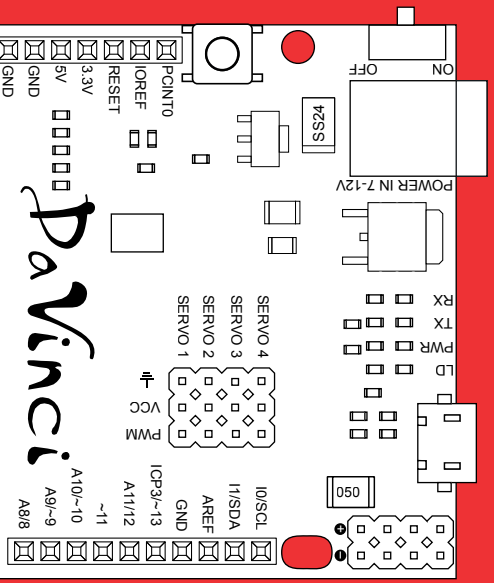
geegrow

Учебное пособие
начинающего изобретателя

Базовый уровень



www.geegrow.ru

Введение

Мы живем в удивительное время бурного развития электроники, когда каждый день появляются разнообразные электронные новинки, стремительно вливающиеся в повседневную жизнь и оказывающие на нее все большее влияние.

Одной из последних тенденций является повсеместное проникновение, так называемого, «интернета вещей» во все сферы промышленности и быта. Глобальная информатизация и связывание в единую сеть всех электронных устройств перестала быть чем то фантастичным и стала реальностью сегодняшнего дня.

Мир движется к будущему, в котором люди, обладающие знаниями и навыками в области электроники, будут иметь больше шансов стать успешнее в учебе и работе. Вот, почему, важно не упустить момент и начать уже сейчас.

На страницах этой книги мы покажем, что конструирование электронных устройств не только гораздо проще, чем кажется, но и может быть на много более интересным и увлекательным занятием, чем принято считать. Расскажем, как с помощью контроллера из семейства Arduino, можно быстро и просто разработать прототип реального устройства, или сконструировать интеллектуальную игрушку.

По мере изучения курса, у Вас появится множество своих идей и все их вы сможете реализовать с помощью нашего конструктора.

Как построен учебный курс

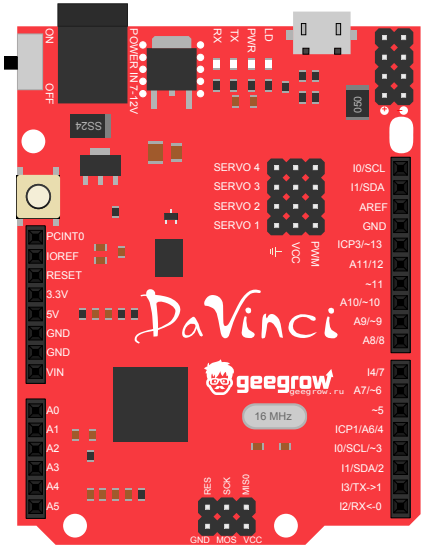
Книга, которую вы держите в руках, посвящена, прежде всего, программированию микроконтроллеров и знакомству с платформой Arduino. Но, так как, полноценное освоение курса не возможно без владения некоторыми базовыми знаниями, книга в равной степени освещает такие темы, как построение алгоритмов, программирование, основы электроники и схемотехники, а так же важнейшие вопросы из курса физики, посвященные электричеству.

Понимая, что конструктор предназначен для различных возрастных категорий, мы постарались максимально подробно осветить все вопросы, освоение которых могло вызвать даже малейшие трудности.

В начале каждого урока даются необходимые для понимания теоретические знания из курса физики и электроники. Затем мы переходим к построению электрической схемы и подробно объясняем принцип работы.
Следующим этапом является разработка программы для Arduino на языке «C/C++». Так же как и в первой части урока, сначала даются теоретические знания из курса программирования, после чего строится блок-схема и составляется управляющая программа.

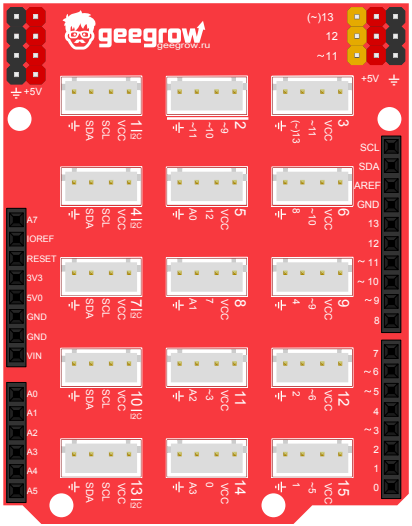
С чем будем работать

Предлагаемый электронный конструктор разработан специально для использования совместно с данным учебным курсом и включает в себя следующие устройства:



1 Контроллер DaVinci

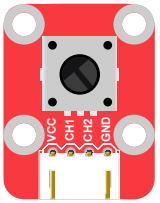
Контроллер, является электронным мозгом, позволяющим получать данные из физического мира, обрабатывать их с помощью написанного вами алгоритма, а затем, основываясь на полученных результатах, определенным образом повлиять на физический мир, посредством различных исполнительных устройств.



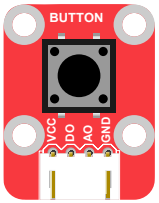
2 Шилд QuatroPortA050

Предназначен для использования совместно с контроллером DaVinci. Позволяет быстро и безопасно подключать внешние модули к контроллеру.

