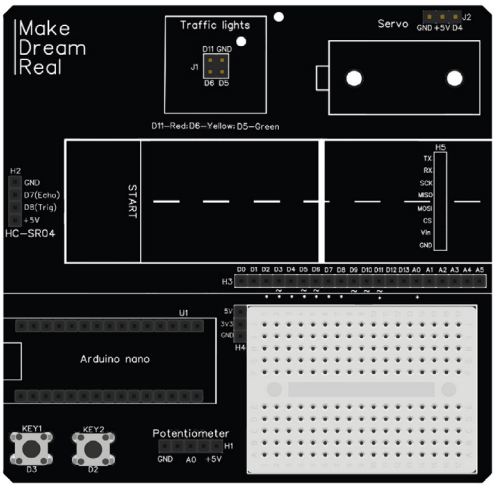
Make
Dream
Real

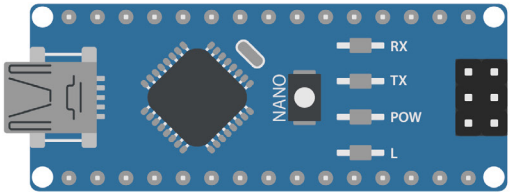
Smart city

Методическое пособие
по программированию и электронике
к набору Smart city

СОСТАВ НАБОРА



Материнская плата

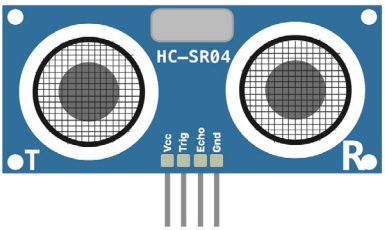


Arduino nano

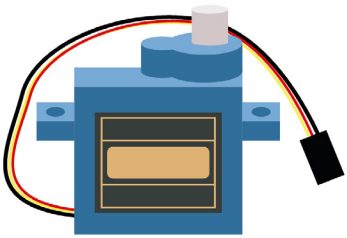
Светодиоды 5мм

Тактовая кнопка

Резисторы 220 Ом
Резистор 10 кОм

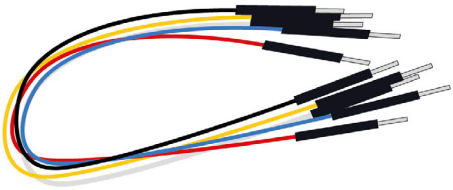


Датчик расстояния
HC-SR04

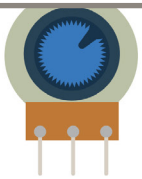


Сервопривод SG90

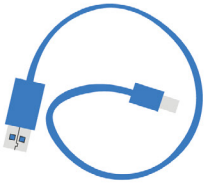
СОСТАВ НАБОРА



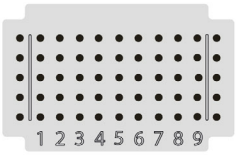
Макетные провода
(папа-папа)



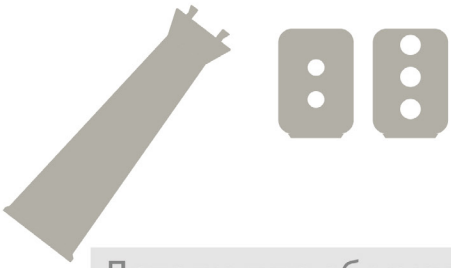
Потенциометр



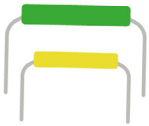
Провод
Mini USB — USB-A



Мини макетная плата



Детали для сборки
светофора



Перемычки

Детали для сборки
шлагбаума

Пинцет

Здравствуй!

Добро пожаловать на курс по набору «Умный город». Здесь наша задача – максимально быстро погрузиться в тему самостоятельного создания электронных устройств. Курс поможет тебе прочувствовать эту сферу и ответить на очень важный вопрос: «А оно мне надо?»
По итогу работы с набором ты самостоятельно соберёшь несколько электронных устройств и научишь их работать совместно.

Перед тем как начнём, дам тебе несколько рекомендаций для максимально эффективного погружения.

Отключай питание при изменении схемы. На курсе мы будем собирать электрические схемы, в том числе тебе и самому точно захочется что-то на ходу поменять. Так вот, при любом изменении в схеме стоит отключить питание, ведь неловкое движение может привести к выходу из строя некоторых компонентов.

Веди записи. Несмотря на то, что ты всегда сможешь вернуться к этому курсу и устранить пробелы, куда более эффективно вести собственный конспект. Такой подход стоит использовать в любом начинании, а не только на уроках в школе или университете, ведь иначе большая часть информации будет потеряна.

Задавай вопросы. Если что-то не получается и остались вопросы, обязательно задавай их в чате или обсуждении в группе социальной сети «ВКонтакте». Наша команда с определённой периодичностью будет проверять их и отвечать.
Но всё-таки, прежде чем писать, полистай выше, вдруг (на самом деле, скорее всего) ответ уже там есть. Что ведёт нас к следующему пункту.

Находи ответы. На самом деле, самым эффективным и полезным способом найти ответ на вопрос является самостоятельный поиск информации из доступных источников. Так ты гораздо лучше усвоишь материал и параллельно освоишь очень важный навык, особенно актуальный в наше время – самостоятельный поиск информации.

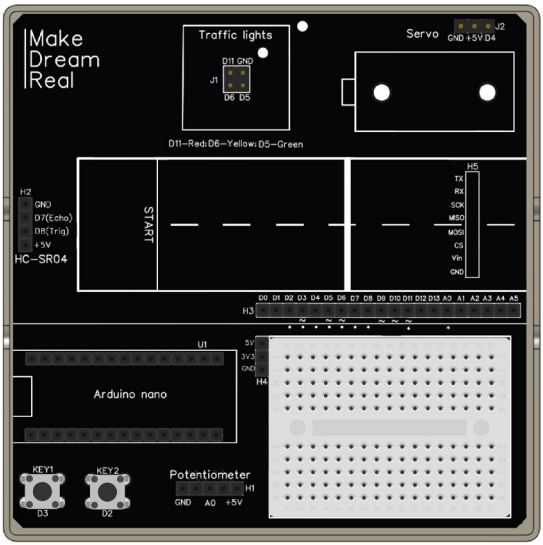
4 **Делай перерывы.** В любой деятельности важно соблюдать режим труда и отдыха. Это, опять же, значительно увеличивает продуктивность. Постарайся найти для себя оптимальное время для концентрации на деле, обычно это от 45 минут до 1,5 часов. После чего делай перерыв, желательно на активную деятельность. Не стоит пытаться дойти до конца урока, если чувствуешь, что устал или информация сегодня не заходит, в таком случае лучше вернуться позже. И уж точно не стоит пытаться поглотить весь курс за один присест.

5 **Не бойся мечтать.** Чем глубже ты будешь погружаться, тем больше будешь понимать, что всё, на самом деле, довольно просто и тебе под силу создать что угодно. Поэтому не бойся мечтать, ведь главная цель всех наших курсов – научить тебя создавать всё, что ты только захочешь.

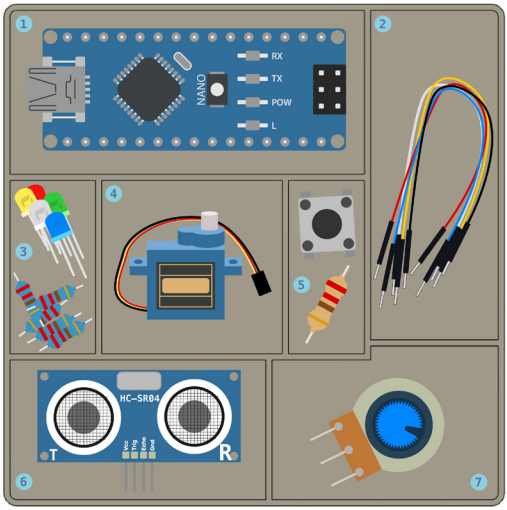
Ну а теперь поехали!

Давай разбираться. Для начала познакомимся с набором.

Откроем пластиковую крышку (кстати, коробка и ещё некоторые вещи в наборе созданы при помощи 3D-принтера). Тут нас сразу встречает материнская плата, на которой мы и будем размещать всё, что ты найдёшь ниже.



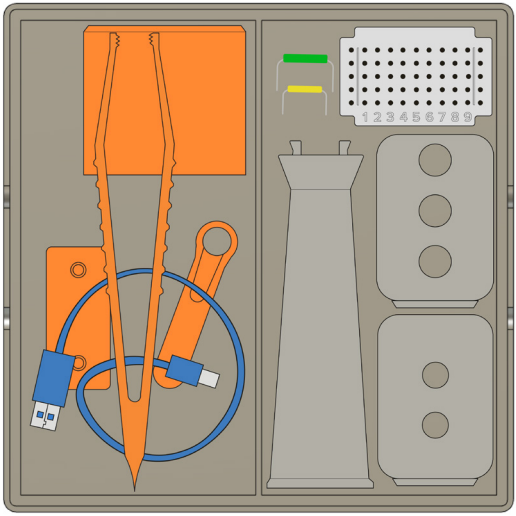
Подробнее с ней мы познакомимся позже. Аккуратно достанем материнскую плату, под ней располагается органайзер.



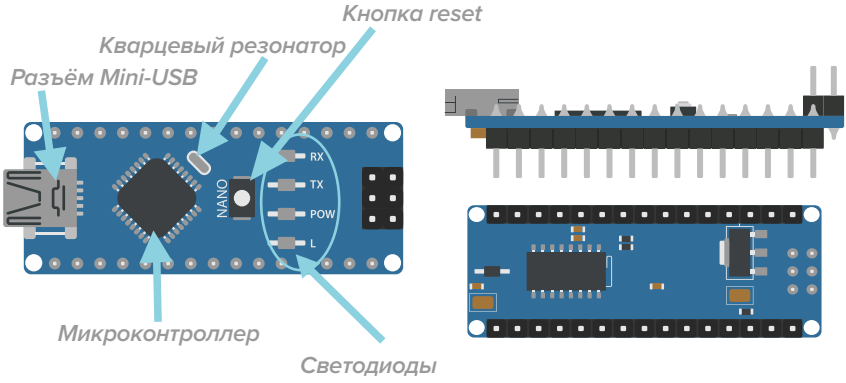
1 — Arduino nano
2 — Макетные провода «папа-папа»
3 — 5 светодиодов 5 мм и 5 резисторов 220 Ом
4 — Сервопривод SG90
5 — Тактовая кнопка и резистор 10 КОм
6 — Датчик расстояния HC-SR04
7 — Потенциометр

В его ячейках находятся все необходимые компоненты. Подробно каждый из них мы рассмотрим на уроках. А пока аккуратно достанем органайзер. Под ним хранятся детали для сборки светофора и шлагбау-

ма, провод для подключения платы к компьютеру и пинцет для удобного извлечения мелких компонентов из органайзера.



Вернёмся к органайзеру. Сегодня мы познакомимся с «сердцем», вернее, «мозгом» набора – платой Arduino Nano. Она находится в этой ячейке в специальном антистатическом пакете. Аккуратно достань её из пакета и давай разглядывать.



Практически в центре платы находится такой чёрный квадрат с ножками по периметру. Это микроконтроллер. Именно эта штука и является «мозгом» всех более-менее автоматизированных устройств (того же светофора, шлагбаума, микроволновки, компьютерной мышки, робота-пылесоса и так далее вплоть до спутников и ракет). По сути дела, его можно назвать микрокомпьютером, ведь внутри него есть ядро (процессор), оперативная память, постоянная память и ещё куча всякой периферии, с которой мы познакомимся позже на уроках. А пока обрати внимание на ножки по периметру микроконтроллера. Правильно эти ножки называть «пины» и именно ими микроконтроллер взаимодействует с окружающим миром: принимает сигналы от датчиков, управляет моторами, лампочками и так далее. Все эти пины выведены по краям платы Ардуино для удобного подключения к ним.

С микроконтроллером более-менее разобрались, теперь посмотрим, что тут на плате есть ещё. Начнём, пожалуй, с кнопочки reset – при нажатии на неё программа начнётся заново. Рядом с ней расположены 4 светодиода, этот RX сигнализирует о приёме данных с компьютера, TX – о передаче данных на компьютер (они будут мигать во время загрузки программы в плату). Далее PWR – сигнализирует о наличии питания, и последний, L – он подключён к пину 13 и мы можем его запрограммировать, чем мы и займёмся на следующем уроке. С другой стороны от кнопки расположен такой блестящий маленький компонент. Это кварцевый резонатор, и именно эта штука определяет, сколько операций в секунду будет выполнять микроконтроллер, в данном случае это 16 МГц или 16 миллионов операций в секунду. Просто как факт: стандартные компьютерные процессоры работают на частоте около 3 ГГц, то есть выполняют около 3 миллиардов операций в секунду.

Идём дальше. На краю платы расположен разъём mini-USB для обмена данными с компьютером и загрузки программ. На другом конце – пины для подключения программатора (устройства для загрузки программы напрямую в микроконтроллер); пока мы только начинаем, не используйте эти пины. На нижней стороне платы расположен ещё один длинный прямоугольник с ножками по краям. Это микросхема, которая обеспечивает обмен данными и загрузку программ по USB. Остальные маленькие штуки – это резисторы, конденсаторы и так далее, они обеспечивают стабильное напряжение питания и защищают важные элементы платы.

На этом мы заканчиваем вводное занятие. На следующем занятии мы установим программное обеспечение (ПО) для работы с платой Arduino.

Первая программа

На этом уроке мы научимся мигать встроенным в плату светодиодом. Но для начала стоит разобраться, где и как мы будем это делать. Для работы с набором потребуется установить всего две программы. Первая – среда разработки Arduino IDE, которая пригодится нам для написания и загрузки программ. Скачать её можно с официального сайта по ссылке:

https://www.arduino.cc/en/software

На странице сайта необходимо пролистать вниз до версии программы Legacy IDE (1.8.X) и скачать версию для вашей операционной системы.

Вторая – драйвер для того, чтобы плата распознавалась компьютером. Скачать его можно по ссылке:

http://makedreamreal.ru/ch340/

Если что-то не получилось, можешь посмотреть подробную инструкцию по установке и устранению неисправностей в группе социальной сети «ВКонтакте», в разделе «Статьи».

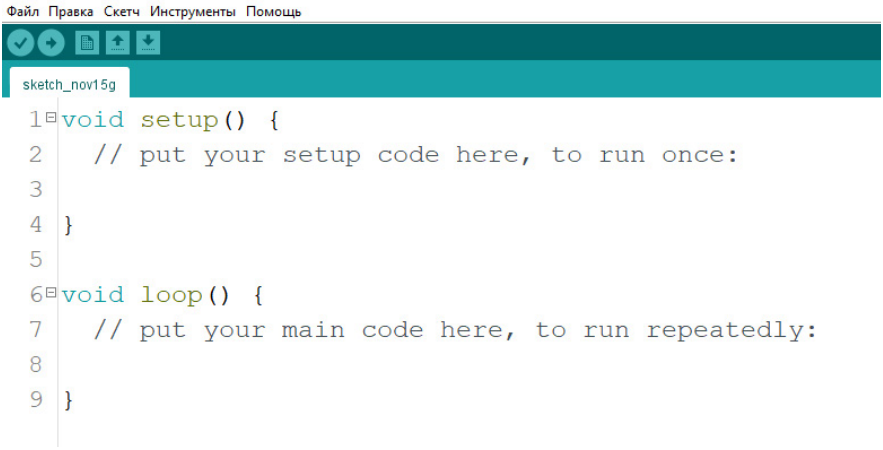
https://vk.com/@makedreamreal-arduino-ide

https://vk.com/@makedreamreal-ch340

Если необходимое ПО уже установлено, время написать нашу первую программу!

Откроем программу Arduino IDE и посмотрим, что она из себя представляет.

IDE расшифровывается как «Интегрированная среда разработки» и в нашем случае представляет собой текстовый документ. Начнём с интерфейса. Это то, как программа выглядит и где расположены её элементы. Сегодня мы рассмотрим только основные возможности, но по мере прохождения курса будем изучать и остальные функции программы.



Скриншот окна программы Arduino IDE

В самом центре расположен текстовый редактор, здесь и будем писать нашу программу. Изначально уже написаны несколько строк, об этом чуть позже. Выше текстового поля расположены важные кнопки:

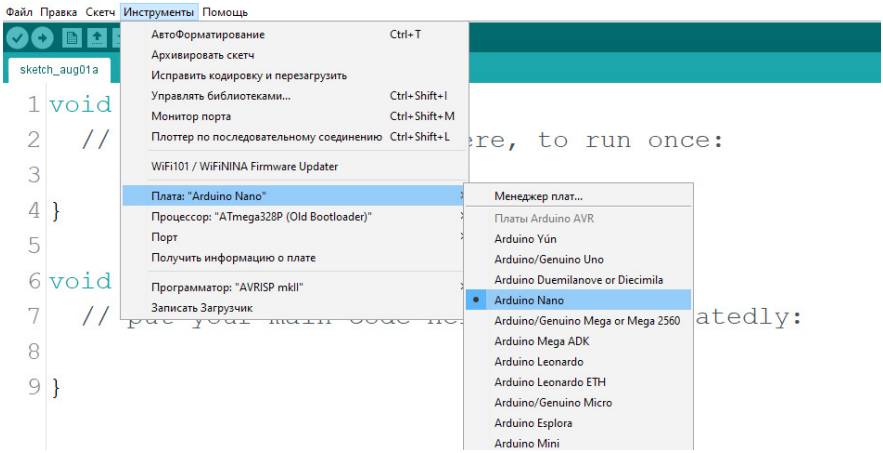
(проверить) – при нажатии на неё начнётся так называемая сборка программы, то есть перевод на машинные команды, понятные микроконтроллеру. Этот процесс называется **компиляция**, а программа, выполняющая сборку, – **компилятор**. В момент компиляции и происходит проверка на ошибки в написанном коде.

Правее находится кнопка **(загрузить)** – при нажатии на неё сначала происходит та самая компиляция, затем ваш код, уже переведённый на понятные контроллеру команды, загружается в микроконтроллер.

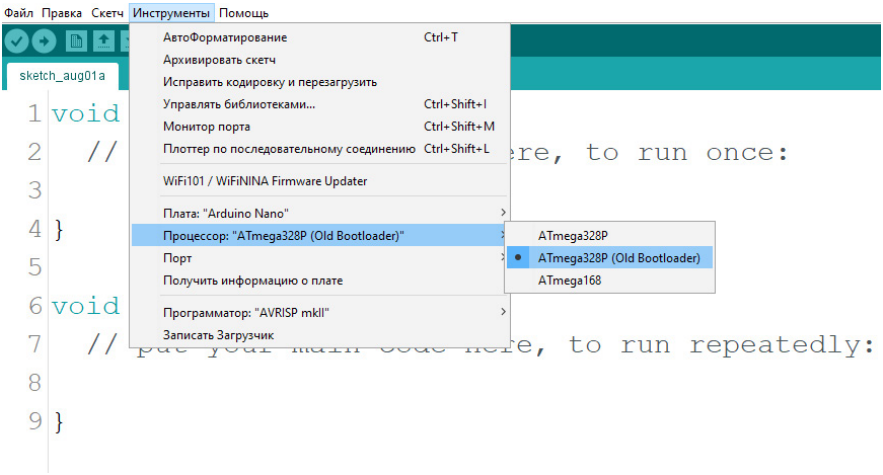
Следом идут уже привычные для большинства программ кнопки: **(новый)** – для создания новой программы, **(открыть)** – для открытия ранее написанных вами программ и **(сохранить)**.

Ещё выше расположены вкладки меню.

Сейчас нас интересует вкладка «Инструменты» и её пункты: «Плата», где среди большого многообразия плат необходимо выбрать Arduino Nano;



далее, снова во вкладке «Инструменты», в пункте «Процессор» выбрать AtMega328P (Old Bootloader)



и последний пункт, который нас интересует, – «Порт». Его необходимо выбрать, когда плата уже подключена к компьютеру, о нём я напомню перед загрузкой вашей первой программы.

В самом низу на чёрном фоне находится так называемый лог выполнения программы. Здесь будут написаны отчёт и выявленные

ошибки. А чуть выше на зелёно-синей полоске – статус. Нажмём, например, на кнопку «Проверить». И увидим, что статус – ***компиляция завершена***, а в логе появилась информация о занимаемой и доступной памяти.

В нижнем правом углу указаны выбранная плата (информация лога справедлива именно для указанной тут платы) и порт.

Так, с интерфейсом примерно разобрались. Казалось бы, можно наконец программировать! Но давай сначала ответим на вопрос «А что вообще такое программирование?»

Программирование – это, по сути дела, написание конкретных действий (команд или инструкций), которые в нашем случае должен выполнить микроконтроллер. Давай разбираться на примере конкретной задачи.

Итак, наша цель сейчас – заставить мигать встроенный светодиод. Первым делом перед написанием программы необходимо придумать алгоритм её работы. Алгоритм – это последовательность действий.

Какой алгоритм необходим для выполнения задачи?

Мы сказали, что алгоритм – это последовательность действий, то есть сейчас нам необходимо назвать конкретные действия, чтобы заставить лампочку мигать. Первое действие – включить лампочку, далее, казалось бы, выключить и начать сначала. Алгоритм это? Да. Но какой будет результат работы такого алгоритма? Будет ли светодиод в этом случае мигать? Как ни странно, да! Но, на самом деле, мы будем видеть, как лампочка просто горит, причём горит вполсилы. Почему именно вполсилы, выясним чуть позже, но почему мы не будем видеть мигания, если они есть? Всё дело в количестве операций в секунду, которое выполняет микроконтроллер. На самом первом занятии мы выяснили, что микроконтроллер на плате Ардуино Нано выполняет 16 миллионов операций в секунду! Как ни странно, на то, чтобы зажечь или потушить лампочку, уходит больше, чем одна операция (об этом как-нибудь в другой раз), но, так или иначе, нетрудно догадаться, что мигать эта лампочка будет сотни тысяч раз в секунду, а то и больше. Наш человеческий глаз способен различить всего лишь 24 кадра в секунду, то есть не более 24 вспышек лампочки за секунду. А тут сотни тысяч! Так, к чему это я… Ах да, так что же необходимо добавить в наш алгоритм? Задержку! И тогда получится: включить, подождать (например, секунду), выключить, подождать (ещё, например, секунду) и начать сначала.

