программирования C = C programming language. — 2-е изд. — М.: «Вильямс», 2007. — С. 304. — ISBN 0-13-110362-8
2. Шилдт Г., «C++ для начинающих» – М.: Эком, 2008. – С. 640 – ISBN: 978-5-9900924-9-5.
3. Джереми Блюм, «Изучаем Arduino» – – СПб.: «БХВ-Петербург», 2015 – С. 336 – ISBN 978-5-9775-3585-4.

Online-ресурсы

1. http://arduino-diy.com/arduino-processing-osnovi
2. http://sebeadmin.ru/kak-polzovatsa-multimetrom.html
3. http://arduino.ru/
4. http://pashkevich.me/article/6.html
5. https://uscr.ru/drebezg-kontaktov-i-sposoby-podavleniya-drebezga/
6. http://simparduino.blogspot.ru/2016/08/arduino.html