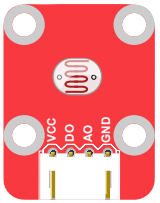
Сенсоры



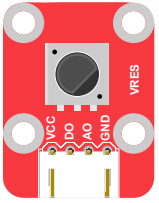
3 Энкодер



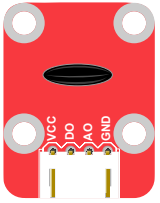
4 Тактовая кнопка



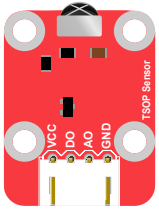
5 Фоторезистор



6 Переменный резистор

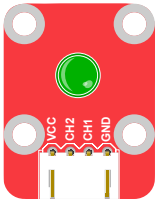


7 Термистор

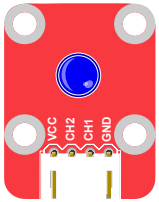


8 ИК-приемник

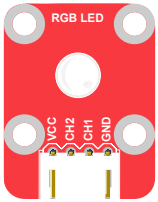
Исполнительные и индикаторные устройства



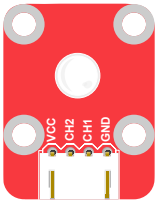
9 Зеленый светодиод



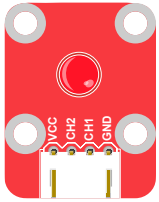
10 Синий светодиод



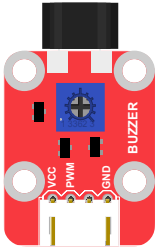
11 RGB светодиод



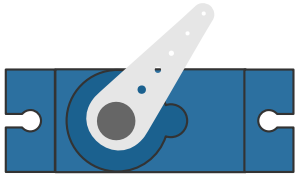
12 Белый светодиод



13 Красный светодиод

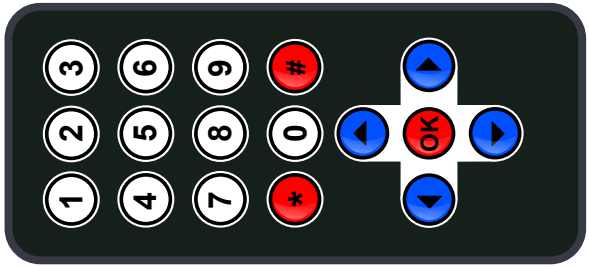


14 Зуммер



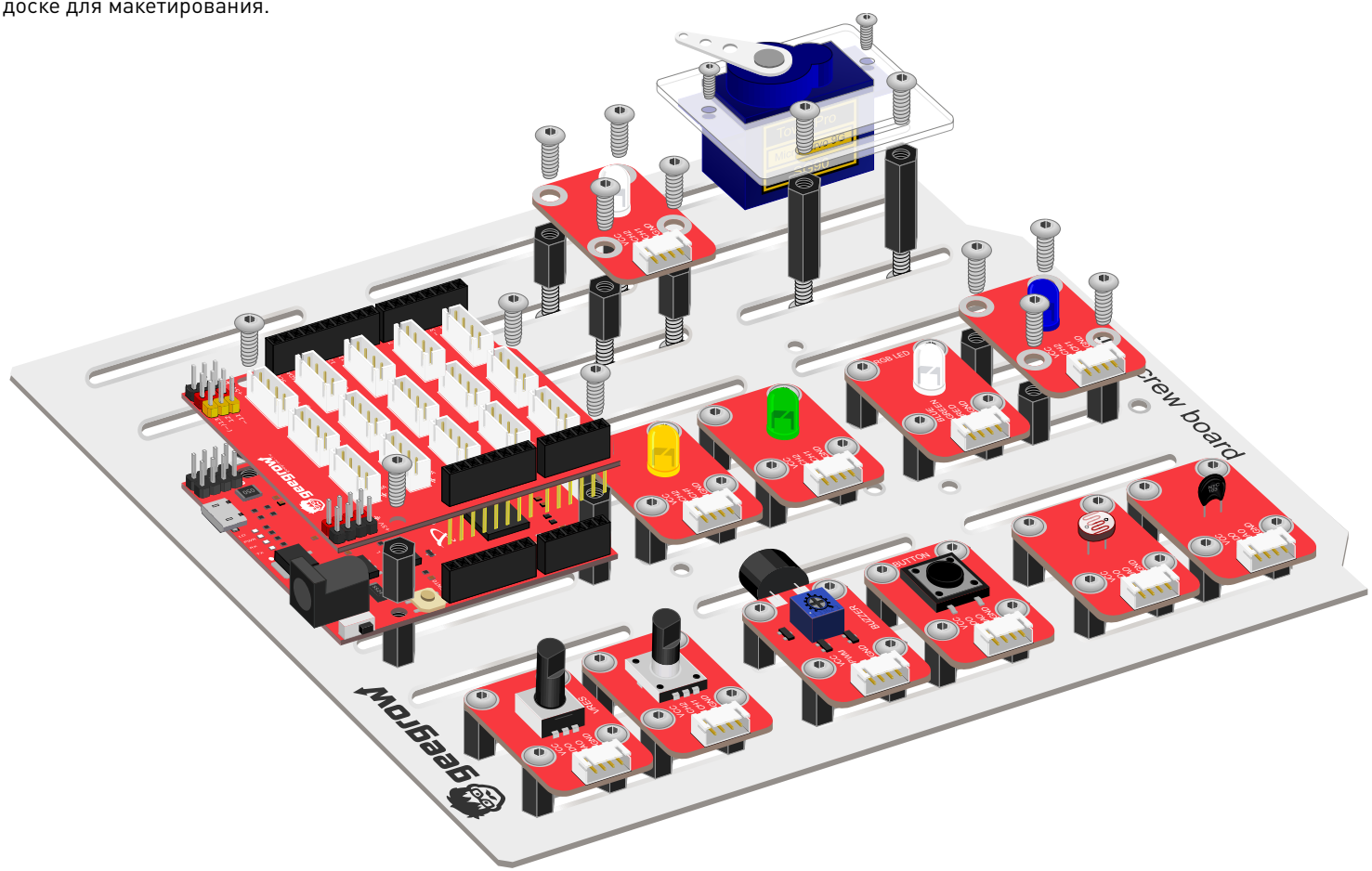
15 Сервопривод 0°..180°

16 Макетная доска для прототипирования



17 ИК-пульт

Чтобы было удобнее работать с конструктором и модули не валялись на столе, можно закрепить все на доске для макетирования.