Это алгоритм просто мигания лампочки, теперь превратим его в алгоритм, который будет выполнять микроконтроллер. Чтобы включить или выключить светодиод, нужно записать либо высокий, либо низкий сигнал на ножку 13. Ведь, как мы уже выяснили ранее, именно к ножке номер 13 подключён встроенный в плату светодиод. Почему именно записать? Дело в том, что в микроконтроллере отдельные ножки рассма-

триваются как отдельные ячейки памяти. Так, что-то мы уже куда-то в дебри лезем, вернёмся к алгоритму. Теперь наш алгоритм будет выглядеть так: записать в ножку 13 высокий сигнал; ничего не делать 1 000 миллисекунд («Ардуинка» засекает задержку именно в миллисекундах); записать низкий сигнал в ножку 13; ничего не делать 1 000 миллисекунд. И всё это в бесконечном цикле.

Последнее, с чем осталось разобраться, – это одна-единственная формальность, необходимая для работы микроконтроллера. Дело в том, что этому самому микроконтроллеру очень важно знать, как работать с ножкой – на вход или на выход. То есть ножка будет отдавать сигнал или получать. Попробуй самостоятельно ответить, как будет работать ножка 13 в нашей программе – отдавать сигнал лампочке (то есть на выход) или получать его (то есть на вход).

Ножка будет отдавать сигнал светодиоду, то есть на выход. Об этом микроконтроллеру нужно написать только один раз в самом начале выполнения программы. Так нужно делать для каждой ножки, которую мы будем использовать, и важно об этом не забывать, иначе программа может работать неправильно.

Вот мы и разобрались с алгоритмом, теперь можем переходить непосредственно к написанию нашей первой программы. Как я говорил ранее, делать это мы будем в текстовом редакторе в самом центре окна программы Arduino IDE. Сразу скажу, писать код мы будем на языке программирования C++. Для начала давай разберёмся с тем, что там уже понаписано.

```
void setup( ) {
  //put your setup code here, to run once:

}

void loop( ) {
  //put your main code here, to run repeatedly:

}
```

Итак, изначально код состоит из двух похожих конструкций, называемых **функциями**. У каждой функции есть так называемое **тело**, оно обозначается фигурными скобками. Именно тут мы и будем писать команды нашего алгоритма. У функций есть имена. Верхнюю зовут ***setup***, а нижнюю – ***loop***. Чем же отличаются эти две функции? Сейчас внимательно. Тело функции setup выполняется только один раз в самом начале программы, а тело функции loop выполняется бесконечно по кругу и остановится только при отключении питания или при специальных режимах работы микроконтроллера. Для простоты приведу ещё пример. Эти самые функции можно представить как два листочка бумаги. На листочке setup написано то, что нужно сделать один раз в самом начале, а на листочке loop – то, что нужно повторять. Фигурные скобочки как бы указывают границы этих листочков.

Об остальных словах и символах мы поговорим позже, чтобы не перегружать урок. Единственное, о чём ещё необходимо сказать, – это комментарии. Они начинаются значком «//» и выделены серым цветом. Всё, что написано в строчку после этого значка, программой-компилятором не учитывается и служит только для того, кто программу пишет. Это своего рода заметки, которыми не стоит пренебрегать, но об этом тоже позже. Сейчас можешь их удалить.

Вот, наконец, мы и можем приступить к написанию кода. Первым делом вспомним наш алгоритм:

ножка 13 работает на выход → начало цикла → записать в ножку 13 высокий сигнал → ничего не делать 1 000 миллисекунд → написать низкий сигнал в ножку 13 → ничего не делать 1 000 миллисекунд → перейти к началу цикла.

Думаю, ты догадался, что работу цикла берёт на себя функция loop. Что же касается остальных команд, приступим, глядя на алгоритм.

Нужно сказать, что ножка работает на выход. Сделать это надо один раз, в самом начале программы, в теле функции ***setup*** (так как оно как раз выполняется один раз в начале программы).

Команда будет выглядеть так:

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

Расшифруем. *pin* – так правильно называть ножки «Ардуинки». Далее слитно, с заглавной буквы *Mode* – переводится как «режим работы». Затем в круглых скобочках, через запятую указываем, какая ножка как работает. Ножка у нас 13, работает на выход – пишем большими буквами *OUTPUT*. В конце каждой команды в языке C++ необходимо ставить точку с запятой. Ещё очень важный момент из языка C++: заглавные и строчные буквы – это разные буквы, и команды и ключевые слова важно писать, обращая на это внимание. У нас получилось: режим работы ножки 13 на выход.

Далее по алгоритму программа идёт в цикле, соответственно, будем

писать команды в теле функции ***loop*** (так как тело функции ***loop*** выполняется в бесконечном цикле).

Судя по алгоритму, нам надо записать высокий сигнал в пин 13.

Команда будет выглядеть так:

digitalWrite(13, HIGH);

digital – переводится как «цифровой», то есть либо 0, либо единичка. В нашем случае 1. Подробнее о цифровом сигнале будет в следующих уроках. Далее слитно, с большой буквы *Write* – переводится как «писать» или в нашем случае «записать». В круглых скобочках – в какую ножку какой сигнал. Ножка 13, сигнал высокий – пишем большими буквами *HIGH*. Не забыли поставить точку с запятой. Получилось: в ножку 13 записать высокий цифровой сигнал.

Следующее действие – это задержка на одну секунду. Команда выглядит так: *delay(1000)*, переводится как «задержка», в скобочках – время в миллисекундах. Не забыли точку с запятой.

Вот что получилось:

```
void setup( ) {
  pinMode(13, OUTPUT);
}
void loop( ) {
  digitalWrite(13, HIGH);
  delay(1000);
}
```

Прежде чем двигаться дальше, попробуй дописать программу самостоятельно, зная, что низкий сигнал пишется как LOW.

Я просто скопирую уже написанные в теле функции loop команды и вставлю их ниже. Изменю только HIGH на LOW.

```
void setup( ) {
  pinMode(13, OUTPUT);
}
void loop( ) {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

Всё готово! Прочитаем, что у нас получилось. Один раз, в начале программы – режим работы ножки 13 на выход. Далее в бесконечном цикле – в ножку 13 записать высокий цифровой сигнал, задержка 1 000 миллисекунд, в ножку 13 записать низкий цифровой сигнал, задержка 1 000 миллисекунд.

Теперь давай загрузим программу в микроконтроллер. В начале урока мы уже настроили Arduino IDE для работы с платой Arduino Nano. Единственное, что осталось сделать, – это указать, к какому порту подключилась наша плата. Открой пункт меню «Инструменты», выбери пункт «Порт». Если он серенький и не нажимается, то оставь – всё хорошо. Если нет, запомни те порты, что там есть, ткни в свободное место, чтобы закрыть «Инструменты». Теперь подключи плату к компьютеру проводом USB-A — mini-USB. Немного подождём, пока плата определится компьютером. Снова заходим в «Инструменты», «Порт» и выбираем тот, что появился. Теперь можем нажимать кнопку «Загрузить» – происходит компиляция, загрузка, и-и-и-и... Ура, магия, да и только!

Если вдруг порт не определился или вылезли ошибки, посмотри инструкцию, приложенную ниже. А так – поздравляю! Первая программа готова и работает! В качестве домашнего задания можешь поиграть со значением задержки или как-то усложнить мигание – например, отправить сообщение SOS азбукой Морзе.

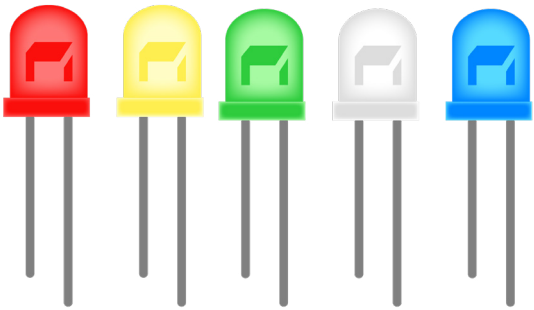
Светофор

Сегодня мы начнём двигаться к созданию нашего первого устройства – светофора.

На прошлом уроке мы уже научились мигать встроенным светодиодом, а сейчас пришло время подключить внешний светодиод.

Для начала ещё раз напомню про важность ведения самостоятельных записей, а сегодня будет, что законспектировать! Это очень поможет в понимании материала.

Итак, светодиод. В твоём наборе их целых 5, находятся они в одной ячейке, вместе с ещё очень важными штуками – резисторами, о них поговорим чуть позже.



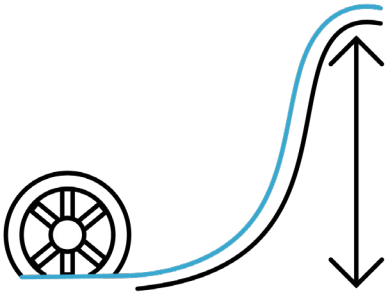
Для наших первых экспериментов лучше использовать светодиоды белого или синего цвета, чтобы не повредить те, которые будем использовать для сборки светофора. Возьми светодиод и попробуй самостоятельно ответить на вопросы. Для чего вообще они применяются? Где используются? Для удобного извлечения мелких деталей из органайзера используй пинцет.

Помимо освещения, из них создают целые экраны, расположенные, например, вдоль дорог, дорожные знаки, и современные светофоры, конечно же, включают, как правило, не один, а целую группу светодиодов на каждый цвет. Ну и одно из важнейших применений светодиодов в современной электронике – индикация. Взглянув практически на любое электронное устройство, можно обнаружить там светодиоды, отвечающие за отображение наличия электропитания, статус выполнения встроенных функций и так далее.

Для начала нужно разобраться с подключением светодиода. А для этого придётся немного окунуться в физику. Уверен, у тебя на слуху такие понятия, как **ток**, **напряжение** и **сопротивление**. Как минимум в школе, примерно в 8-м классе, с этими понятиями знакомят. И, возмож-

но, в голове ещё даже остались какие-то заученные формулы и определения из учебника. Так или иначе, сейчас наша задача – **не заучивать всякие умные формулировки, а прежде всего понять, почувствовать и осознать,** что всё-таки скрывается за этими до боли знакомыми, но такими непонятными словами.

Представим себе гору, с которой срывается водопад. У подножия горы установим водяную мельницу (в этом сравнении она заменит нам светодиод).



Чтобы было проще, немножко упростим физику водопада и примем, что чем выше источник, тем сильнее будет раскручиваться водяная мельница. Итак, пользуясь уже существующими представлениям у тебя в голове, попробуй догадаться, где в этой метафоре ток, напряжение и сопротивление. Снова рекомендую постараться порассуждать самостоятельно.

Одно из умных определений напряжения гласит: напряжение – это разность электрических потенциалов между двумя точками. Понятнее, думаю, не стало. Чтобы разобраться с этим определением и найти напряжение в нашей иллюстрации, придётся понять, что такое этот электрический потенциал. Тут уже обойдёмся без точных определений и снова прибегнем к сравнению.

Надеюсь, что в школе ты уже слышал про потенциальную и кинетическую энергии. Так или иначе, бегло напомню. У нас сейчас нет задачи глубоко погружаться в физику и определения, а целью является только общее понимание и представление, поэтому обойдёмся без формул.

Начнём. Возьми ручку или карандаш. Подними на небольшое расстояние над столом. Потенциально, объект может упасть. И что тогда будет? Он наберёт скорость и ударится о стол. Получается, сейчас у него есть какая-то запасённая энергия или, как её правильно называть, потенциальная энергия. Помимо прочего, зависит она от высоты. Не будем сейчас вдаваться в физику 7-го класса, а просто позволим себе упрощение потенциальной энергии до слова «потенциал».

Так вот, вернёмся к нашему водопаду и присмотримся к потенциалу воды относительно уровня земли. На вершине он больше нуля и завсит высоты, а на уровне земли вся потенциальная энергия превратилась в кинетическую, которая теперь тратится на вращение мельницы.

Вспомним определение напряжения: напряжение — это разность электрических потенциалов между двумя точками. Что в нашем сравнении — напряжение? Если упростить потенциальную энергию до слова «потенциал», станет очевидно: напряжение здесь — высота горы (источника)!

Теперь ток. По определению, ток — это движение заряженных частиц. И здесь всё просто: в нашей метафоре его представляет водопад (вода). Но, как правило, нас интересует не ток, а величина, называемая силой тока. Если упростить определение и перенести его в наше сравнение, то сила тока (сила воды) — это количество воды, ударяющейся о лопасти водяной мельницы за секунду. И тут уже несложно догадаться, что чем выше источник, тем быстрее разгоняется вода и тем большее её количество успевает толкнуть лопасти водяной мельницы за секунду.

Теперь представим, что мы не рассчитали, что гора окажется настолько высокой, когда ставили мельницу, и стало понятно, что вода слишком сильно бьёт по лопастям и наша конструкция так долго не выдержит, или в случае электричества напряжение слишком велико и сила тока, соответственно, тоже, что уменьшает срок службы светодиода. Что делать?

Необходимо добавить сопротивление. В случае водопада можно построить стенку или просто накидать камней, которые будут забирать часть энергии, а в случае электрической цепи необходимо добавить резистор. Чем больше сопротивление, тем больше энергии будет поглощено резистором. Это эквивалентно тому, как если бы мы уменьшили напряжение на соответствующую величину (по сути, уменьшили бы высоту горы). То есть потенциал после резистора меньше, чем перед ним. Соответственно, на его концах можно измерить напряжение. Это изменение потенциала называется падением напряжения.

Разумеется, сопротивление резистора необходимо подобрать подходящее, и как это сделать, мы разберём в следующей части урока.

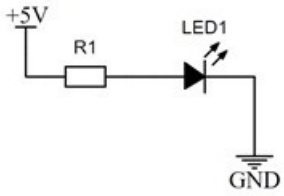
Принципиальная электрическая схема и расчёт резистора

Сейчас мы соберём первую простенькую электрическую схему, основываясь на понятиях, изученных ранее.

Напомню: мы плавно движемся к созданию первого электронного устройства, и сейчас наша задача – подключить только один светодиод (лампочку).

Начнём с принципиальной электрической схемы нашего устройства. На такой схеме в развёрнутом виде отражают все электрические соединения необходимых компонентов.

Для начала «умное» – графическое обозначение будет выглядеть так:



Электрическая схема подключения светодиода

Разберём его подробнее. На самом деле, если вспомнить аналогию про водопад и водяную мельницу, всё довольно просто, а главное – очень похоже.

В верхнем левом углу такой символ: 

. В нашем случае это источник сигнала с потенциалом, равным 5 вольт. По сути дела, это плюс, как у батарейки. (Кстати, вольт – это единица измерения напряжения. Ну, как метр у расстояния.) Как мы помним, напряжение – это разность потенциалов. То есть сразу вопрос: 5 вольт относительно чего? Относительно, как ни странно, земли. Обрати внимание в правый нижний угол: там такой значок 

с обозначением GND (от английского ground – земля). Это минус нашей схемы – опять же, как у батарейки. Светодиод на схеме обозначен так:  .

На его обозначении остановимся подробнее. Можно назвать «стрелочка-стеночка». Вспомним, что ток – это движение заряженных частиц. В каком направлении идёт ток? От плюса к минусу или от минуса к плюсу?

Общепринято направление от плюса к минусу. И в случае подключения светодиода это очень важно. На схеме он подключён стрелочкой в направлении тока, а в обратном направлении путь как бы преграждает стеночка.

Обрати внимание на любой светодиод из твоего набора: одна из его «ног» длиннее, и это неспроста. Та, что длиннее, – это плюс, а та, что короче, – минус. Но что будет, если подключить светодиод не в том направлении? Помнишь про стрелочку-стеночку? Так вот, в этом случае светодиод просто будет препятствовать протеканию тока, и всё. В схеме в этом месте будет будто разрыв цепи. Если, конечно, напряжение не окажется слишком уж большим – тогда условная стеночка просто не выдержит и светодиод сгорит или даже взорвётся! В нашем же случае при неправильной полярности светодиода ничего плохого не произойдёт.

Итак, теперь самое сложное, сосредоточимся. Светодиод питается не напряжением, а током, то есть, чем выше ток, тем ярче светодиод светится. И, разумеется, предел этому есть, он указан в документации на конкретный светодиод, но чаще всего для таких вот пятимиллиметровых светодиодов не превышает 20 мА.

И тут мы подбираемся к самому интересному.

Закон Ома! Прости, совсем уж без формул нельзя, но обещаю, тут нет ничего сложного.

> **К слову, ток измеряют в амперах, а приставка «милли» означает деление на тысячу. Также она справедлива для всех величин – например, как в случае с метром и миллиметром**

Итак, закон Ома выглядит так:

$$I=\frac{U}{R} \qquad R=\frac{U}{I} \qquad U=IR$$

Он содержит всего три буквенных обозначения: **U** – напряжение, **I** – ток, **R** – сопротивление. И самое главное, что сходу нужно понять, – закон показывает отношения всех этих трёх величин друг к другу.

Например. Приглядимся ко второй формуле. Если напряжение увеличивать (высота горы увеличивается), а сопротивление не изменять, то что произойдёт с током? Исходя из метафоры с водопадом, мы помним, что ток тоже увеличивается, а, исходя из математики, если левая часть равенства увеличивается, то и правая тоже должна увеличиваться. И на-

оборот. Можешь самостоятельно немного поиграть с этой формулой, не забывая сверяться со сравнением про водопад.

Следующее, что очень важно, – благодаря только этой формуле мы можем рассчитать необходимое сопротивление, которое нужно добавить в схему, чтобы светодиод не сгорел.

Мы уже знакомились с таким понятием, как падение напряжения. Так вот, на светодиоде падение напряжения тоже происходит. Зависит оно, как ни странно, от цвета светодиода, но в дебри опускаться не будем, ограничимся лишь тем, что минимальным падением напряжения обладает красный светодиод – примерно 1,8 вольта, на это значение и будем опираться для всех. К слову сказать, красный светодиод можно питать напрямую напряжением 1,8 вольта без всяких резисторов, но напряжение Arduino Nano составляет 5 В.

Теперь можно рассчитывать необходимое сопротивление для безопасной работы нашей схемы.

Итак, $R=\frac{U}{I}$

U = напряжение питания – падение напряжения на светодиоде =

= 5 – 1,8 = 3,2 В.

Ток = максимальный для светодиода = 20 мА = 0,02 А.

Выходит,

R = 3,2 / 0,02 = 160 Ом (единица измерения сопротивления Ом). Мы получили минимально допустимое сопротивление, которое необходимо использовать при подключении красного светодиода, при котором он покажет свою максимальную яркость. Для остальных это значение, как правило, меньше, но пока мы только начинаем, примем 160 Ом как минимально допустимое, чтобы избежать ошибок. Чем больше будет сопротивление, тем меньше окажется яркость светодиода, поэтому в твоём наборе лежат резисторы на 220 Ом для более комфортной работы. Кстати, обрати внимание на разноцветные полосы на корпусе резистора: в этих цветах и зашифровано его номинальное сопротивление. Можешь попробовать самостоятельно расшифровать по специальной таблице.

4-х полосный резистор
220Ω ±5%

Четырёхполосные резисторы имеют две полосы сопротивления, одну полосу множителя и одну полосу погрешности.

	1 полоса	2 полоса	Множитель	Допустимое отклонение
чёрный	0	X	X1Ω	
коричневый	1	1	X10Ω	±1%
красный	2	2	X100Ω	±2%
оранжевый	3	3	X1кΩ	
жёлтый	4	4	X10кΩ	
зелёный	5	5	X100кΩ	±0.5%
синий	6	6	X1МΩ	±0.25%
фиолетовый	7	7	X10МΩ	±0.10%
серый	8	8		±0.05%
белый	9	9		
золотой	X	X	X0.1Ω	±5%
серебряный	X	X	X0.01Ω	±10%

На самом деле, для полной картины надо сказать, что у резисторов есть ещё одна важная характеристика — максимальная рассеиваемая мощность, измеряемая в ваттах. Она определяет, как много энергии светодиод способен погасить без излишнего износа. Более подробно об этом будет рассказано в других уроках.

Подключаем светодиод

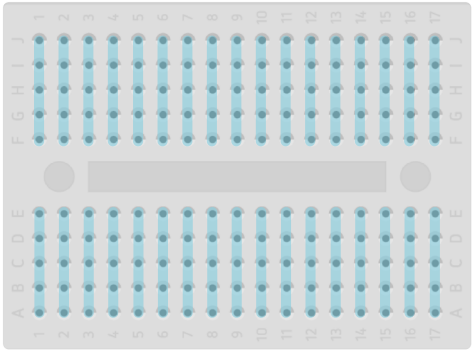
Итак, с принципиальной электрической схемой разобрались. Настало время применить знания на практике.

Но каким же образом собрать эту схему в реальности? Как соединить ножки компонентов друг с другом?

Уверен, ты что-то знаешь про пайку – такой способ соединения, например проводов, при помощи расплавленного металла. Но пайка – процесс небыстрый, если нет определённой сноровки, да и инструменты имеются далеко не у каждого. В конце концов, мы просто хотим собрать макет схемы, посмотреть, работает ли она, а после — разобрать. Короче, пайка в данном случае — точно перебор.

Можно попробовать скрутить ножки компонентов между собой, ведь даже просто соприкоснувшись, они уже будут в электрическом контакте. Догадываешься, чем плох этот способ? Слишком низкая надёжность! Если в случае с подключением светодиода довольно просто обнаружить, где появился разрыв, то в случае с более сложной схемой – даже того же светофора – можно просто непрерывно поправлять контакты, и ни о коком удобстве речи не идёт.

И нам повезло, что существует ещё один вариант – использовать макетную плату. В твоём наборе она уже приклеена на материнскую плату. Прямоугольная штука с кучей отверстий в нижнем правом углу. Создана она специально для того, чтобы быстро собрать макет схемы. Поэтому давай разбираться, как с ней работать.



Макетная плата состоит из так называемых шин. Шина – это, по сути, общий провод, и при подключении к ней контактов компонентов они электрически соединяются. Присмотримся ближе. К каждой шине можно подключить до пяти контактов. В центре макетной платы и между ши-

нами – разрыв, то есть, чтобы соединить контакты, необходимо попасть каждым в одно из пяти отверстий одной шины. Если соединить нужно более пяти контактов, можно использовать перемычки или провода для объединения нескольких шин в одну.

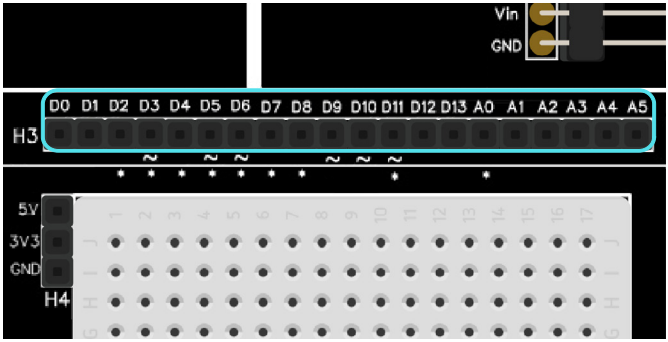
Фух! С макетной платой разобрались, с принципиальной электрической схемой тоже. Теперь, наконец, можем приступать к сборке!

Важно помнить, что **при сборке схемы и любом изменении в ней необходимо отключать питание, чтобы случайно не зацепить ничего лишнего.** Иначе неловкое движение может привести к выходу из строя компонентов, в том числе и платы Arduino.

Установим Arduino Nano в её посадочное место на материнской плате, обращая внимание на ключ, указывающий расположение разъёма mini-USB. (Обрати внимание на точное попадание контактов Arduino на свои места). Плата может вставляться довольно туго, но ещё раз проверив совпадение контактов с посадочными местами, можешь надавить посильнее, главное старайся равномерно распределять усилие.

Теперь контакты платы Arduino доступны на материнской плате. Ты можешь увидеть их прямо над макетной платой (её часто зовут макеткой).

И тут стоит вспомнить самое первое ознакомительное занятие, где мы узнали, как же на самом деле правильно называть контакты. Напоминаю – пины.



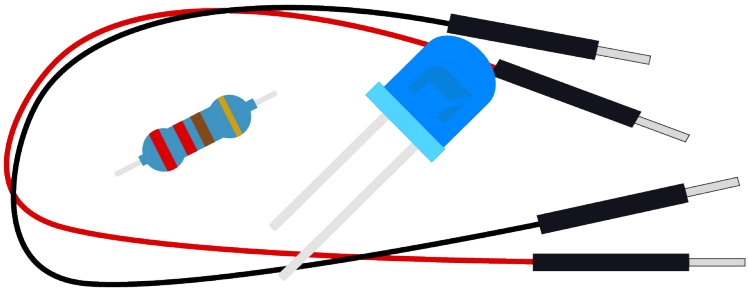
Итак, на материнской плате все пины подписаны так же, как и на плате Arduino Nano. Помимо этого, некоторые из них помечены значками точки и волны (тильды). Со значением значка волны мы разберёмся позже. Что же касается точки: если пин помечен точкой, это значит, он задействован где-то ещё на материнской плате. То есть к нему либо подключён светодиод светофора, либо пин датчика расстояния и так далее. На материнской плате все задействованные пины подписаны возле того

места, где используются, так что если сейчас там ничего не установлено, можешь смело использовать их для сборки схемы на макетке.

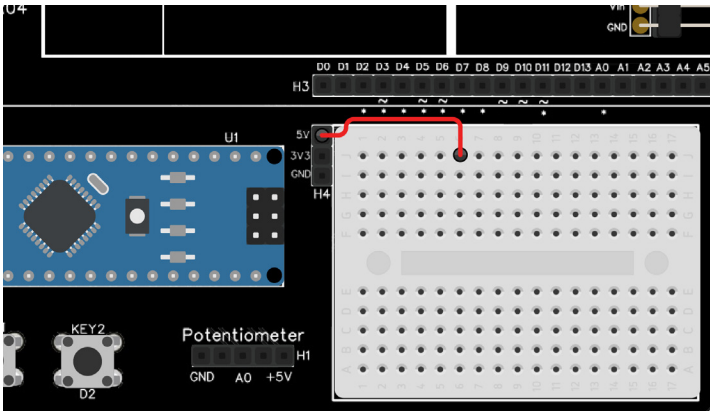
Также обрати внимание на верхний левый угол макетной платы, прямо там расположены выходы питания 5 В, 3,3 В и GND (земля).

Что ж, приступим к сборке! Глядя на принципиальную электрическую схему, будем её собирать. Ещё раз убедись, что питание платы отключено.

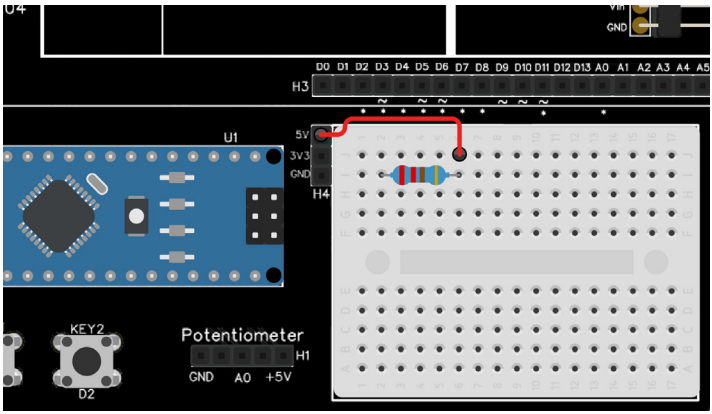
Приготовь всё, что понадобится для сборки: 2 провода «папа-папа» (желательно чёрный и красный), светодиод (желательно белый или синий) и резистор из ячейки со светодиодами (с голубым корпусом).



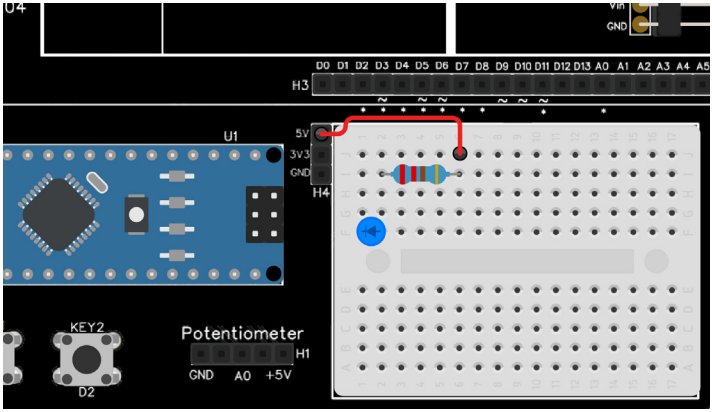
Итак, пойдём слева направо. Сначала идёт выход питания 5 В, подключим к нему один конец провода «папа-папа» **(обычно провод плюса питания красный, старайся придавать используемым цветам проводов смысл, так тебе будет легче в будущем вносить изменения в схему)**. Второй конец красного провода «папа-папа» подключим в любую свободную шину макетной платы.



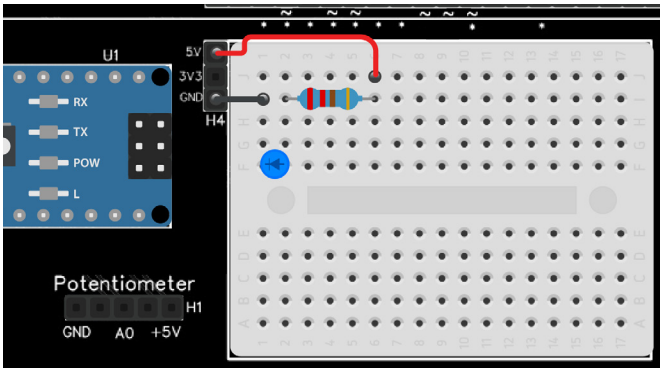
Далее, опять же глядя на схему, к той же шине, куда подключали провод, подключим один контакт резистора, а второй контакт – в свободную шину.



Дальше вспоминаем, что у светодиода есть полярность, то есть плюс и минус, плюс – длинная ножка, минус – короткая. Снова, глядя на схему, вспоминаем, где там у светодиода плюс, и подключаем длинную ножку светодиода в общую шину со свободной ногой резистора, а короткую ножку – в соседнюю свободную шину.



И, наконец, берём, желательно чёрный, провод и одним концом соединяем его с шиной минуса светодиода, а другим попадаем в GND.



Подключаем провод USB к плате Ардуино, и если всё сделали правильно – о чудо, да будет свет!

В качестве самостоятельной работы попробуй подключить ещё один светодиод. Не забудь отключить питание.

Управление внешним светодиодом

На прошлом занятии мы научились подключать светодиод. Но наша цель – собрать устройство, которым мы сможем управлять, а сейчас светодиод подключён к плюсу и минусу питания напрямую, и, соответственно, он включён всегда, когда есть питание. На самом первом занятии мы уже разобрались, что у микроконтроллера есть так называемые пины, которыми он может управлять. Впрочем, это ты уже умеешь, ведь мы уже написали первую программу, целью которой было мигать встроенным в плату светодиодом. И подключён он был к ножке (пину) 13.

На прошлом уроке мы уже узнали, где находятся пины Ардуино на материнской плате. И сейчас, не забыв отключить питание платы, вытащим один конец провода, который подключён к 5 В и подключим его к любому пину от D0 до A5. Все эти пины могут отдавать как высокий (5 В), так и низкий (GND) сигнал.

Теперь вернёмся к скетчу (так принято называть программы в Arduino IDE), который мы написали в прошлый раз, и сделаем так, чтобы он управлял другим пином. Попробуй сначала самостоятельно изменить программу.

Чтобы всё заработало, достаточно поменять номер пина в программе на тот, к которому подключена лампочка, и загрузить программу. Важное замечание: если ты выбрал ножку A0 – A5, то в программе к ней так и надо обращаться с символом A в английской раскладке. Загружаем программу, и я надеюсь, что у тебя всё получилось. Если нет, то добро пожаловать в чат в нашей группе в социальной сети «ВКонтакте», и я напоминаю, что прежде чем задавать вопрос, попробуй сначала найти ответ среди уже написанных.

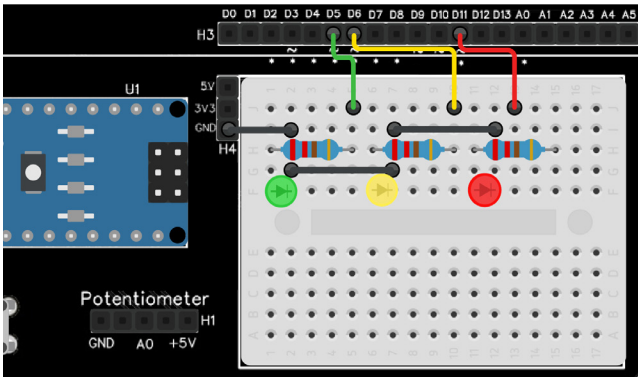
Ну и по традиции – самостоятельная работа. В следующей части урока мы будем на макетной плате собирать схему светофора. Я настоятельно рекомендую тебе сначала попробовать всё подключить и написать программу самостоятельно.

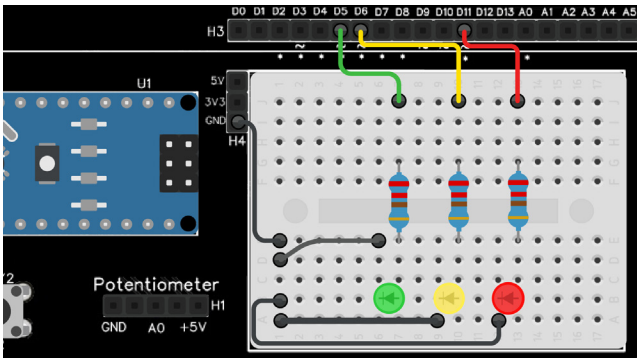
Светофор на макетке

Мы продолжаем двигаться к сборке нашего первого устройства. Осталась всего пара шагов. И сейчас мы соберём и запрограммируем светофор на макетной плате. Надеюсь, ты постарался решить эту задачку самостоятельно. Если нет, то всё ещё рекомендую тебе попробовать сделать это.

Итак, мы уже успешно умеем подключать и программировать одну лампочку. Получается, для работы светофора необходимо подключить ещё две. Делаем это по аналогии с первой. Единственный вопрос, который у тебя может возникнуть, – где взять столько контактов GND? Всё очень просто. Когда мы соединяем проводом контакт GND и любую шину макетной платы, все отверстия этой шины становятся GND. То есть появляются ещё четыре выхода GND. И так можно множить любые пины, пока не кончится макетная плата.

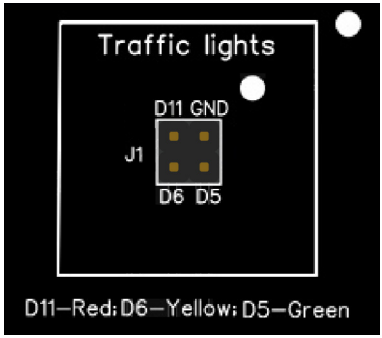
Так вот, вернёмся к сборке светофора. Сделать это можно по такой схеме:





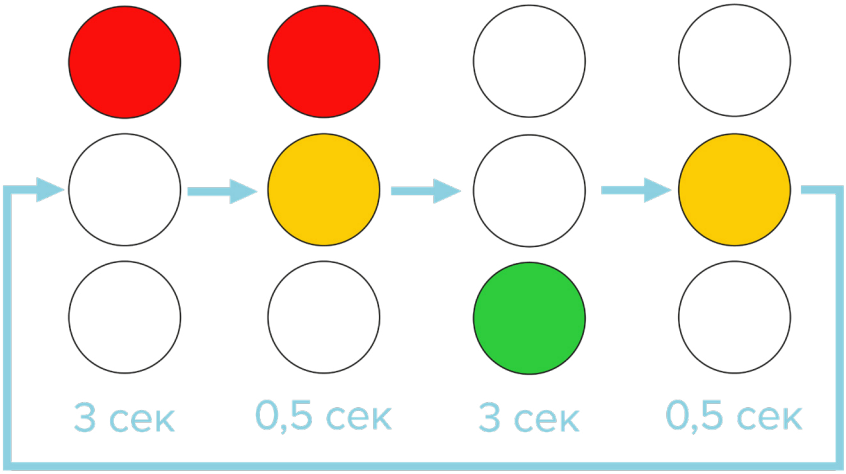
Это одинаковые схемы, отличается только расположение компонентов. Можешь выбрать то, которое тебе нравится, или придумать своё.

Обрати внимание на пины подключения светофора на материнской плате. Рекомендую сразу выбрать их и для сборки схемы на макетке.



Если возникают трудности со сборкой, то стоит вернуться к урокам, где мы подключали светодиод.

Когда схема собрана, можно приступать к написанию программы. И начать, как ни странно, необходимо с алгоритма. В этот раз для максимального понимания алгоритм нарисую.



Вначале горит красный свет. Ниже указано, как долго он должен гореть. Например, 3 секунды или 3 000 миллисекунд. Далее следует

одновременное включение жёлтого и красного, например, на полсекунды или 500 миллисекунд. Затем зелёный на 3 секунды и жёлтый на 500 миллисекунд. И так далее по кругу до бесконечности.

Теперь можно приступать к написанию программы. Открою пустой скетч и, вспоминая предыдущие уроки, пользуясь конспектами, приступаю.

Всё ещё предлагаю попробовать написать программу самостоятельно.

Первым делом, как всегда, настрою правильную работу используемых пинов функциями *pinMode* в теле функции *setup* для выполнения один раз в начале программы. Очевидно, что для каждого пина функция *pinMode* должна быть отдельной.

Далее в теле функции *loop* напишем реализацию нашего алгоритма для выполнения в бесконечном цикле. Первым делом, судя по алгоритму, необходимо включить красный свет. Пишем:

```
digitalWrite(
```

И в этот момент необходимо посмотреть на материнскую плату или вспомнить, куда подключён красный светодиод. Боюсь представить, что бывает в больших проектах, где задействовано около сотни пинов! А ведь существует ещё проблема поддержки и модернизации программного кода. Только представь, что спустя месяц ты захотел внести изменения в программу на устройстве, которое уже даже установлено в корпусе, – реально ли вспомнить, какие пины для чего были задействованы?

Или ещё проблема: допустим, мне захотелось поменять используемый пин, а уже написана программа на тысячи строк, где номер нужного пина встречается довольно часто. Неужели менять всё вручную? Выход, конечно же, есть – переменные! Ну или, как в нашем случае, константы. Сейчас поясню. Дело в том, что в таких случаях ножкам принято давать имена, то есть в самом верху программы, даже над функцией setup необходимо объявить переменные для используемых пинов. Давай разбираться, что же такое переменная, прежде чем будем её использовать.

Переменная – это, по сути, ячейка памяти, которая имеет своё уникальное название и хранит числа соответственно своему размеру. К переменной мы можем обратиться по её имени и получить значение либо изменить его. Можно представить её как контейнер, которому можно дать название и хранить там число. Число это, разумеется, не бесконечное, и, на самом деле, существуют различные типы переменных для хранения разных типов и объёмов данных. С ними мы познакомимся позже на примерах, а пока вернёмся к используемым пинам. Создадим для них переменные. Делается это, как я сказал, в самом верху программы. Для объявления новой переменной существует команда *int* (от английского слова integer – целое число). Далее через пробел следует имя переменной. Затем через равно – значение переменной. Не забудем точку с запятой. И, как было оговорено раньше, в нашем случае это не перемен-

ная, а константа. Мы не будем изменять её значение по ходу программы. Поэтому стоит перед int дописать const. Так компилятор сможет лучше оптимизировать код, и программа станет чуть легче и быстрее.

Объявим константы для используемых пинов. Сделать это можно как в строчку – друг за другом, – так и в столбик. Имена им дадим по цветам (red, yellow, green). Важно помнить, что имена переменных должны быть значащими, то есть что-то значить. Не стоит называть переменные как попало, ведь тогда в коде будет просто невозможно ориентироваться.

```
const int red = 11, yellow = 6, green = 5;
```

Теперь мы можем обращаться к пинам по имени. Заменим номера на имена в уже написанных командах и продолжим реализацию алгоритма в теле функции *loop*.

Итак, первым делом включаем красный.

```
digitalWrite(red, HIGH);
```

А как с остальными цветами в этот момент? По умолчанию предполагается, что там низкий сигнал. Но хорошим тоном будет всё-таки утвердить их низкое состояние – например, в теле функции ***setup*** указав всем светодиодам низкий сигнал. Теперь всё по фэншую, продолжаем.

Красный должен гореть 3 секунды, тогда пишем

```
delay(3000);
```

далее к красному добавляется жёлтый, и задержка здесь уже 500 миллисекунд, судя по алгоритму.

Далее выключаем красный, выключаем жёлтый и включаем зелёный.

Задержка снова 3000 миллисекунд.

И наконец выключаем зелёный и включаем жёлтый на полсекунды.

Не забудем выключить жёлтый.

```
// пины светофора
const int red = 11, yellow = 6, green = 5;

void setup( ) {
  pinMode(red, OUTPUT);      // пин 11 на выход
  pinMode(yellow, OUTPUT);  // пин 6 на выход
  pinMode(green, OUTPUT);   // пин 5 на выход

  // начальное состояние, все выключены
  digitalWrite(red, LOW);      // выключаем красный
  digitalWrite(yellow, LOW);  // выключаем жёлтый
```