Знакомство с контроллером

Основой конструктора является плата-контроллер DaVinci. Он выполняет роль миниатюрного компьютера, в который загружается написанная пользователем программа.
Загруженная программа будет циклично исполняться, раз за разом, общаясь с физическим миром через порты ввода/вывода.

Архитектура контроллера аналогична архитектуре популярной платы Arduino Leonardo и расширяет ее возможности, имея ряд улучшений и особенностей, с которыми вы сейчас познакомитесь.

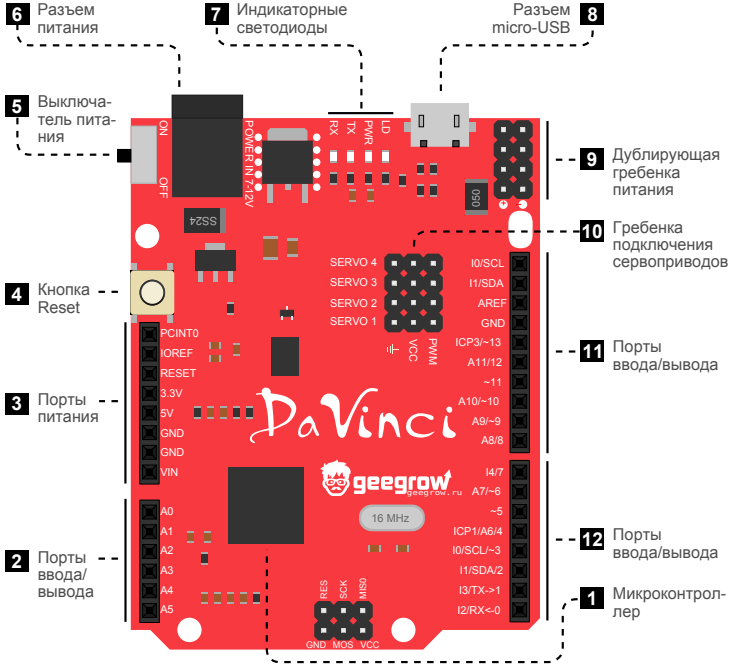


Рис.1.1 Назначение элементов контроллера DaVinci

1 Микроконтроллер ATmega32U4, имеет 14 цифровых портов ввода/вывода, 7 из которых могут быть ШИМ (PWM) выводами. Кроме того на плате имеется 11 аналоговых входов (АЦП), 32Кб флеш-памяти, 2Кб ОЗУ, 1Кб EEPROM, USB порт, интерфейсы I2C и ISP. Не пугайтесь, во всех перечисленных терминах мы разберемся позже.

2 **11** **12** Порты ввода/вывода — предназначены для подключения аналоговых и цифровых устройств. Порты A0..A11 умеют работать с Аналого-цифровым Преобразователем (АЦП), с которым вы познакомитесь в одном из следующих уроков. Их можно использовать для подключения аналоговых модулей.

3 Порты питания — служат для подачи питания на подключаемые модули. Порт VIN подключен к входной шине питания и напряжение на нем может быть **больше 5В**. Будьте внимательны!

4 Кнопка Reset — осуществляет перезагрузку контроллера. Она бывает полезна если контроллер завис или, если вы хотите запустить выполнение программы сначала.

5 Выключатель питания — полностью обесточивает плату. Его удобно использовать, если не хочется каждый раз вынимать питающий кабель.

6 Разъем для подачи внешнего питания от стабилизированного источника постоянного напряжения 7..12В. Источником может быть блок питания, сборка аккумуляторов или батареек.

7 Индикаторные светодиоды — помогают понять, чем занят контроллер.
TX и RX — мигают, если контроллер обменивается данными с компьютером.
PWR — загорается при подаче питания.
LD — подключен к порту контроллера номер 13 и, при необходимости, может использоваться пользовательской программой.

8 Разъем micro-USB — используется для загрузки программы в контроллер. В момент, когда контроллер подключен к USB порту

компьютера, он не нуждается в дополнительном внешнем питании. Питание поступает от компьютера.

9 Дополнительная гребенка питания для подачи питающего напряжения от контроллера к внешним модулям.

10 Гребенка для подключения сервоприводов напрямую к контроллеру, без дополнительных шилдов.

Функциональное назначение портов

Для простоты и удобства порты контроллера можно разбить на 6 групп согласно их назначению Рис 1.2.