```
 digitalWrite(green, LOW);   // выключаем зелёный
}

void loop( ) {
  digitalWrite(red, HIGH);       // включаем красный
  delay(3000);                   // ждём 3 сек.
  digitalWrite(yellow, HIGH);  // включаем жёлтый
  delay(500);                    // ждём полсек.
  digitalWrite(red, LOW);        // выключаем красный
 digitalWrite(yellow, LOW);    // выключаем жёлтый
  digitalWrite(green, HIGH);   // включаем зелёный
  delay(3000);                   // ждём 3 сек.
  digitalWrite(green, LOW);    // выключаем зелёный
  digitalWrite(yellow, HIGH);  // включаем жёлтый
  delay(500);                    // ждём полсек.
  digitalWrite(yellow, LOW);  // выключаем жёлтый
}
```

Вот и всё! Можно загружать и проверять.

Теперь светофор на макетной плате готов и работает! На следующем занятии уже будем собирать схему в корпусе. Обрати внимание, что в корпусе светофора присутствует маленькая макетная плата. Можешь попробовать прикинуть, как уместить всю схему светофора на ней, да ещё чтобы всё аккуратно закрылось. Для этого в ячейке с деталями светофора есть 2 перемычки.

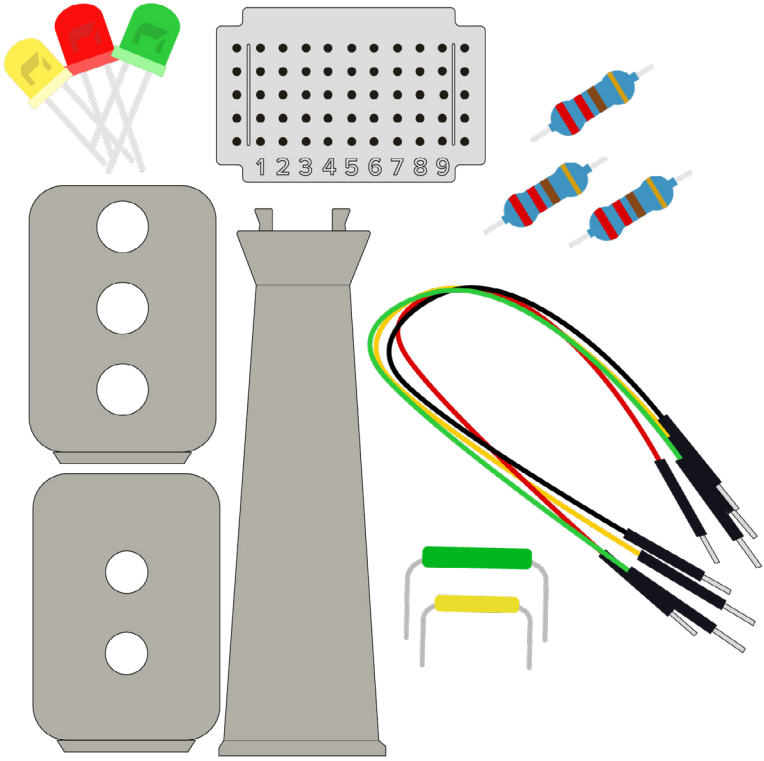
Ну и дополнительно можешь попробовать добавить мигающий зелёный в алгоритм нашего светофора.

Светофор в корпусе

И вот, наконец, время пришло! Сегодня мы завершим сборку нашего первого электронного устройства! Приступим.

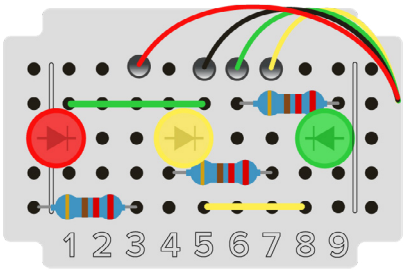
Подготовим все детали для сборки:

- провода чёрного, красного, зелёного и жёлтого цветов;
- две перемычки для макетной платы;
- детали для сборки светофора;
- 3 резистора и 3 светодиода: красный, зелёный, жёлтый.

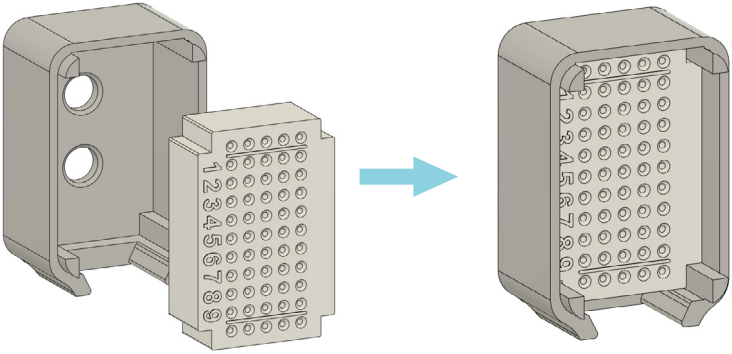


Надеюсь, ты уже попробовал собрать схему самостоятельно перед тем, как увидеть мой вариант.

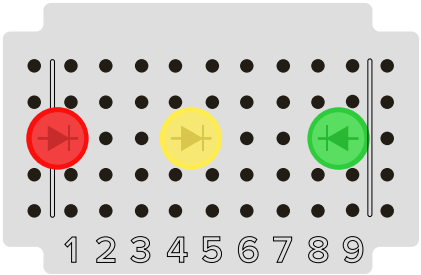
Короче говоря, схема готового светофора будет выглядеть так:



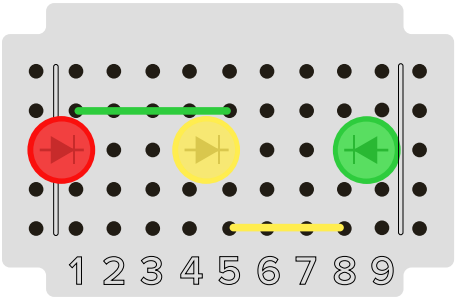
Перед началом сборки схемы необходимо разместить макетную плату в задней крышке светофора.



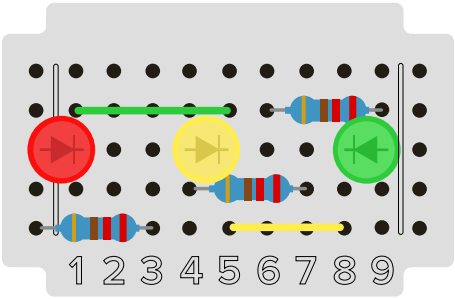
Перейдём к пошаговой сборке. Для начала необходимо установить светодиоды в правильных местах – так, чтобы при закрытии корпуса они точно совпадали с отверстиям, при этом важно соблюсти полярность установки. Тут на всякий случай снова напомню: длинная ножка – это плюс, короткая – минус. На этой схеме изображена правильная установка светодиодов.



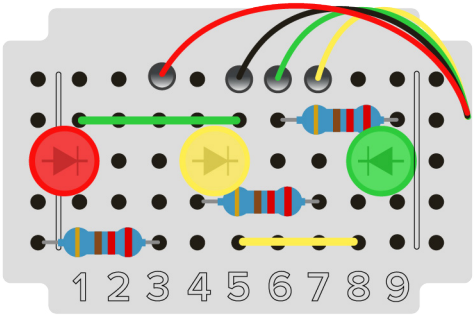
Далее объединим все минусы светодиодов перемычками. У тебя в наборе их две: зелёная (подлиннее) и жёлтая (покороче). В моём варианте они расположены так.



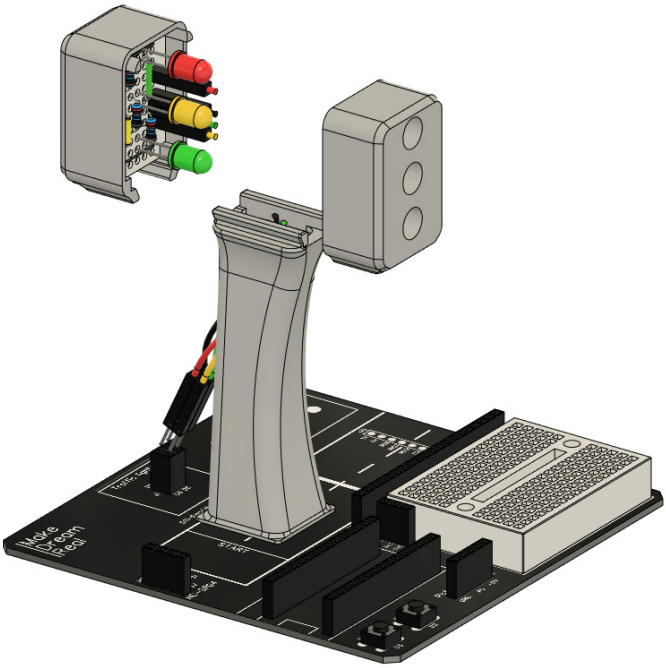
Теперь установим резисторы. Их подключаю к плюсам светодиодов так, чтобы вторая нога резистора оказалась на свободной шине. Получилось как-то так:



Ну и наконец подключаю провода в соответствии с цветами светодиодов. Важно установить их скраю, чтобы внутренние элементы корпуса не соприкасались с ними.



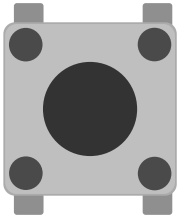
Вот схема и готова. Теперь упакуем всё в корпус, как показано на картинке. И подключим провода согласно подписям на плате.



Кнопка

Введение

В этом уроке мы познакомимся с нашим первым устройством ввода – кнопкой. Они есть как на материнской плате, так и в органайзере.



По традиции начну с вопроса. Как ты думаешь, где и как применяются именно такие кнопки? Попробуй порассуждать самостоятельно.

Кнопка – это простейшее устройство ввода, и, как правило, она встречается не реже светодиода. Её применяют для перезагрузки, изменения режима работы, ввода каких-либо данных и так далее.

Итак, для начала работы с кнопкой разберёмся, как она устроена. Физически она выполняет очень простую функцию – замыкает и размыкает контакт. В нашем случае мы имеем дело с по умолчанию разомкнутой тактовой кнопкой без фиксации. Иначе говоря, если кнопка не нажата, контакт разомкнут, а если кнопка нажата, контакт замыкается. При этом, если кнопку отпустить, она автоматически размыкает контакт.

Наша кнопка имеет четыре ножки, и очень важно понять, как они соединены внутри корпуса. Обрати внимание на рисунок:

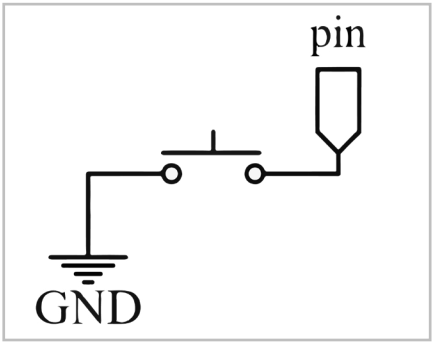
Разомкнуты контакты с одной стороны корпуса и замыкаются при нажатии на кнопку, а контакты по разные стороны попарно замкнуты. Даль-

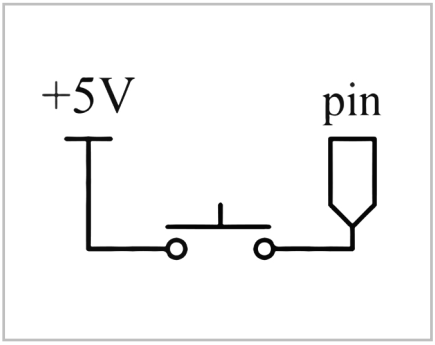
ше я ещё напомню об этой особенности.

Подключение

Итак, как же подключить кнопку к микроконтроллеру? Ранее я мельком упоминал, что пины микроконтроллера умеют не только отдавать сигнал, но и принимать, иначе говоря, читать состояние пина. Именно это свойство мы и будем использовать, чтобы заметить нажатие на кнопку.

Разберёмся подробно с подключением кнопки к Ардуино. По факту, мы имеем всего два контакта, один из которых мы, очевидно, подключаем к любому пину. Куда же подключить второй? Мы знаем, что цифровые ноги Ардуино умеют замечать только два вида сигнала: либо высокий (5 вольт), либо низкий (минус, земля). Таким образом, если подключить второй контакт либо к земле, либо к 5 вольтам, при нажатии на кнопку мы будем получать на пине соответствующий сигнал.





Но! Какой сигнал будет в обоих случаях, если кнопка не нажата? Попробуй самостоятельно предположить.

Разумеется, сигнал не может быть никаким, он обязательно либо 0, либо 1.

Правильный ответ – случайный. Дело в том, что в момент, когда контакт не подключён к определённому источнику сигнала, он как бы висит в воздухе и является антенной, что в итоге приводит к появлению случайного сигнала из-за внешних шумов.

Что же делать? Необходимо точно определить состояние пина, когда кнопка не нажата. Что произойдёт, если мы просто подключим ножку к противоположному источнику сигнала (то есть в случае, если кнопка подключена к GND, ножку мы подключим к 5 вольтам, и наоборот)?

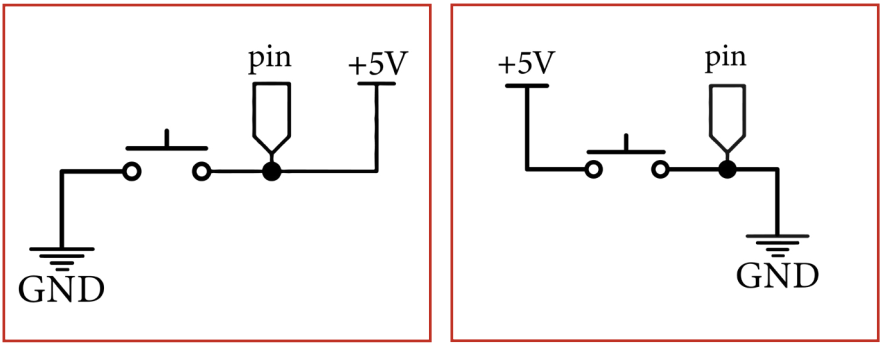


Схема с коротким замыканием

Теперь состояние ножки, когда кнопка не нажата, точно определено. Но какой сигнал будет, если теперь кнопку нажать? Я говорил, что сигнал не может быть никакой. Но в данном случае так, наверное, действительно можно сказать, ведь в этот момент «Ардуинка» может испустить дух... Знаешь ли ты что-нибудь про короткое замыкание? В данном случае, когда мы нажимаем на кнопку, происходит прямой контакт 5V и GND с минимальным сопротивлением, что приводит к бесконтрольному повышению температуры проводников, которые, как правило, от такого плавятся в самом тонком месте. В нашем случае таким местом является микроконтроллер, так как проводники внутри него невероятно тонкие.

В случае Ардуино при коротком замыкании возможны несколько вариантов исхода.

1. **Самый безобидный.** На плате гаснут все лампочки, но всё остаётся целым благодаря встроенной защите источника питания, в нашем случае – разъёма USB компьютера.

2. **Аналогичен первому.** Единственное, после проверки и повторного подключения PWR всё равно не горит. При этом, если схема точно собрана правильно, возможно, что в разъёме компьютера сработал предохранитель, который восстанавливается при перезагрузке компьютера.

3. **В продолжение второго,** если после перезагрузки компьютера светодиод PWR всё равно не загорается, это может свидетельствовать о «смерти» конкретного разъёма USB на компьютере или «смерти» платы Ардуино.

Возвращаемся

Так как же всё-таки правильно подключить кнопку к Ардуино? На электрических схемах выглядеть это будет так:

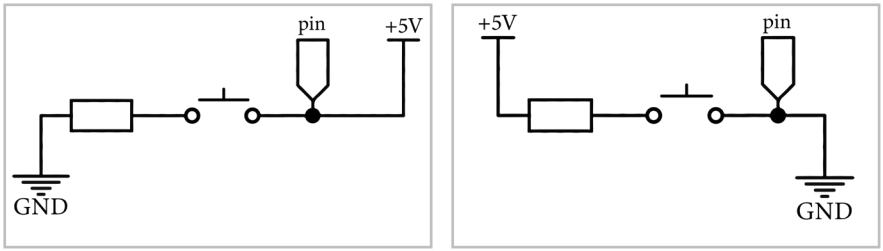


Схема со стягивающим (pull down) резистором и схема с подтягивающим (pull up) резистором

Необходимо использовать резистор, как правило, на 10 кОм, чтобы максимально уменьшить ток через него. Если этот резистор подключён к 5 вольтам, его называют подтягивающим, а если к земле, то стягивающим. И теперь, когда кнопка не нажата, сигнал явно определён через резистор, а когда кнопка нажата, наибольший ток пойдёт по пути наименьшего сопротивления, то есть через кнопку, и ножка «Ардуинки» зафиксирует другой сигнал.

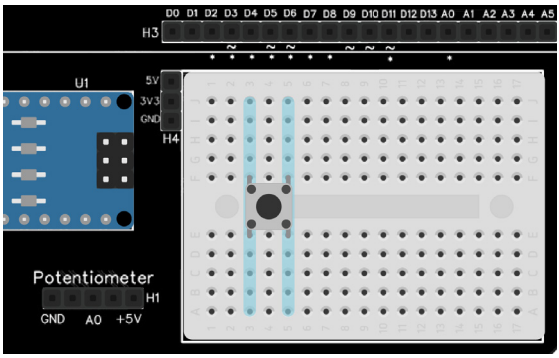
А сейчас предлагаю собрать одну из этих схем на макетной плате.

Пусть это будет схема с подтягивающим резистором (позже объясню, почему это важно).

По традиции попробуй собрать схему самостоятельно, прежде чем смотреть дальше, но помни, что важно быть очень осторожным...

Ну что ж, приступаем!

Для начала, кнопку принято размещать в центре макетной платы, при этом кнопка соединяет шины с обеих сторон макетной платы.



Об этом обязательно стоит помнить.

А далее, как всегда, глядя на электрическую схему, начинаем собирать её.

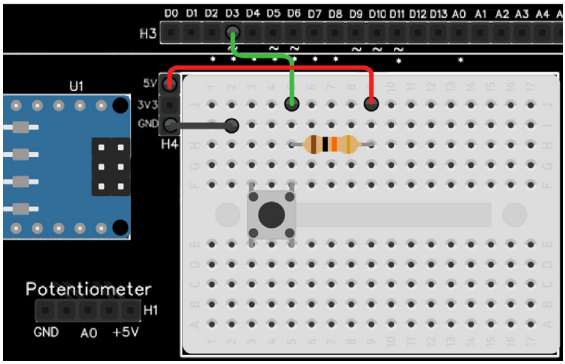
Вставлю провод одним концом в 5V, другим – в свободную шину макетной платы.

По схеме одна ножка резистора подключена к 5V, другая – к контакту кнопки. Подключаем.

На схеме видим, что этот же контакт подключён ещё и к пину Ардуино, соединяем проводом: один конец подключаем к любому пину (в моём случае это пин 3), другой – в ту же шину макетной платы.

Снова смотрим на схему. Другая ножка кнопочки подключена к GND, поэтому подключаем провод к контакту GND и соответствующей шине.

Вот схема и готова.



Программирование

Теперь пора приступать к написанию первой программы.

Сначала реализуем самый простой алгоритм. Если кнопка нажата, то встроенный светодиод на ножке 13 горит, если не нажата, встроенный светодиод не горит.

Можно ещё сформулировать по-другому, это поможет нам в дальнейшем. Если кнопка нажата, то лампочка горит, иначе (то есть в любом другом случае) не горит. Приступим!

Для начала, как всегда, создадим необходимые переменные. В нашем случае led для ножки 13 и button для ножки 5, к которой подключена кнопка.

```
const int led = 13;
const int button = 5;
```

Напомню, имена переменной ты можешь придумать и самостоятельно, но важно, чтобы они были значащими, то есть их название должно передавать скрытый за ними смысл.

С переменными справились, идём дальше. Необходимо задать режимы работы используемых пинов. Делаем это в теле функции setup.

Для ножки, к которой подключён светодиод, ничего не изменилось, её режим работы всё так же на выход, а вот для пина кнопки – на вход, ведь мы будем читать состояние на этой ножке (то есть она как бы принимает данные на вход). Итого, получим:

```
void setup() {
  pinMode(led, OUTPUT);
  pinMode(button, INPUT);
}
```

Далее время реализовывать наш алгоритм. И тут мы рассмотрим два варианта его реализации.

Первый, самый очевидный: будем действовать, прямо как написано. Всё начинается со слова «если», получается, пришло время для знакомства с новой конструкцией в языке C++. Синтаксис будет выглядеть так:

```
if(условие){
 тело
}
```

Если условие в круглых скобках верно, то выполним то, что в фигурных скобках.

В нашем случае условие звучит так: «Если кнопка нажата». Существует специальная команда для считывания состояния пина и записывается она так: digitalRead(номер пина). То есть, по логике, у нас должна получиться запись:

```
if(digitalRead(button) = ?)
```

А чему равно? Давай-ка взглянем на схему и подумаем. Если мы нажимаем на кнопку, то какой сигнал будет на ножке 5? Он будет низкий, то есть 0. Так как кнопка замкнёт пин на GND. Выходит, запись будет выглядеть так?

```
if(digitalRead(button) = 0)
```

Здесь всё ещё есть одна ошибка. Дело в том, что в языке C++ знак **(оператор) =** означает присвоение, то есть его использование предполагает, что значение справа от него записывается в переменную слева. Но мы тут не хотим ничего записывать, мы хотим сравнить. Для этого существует **оператор ==**. Он означает сравнение. Таким образом, правильная запись будет выглядеть так:

```
void loop() {
  if(digitalRead(button) == 0){
  }
}
```

Помимо == ещё существуют:
> Больше
>= Больше или равно
< Меньше
<= Меньше или равно
!= Не равно

И теперь, если на пине 3 появится низкий сигнал, то есть если кнопка будет нажата, тогда выполнятся команды, написанные в фигурных скобках. А что, собственно, в них написать? Судя по алгоритму, включить лампочку. А эту команду ты уже знаешь:

```
digitalWrite(led, HIGH);
```

Если загрузить этот код, то результат будет выглядеть так: вначале лампочка выключена, если нажать кнопку, светодиод включится, и на этом всё! Дальше, сколько ни нажимай и ни отпускай, состояние светодиода не изменится, ведь нигде в программе мы его не выключаем. Необходимо этим заняться. Для этого мы можем опять писать

```
if(digitalRead(button) == 1)
```

то ...

Но есть другой, более короткий вариант. Мы перефразировали наш алгоритм, и вторая его часть звучала так: «иначе выключить». В языке C++ предусмотрена подобная запись. Выглядеть это будет так:

```
else {
}
```

Ну и в фигурных скобках отключаем лампочку.

Вот и всё, код готов, можем загружать и проверять.

```
const int led = 13;                  // пин светодиода
const int button = 3;                // пин кнопки
void setup( ) {
  pinMode(led, OUTPUT);              // пин 13 на выход
  pinMode(button, INPUT);            // пин 3 на вход
}
void loop( ) {
  if(digitalRead(button) == 0){      // если кнопка нажата
     digitalWrite(led, HIGH);        // включить светодиод
  } else {                           // иначе
    digitalWrite(led, LOW);          // выключить светодиод
  }
}
```

Не лишним будет напомнить, что перед загрузкой необходимо во вкладке «Инструменты» проверить пункты: плата – Arduino Nano, процессор – ATmega328P (Old bootloader) и выбрать порт, к которому подключена плата.

Теперь самостоятельно попробуй сделать так, чтобы, если кнопка нажата, лампочка не горела, а если не нажата, горела. То есть инвертируй её состояние.

Это можно сделать несколькими способами:

1. Изменить условие, заменив 0 на 1. Получится: если на ножке высокий сигнал, то включить светодиод.

2. Изменить условие, заменив оператор сравнения на «не равно», он будет выглядеть так: !=. В языке C++ символ ! значит «не». Получится: если на ножке не низкий сигнал, то включить лампочку.

3. Заменить аргумент HIGH на LOW в обоих случаях. Тогда выйдет: если кнопка нажата (если низкий сигнал на ножке 3), то выключить лампочку, иначе включить.

Уверен, можно придумать ещё варианты, но пока этого достаточно.

Ранее я обещал, что мы рассмотрим два варианта реализации алгоритма. И сейчас напишем его в одну строчку! Но, как всегда, сначала предлагаю тебе догадаться, как это можно сделать. Подсказка: **if** нам вообще не понадобится.

Так вот. Выглядеть это будет так:

```
void loop() {
  digitalWrite(led, !digitalRead(button));
}
```

Записать в ножку 13 сигнал, противоположный прочитанному с ножки 3. Противоположный потому, что мы хотим, чтобы при нажатии лампочка включалась. Соответственно, если мы хотим, чтобы лампочка выключалась, тогда просто уберём «!».

Ну и пришло время раскрыть тебе главную тайну! Почему я предложил собирать именно такую схему, с подтягивающим резистором. Дело в том, что в каждую ножку Ардуино этот резистор уже встроен! Но его нужно включить. Делается это изменением режима работы пина, к которому подключена кнопка. Вместо INPUT необходимо написать **INPUT_PULLUP**, что расшифровывается как «на вход с подтягивающим резистором». К сожалению, режим **INPUT_PULLDOWN**, то есть со стягивающим резистором, в нашем микроконтроллере не доступен, но встречается в других, а в некоторых может не быть ни того, ни другого, так что тут нужно уже изучать документацию на конкретный микроконтроллер. Для проверки можешь убрать из схемы резистор с проводом к 5 вольтам и убедиться, что всё работает. Только не забудь отключить питание при изменении в схеме.

На материнской плате присутствуют ещё две кнопки, подписанные как **KEY1** и **KEY2**, подключённые к ножкам 3 и 2 соответственно. При этом на плате нет никаких резисторов, что намекает на то, что использовать их нужно со встроенным подтягивающим резистором.

В качестве самостоятельной работы предлагаю тебе придумать различные алгоритмы для работы со светофором, используя встроенные в плату кнопки.

Переключатель

Сейчас наша задача – реализовать куда более интересный алгоритм. В электронных устройствах чаще всего мы отслеживаем именно клик, то есть нажатие и отпускание кнопки. В нашем наборе мы сможем это использовать, например, для переключения сигнала светофора. Давай сделаем это!

Начнём с переключения состояния одной лампочки. Есть несколько вариантов реализации этой задачи, но мы рассмотрим самый распространённый и, наверное, оптимальный. Будем отслеживать **конкретный момент нажатия** и **конкретный момент отпускания кнопки**. Алгоритм будет выглядеть так: «Если кнопка нажата и до этого не была нажата, то запомним, что она нажата, и тут можно что-то сделать, например переключить состояние лампочки. Далее, если кнопка не нажата и до этого была нажата, то запомним, что она не нажата, тут тоже можно что-то сделать, например переключить состояние лампочки». По ходу написания программы будет куда понятнее.

Приступим к написанию программы. Основу мы уже изучили: переменные, режимы работы ног.