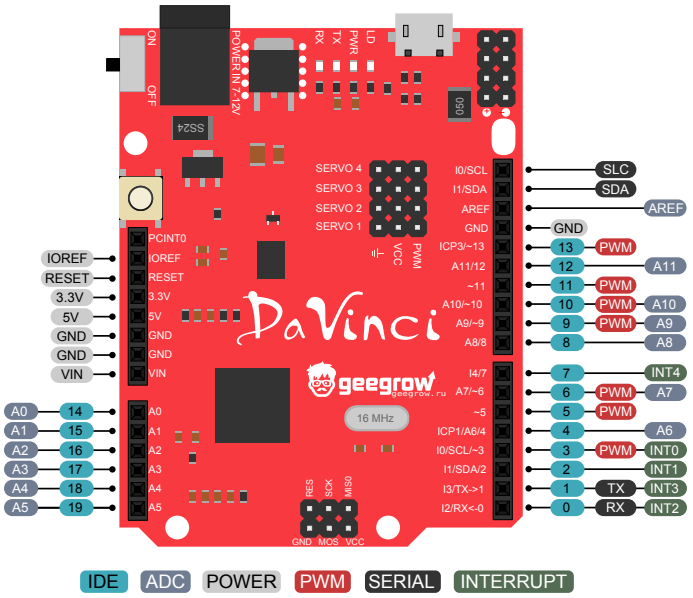


Рис.1.2 Функциональное назначение портов контроллера

IDE — нумерация портов согласно принятой в среде программирования Arduino IDE.

ADC — порты АЦП, позволяют обрабатывать аналоговые, плавно меняющиеся, сигналы.

POWER — порты, предназначенные для питания внешних модулей и других нужд.

PWM — порты с возможностью генерации ШИМ сигнала. Позволяют плавно управлять внешними модулями, такими, как светодиоды, моторы и т.д. Подробнее эта тема будет рассмотрена в последующих уроках.

SERIAL — порты, предназначенные для подключения устройств через последовательный интерфейс.

INTERRUPT — порты, работающие с внешними прерываниями. Работа с прерываниями будет рассмотрена в одном из уроков.

Шилд для подключения внешних модулей

Контроллер, сам по себе, не очень удобен при работе с внешними модулями. Он имеет ограниченное количество портов питания, которые, к тому же, расположены далеко от портов ввода/вывода. Чтобы сделать работу с внешними модулями удобнее, мы разработали специальный переходной шилд QuatroPort A050.
Как видно из Рис 1.3, он дублирует боковые разъемы контроллера и, дополнительно имеет 15 белых разъемов для подключения модулей. В каждом разъеме находится 4 контакта. Два крайних предназначены для подачи питания, а два центральных для передачи сигнала.

Разъемы для подключения внешних модулей сгруппированы по назначению. Некоторые из них служат для подключения цифровых I2C модулей, другие лучше использовать для подключения модулей нуждающихся в портах ШИМ и АЦП.
Дополнительно, на шилде имеется дублирующая гребенка пита-

ния и гребенка 3x3 для подключения сервоприводов к портам 13, 12, 11.

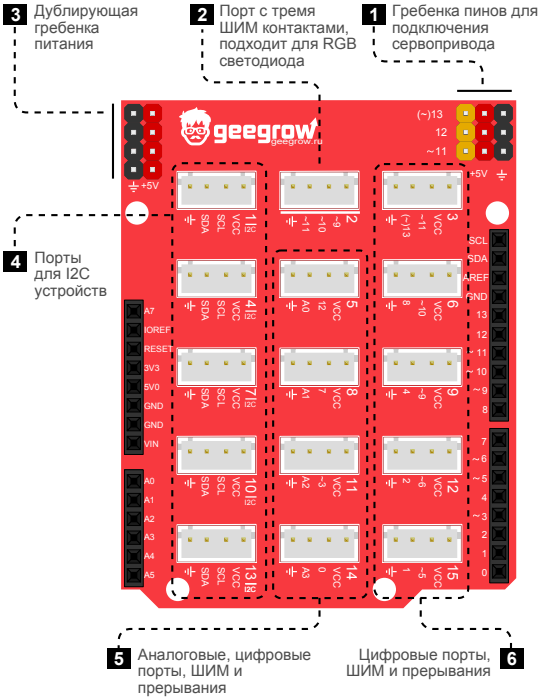


Рис.1.3 Шилд QuatroPort A050 с обозначением групп разъемов

Во всех уроках мы будем указывать, какие именно разъемы и порты использовать для экспериментов. Но в будущем, пока вы не начнете чувствовать себя более уверенно в обращении с контроллером и шилдом, мы рекомендуем пользоваться таблицей совместимости. С ее помощью можно быстро определить наиболее предпочтительный для подключения конкретного модуля разъем.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Светодиод		●●	●●		●●	●●		●●	●●		●●	●●		●●	●●
RGB Светоди		●●													
Зуммер			●●			●●			●●		●●	●●		●●	●●
Кнопка			●		●	●		●●	●		●●	●		●●	●
Потенциометр					●			●			●			●	
Энкодер			●		●	●		●	●		●	●●		●	●●
Фоторезистор					●	●		●	●		●	●		●	
Термистор					●	●		●	●		●	●●		●	
ИК-датчик			●●		●●	●●		●●	●●		●●	●●		●●	●●

●● — порты обеспечивающие полную совместимость с подключаемыми модулями.
● — порты, подключение к которым выбранных модулей допустимо, но не обеспечивает возможности полного использования всех функций.

Установка среды Arduino IDE

Платформа Arduino подразумевает совместное использование контроллеров этого семейства и, специально разработанной, среды программирования, называемой Arduino IDE (Integrated Development Environment).

Перед тем как подключать контроллер, на компьютер необходимо установить среду разработки Arduino IDE. На официальном сайте разработчика можно скачать последнюю версию Arduino IDE всех популярных операционных систем (Windows, Mac OS и Linux). Ссылка для скачивания https://www.arduino.cc/en/Main/Software.

Подключение контроллера

После установки Arduino IDE, можно подключить контроллер к компьютеру с помощью USB-кабеля. В течение нескольких секунд, операционная система обнаружит новое устройство. В некоторых случаях может потребоваться установка драйверов. Прочитать о том, как это сделать, вы можете на нашем сайте, по ссылке http://www.geegrow.ru/articles/how-to/connect-davinci.