```
const int led = 13;                    // пин светодиода
const int button = 3;                  // пин кнопки

void setup() {
  pinMode(led, OUTPUT);                // пин 13 на выход
  pinMode(button, INPUT_PULLUP);       // пин 3 на вход
}

void loop() {

}
```

А вот в теле функции **loop** всё и происходит.

Для начала покажу тебе важный приём. В предыдущих примерах мы читали состояние ножки прямо в круглых скобочках условия, но обычно, особенно когда значение нужно в нескольких условиях, прочитанное значение сначала записывают в переменную, которая уже и использу-

ется для сравнения. Делается это для порядка, ведь возможен вариант, когда состояние кнопки может поменяться прямо в момент начала второго условия – тогда получится, что выполнятся и первое, и второе, а это, в свою очередь, не всегда нужно. Поэтому в самом начале сохраним состояние ножки в переменную. Создадим её прямо в теле функции loop, что сделает эту переменную локальной, то есть использовать её можно будет только в теле этой функции, в других она не видна. Делается это для экономии оперативной памяти микроконтроллера. Переменные, объявленные вне тел функций, называют глобальными, и их можно использовать во всей программе.

Поскольку имя должно быть значащим, предлагаю назвать её buttonState (состояние кнопки). Итак, пишем:

```
int bottonState = digitalRead(button);
```

Надеюсь, ты помнишь, что int значит «целое число», а тут у нас возможны всего два варианта – 0 и 1. Они вполне себе целые, и для создания таких переменных можно использовать int, но обычно принято использовать специальный тип данных для двоичных чисел – **bool**. Так что предлагаю заменить **int** на **bool** – опять же, для порядка.

Едем дальше. Из алгоритма можно увидеть, что нам предстоит запоминать предыдущее состояние кнопки, вернее, пина. Делать это тоже будем в специальную переменную. Объявить её можно в самом верху, то есть вне тел всех функций. Во-первых, это сделает её глобальной, то есть она будет видна везде, во всех функциях, что нам сейчас не особо важно, ведь нам она нужна только в теле **loop**. Ну а во-вторых, если мы создадим её в теле **loop**, при каждом проходе цикла она будет создаваться заново, что будет сбрасывать её значение. Кстати, если значение по умолчанию не задать, оно будет автоматически приравниваться к нулю, но для порядка, даже если мы и так хотим записать туда 0, всё равно приравнивают к нулю, ведь бывают другие платформы и языки, где автозаполнение не предусмотрено и туда может быть записано любое случайное значение. Создаём, как уже умеем:

```
bool lastStateButton = 1;
```

и изначально приравниваем к состоянию «не нажата», предполагая, что при включении питания её никто не трогает.

Ну а дальше как слышим, так и пишем: если кнопка нажата и её прошлое состояние – «не нажата», то...

```
if(buttonState == 0 && lastButtonState == 1){
```

Знак && обозначает «и». Его применяют, когда нужно, чтобы тело выполнилось только в случае, если оба условия верны. Помимо этого, часто используют:
|| - «или, верно хотя бы одно»

Теперь запомним, что она нажата:

```
lastButtonState = 0;
```

Ну и далее как-то можем отреагировать на изменение состояния кнопки. Пока продолжим. Второе условие: если кнопка не нажата и до этого была нажата, то...

```
if(buttonState == 1 && lastButtonState == 0){
```

запомним, что она нажата:

```
lastButtonState = 1;
```

и тут можно отреагировать на отпускание кнопки.

Есть ещё одна важная деталь. Дело в том, что при нажатии на кнопку и при её отпускании происходит такое явление, которое называют дребезгом контактов. То есть контакты при сближении замыкаются и размыкаются несколько раз. Связано это с упругостью материала контактов кнопки, из-за чего они некоторое время отскакивают друг от друга. Происходит это невероятно быстро и длится около 10 мс. Но тут стоит вспомнить, что наш микроконтроллер делает 16 млн операций в секунду, что позволяет ему зафиксировать дребезг, который будет воспринят как несколько нажатий на кнопку, и, соответственно, результат будет случайным. Для того чтобы избежать влияния дребезга, воспользуемся самым лёгким способом. Просто поставим задержу в конце обоих тел условий, чтобы дать время дребезгу успокоиться. Пусть это будет

```
delay(50);
```

чтоб наверняка.

Осталось, собственно, изменить состояние лампочки. Сделать это можно в двух местах: при нажатии на кнопку или при отпускании, тут уж на твоё усмотрение. Предлагаю попробовать оба варианта. Для начала давай отреагируем именно в момент нажатия. В тело первого условия добавим изменение состояния ножки 13. По традиции попробуй сделать это самостоятельно.

Тут, как всегда, есть несколько вариантов. И, учитывая то, чему мы уже научились, очевидным кажется вариант использования конструкции, похожей на ту, что мы использовали для отслеживания нажатия на кнопку. Но покажу тебе классный лайфхак. Дело в том, что читать состояние ножки можно при любом режиме её работы, то есть ничего нам не мешает прочитать состояние лампочки. И реализовать это можно так: **digitalWrite(led, !digitalRead(led))**; записать в ножку 13 **НЕ** сигнал, прочитанный с ножки 13. Вот и всё.

```
const int led = 13;      // пин светодиода
const int button = 3; // пин кнопки

// переменная для хранения предыдущего состояния кнопки
bool lastButtonState = 1;

void setup() {
  pinMode(led, OUTPUT);                 // пин 13 на выход
  pinMode(button, INPUT_PULLUP);  // пин 3 на вход
}

void loop() {
  // читаем состояние кнопки в переменную
  bool buttonState = digitalRead(button);

  // Если кнопка нажата и до этого не была нажата
  if (buttonState == 0 && lastButtonState == 1) {
    lastButtonState = 0; // запомним, что она нажата
    // инвертируем состояние светодиода
    digitalWrite(led, !digitalRead(led));
    delay(50); // задержка против дребезга
  }
  // Если кнопка не нажата и до этого была нажата
  if (buttonState == 1 && lastButtonState == 0) {
    lastButtonState = 1; // запомним, что она не нажата
    delay(50); // задержка против дребезга
  }
}
```

Теперь предлагаю ознаменовать окончание знакомства с кнопкой написанием программы для переключения сигнала светофора. И прошу тебя написать её самостоятельно. Напомню, что мышление и необходимые навыки прокачиваются только при самостоятельной работе. И тут уже неважно, получится решить эту задачку или нет. Сам факт попытки уже значительно тебя прокачает. Ну а неудача так вообще по итогу даёт максимальный рост, ведь помогает придумать правильные вопросы.

Serial порт и общение с компьютером

В этом уроке мы научимся обмениваться данными с компьютером. Это позволит выводить необходимую информацию на экран компьютера и управлять Ардуино при помощи клавиатуры.

Обмен данными с компьютером применяется чаще всего для отлова ошибок при разработке устройства. В любой точке программы на экран компьютера можно вывести необходимую информацию: результаты вычислений, значения переменных, состояние ножки и так далее. Весь этот процесс называется отладкой программы.

Также можно создать программу для компьютера, где уже красиво разместить вывод необходимой информации, кнопки для управления устройством – короче, всё, что душе угодно, но об этом как-нибудь в другой раз.

Ну а пока познакомимся с простейшим способом реализации этой задачи.

Для начала разберёмся, каким образом происходит общение. Для этого попробуй ответить на вопрос, что такое USB, HDMI, ETHERNET? Что их объединяет? Сюда же можно отнести Wi-Fi, Bluetooth и так далее. Всё это – интерфейсы передачи данных. Все они используют разное количество проводов, разные протоколы передачи и даже разные способы соединения. Так что же такое протокол? Если говорить просто, протокол – это последовательность действий при передаче данных. Например, протокол общения людей – поздоровались, пожали руки, обменялись информацией, попрощались, пожали руки.

Так и в электронике – соблюдение подобных последовательностей необходимо.

Дело в том, что в микроконтроллере нет встроенного протокола USB и для общения с компьютером он использует протокол UART. Не пугайся, это просто название очередного интерфейса, и вдаваться в подробности его работы мы не будем.

Но как же «Ардуинка» и компьютер могут обмениваться данными через USB, если в микроконтроллер его не завезли? Всё благодаря USB-UART преобразователю, он расположен на обратной стороне платы Ардуино Нано. Это своего рода переводчик. Подключён он к ножкам 0 и 1, подписанным как TX (transmit – передача) и RX (receive – приём). На плате также есть соответствующие лампочки, они подсвечивают буквально каждый бит пролетающей информации, можешь обратить внимание

при следующей загрузке программы. Эти пины участвуют в прошивке микроконтроллера, поэтому их лучше не использовать в своих проектах без особой необходимости.

Итак, на самом деле, с точки зрения написания кода тут всё довольно просто. Так что давай прям сходу разбираться.

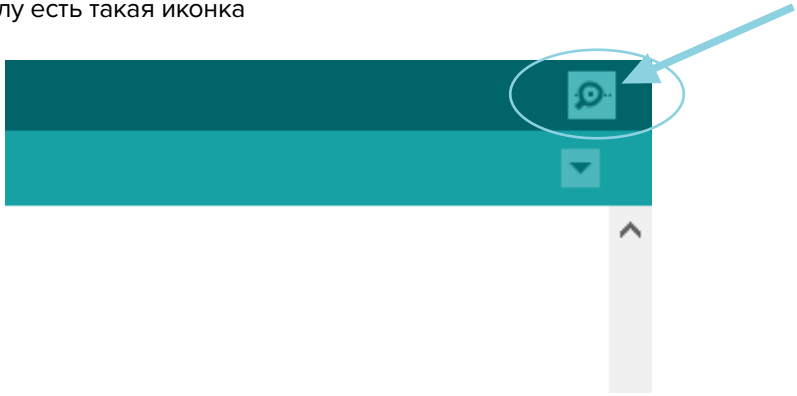
Для начала необходимо, как ни странно, начать общение с компьютером, для этого существует команда *Serial.begin()*. Писать её следует в теле функции **setup**. Сразу поясним важные моменты. Слово Serial переводится как «последовательный» и используется именно оно, так как UART принято называть «Последовательный порт передачи данных». Данные, собственно, передаются последовательно, то есть друг за другом, так, соответственно, и читаются. То есть от компьютера, например, Ардуино получила несколько байт, они скопились в так называемом буфере, как бы в очереди. Микроконтроллер прочитал один, что-то там с ним сделал, удалил, когда необходимо, прочитал из буфера следующий и так далее.

Так вот, каждый раз, когда мы хотим что угодно сделать с последовательным портом, мы должны использовать следующую конструкцию – *Serial*, после чего уже писать конкретную команду, обозначающую необходимое действие, в данном случае *begin*. В скобках у конкретно этой команды необходимо указать скорость общения с компьютером, причём не абы какую, а одну из стандартных. Мы укажем одну из самых популярных – 9600 бод. Этих стандартных скоростей много – важно, чтобы у обоих устройств они были одинаковые, иначе корректно передать данные не получится.

Теперь давай отправим что-нибудь компьютеру. Сделаем это тоже в теле функции setup, чтобы отправилось один раз. Для этого существует команда *Serial.print()*. В скобочках в двойных кавычках пишем текст или, как правильно говорить, строку, которую хотим отправить. Напишем по традиции Hello, world! (Привет, мир!)

```
void setup( ) {
  Serial.begin(9600);
  Serial.print("Hello, world!");
}
void loop( ) {
}
```

Загружаем. Чтобы увидеть результат работы программы, необходимо открыть монитор последовательного порта в **Arduino IDE**. В верхнем правом углу есть такая иконка



Если на неё нажать, открывается консоль. Важно, чтобы при этом плата была подключена к компьютеру и был выбран порт.

Здесь в центре выводятся полученные данные, в верхней части есть строка для отправки и кнопка «Отправить». В нижнем правом углу – настройка скорости передачи данных. Нужно, чтобы была такая же, как и на «Ардуинке». Вот и всё! Можешь поэкспериментировать, перенести отправку в тело функции **loop** и посмотреть, что будет.

Сообщение выводится в строчку и перестаёт быть достаточно читаемым. Удобнее было бы выводить его в столбик. Для этого существует команда *Serial.println()*, где ln обозначает line – новая строка. Эта команда

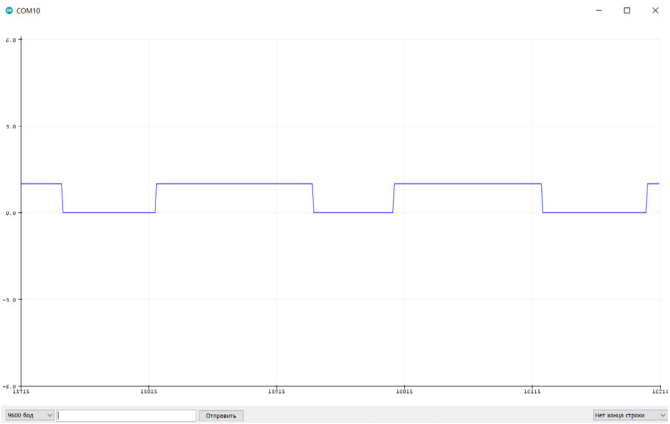
автоматически переносит строку после вывода сообщения. Загрузим и посмотрим, как результат выглядит теперь.

Есть и другой способ. В конце строки можно написать символ переноса строки **\n**, вот так: *Serial.print("Hello, world!\n")*. В некоторых случая это бывает удобно. При загрузке получим такой же результат.

Теперь давай попробуем выводить состояние кнопки. Попробуй, не подглядывая, написать программу, которая это делает, используя кнопки, доступные на материнской плате

```
const int button = 3; // пин кнопки

void setup() {
  // начать общение с компьютером на скорости 9600 бод
  Serial.begin(9600);

  // режим работы пина на вход с подтягивающим резистором
  pinMode(button, INPUT_PULLUP);
}

void loop() {
  // выводим состояние кнопки в монитор порта
  Serial.println(digitalRead(button));
}
```

Теперь, если кнопка нажата, выводится 0, а если не нажата – 1. Помимо консоли существует ещё один вариант вывода – в виде графика. Открыть его можно во вкладке «Инструменты», «Плоттер по последовательному соединению». Но для его открытия консоль должна быть закрыта, или порт будет считаться занятым. Теперь мы наблюдаем график нажатия на кнопку.



Что ж, отправлять данные мы научились. Теперь давай разберёмся с получением. Для начала напишем простенькую программку, которая будет отправлять то, что получила.

Для считывания данных из буфера существует команда *Serial.read()*. Важно понимать, что она не просто читает данные, но и удаляет после прочтения. Поэтому принято читать их в переменную.

Мы уже знакомы с такими типами данных, как **int** и **bool**, но данные с компьютера отправляются как символ. Для работы с ними существует специальный тип данных – **char**. В итоге чтение будет выглядеть так:

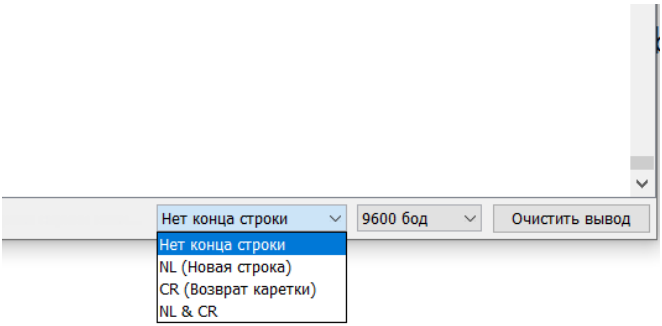
```
char data = Serial.read();
```

Теперь в переменной data хранится символ, прочитанный из буфера. Чтобы его вывести, используем уже знакомую команду *Serial.print()*. Если использовать *Serial.println*, то, что мы отправим, будет выводиться посимвольно в столбик.

Но для нормальной работы написанного недостаточно. Дело в том, что мы выводим содержимое буфера, даже когда нам ничего не отправляют, соответственно, выводим мы по большей части всякий мусор. Чтобы выводить только полученную информацию, нам необходимо сначала убедиться, что в буфере что-то появилось, то есть если данные доступны, то их читаем и выводим. Для получения информации о доступных данных существует команда *Serial.available()*, она возвращает количество доступных байт в буфере. Соответственно, если их количество больше нуля, можно читать.

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

void loop( ) {
  if(Serial.available( ) > 0){    // Если данные доступны
    char data = Serial.read(); // прочитаем их в переменную
    // выводим полученные данные в столбик
    Serial.println(data);
  }
}
```

Теперь можем загружать и проверять. Пробуем что-нибудь отправить и видим ответ. Тут необходимо обратить внимание на ещё одну важную настройку в нижнем правом углу консоли монитора порта.



Здесь указывается автоматически отравляемый символ в конце отправки нашего сообщения. Не будем сейчас заострять на нём внимание, просто укажем пункт «Нет конца строки».

Теперь попробуем написать программу, которая будет включать встроенный светодиод, если получена 1, и выключать, если получен 0.

Для начала создадим переменную для хранения номера пина, к которому подключён светодиод.

```
const int led = 13;
```

Укажем его режим.

```
pinMode(led, OUTPUT);
```

И приступим к алгоритму.

Казалось бы, всё просто, пишем:

```
if(data == 1){
  digitalWrite(led, HIGH);
} else if(data == 0){
  digitalWrite(led, LOW);
}
```

Но такая программа работать не будет! Ведь мы отправляем не просто 1 или 0, как целое число, а отправляем именно символ 1 или 0. Существует специальная таблица кодировок, называется ASCII.

ASCII таблица

DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CHAR
0	00	000		32	20	040	SP	64	40	100	@	96	60	140	`
1	01	001		33	21	041	!	65	41	101	A	97	61	141	a
2	02	002		34	22	042	"	66	42	102	B	98	62	142	b
3	03	003		35	23	043	#	67	43	103	C	99	63	143	c
4	04	004		36	24	044	$	68	44	104	D	100	64	144	d
5	05	005		37	25	045	%	69	45	105	E	101	65	145	e
6	06	006		38	26	046	&	70	46	106	F	102	66	146	f
7	07	007		39	27	047	'	71	47	107	G	103	67	147	g
8	08	010		40	28	050	(	72	48	110	H	104	68	150	h
9	09	011		41	29	051	)	73	49	111	I	105	69	151	i
10	0A	012		42	2A	052	*	74	4A	112	J	106	6A	152	j
11	0B	013		43	2B	053	+	75	4B	113	K	107	6B	153	k
12	0C	014		44	2C	054	,	76	4C	114	L	108	6C	154	l
13	0D	015		45	2D	055	-	77	4D	115	M	109	6D	155	m
14	0E	016		46	2E	056	.	78	4E	116	N	110	6E	156	n
15	0F	017		47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020		48	30	060	0	80	50	120	P	112	70	160	p
17	11	021		49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022		50	32	062	2	82	52	122	R	114	72	162	r
19	13	023		51	33	063	3	83	53	123	S	115	73	163	s
20	14	024		52	34	064	4	84	54	124	T	116	74	164	t
21	15	025		53	35	065	5	85	55	125	U	117	75	165	u
22	16	026		54	36	066	6	86	56	126	V	118	76	166	v
23	17	027		55	37	067	7	87	57	127	W	119	77	167	w
24	18	030		56	38	070	8	88	58	130	X	120	78	170	x
25	19	031		57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032		58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033		59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034		60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035		61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036		62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037		63	3F	077	?	95	5F	137	_	127	7F	177	DEL

В ней указано, как зашифрованы символы в различных системах исчисления. Например, символ единицы в десятичной системе исчисления – это 49. Соответственно, необходимо сравнивать именно с этим числом.

Получается, нужно каждый раз залезать в эту таблицу? Конечно нет, это страшно неудобно. Для того чтобы указать, что мы сравниваем именно с символом, необходимо в условии написать 1 и 0 в одинарные кавычки.

```
const int led = 13;
void setup( ) {
  Serial.begin(9600);
  pinMode(led, OUTPUT);
}
void loop() {
  if(Serial.available() > 0){
    char data = Serial.read();
    if(data == '1'){
      digitalWrite(led, HIGH);
    } else if(data == '0'){
      digitalWrite(led, LOW);
    }
  }
}
```

Теперь всё должно работать. Таким образом мы можем реагировать на любой символ с клавиатуры.

Кстати, уверен, ты обратил внимание на запись

```
if(){
}else if(){
}
```

Дело в том, что связанные друг с другом условия принято писать именно так. И таких связанных условий может быть сколько угодно. Прелесть в том, что как только одно из условий окажется верным, остальные даже не проверяются. Если необходимо каждый раз проверять и остальные, использовать else не нужно. Ну и, конечно же, в такой конструкции в конце может быть ещё и просто else. То есть:

```
 if(){
}else if(){
} ...
 else{
}
```

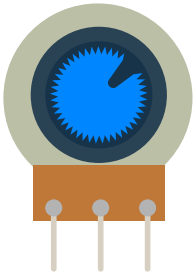
Вот, собственно, и всё, что тебе пока необходимо знать про последовательный порт.

В качестве самостоятельной работы попробуй написать программу, которая будет переключать состояние светодиода или светофора при отправке одного и того же символа.

Потенциометр

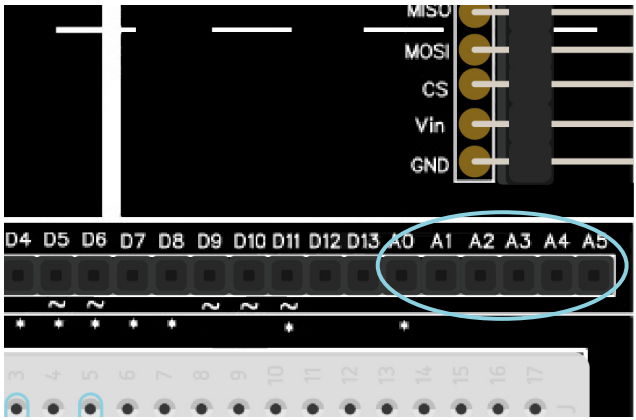
В этом уроке мы познакомимся с аналоговым сигналом и научимся работать с потенциометром.