Знакомство с Arduino IDE

Теперь все готово к началу работы. Давайте запустим среду Arduino IDE и познакомимся с интерфейсом, Рис. 1.4.

Окно условно можно разбить на две части. Верхняя часть содержит пункты меню и органы управления:

1. Verify: кнопка компиляции и проверки корректности кода
2. Upload: кнопка загрузки скетча (программы) в контроллер
3. New: открывает новое окно редактирования кода
4. Open: позволяет открыть для редактирования ранее созданный скетч, хранящийся на диске
5. Save: сохраняет текущие изменения
6. Serial Monitor: открывает окно, отображающее информацию об обмене данными с вашим контроллером DaVinci. Эта возможность может быть использована для отладки кода.

Средняя часть предназначена для редактирования кода:

7. Название скетча (программы), который вы в данный момент редактируете.
8. Текстовое поле, в котором содержится код программы.
9. Текстовая консоль, отображающая сервисные сообщения среды о ходе выполнения компиляции, загрузке скетча, ошибках и т.д.

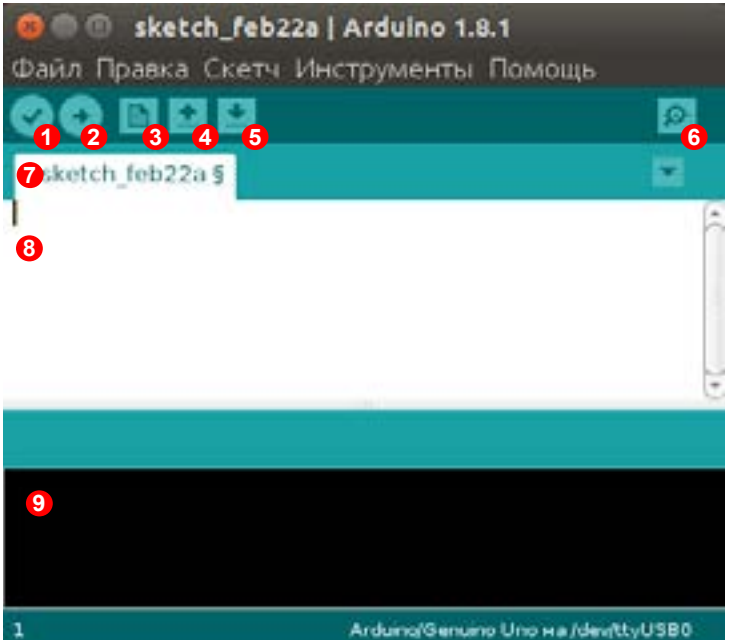


Рис.1.4 Интерфейс среды разработки Arduino IDE

Теперь, познакомившись с основными органами управления, можно продолжить работу. Для этого необходимо выполнить два действия.

Для начала необходимо выбрать правильную плату из списка. Для этого зайдите в пункт меню Инструменты > Плата (зайдите в Tools > Board, если используете англоязычную версию). Для работы с контроллером DaVinci, в выпадающем меню выберите Arduino Leonardo. Как было сказано выше, контроллер DaVinci полностью аналогичен модели Arduino Leonardo и не требует специального программного обеспечения.

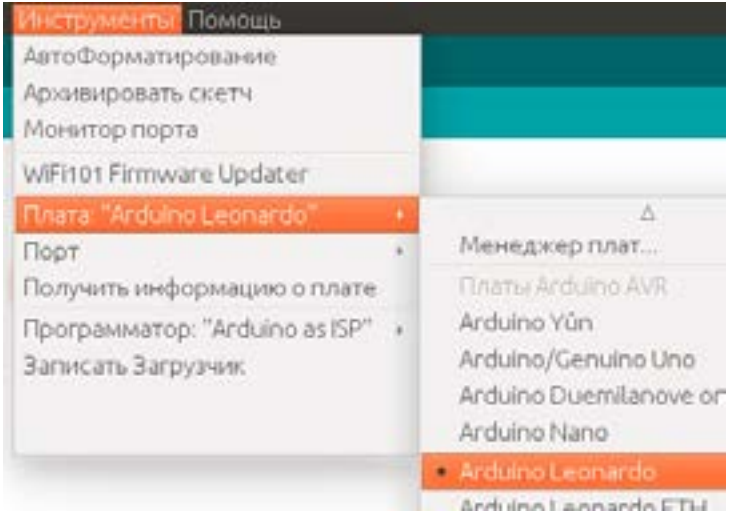


Рис.1.5 Arduino IDE - выбор контроллера

Теперь осталось выбрать порт. Для этого перейдите в пункт меню Инструменты > Порт (Tools > Port в англоязычной версии) и выберите порт соответствующий контроллеру. В нашем случае он называется /dev/tty/ACM0 (Arduino Leonardo) Рис. 1.6.

На разных операционных системах порт может называться по-разному. Если вы не можете определить какой порт соответствует контроллеру, отключите его, потом подключите заново и посмотрите какой порт появился.