Итак, начнём с потенциометра – это вот эта крутилка:



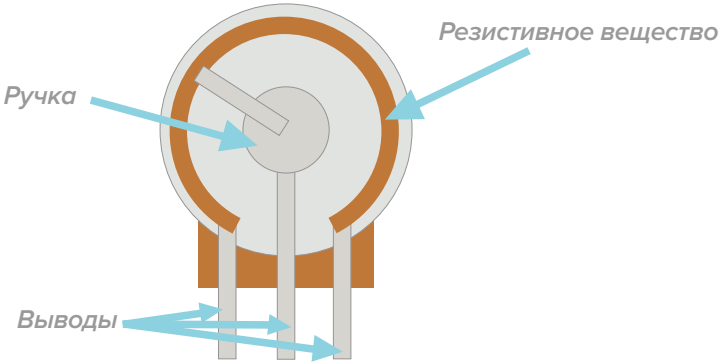
С уроков физики в школе он может быть знаком тебе как переменный резистор. Смысл его в том, что при повороте ручки изменяется сопротивление на его выходе, но не будем забегать вперёд. Для начала, как всегда, разберёмся, а где эта штука применяется. Конкретно такие, как в этом наборе, применяются для настройки чего угодно, будь то громкость или какой-то другой параметр. Помимо этого, потенциометры включены в состав джойстиков, например в геймпаде игровой приставки.

Для того чтобы получать данные с потенциометра, необходимо измерить напряжение на его выходе, и с этим нам помогут специальные ножки микроконтроллера, которые подписаны буквой A (A0-A5).



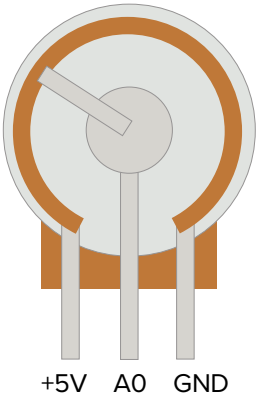
Если посмотреть не на материнскую плату, а на плату Ардуино, то можно найти ещё контакты A6 и A7, но их использование ограничено только измерением напряжения, тогда как ножки A0-A5 являются ещё и цифровыми, то есть ими можно пользоваться ровно так же, как остальными.

Для начала рассмотрим, как устроен потенциометр.



Две его крайние ножки подключены к материалу, имеющему постоянное сопротивление. Центральная подключена к скользящему контакту, который в зависимости от угла поворота ручки перемещается между крайними контактами. Таким образом, сопротивление между любой крайней и средней ножкой можно изменять от 0 Ом до максимального, указанного на корпусе.

Правильное подключение потенциометра выглядит так:



Таким образом, напряжение на средней ноге потенциометра может изменяться от 0 до 5 вольт.

Сигнал, который мы получим с потенциометра, называется аналоговым.

Что же такое аналоговый сигнал? До этого мы имели дело только с цифровыми сигналами, которые имеют всего два состояния – высокий или низ-

кий, то есть 1 или 0. В случае «Ардуинки» 1 – это 5 вольт, 0 – это 0 вольт.

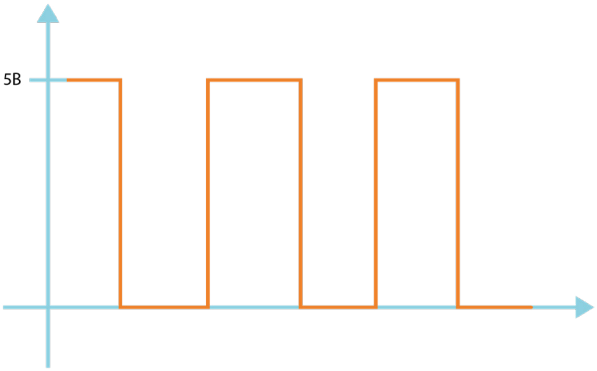


График цифрового сигнала

При этом цифровые ножки не умеют измерять напряжение, ведь ограничены только двумя значениями. Аналоговый же сигнал предполагает, что значение напряжения может быть любым, в нашем случае — в промежутке от 0 до 5 вольт.

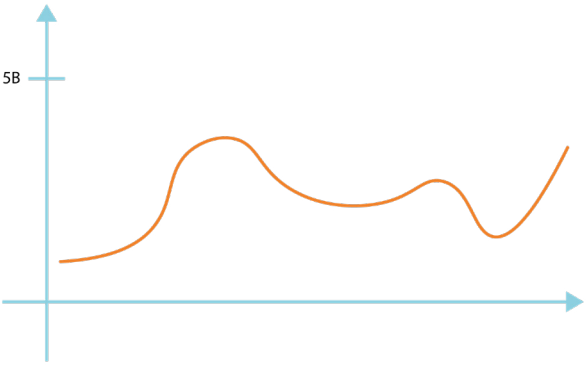
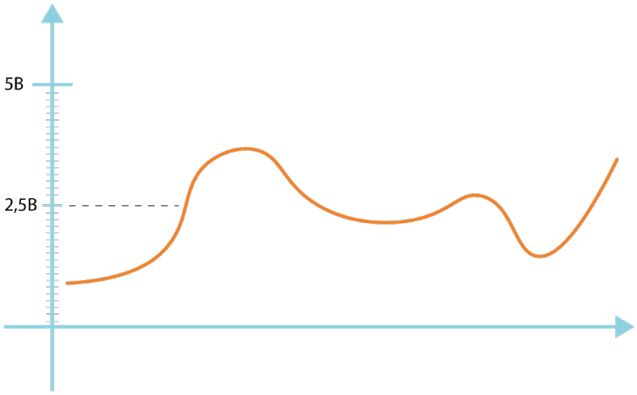


График аналогового сигнала

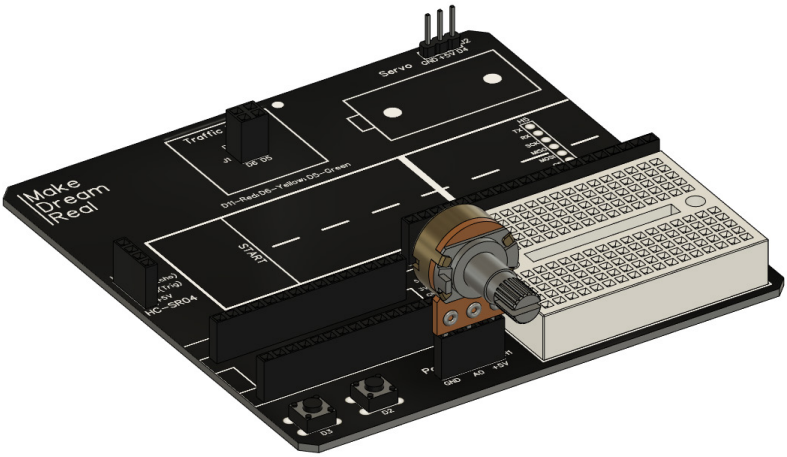
Как же МК может принять такой сигнал, если вся логика внутри него понимает только 0 или 1? Всё дело в аналого-цифровом преобразователе, или АЦП. Это встроенная, или, как говорят, аппаратная возможность микроконтроллера на плате Ардуино. АЦП, как следует из названия, преобразует

аналоговый сигнал в цифровой. Он, по сути, делит диапазон от 0 до 5 вольт на 1 024 отрезка и сопоставляет значение напряжения с одним из них.



Ножки A0-A7 подключены к такому преобразователю, и поэтому их называют аналоговыми. При этом в наш микроконтроллер не завезли ЦАП (цифро-аналоговый преобразователь), который умеет делать из цифрового сигнала аналоговый. В связи с этим ни один пин Ардуино не умеет выдавать аналоговый сигнал.

Что ж, с подключением вроде разобрались. Можно собрать схему на макетной плате либо сразу установить потенциометр на своё место на материнской плате.



Переходим к программе. Сейчас наша задача – вывести значения потенциометра на экран компьютера.

Начинаем, как всегда, с создания переменных. Обрати внимание, что обращаться к аналоговым ножкам следует не просто по номеру, а с буквой – A0.

```
const int pot = A0;
```

Мы будем выводить значения на экран компьютера. Соответственно, в теле функции setup необходимо начать общение с компьютером – по традиции на скорости 9600 бод.

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

Далее, как всегда, режим работы ножки, в нашем случае – на вход.

```
pinMode(pot, INPUT);
```

Ну и теперь всё как в программе с выводом состояния кнопки. В теле функции loop для начала прочитаем значение аналоговой ножки в переменную.

```
int potValue = analogRead(pot);
```

И отправим её на экран.

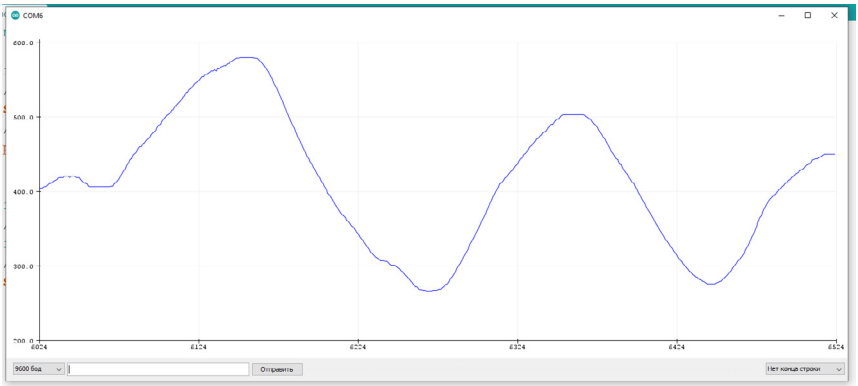
```
Serial.println(potValue);
```

Вот что у нас получилось:

```
const int pot = A0; // пин потенциометра

void setup() {
  // открываем последовательный порт
  Serial.begin(9600);
  // ножка A0 на вход
  pinMode(pot, INPUT);
}

void loop() {
  // получаем значение ножки A0
  int potValue = analogRead(pot);
  // выводим значение с кнопки в последовательный порт
  Serial.println(potValue);
}
```

Теперь, если открыть монитор порта после загрузки, увидим значения от 0 до 1 023 в зависимости от положения ручки.

Куда интереснее смотреть на вывод в виде графика. Открываем «Инструменты» ➜ «Плоттер по последовательному соединению» (не забываем закрыть монитор порта) и наблюдаем график в реальном времени.



Вот и всё! Теперь ты умеешь работать с аналоговыми датчиками.

Ну а бонусом покажу интересный эксперимент.

Попробуй извлечь потенциометр и подключить к пину A0 провод «папа-папа» одним концом, а второй оставь висящим в воздухе.

Мы уже говорили на уроке про подключение кнопки, что в таком случае провод становится антенной и сигнал на нём зависит от внешних факторов. Посмотри на график теперь. Попробуй подносить руку к проводу, коснуться его изоляции и так далее.

Такой вот интересный эксперимент. Ну а на этом первая часть урока закончена.

В качестве самостоятельной работы можешь написать программу, которая включает светодиод, если значение потенциометра больше среднего. Ну и программу, которая включает определённые цвета светофора в разных диапазонах поворота ручки потенциометра.

ШИМ и потенциометр

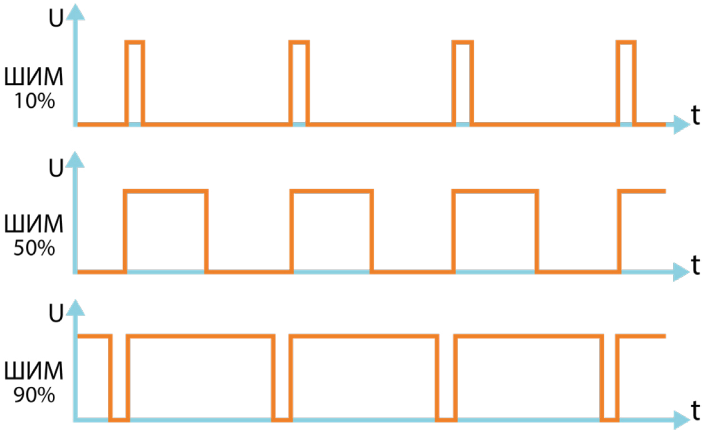
В этой части урока разберём задачу регулировки яркости светодиода при помощи потенциометра. С потенциометром уже разобрались, остаётся яркость. Как вообще её регулировать, если «Ардуинка» умеет выдавать только высокий или низкий сигнал?

Можно заменить резисторы или поставить вместо них регулируемые. Тогда за счёт большего падения напряжения на них яркость светодиодов действительно уменьшится, но это в нашем случае, мягко говоря, неудобно, да и на резисторе будет рассеиваться много энергии.

Или можно подавать на светодиоды аналоговый сигнал, но, как я говорил ранее, Arduino Nano так не умеет.

Делается это при помощи технологии, которая называется ШИМ, или широтно-импульсная модуляция. Давай разбираться.

В уроке про первую программу мы говорили, что если включать и выключать светодиод слишком быстро, мы увидим свечение только вполсилы. Но почему так происходит? Дело в том, что светодиод слишком быстро меняет своё состояние и при этом он одинаковое время включён и выключен, в результате чего мы и видим свечение вполсилы. Это, по сути, есть ШИМ.

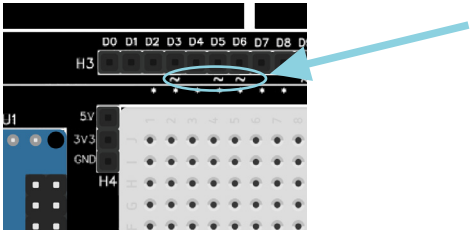


Если уменьшать ширину импульса, то есть времени свечения, светодиод будет гореть тусклее. Если наоборот увеличивать, то ярче.

Получается, широтно-импульсная модуляция – регулировка ширины импульсов. Вот и всё!

Кстати, именно такую технологию используют в RGB-подсветке и даже в экране смартфона!

Теперь к программе. Можно, конечно, сделать всё программно и регулировать ширину импульса (время свечения) при помощи, например, delay, но это просто займёт ядро микроконтроллера и сильно усложнит дальнейшее написание программы. Есть способ получше. Некоторые пины поддерживают автоматическую генерацию ШИМ, то есть ядро в этом процессе не участвует. Такие пины помечены значком волны на материнской плате.



Теперь можно приступать к коду.

Для начала напишем простейшую программу, в которой яркость меняется ступенчато, то есть сначала минимальная, потом средняя, затем максимальная и обратно. Как ты можешь заметить, ножка 13 не поддерживает ШИМ, так что придётся подключать внешний светодиод или использовать светодиоды светофора.

Команда для регулировки ширины импульсов выглядит так:

analogWrite(led, значение от 0 до 255).

255 – это максимальная яркость.

Далее попробуй написать программу самостоятельно, не подглядывая.

В итоге должно получиться что-то такое:

```
const int redLed = 11;              // пин красного светодиода

void setup() {
  pinMode(redLed, OUTPUT); // пин 11 на выход
}

void loop() {
// задаю значение яркости светодиода
analogWrite(redLed, 10);
delay(500);                         // задержка
analogWrite(redLed, 150);
delay(500);
analogWrite(redLed, 255);
delay(500);
}
```

Если загрузить эту программу, мы увидим, как яркость меняется ступенчато.

Теперь попробуем плавно регулировать яркость от меньшего к большему и наоборот.

По сути, нам нужно просто увеличивать значения яркости от 0 до 255 и наоборот.

Как бы ты писал такую программу? Написать 510 строк для этого – очень плохой вариант. В подобных случаях удобно использовать циклы. В языке C++ их несколько, и сейчас мы познакомимся с одним из них. Синтаксис будет выглядеть так:

```
for(переменная счётчик; условие; изменение переменной){
}
```

Сначала обозначаем переменную, которая будет считать итерации (проходы), её можно создать прямо здесь. Например:

```
for(int i = 0;
```

Кстати, через запятую можно создать несколько переменных. Например:

```
for(int i = 0, j=255;
```

Потом идёт условие – пока оно верно, цикл выполняется. Например:

```
for(int i = 0; i < 255;
```

Здесь тоже можно использовать двойные условия по типу:

```
for(int i = 0, j=255; i < 255 && j > 0;
```

Далее изменение счётчика.

Например:

```
for(int i = 0; i < 255; i++)
```

Также можно изменять несколько переменных. Например:

```
for(int i = 0, j=255; i < 255 && j > 0; i++, j--)
```

> ++ значит увеличение на 1. Эта запись эквивалентна записи i=i+1 и i+=1. Таким образом можно сокращать вычисления.
> i+=a ← i=i+a;
> i-=a ← i =i-a;
> i*=a ← i=i*a;
> i/=a ← i=i/a;
> i%=a ← i=i%a – возвращает остаток от деления;
> где a – любое число

В нашем случае цикл, увеличивающий яркость, будет выглядеть так:

```
for(int i = 0; i < 255; i++){
   analogWrite(redLed, i);
}
```

Традиционно для счётчика используют переменную с именем **i**. Заметим, что переменную, объявленную в цикле, можно использовать только в теле цикла. Когда цикл заканчивается, переменная удаляется.

Но если попробовать загрузить такую программу, мы увидим, что светодиод просто горит и никак не изменяет свою яркость. Всё дело в том, что цикл проходит слишком быстро, и мы просто не успеваем заметить изменение яркости. Чтобы это исправить, добавим в конце цикла

```
delay(10);
```

Соответственно, значение задержки будет регулировать скорость изменения яркости.

Попробуй написать цикл уменьшения яркости самостоятельно.

Итоговая программа будет выглядеть так:

```
const int redLed = 11;              // пин красного светодиода

void setup() {
  pinMode(redLed, OUTPUT); // пин 11 на выход
}

void loop() {
  for(int i = 0; i < 255; i++){      // цикл от 0 до 255
    analogWrite(redLed, i);
    delay(10);
  }
  for(int i = 255; i > 0; i--){              // цикл от 255 до 0
    analogWrite(redLed, i);
    delay(10);
  }
}
```

Теперь давай напишем программу, которая будет регулировать яркость светодиода в зависимости от угла поворота ручки потенциометра. Как обычно, попробуй написать сам.

Пожалуй, больше не буду акцентировать внимание на переменных и режимах работы. Перейдём сразу к делу. Если напрямую передать значение ножки A0 в изменение яркости, получится не совсем то, что мы хотели. Как только значение доберётся до 256, переменная переполнится и пойдёт по второму кругу, и так 4 раза за весь поворот ручки потенциометра. Соответственно, необходимо преобразовать значения из диапазона от 0 до 1023 в диапазон от 0 до 255. Сделать это можно, просто поделив 1023 на 255. Мы получим 4,01... И теперь, разделив полученное значение на это число, мы переведём его в нужный диапазон. Но для такого случая в **Arduino IDE** есть уже встроенная функция **map**, которая возвращает уже преобразованное значение. Она принимает сразу 5 аргументов:

map(преобразуемое значение, минимальное значение исходного диапазона, максимальное значение исходного диапазона, минимальное значение целевого диапазона, максимальное значение целевого диапазона).

В итоге получится как-то так:

```
const int pot = A0;                  // пин потенциометра
const int redLed = 11;               // пин красного светодиода

void setup() {
  pinMode(pot, INPUT);           // ножка A0 на вход
  pinMode(redLed, OUTPUT);// пин 11 на выход
}

void loop() {
  // получаем значение ножки A0
  int potValue = analogRead(pot);
  // перобразуем диапазон значения
  potValue = map(potValue, 0, 1023, 20, 255);
  // устанавливаем яркость
  analogWrite(redLed, potValue);
}
```

На этом, пожалуй, закончим.

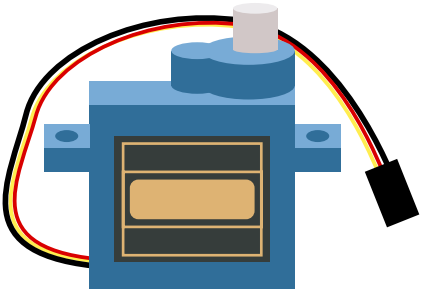
В качестве самостоятельной работы попробуй:

1) Написать программу плавного перетекания цвета светофора из одного в другой при помощи циклов. То есть горит, например, красный. Его яркость плавно уменьшается, и одновременно с этим плавно увеличивается яркость жёлтого, пока красный полностью не погаснет, а жёлтый не дойдёт до максимальной яркости. Далее так же из жёлтого в зелёный и обратно.

2) Регулировать ручкой потенциометра это перетекание света.

Сервопривод

В этом уроке мы познакомимся с сервоприводом. Это такой моторчик.



И сразу важное предостережение: **руками крутить вал не стоит**, это может «убить» сервопривод.

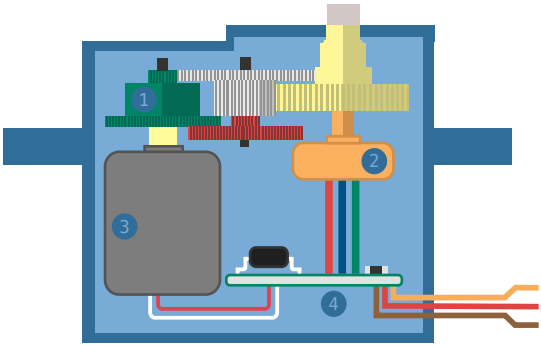
И чтобы понять, почему, давай разбираться с его устройством.

Что вообще такое сервопривод? И чем он отличается от обычного моторчика с редуктором? Его главная фишка в том, что он умеет поворачиваться в определённый угол от 0 до 180 градусов.

Например, вал находится в положении 10 градусов, если «серва» получит команду – скажем, 90 градусов, – вал переместится в это положение и будет удерживать его. Если ещё раз отправить 90 градусов, то вал останется на месте, ведь он уже и так в нужном положении.

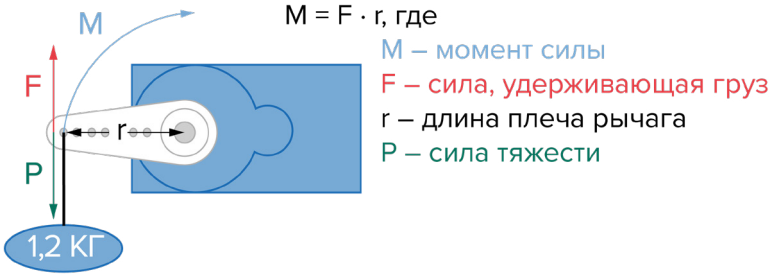
Сервоприводы часто используют в радиоуправляемых моделях и в различных робототехнических проектах. Мы же будем использовать его как шлагбаум.

Что же у него внутри?



1 — Редуктор
2 — Потенциометр
3 — Двигатель
4 — Блок управляющей электроники

Электродвигатель приводит вал в движение через редуктор (набор шестерней). Как ты думаешь, для чего нужен этот редуктор? Дело в том, что электродвигатель сам по себе очень слабый и очень быстро крутится, а редуктор преобразует скорость вращения в силу, или, как правильно говорить, крутящий момент. В нашем случае крутящий момент может доходить до 1,2 кг/см. Это значит, что на рычаге 1 см от вала сервопривод может удержать до 1,2 кг.

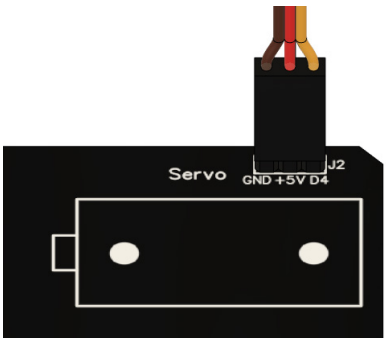


В нашем сервоприводе редуктор пластиковый, и, если пытаться дать ему нагрузку больше, зубчики шестерёнок могут сломаться и сервопривод выйдет из строя.

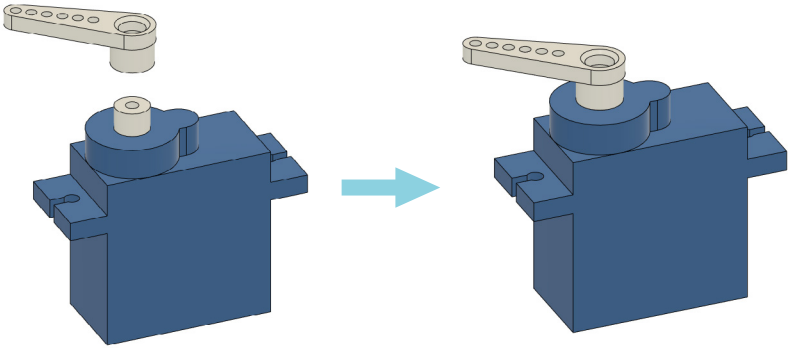
Идём дальше. Как же он понимает, в каком положении находится? К выходному валу присоединён потенциометр. По принципу и устройству ровно такой, с каким мы уже знакомы. Благодаря ему сервопривод и понимает, в каком положении находится.

Ну и остаётся плата управления, которая принимает сигналы извне от потенциометра и передает управляющий сигнал на электродвигатель.

У сервопривода три провода. Коричневый – земля, красный – плюс питания, жёлтый – сигнал. Можешь сразу подключить его на своё место на материнской плате.



В корпус пока собирать не обязательно. Единственное, для наглядности лучше использовать насадку на вал.



Теперь можем переходить к программе.

Для управления сервоприводом применяют ШИМ сигнал, то есть от длины импульса будет зависеть положение вала. Но это не совсем удобно. Для упрощения работы с сервоприводом мы будем использовать библиотеку.

Библиотека – это, по сути, кусок программы, в котором созданы команды, в нашем случае для работы с «сервой». И чтобы эти команды можно было использовать, библиотеку необходимо подключить. В самой первой строчке пишем:

```
#include <Servo.h>
```

Команды, начинающиеся с решётки, называются директивами препроцессора. Они выполняются перед компиляцией. В данном случае вставляется содержимое файла **Servo.h**, который находится в папке с библиотеками в недрах программы **Arduino IDE**.

Не забудем создать переменную для пина сервопривода:

```
const int servoPin = 4;
```

Теперь, когда библиотека подключена, мы можем пользоваться её командами. Для начала необходимо объявить новый сервопривод. Делается это вне тел функций — по сути, как глобальная переменная.

```
Servo ser;
```

Тут, как всегда, имя можно выбирать на своё усмотрение.

Далее в теле функции setup необходимо указать, к какому пину подключён сервопривод.

```
ser.attach(servoPin);
```

Обрати внимание: синтаксис как в случае с использованием последовательного порта. Каждый раз, когда мы что-то хотим сделать с серво-

приводом, нужно писать его имя, точку, потом команду.

Как только команда *ser.attach(4)* выполнится, сервопривод начнёт удерживать своё положение. Если необходимо его отключить, существует команда *ser.detach()*, тут она нам не пригодится.

Чтобы указать угол поворота, используем команду

ser.write(от 0 до 180);

Попробуй сам написать программу, которая заставит сервопривод двигаться ступенчато, в положение, скажем, 0, 90, 180 и обратно.

Код будет выглядеть так:

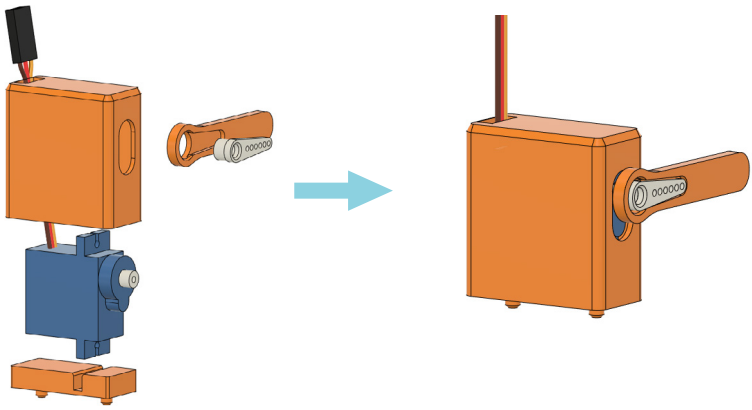
```
#include <Servo.h>          // подключаем библиотеку

const int servoPin = 4;     // пин сервопривода

Servo servo;                // объявляем новый сервопривод

void setup() {
  servo.attach(servoPin); // включаем сервопривод на 4 пине
}

void loop() {
  servo.write(0);           // устанавливаем угол «сервы»
  delay(1000);              // даём время повернуться
  servo.write(90);
  delay(1000);
  servo.write(180);
  delay(1000);
}
```

Важно помнить, что сервоприводу нужно время для поворота, поэтому без задержки никак.

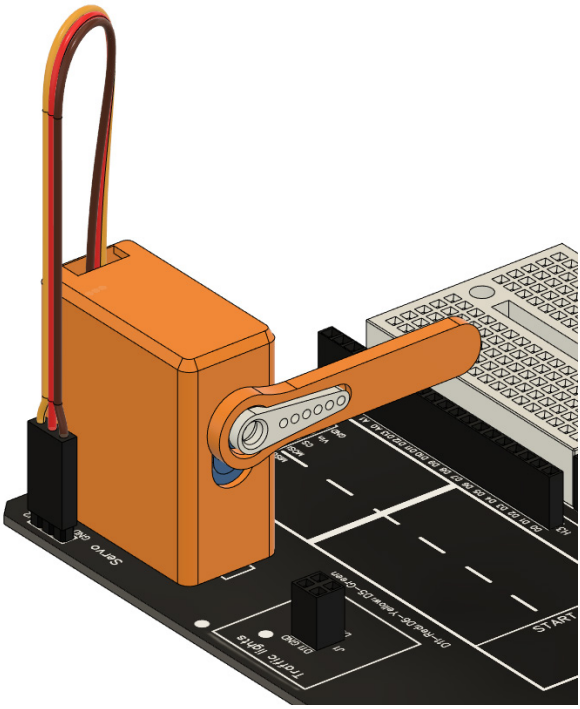
Перед сборкой шлагбаума стрелы необходимо установить вал в крайнее положение «закрыт», чтобы избежать столкновения с материнской платой при работе. Для этого загружу такую программу:

```
#include <Servo.h>          // подключаем библиотеку

const int servoPin = 4;     // пин сервопривода

Servo servo;                // объявляем новый сервопривод

void setup() {
  // включаем сервопривод на 4 пине
  servo.attach(servoPin);
  // устанавливаем сервопривод в положение — «закрыт»
  servo.write(0);
}

void loop() {
}
```

Теперь можно приступать к сборке шлагбаума по схеме ниже.



Установим шлагбаум на материнскую плату и подключим провод, соблюдая цвета.



Уверен, что теперь для тебя не составит труда написать программу для управления сервоприводом. Поэтому вот тебе задания для самостоятельной работы.

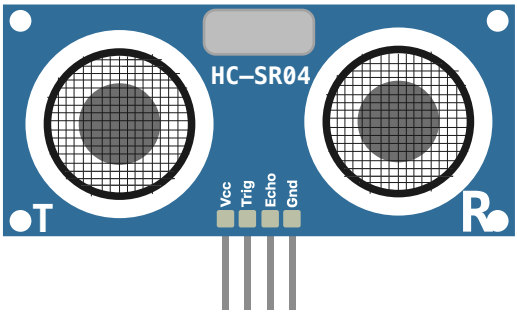
1) Добавить шлагбаум к алгоритму светофора.

2) Управлять шлагбаумом при помощи потенциометра.

3) Изменять состояние шлагбаума по нажатию на кнопку.

Датчик расстояния

В этом уроке мы познакомимся с ультразвуковым датчиком расстояния.



Мы уже собрали макет шлагбаума со светофором. Но как бы это работало в реальности?

Автомобиль дожидается разрешающего сигнала светофора и открытия шлагбаума, начинает движение, но по какой-либо причине останавливается прямо на линии стрелы шлагбаума. Как только время зелёного цвета закончится, шлагбаум ударит по автомобилю. Для того чтобы подобных случаев не происходило, применяют различные датчики. В нашем случае будем измерять расстояние перед шлагбаумом для обнаружения препятствий.

Такие датчики часто применяются, например, в роботах для определения расстояния до препятствий. Кстати, автомобильные парктроники тоже являются ультразвуковыми датчиками расстояния, просто другой конструкции.

Как же этот датчик может определять расстояние, используя звук? Обрати внимание, у датчика есть два таких круглых глазка. Один – это динамик, другой – микрофон. У него четыре ножки. Две, как всегда — это питание. Ножка trig принимает сигнал от «Ардуинки». Если подать на неё импульс длительностью 10 мкс, датчик произведёт измерение в автоматическом режиме. Динамик включится, звук отразится от препятствия и вернётся в микрофон.

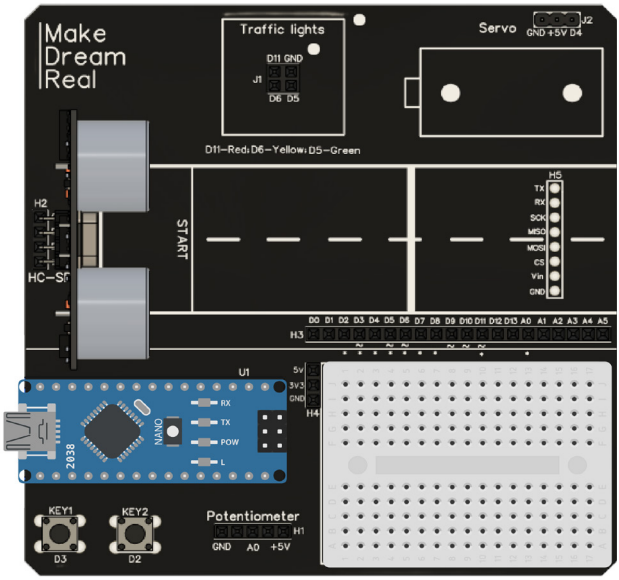
> Динамик работает на частоте 40 кГц, это нужно для того, чтобы датчик мог отличить свой звук от шумов, ну и эта частота вдвое выше, чем максимальная, которую человек может услышать.

После измерения на ножке echo появится импульс, равный по времени путешествию звука. Далее, уже зная скорость звука, можно

вычислить расстояние, им пройденное.

Вот, на самом деле, и всё. Теперь можно переходить к написанию программы.

Установи датчик на его место на материнской плате, только не перепутай, он должен смотреть на плату.



Для работы с этим датчиком существуют различные библиотеки, но сейчас мы попробуем обойтись без них и взаимодействовать с датчиком напрямую.

Напишем программу, которая будет выводить данные с датчика на экран компьютера.

Вначале всё стандартно. Переменные, начать общение по последовательному порту, режимы работы пинов. Будь внимателен при установке режима работы. Ножка, подключённая к контакту trig, работает на выход; ножка, подключённая к контакту echo, – на вход.

```
const int trig = 8, echo = 7;
void setup() {
  Serial.begin(9600);
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
}
```

Теперь получим расстояние. Как мы разобрались ранее, для измерения расстояния на ножку trig необходимо подать импульс 10 мкс, то есть на ножке 8 должен быть высокий сигнал целых 10 мкс. Только подумай,

насколько это быстро. Приставка «микро» означает миллионные доли. Ну да ладно. Так вот, для начала записываем высокий сигнал в пин 8.

```
digitalWrite(trig, HIGH);
```

Теперь надо засечь 10 мкс, delay для этого не подходит, ведь в качестве аргумента он принимает мс. Для таких случаев существует команда *delayMicroseconds()*, которая в качестве аргумента принимает, как ни странно, микросекунды.

```
delayMicroseconds(10);
```

Ну и после задержки нужно записать низкий сигнал:

```
digitalWrite(trig, LOW);
```

Теперь датчик начал измерение, по итогам которого на ножке echo появится высокий сигнал, длительностью равный задержке между включением динамика и приёмом звука в микрофон. Соответственно, нам нужно как-то этот сигнал измерить. Для этого используется функция *pulseIn()*. Она принимает два аргумента: ножка, высокий или низкий сигнал измерять. Длительность возвращает в миллисекундах. Давай запишем его в переменную. int для такого случая уже не подходит. Дело в том, что переменная типа int занимает всего 2 байта и в неё можно вместить значение от -32768 до 32767, а у нас тут микросекунды и значение с датчика может быть получено куда большее, чем максимальное значение диапазона int. Есть лайфхак, который поможет увеличить максимальное значение диапазона: можно просто убрать знаковые значения, то есть отрицательные. Для этого существует запись *unsigned int*, что значит «беззнаковый», и тогда диапазон станет от 0 до 65535. Но, как ни странно, его нам тоже не хватает.

Здесь нужно использовать тип данных с большим диапазоном. И такой есть, зовут его long. Занимает 4 байта, и диапазон, соответственно, от -2147483648 до 2147483647. Для него также справедливо использование приставки unsigned, что, на самом деле, более грамотно, ведь длительность просто не может быть отрицательной. И тогда диапазон станет от 0 до 4294967295, что, разумеется, избыточно, зато грамотно.

```
unsigned long duration = pulseIn(echo, HIGH);
```

Теперь время нужно перевести в сантиметры, для этого необходимо учитывать скорость звука, температуру воздуха, молекулярную массу воздуха и всякое такое. Но для нормальных условий используют просто коэффициент, равный 29,1. Обрати внимание, что число это нецелое. Можно, конечно, округлить и писать просто 29, но для большей точности лучше использовать тип данных **float**. Он создан для работы с числами с плавающей точкой, то есть для десятичных дробей.

Казалось бы, следующая строчка должна выглядеть так:

```
float cm = duration / 29.1 / 2;
```

Так вот, в нашей формуле **duration** – это целое число, 29,1 – с плава-

ющей точкой, 2 – снова целое, а ответ должен получиться с плавающей точкой. Дело в том, что в таких случаях необходимо приводить все числа к одному типу данных, иначе «Ардуинка» в процессе вычислений может дать неправильный ответ, ведь никаким округлением по правилам она заниматься не будет. Так что правильная запись будет выглядеть так:

```
float cm = float(duration) / 29.1 / 2.0;
```

Кстати, на 2 делим, так как звук проходит расстояние от датчика до препятствия дважды: туда и обратно.

Ну вот и всё! Теперь у нас есть переменная cm, в которой хранится расстояние до объекта. Осталось вывести её на экран.

```
const int trig = 8, echo = 7;   // пины датчика расстояния

void setup() {
  Serial.begin(9600);     // Начинаем обмен данными по UART
  pinMode(trig, OUTPUT);    // пин 8 на выход
  pinMode(echo, INPUT);     // пин 7 на вход
}

void loop() {
  // импульс 10 мкс
  digitalWrite(trig, HIGH);
  delayMicroseconds(10);
  digitalWrite(trig, LOW);

  // получаем время путешествия звука
  unsigned long duration = pulseIn(echo, HIGH);
  // преобразуем в расстояние
  float cm = float(duration) / 29.1 / 2.0;
  // выводим в консоль
  Serial.print("Distance = ");
  Serial.print(cm);
  Serial.println(" cm.");
}
```

Загружаем программу и наблюдаем результат в мониторе порта.

Если расстояние не изменяется, наклони датчик чуть вверх – возможно, он измеряет расстояние до объектов на плате.

Функции и цикл while

Общая программа со светофором и шлагбаумом становится всё больше, и сейчас наша задача – добавить сюда измерение расстояния перед закрытием шлагбаума.

Ничего сложного: можно просто вставить код, полученный в первой части урока, в необходимое место, и всё. Но для удобства предлагаю создать собственную функцию измерения расстояния.

Для начала немного об этих самых функциях. В недрах программы Ардуино находятся различные библиотеки, с этим ты уже знаком. Одна из них – это библиотека **Arduino.h**, в ней расписаны такие функции, как **pinMode**, **digitalWrite**, **digitalRead**, ну и так далее. В предыдущих уроках я в основном обзывал их командами, чтобы тебя не путать. На самом же деле, их правильно называть функциями – точно так же, как **setup** и **loop**, – просто их тела расписаны в библиотеке **Arduino**, что позволяет нам, не задумываясь, ими пользоваться.

И вот пришло время создать свою функцию. Давай рассмотрим несколько вариантов сразу на примере. За основу возьмём уже написанную программу.

Над функцией setup принято объявлять собственные функции, ведь их, как и переменные, необходимо объявлять до использования. Имена функций тоже придумываем самостоятельно. В нашем случае функция измеряет расстояние, поэтому назову её **GetDistance**. Ты уже много раз видел, как объявляют функции, на примере **setup** и **loop**. Исходя из этого, запись будет выглядеть так

```
void GetDistance(){
}
```

Тут всё стандартно: имя, круглые скобочки, тело. Но что же всё-таки значит это слово **void**? Переводится оно как «пустой», что в нашем случае означает отсутствие типа данных, то есть функция ничего не возвращает. Примером таких функций может служить **pinMode** или **digitalWrite**. Они просто выполняют какую-то задачу и по итогу их нельзя приравнять ни к какому типу данных. Такие функции, как **digitalRead** или **pulseIn**, помимо выполнения каких-либо действий, ещё и возвращают значение, которое можно записать, например, в переменную.

В нашем случае правильно будет объявить функцию именно с типом данных и возвращать расстояние в сантиметрах. Осталось только определиться с типом данных. Переменная cm у нас **float**, значит и функция должна быть **float**:

```
float GetDistance(){
}
```

В тело функции копируем код, полученный в первой части урока, и, чтобы функция вернула значение, ниже, в её тело запишем:

```
return cm;
```

После чего нашу функцию можно будет использовать, например, так:

```
const int trig = 8, echo = 7;

// Функция измерения расстояния
float GetDistance() {
  digitalWrite(trig, HIGH);
  delayMicroseconds(10);
  digitalWrite(trig, LOW);

  unsigned long duration = pulseIn(echo, HIGH);
  float cm = float(duration) / 29.1 / 2.0;
  return cm; // Возвращаем расстояние в см
}

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

  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
}
void loop() {
  // Выводим в порт расстояние в см
  Serial.print("Distance = ");
  Serial.print(GetDistance());
  Serial.println(" cm.");
}
```

Обрати внимание: круглые скобочки нужно ставить обязательно.

Теперь давай всё-таки добавим в наш основной код измерение расстояния. Пусть если перед шлагбаумом препятствие, система ждёт, пока оно исчезнет.

Первым делом в основную программу добавим всё, что связано с датчиком расстояния: переменные, функцию, режимы работы ног.

Теперь находим место в нашем алгоритме, где опускается шлагбаум – в идеале это должно происходить уже после переключения светофора на красный. Если написать тут что-то вроде:

```
if(GetDistance() < 5){
```

```
}else{
  servo.write(180);
}
```

то даже если препятствие будет обнаружено, система просто продолжит работу, не закрывая шлагбаум, а нам нужна полная остановка системы. И в данном случае лучше всего использовать цикл **while**. Его синтаксис выглядит так:

```
while(условие){
}
```

Пока условие в круглых скобках верно, цикл выполняется.

В нашем случае:

```
while(GetDistance() < 5){
}
```

Кстати, когда тело цикла остаётся пустым, как в нашем случае, можно заменить его на точку с запятой:

```
while(GetDistance() < 5);
```

Теперь, пока расстояние меньше 5 см, система будет ждать выход из цикла.

Ну вот, пожалуй, и всё. Мы закончили знакомиться со всеми компонентами набора и теперь наша задача – заставить их работать вместе.

В итоге всё должно работать так:

Светофор и шлагбаум работают в автоматическом режиме, при этом если перед шлагбаумом препятствие, а пора перекрывать движение, загорается красный свет и система ждёт, пока препятствие исчезнет. Если шлагбаум нельзя закрыть 10 секунд и больше, то необходимо включить жёлтый мигающий. Вернуть систему обратно можно по нажатию на KEY2. Кнопка KEY1 досрочно переключает состояние светофора. Кнопка KEY2 включает и выключает мигающий жёлтый. Потенциометр настраивает яркость светодиодов светофора.

Следующие уроки будут посвящены разработке ПО для реализации описанной задачи.

Можешь самостоятельно подумать, почему тот подход, который мы использовали до этого, уже не будет работать, что мешает реализовать одновременную работу всех компонентов системы?

millis(), или прощай, delay()

Вот мы и добрались до финала. Всё, что будет происходить дальше, можно назвать только одним словом – ХАРДКОР. В этом уроке ты познакомишься с новыми принципами программирования и, несомненно, испытаешь стресс. Поэтому я напомню, что очень важно делать перерывы, особенно если устал и ничего не понимаешь. Не всё будет понятно с первого раза, поэтому изучай этот урок не торопясь. По мере прохождения старайся останавливаться и попытаться продвинуться вперёд самостоятельно, как бы опережая. Так ты извлечёшь максимальную пользу из материала.

Приступим к решению нашей задачки. Продублирую ещё раз её условие:

Светофор и шлагбаум работают в автоматическом режиме, при этом если перед шлагбаумом препятствие, а пора перекрывать движение, загорается красный свет и система ждёт, пока препятствие исчезнет. Если шлагбаум нельзя закрыть 10 секунд и больше, то необходимо включить жёлтый мигающий. Вернуть систему обратно можно по нажатию на KEY2. Кнопка KEY1 досрочно переключает состояние светофора. Кнопка KEY2 включает и выключает мигающий жёлтый. Потенциометр настраивает яркость светодиодов светофора.