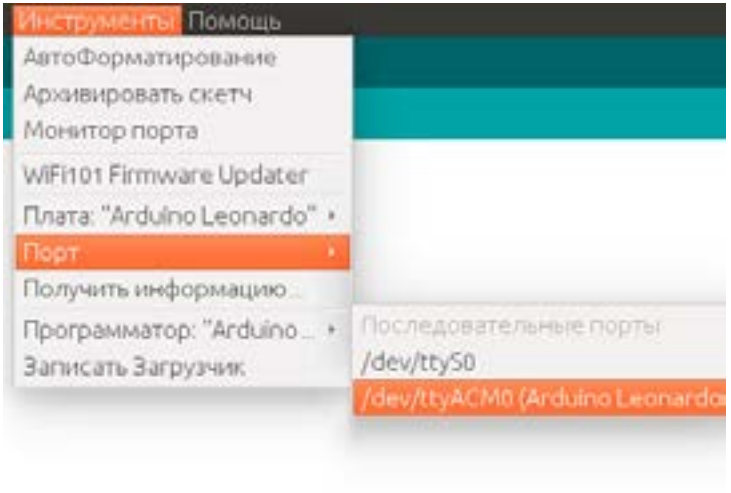


Рис.1.6 Arduino IDE - выбор порта

На этом все подготовительные операции можно считать завершенными. Давайте загрузим в контроллер первый тестовый скетч из числа стандартных примеров среды Arduino IDE. Для этого в меню выберем Файл > Примеры > 01.Basics > Blink (или File > Examples > 01.Basics > Blink в англоязычной версии).

В результате откроется новое окно с исходным кодом примера. Нажимаем на кнопку Upload. После завершения загрузки скетча, в строке статуса над консолью появится надпись: Done uploading, что значит — загрузка завершена. Светодиод LD на плате контроллера будет мигать с частотой 1 раз в 2 секунды.

Урок 1. Блок-схемы

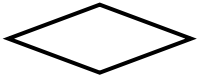
Информация, описанная в этом уроке, будет полезна, прежде всего, новичкам, не имеющим практического опыта в программировании. Составление блок-схем помогает абстрагироваться от конкретного языка программирования и сосредоточиться непосредственно на разработке алгоритма работы программы.

Теория:

Составление блок-схем позволяет описать алгоритм с использованием геометрических фигур — «блоков». Каждая вычислительная операция размещается в отдельном блоке. Блоки алгоритма могут быть связаны между собой различным образом, образуя как линейные участки алгоритма, так и ветвления с использованием условных блоков. Рассмотрим, какие основные типы блоков применяются при составлении блок-схем:

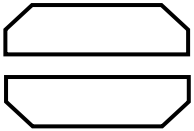
Терминатор или блок *начало-конец* — обозначает вход в программу и выход из нее.

Блок команды или *процесса* — отвечает за выполнение операций изменения данных, арифметических действий и т.д.



Условный оператор — осуществляет проверку условия и имеет два выхода «Да» и «Нет».

Блок цикла со счетчиком. Цикл с предусловием for. Тело цикла выполняется пока не будет достигнуто условие окончания цикла.



Парный блок циклов с пред- и постусловием типа while, foreach и т.д. Операции тела цикла размещаются между блоками. Заголовок цикла и изменения счетчика цикла записываются внутри верхнего или нижнего блока — в зависимости от типа цикла.

Блок ввода-вывода данных — обозначает получение данных (с клавиатуры или другого устройства), а так же за вывод, например на экран монитора.

Вызов подпрограммы — обозначает обращение к внешней функции, содержащей программный код.

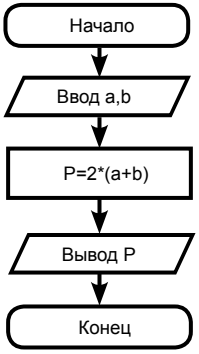
Соединитель — предназначен для соединения линий связи между двумя частями алгоритма. Применяется, если блок-схема не помещается на лист. Внутри блока помещается идентификатор (имя) связывающий два соединителя.

Это лишь часть применяемых при построении блок-схем обозначений, но этого вполне достаточно для наших целей.

Практика:

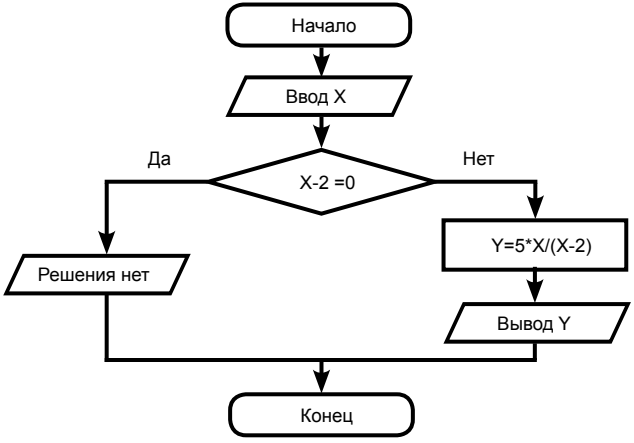
Чтобы лучше понять материал, давайте решим несколько задач, составив блок-схемы алгоритмов.

Задача 1.1 на составление линейного алгоритма.
Вычислить периметр прямоугольника по двум его сторонам a и b.

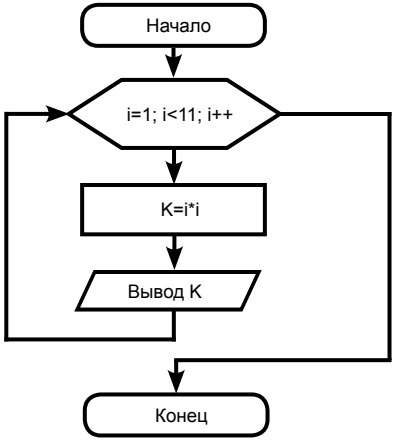


Задача 1.2 на составление алгоритма с ветвлением.
Вычислить значение выражения 5*X/(X-2).

Ясно, что ответ может быть найден только в том случае, когда значение под дробной чертой не равно нулю. Составим блок-схему исходя из этого условия.



Задача 1.3 на составление циклического алгоритма.
Вывести на экран квадраты первых десяти натуральных чисел.