Казалось бы, автоматическая работа светофора и шлагбаума, да ещё и с датчиком расстояния уже была реализована в предыдущих уроках. Но давай подумаем. Например, потенциометр должен в любой момент регулировать яркость светофора, но даже если вставить код, за это отвечающий, после каждой строчки основной программы, мы всё равно не сможем плавно регулировать яркость, так как во время работы функции *delay* это неизбежно приведёт к задержкам. Дело в том, что функция *delay* останавливает работу микроконтроллера, ведь всё, что он делает во время её работы, крутится в цикле, пока не пройдёт заданное время. По сути, основную часть времени микроконтроллер занимается просто отсчётом времени, что очень затрудняет реализацию нашей задачи.

Тут стоит сказать, что существует функция **yield()**. Она постоянно вызывается при работе *delay* и, соответственно, в её теле можно расписать то, что нужно делать постоянно, в нашем случае – корректировать яркость. Но дело в том, что использование этой функции может привести к сильному усложнению и нечитаемости кода, да и в целом её использование является плохим тоном.

В этой части урока мы научимся избавляться от *delay*, используя встроенный таймер, который начинает отсчёт каждый раз при подключении питания к плате. И сейчас нашей задачей будет переписать программу управления светофором, используя этот таймер.

Также в этом уроке мы начнём знакомиться с понятием «архитектура программы» и научимся более грамотно писать код.

Переменные

Начнём с создания переменных. Сейчас быстро освежим, что мы уже знаем касательно этой темы, и добавим немного нового.

Для объявления значений, которые не будут изменяться в процессе работы программы, принято использовать константу. Её значение можно только прочитать, изменить нельзя. В нашем случае это бы выглядело так:

```
const int redLed = 11;
```

Помимо этого, константы принято писать заглавными буквами, то есть так:

```
const int RED_LED = 11;
```

И ещё очевидно, что номер ножки не может быть отрицательным. На что может указать ключевое слово **unsigned**. Плюс ко всему, тип данных *unsigned int* занимает 2 байта и может вместить в себя значения от 0 до 65535. Такой большой объём нам не нужен, и в случае объявления номера ножки можно обойтись одним байтом, который способен вместить значения от 0 до 255. Так что максимально грамотная запись будет выглядеть так:

```
const unsigned byte RED_LED = 11;
```

Кстати, тип данных byte по умолчанию беззнаковый, так что указывать это не обязательно.

```
const byte RED_LED = 11;
```

Фу-у-ух! Что-то как-то много. Это всё связно с тем, что существуют правила при оформлении программ. Они не относятся к синтаксису и при компиляции не вызовут ошибок, но соблюдать их нужно обязательно. Сравнить это можно с оформлением текста в тетради по русскому языку. Все знают, что писать нужно строго по линиям, а не как попало – например по диагонали, – нельзя писать на полях тетради и так далее. Благодаря соблюдению этих правил текст гораздо проще воспринимать. Так же и в программировании: если не соблюдать подобные правила вроде отступов или имён переменных, код становится нечитаемым и даже его автор через время не сможет в нём разобраться и уж тем более вносить изменения. Так что настоятельно прошу тебя внимательно прислушиваться к тому, о чём будет идти речь далее.

Вернёмся к объявлению номеров ножек. Дело в том, что при объявлении переменной выделяется область в оперативной памяти, а места там у микроконтроллера, прямо скажем, не ахти, и в нашем случае – всего лишь 2 килобайта. Этого, на самом деле, хватит для большинства задач, но грамотнее будет экономить. Для этого необходимо писать значение констант каждый раз напрямую, то есть не используя переменные. Но неужели это откатит нас назад и придётся жертвовать удобством? Конечно нет, давать имена пинам очень и очень важно, и для этого мы будем использовать

директиву препроцессора **#define**.

```
#define RED_LED 11
```

В уроке про сервопривод мы уже встречались с командами препроцессора, но я напомню. Такие команды выполняются перед компиляцией. В данном случае перед загрузкой программы все имена **RED_LED** будут заменены на 11 во всей программе. То есть директива define заменяет левую часть на правую.

millis()

Фух, с ножками разобрались, теперь пора переходить к отказу от **delay**. Для начала напишем простую программу мигания светодиодом. Пусть мигать будет встроенный светодиод на пине 13. Начало, как всегда:

```
#define LED 13 // пин встроенного светодиода
void setup() {
  pinMode(LED, OUTPUT); // пин 13 на выход
}
```

А вот теперь самое интересное. Я уже говорил про встроенный таймер, который начинает отсчёт в момент старта программы. Соответственно, в любой момент её работы можно получить значение этого таймера. Делает это функция **millis()**, она возвращает значения таймера в миллисекундах. Важно сказать, что считать таймер может не до бесконечности, его значение переполнится после 4 294 967 295 мс или примерно 50 дней, далее таймер обнуляется и начинает считать заново.

Теперь давай научимся это использовать. Алгоритм программы будет звучать так: «Если прошло 1000 мс, то изменить состояние светодиода». Для того чтобы понять, что время прошло, необходимо текущее время с чем-то сравнивать. Для этого создадим переменную типа *unsigned long*:

```
unsigned long lastTime = 0;
```

Она будет хранить предыдущее время изменения состояния. И тогда алгоритм будет: «Если предыдущее время меньше, чем текущее, на 1000 мс, измени состояние светодиода». И в итоге получится:

```
#define LED 13 // пин встроенного светодиода

// предыдущее время изменения состояния светодиода
unsigned long lastTime = 0;

void setup() {
  pinMode(LED, OUTPUT); // пин 13 на выход
}

void loop() {
  // если последнее сохранённое время меньше, чем текущее, на 1000 мс
  if(millis() - lastTime >= 1000){ // если прошло 1000 миллисекунд
    lastTime = millis(); // запомним время изменения состояния
    // инвертируем состояние ножки 13
    digitalWrite(LED, !digitalRead(LED));
  }
}
```

Обрати внимание, необходимо каждый раз запоминать время изменения состояния светодиода.

Теперь самостоятельно попробуй написать программу мигания сразу двумя лампочками с разной частотой.

Ну, на этом с millis вроде всё. В качестве самостоятельной работы попробуй написать код работы светофора без delay. А как это можно сделать, мы разберём в следующей части урока.

Машина состояний и алгоритм светофора

Напомню, что для максимально продуктивного прохождения урока следует периодически останавливаться в поглощении материала и пытаться продвигаться вперёд самостоятельно. А в простом списывании смысла не будет.

Ну что ж, пришло время разобраться с реализацией светофора без использования *delay*. На самом деле, существует очень много вариантов решения этой задачки, мы же рассмотрим подход с использованием так называемой машины состояний, или конечного автомата. Это такой способ сделать программу более читаемой, соответственно, и более удобной для разработки. Здесь мы будем работать с состояниями системы, в нашем случае — светофора. Вот какие у светофора состояния? Красный, красный и жёлтый вместе, зелёный, зелёный мигающий, жёлтый, жёлтый мигающий. Всё, что нам нужно делать, – просто переключаться между этими состояниями в зависимости от алгоритма. В нашем случае нужно просто переключать первые пять состояний в цикле, а шестое включать в определённых ситуациях.

Для начала про оформление этих состояний. Объявляются они вверху при помощи директив define таким образом:

```
// Состояния светофора
#define RED 0
#define RED_YELLOW 1
#define GREEN 2
#define GREEN_BLINK 3
#define YELLOW 4
#define YELLOW_BLINK 5
```

У каждого состояния есть свой номер, начинать нумерацию принято с нуля.

При автоматическом режиме работы светофора, если прошло определённое время, следует включить следующее состояние. Но время это будет разным для разных состояний, соответственно, его тоже нужно задать там же вверху.

```
// Время состояний
#define RED_TIME 10000
#define RED_YELLOW 1000
#define GREEN_TIME 10000
#define BLINK_GREEN_DELAY 1000
#define YELLOW_TIME 1000
```

```
#define BLINK_YELLOW_TIME 1000
```

Ещё нам понадобится переменная для хранения текущего состояния и предыдущего времени изменения состояния.

```
byte trafficLightsState = RED; // текущее состояние светофора
// предыдущее время изменения состояния светофора
unsigned long lastTime = 0;
```

Ну а дальше по ходу разберёмся.

Нам необходимо включать состояние, если оно текущее. Для этого можно использовать if таким образом:

```
if(trafficLightsState == RED){

}
else if(trafficLightsState == RED_YELLOW){

}
else if(trafficLightsState == GREEN_BLINK){

}
else if(trafficLightsState == GREEN_BLINK){

}
else if(trafficLightsState == YELLOW){

}
else if(trafficLightsState == YELLOW_BLINK){

}
```

Переключение должно происходить только спустя определённое время, поэтому заключим выбор состояния в условие, сравнивающее millis() с предыдущим изменением. Получается, ещё нужна переменная, хранящая время до смены состояния.

```
unsigned long stateTime; // Время до смены состояния
...
if(millis() - lastTime >= stateTime){
   lastTime = millis();
   if(trafficLightsState == RED){
...
```

Что нужно делать, если текущее состояние светофора – красный? Очевидно, включить красный светодиод, а остальные выключить. Здесь же хорошо бы задать время работы режима и следующий режим.

```
if(trafficLightsState == RED){
    digitalWrite(RED_LED, HIGH);
    digitalWrite(YELLOW_LED, LOW);
    digitalWrite(GREEN_LED, LOW);
    stateTime = RED_TIME;
    TrafficLightsState = RED_YELLOW;
  }
```

С состоянием красный и жёлтый, зелёный и жёлтый всё так же. А что делать с мигающими состояниями? В случае жёлтого нужно просто изменять состояние светодиода через заданное время. Делать это можно здесь же:

```
else if(trafficLightsState == YELLOW_BLINK){
      digitalWrite(RED_LED, LOW);
      digitalWrite(YELLOW_LED, !digitalRead(YELLOW_LED));
      digitalWrite(GREEN_LED, LOW);
      stateTime = YELLOW_BLINK_TIME;
    }
```

Ну а в случае зелёного нужно просто добавить счётчик количества миганий, и, когда он закончится, переходить к следующему состоянию. Светодиод должен мигнуть 3 раза, то есть изменить состояние 6 раз. Не забудем установить начальные значения счётчика в конце для корректной работы следующей итерации.

```
  byte greenBlinkCount = 6; // Количество изменений состояния при мигании зелёным
  ...
  else if(trafficLightsState == GREEN_BLINK){
      digitalWrite(RED_LED, LOW);
      digitalWrite(YELLOW_LED, LOW);
      digitalWrite(GREEN_LED, !digitalRead(GREEN_LED));
      greenBlinkCount--;
      stateTime = GREEN_BLINK_TIME;
      if(greenBlinkCount == 0){
        greenBlinkCount = 6;
        trafficLightsState = YELLOW;
      }
    }
```

Ну вот и всё!

Программа слишком объёмная, чтобы разместить её здесь, так что можешь увидеть её по ссылке.

vk.com/@makedreamreal-code-millis

Светофор работает без использования delay! Знаю, было непросто, но, надеюсь, тебе всё понятно. В качестве самостоятельной работы попробуй реализовать регулировку яркости при помощи потенциометра.

Регулировка яркости светофора

В этой части урока реализуем регулировку яркости светодиодов светофора в зависимости от положения ручки потенциометра. Тут нет ничего сложного — всё так же, как и в случае с регулировкой яркости одного светодиода, а такое мы уже делали.

Ну, давай по порядку. Буду сразу дополнять код из предыдущего урока. Вступление, как всегда: *#define*, *pinMode*, на этом заострять уже не буду.

Необходимо постоянно читать значение пина A0 и преобразовывать его в диапазон яркости:

```
void loop() {
  // получаем значение потенциометра
  unsigned int brightness = analogRead(POT);
  brightness = map(brightness, 0, 1023, 20, 255); // преобразуем диапазон
```

Теперь в переменной *brightness* хранится яркость, осталось её передать светодиодам. Для этого необходимо просто заменить все функции включения светодиода *digitalWrite(led, HIGH)* на *analogWrite(led, brightness)*. Единственная трудность возникает с состояниями мигания светодиодом. Но об этом чуть позже. Пока разберёмся с яркостью простых состояний.

Итак, ну предположим, заменили мы все функции, и как теперь работает программа? Попробуй проверить.

Если крутить ручку потенциометра, яркость будет меняться только при смене состояния светофора. Но почему так? Дело в том, что состояние мы меняем только спустя время, и команды установки яркости выполняются соответственно. А нам нужно, чтобы яркость обновлялась сразу, как только мы повернули ручку. По сути, нам просто нужно постоянно пробегаться по условиям состояний светофора независимо от времени. При этом состояние всё так же должно меняться с заданным промежутком. Для этого можно в условии проверки времени оставить только изменение состояния, а всё остальное вынести в тело *loop*. Но в теле каждого условия состояние мы устанавливаем новое, соответственно, светофор будет переключаться постоянно, независимо от времени и очень быстро. Для решения этой проблемы необходимо ввести переменную для хранения следующего состояния: *nextTrafficLightsState*. Тогда код в теле *loop* будет выглядеть так:

```
    unsigned int brightness = analogRead(POT); // получаем значение потенциометра
     brightness = map(brightness, 0, 1023, 20, 255); // преобразуем диапазон

     if(millis() - lastTime >= stateTime){
       lastTime = millis();
       trafficLightsState = nextTrafficLightsState;
     }

     if(trafficLightsState == RED){
       analogWrite(RED_LED, brightness);
       digitalWrite(YELLOW_LED, LOW);
       digitalWrite(GREEN_LED, LOW);
       stateTime = RED_TIME;
       nextTrafficLightsState = RED_YELLOW;
     }
   ...
```

Теперь разберёмся с состояниями мигания светодиодом.

Тут придётся немного изменить подход, ведь инвертировать состояния светодиода мы больше не сможем, так как их теперь 256. Начнём с мигания жёлтым светодиодом.

Напоминаю о необходимости пытаться продвигаться вперёд самостоятельно.

Создадим вместо одного режима *YELLOW_BLYNK* два – *YELLOW_BLYNK_ON* и *YELLOW_BLYNK_OFF* – и просто будем переключаться между ними.

```
...
else if(trafficLightsState == YELLOW_BLINK_ON){
    digitalWrite(RED_LED, LOW);
    analogWrite(YELLOW_LED, brightness);
    digitalWrite(GREEN_LED, LOW);


    stateTime = YELLOW_BLINK_TIME;
    nextTrafficLightsState = YELLOW_BLINK_OFF;
  }
  else if(trafficLightsState == YELLOW_BLINK_OFF){
    digitalWrite(RED_LED, LOW);
    digitalWrite(YELLOW_LED, LOW);
    digitalWrite(GREEN_LED, LOW);

    stateTime = YELLOW_BLINK_TIME;
    nextTrafficLightsState = YELLOW_BLINK_ON;
  }
...
```

Теперь с зелёным светодиодом. Применим тот же подход, но в данном случае нужно мигать не бесконечно, а всего 3 раза. Значит, нам пригодится переменная-счётчик *greenBlinkCount*. При этом цикл использовать здесь нельзя, ведь он остановит выполнение нашей программы. Пусть изначально *greenBlinkCount* будет равна 3. Тогда в теле условия *GREEN_BLYNK_ON* будем уменьшать её при каждом новом включении, а когда переменная станет равна 0, установим следующее состояние на *YELLOW*. Уменьшать переменную необходимо именно один раз при включении светодиода, ведь если просто делать это каждый раз при выполнении условия *GREEN_BLYNK_ON*, счётчик уменьшится сразу же. Так что нужно добавить ещё условие. Ну а как это можно реализовать, смотри в итоговой программе этой части урока:

vk.com/@makedreamreal-code-brightness

Шлагбаум и датчик расстояния

В этой части урока добавим шлагбаум и датчик расстояния к нашей программе. Начнём вот с чего.

Для лучшей структуризации кода упакуем работу светофора в отдельную функцию. Над функцией setup объявим функцию с именем *TrafficLights()*, а её тело распишем уже после loop. Объявить функцию после *loop* – неправильно, ведь их, как и переменные, необходимо объявлять до использования. А расписывать тело после *loop* нужно для более удобной работы с кодом, чтобы не нагромождать его в верхней части.

```
...
void TrafficLights();

void setup() {
  ...
}

void loop() {
  TrafficLights();
}

void TrafficLights(){
  ...
}
```

Убедимся, что всё работает, и идём дальше. Код шлагбаума можно добавить в функцию состояний светофора, но это может усложнить работу над программой, поэтому используем другой подход.

Создадим функцию для работы шлагбаума по тому же принципу, что и для светофора. Шлагбаум должен закрываться, когда состояние светофора становится *RED*, и открываться, когда *YELLOW*, на это и будем опираться. Для начала добавим библиотеку, обозначим ножку сервопривода, создадим новый объект с именем *barrier* и добавим ещё две константы, хранящие значения углов для состояний «откры-

то» и «закрыто». Не забудем включить сервопривод на ножке в теле функции *setup*.

```
#include <Servo.h>
...
#define SERVO 4 // пин сервопривода
...
// Углы шлагбаума
#define OPEN 180
#define CLOSE 90
Servo barrier;
...
void Barrier();

void setup() {
  barrier.attach(SERVO);
  ...
}

void loop() {
  TrafficLights();
  Barrier();
}
...
void Barrier(){

}
```

Так-с, надеюсь, ты не забываешь сначала пробовать решить задачу самостоятельно?

Теперь в теле Barrier напишем условия: если красный, то закрыть, если зелёный, то открыть.

```
void Barrier(){
  if(trafficLightsState == RED){
    barrier.write(CLOSE);
  } else if(trafficLightsState == GREEN){
    barrier.write(OPEN);
  }
}
```

Казалось бы, всё просто. Но теперь нужно отслеживать препятствие перед шлагбаумом и, если оно не исчезает в течение определённого времени, включить мигающий жёлтый. Тут уже нужно прямо подумать над алгоритмом. Но для начала добавим всё очевидное: номера ножек датчика расстояния и режимы их работы, функцию получения расстояния из предыдущего урока, константу для хранения того самого времени ожидания. Очевидно, мы будем засекать время, используя *millis()*. Получается, нам понадобится соответствующая переменная.

Чтобы понять, что ещё необходимо добавить, надо продумать алгоритм. В очередной раз рекомендую помучиться с этой задачкой самому. В данном случае нужно усиленно пробовать, тестировать, и тут тебе поможет последовательный порт, куда можно выводить сообщения, указывающие, куда заходит программа, куда нет, чему равны переменные и так далее.

В итоге вот возможный вариант реализации этого алгоритма:

vk.com/@makedreamreal-code-barrier

Кнопки

Последнее, что осталось сделать, – это добавить кнопки к нашему алгоритму. **KEY1** должна переключать состояние светофора, а **KEY2** – вводить и выводить из состояния мигающий жёлтый. И тут можно использовать тот же подход, что в уроке про кнопку, но в качестве бонуса за то, что ты добрался до финального урока, познакомимся с внешними прерываниями.

Что же такое прерывания? Некоторые ножки микроконтроллера – в нашем случае 2 и 3 – имеют встроенную функцию вызывать прерывание выполнения программы для осуществления каких-либо действий вне очереди. То есть как только мы нажимаем на кнопку, выполнение программы останавливается, микроконтроллер выполняет необходимые команды и продолжает работу с того же места как ни в чём не бывало.

Давай разбираться сразу на примере. Для начала, как всегда, обозначим ножки и режимы работы. В данном случае режим работы будет *INPUT_PULLUP*, мы говорили об этом в уроке про кнопку.

Теперь необходимо включить прерывания, делает это функция *attachInterrupt(номер, обработчик, режим)*. Разберёмся с аргументами. Номер прерывания соответствует конкретной ножке на плате. В нашем случае пину 2 соответствует номер 0, а ножке 3 – номер 1. То есть в качестве аргумента нужно передавать именно номер прерывания, а не номер ножки. Для ленивых существует функция *digitalPinToInterrupt(номер ножки)*, которая возвращает номер прерывания для указанной ножки. Далее обработчик – это функция, в которой мы и напишем действия, которые необходимо выполнить по прерыванию. Ну и наконец режим. Существуют четыре режима прерывания – по сути, четыре причины возникновения прерывания:

RISING – срабатывает при изменении сигнала с низкого на высокий;

FALLING – срабатывает при изменении сигнала с высокого на низкий;

CHANGE – срабатывает при изменении сигнала;

LOW – срабатывает непрерывно, пока сигнал на ножке низкий.

В нашем случае, пока кнопка не нажата, сигнал высокий, при нажатии он меняется на низкий. Соответственно, нам подходит режим **FALLING**. Если выбрать режим **RISING**, можно реагировать на отпускание кнопки.

Итак, добавим функции-обработчики прерываний и включим внешние прерывания.

```
...
#define KEY1 3 // пин кнопки 1
#define KEY2 2 // пин кнопки 2
...
void KEY1_INT();
void KEY2_INT();
void setup() {
  ...
  pinMode(KEY1, INPUT_PULLUP);  // пин 3 на вход с подтягивающим резистором
  pinMode(KEY2, INPUT_PULLUP); // пин 2 на вход с подтягивающим резистором
  attachInterrupt(digitalPinToInterrupt(KEY1), KEY1_INT, FALLING);
  attachInterrupt(digitalPinToInterrupt(KEY2), KEY2_INT, FALLING);
}
void loop() {
  ...
}
void KEY1_INT() {
}
void KEY2_INT() {
}
...
```

В обработчике должно быть как можно меньше команд, чтобы не задерживать выполнение основной программы, поэтому в них будем просто отмечать факт нажатия на кнопку. Для этого создадим соответствующие переменные.

```
bool key1Pressed = false, key2Pressed = false; // флаги нажатия на кнопки
```

А реагировать будем в отдельной функции. Начнём с кнопки 1.

Если кнопка нажата и горит красный, надо включить зелёный, если горит зелёный, надо включить красный. Важно не забывать обнулить время переключения состояний светофора и сбросить флаг кнопки 1. Также стоит обнулить переменную обнаружения препятствия, иначе в некоторых состояниях код может работать неправильно.

```
void Keys() {
  if (key1Pressed == true) {
    if (trafficLightsState == RED) {
      trafficLightsState = RED_YELLOW;
    } else if (trafficLightsState == GREEN) {
      trafficLightsState = GREEN_BLINK_ON;
    }
    lastTime = millis(); // Обнуляем время
    obstacleDetected = false; // Исправляем баг
    key1Pressed = false;
  }
}
```

Теперь со второй кнопкой. Если кнопка 2 нажата и состояние – мигающий жёлтый, то включить, например, зелёный, иначе включить мигающий жёлтый. Также не забудем обнулить время переключения состояний и сбросить флаг кнопки 2. Вот и вся программа!

vk.com/@makedreamreal-code-buttons