Запись блока объявления цикла нуждается в пояснении.
Строка i=1; i<11; i++ является частью синтаксиса объявления условия цикла for в языке C. Она состоит из трех частей, которые означают буквально следующее:
i = 1; — задать значение i равным 1
i<11; — пока значение i<11 выполнять тело цикла
i++ — после каждой итерации (повторения) увеличивать значение i на единицу.
Таким образом блок вычисления K=i*i и блок вывода K будут выполнены 10 раз. При этом значение переменной цикла i будет изменяться от 1 до 10. В одиннадцатый раз значение переменной цикла i будет равно 11, а значит условие продолжения цикла не выполнится и программа выйдет из цикла.

В последующих уроках циклы и условные блоки будут рассмотрены более подробно. А пока, имеющихся знаний уже достаточно для того, чтобы перейти к изучению языка C++ и работы с контроллером, чем мы и займемся в следующих уроках.

Урок 2. Основные понятия электроники

Введем ряд основных понятий, объясняющих, что такое электричество и рассмотрим основные физические явления, связанные с электричеством.
Это поможет разобраться в том, как работают электронные схемы, как в них течет ток и на что влияют те или иные компоненты.

Электричество

Электричество — совокупность явлений, обусловленных существованием, взаимодействием и движением электрических зарядов.

Электричество впервые было обнаружено еще в VII веке до н.э. греческим философом Фалесом. Он обратил внимание на то, что если кусочек янтаря потереть о шерсть, он начинает притягивать к себе легкие предметы.
Это явление называется статическим электричеством и вы можете повторить его у себя дома, потерев пластмассовую линейку шерстяной тканью и поднеся ее к мелким кусочкам бумаги.

Статическое электричество — явление обусловленное эффектом накопления элементарных носителей заряда, электронов, на поверхности диэлектриков.
Электрон – это элементарная частица, обозначается **e**, имеет отрицательный заряд примерно равный $-1{,}602 \cdot 10^{-19}$ Кл (Кулон).

Электрический ток

Статическое электричество может приводить к возникновению искр и притягивать мелкие предметы, но его нельзя использовать в электронных схемах. Для этого нужно создать электрический ток, то есть привести заряды в движение. Отсюда вытекает определение электрического тока.
Электрический ток — направленное (упорядоченное) движение носителей заряда.
Для измерения величины электрического тока, ввели понятие *силы тока*.

Сила тока — физическая величина, характеризующая количество заряда ΔQ, прошедшего через поперечное сечение проводника за некоторое время Δt. Сила тока обозначается лат. буквой **I** и измеряется в Амперах *(сокращ: А)*.
1Ампер, это достаточно большая величина, и на практике, в электронных схемах, сила тока часто оказывается значительно меньше. Если сила тока меньше 1А, то ее часто обозначают в миллиамперах — мА или mA *($1мА = 1 \cdot 10^{-3} А$)*.

Для наглядности, представим себе резервуар с водой и выходящую из его дна трубу Рис. 2.1. Условимся, что вода это электрический заряд, а молекулы воды — элементарные носители заряда. Тогда количество воды вытекающее из трубы в единицу времени и будет эквивалентно силе тока.

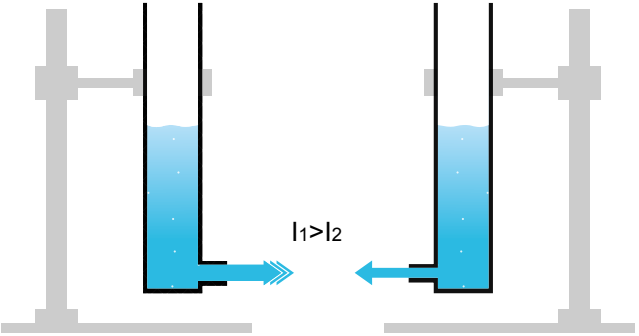


Рис. 2.1 Наглядное представление электрического тока как жидкости

Сопротивление и проводимость

Глядя на рисунок 2.1, естественно предположить, что сила тока зависит от диаметра трубы. Чем толще труба, тем большее количество воды из нее вытекает. И наоборот — чем тоньше труба, тем меньшее количество воды сможет пройти через нее в единицу времени.
Такое же соотношение действует и в случае с электричеством. Чем толще проводник, тем больший ток может по нему течь. Способность тела проводить электрический ток называется *проводи-*

мостью. В электронике, для расчетов, обычно, пользуются величиной обратной по смыслу — *электрическим сопротивлением*.

Электрическое сопротивление — физическая величина, характеризующая свойства проводника препятствовать прохождению электрического тока. Сопротивление обозначается лат. буквой **R** и измеряется в Омах, *(сокращ: Ом, иногда обозначается греч. буквой* **Ω**—омега*)*. В различной литературе можно встретить номиналы сопротивлений указанные в килоомах *(кОм или просто К. 1кОм = 1·10 [3] Ом)* или мегаомах *(МОм или М, 1МОм = 1·10 [6] Ом).*

Сопротивление удобнее всего представить себе на примере двух водопроводных труб. Труба с меньшим диаметром, будет обладать бОльшим сопротивлением. И наоборот, труба с бОльшим диаметром, будет обладать меньшим сопротивлением.

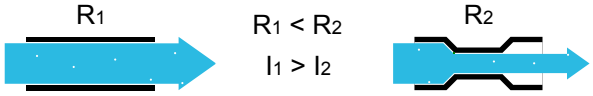


Рис. 2.2 Наглядное представление сопротивления, как двух труб разного диаметра

Напряжение

Снова обратимся к рисунку 2.1. Предположим, что мы хотим увеличить силу потока воды, выходящего из резервуара, но на этот раз изменять диаметр трубы нельзя.
В этом случае, повлиять на объем вытекающей воды, можно только увеличив скорость потока. Для этого добавим воды (заряда) в резервуар, что приведет к увеличению давления внутри него и, соответственно, увеличению скорости движения воды в трубе, Рис. 2.3.

Очевидно, что давление внутри резервуара будет меняться пропорционально высоте водяного столба.
В этом примере высота столба воды внутри резервуара эквивалентна понятию *электрического напряжения*.

Напряжение обозначается буквой **U** и измеряется в Вольтах *(сокращ: В или по англ. V)*. Если значение напряжения меньше 1В, его, как правило, обозначают в милливольтах — мВ или mV *(1мВ = 1·10 [-3] В)*.

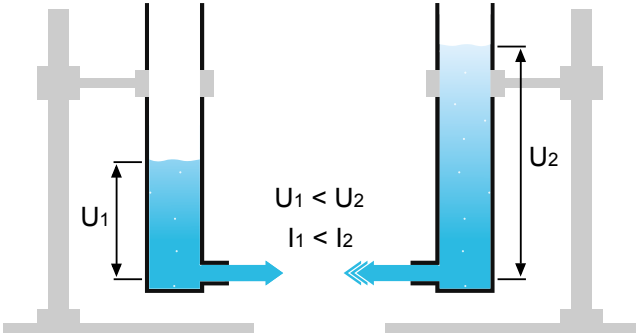


Рис. 2.3 Наглядное сравнение напряжения с уровнем жидкости в резервуаре

Закон Ома

Из рассмотренных экспериментов, наглядно демонстрирующих взаимосвязь между напряжением, током и сопротивлением, можно сделать два вывода:
1) Если сопротивление увеличивается — ток уменьшается. И наоборот, если сопротивление уменьшается — ток увеличивается.
2) Если напряжение увеличивается — ток увеличивается. И наоборот, если напряжение уменьшается — ток тоже уменьшается.

Пользуясь этими утверждениями, выразим ток **I** через напряжение **U** и сопротивление **R**:

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

Полученное выражение называется законом Ома для участка цепи и является основополагающим законом в электронике.

Закон Ома для участка цепи — ток прямо пропорционален напряжению и обратно пропорционален сопротивлению.

Урок 3. Подключение светодиода

В этом уроке вы познакомитесь со светодиодом, научитесь правильно и безопасно его подключать, а так же управлять светодиодом с помощью контроллера.

Теория:.

Светодиод — это полупроводниковый прибор, преобразующий электрический ток непосредственно в световое излучение. По-английски светодиод обозначается как light emitting diode, или LED. При включении светодиода нужно помнить, что это не обычная лампочка, а полупроводниковый прибор, пропускающий ток только в одном направлении и при его подключении важно соблюдать полярность. Если полярность перепутана, ток через светодиод течь не будет.
Выводы светодиода обозначаются как Анод (+) и Катод (-).

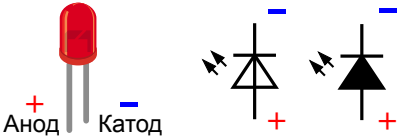


Рис. 3.1 Светодиод

На Рис. 3.1 показано, как выглядит светодиод и как он обозначается на электрических схемах.
Для того, чтобы не перепутать полярность, ножки светодиода специально сделаны разной длины. Длинная ножка, это Анод (+). Он подключается к плюсу источника питания. А короткая ножка, это катод (-). Он подключается к минусу питания, то есть к земле.

К ключевым характеристикам светодиода относятся такие параметры, как рабочий ток и рабочее напряжение. Типовым, для рассматриваемых светодиодов, является напряжение питания от 2.2В до 3В (вольт) и ток порядка 20 мА (миллиампер).
Мы будем питать светодиод от вывода контроллера, потенциал (напряжение) которого равен 5В, что слишком много для светодиода. Чтобы светодиод не сгорел, последовательно с ним включают токоограничительный резистор.

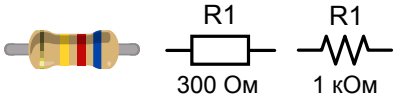
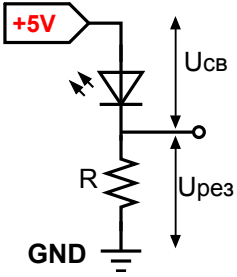


Рис. 3.2 Резистор

Резистор — это пассивный элемент, обладающий постоянным или переменным сопротивлением. Внешний вид резистора и его обозначение на электрических схемах, показано на Рис. 3.2

Используя, рассмотренный в предыдущем уроке Закон Ома, посчитаем величину токоограничительного резистора для нашего светодиода.



Известно:
Uпит = 5В
Uсв = 2.2В
Iсв = Iрез = 20мА (0.02А)
Полное напряжение в цепи складывается из Uсв и Uрез и равно напряжению питания 5В. То есть:
Uрез + Uсв = 5В
Отсюда можно найти падение напряжения на резисторе

Uрез = Uпит - U св = 5В - 2.2В = 2.8В;

Отметим, что резистор и светодиод включены последовательно, а значит протекающие через них токи равны.
Пользуясь Законом Ома, выразим сопротивление резистора R через протекающий по нему ток Iрез и падение напряжения Uрез:

$$R = \frac{Uрез}{Iрез} = \frac{2.8В}{0.02А} = 140Ом$$

Итак мы вычислили, что при токе 20мА, сопротивление резистора

должно быть 140 Ом. Но, чтобы исключить работу светодиода на предельных параметрах, увеличим сопротивление примерно в 2 раза, до 300 Ом. В результате, ток уменьшится в 2 раза и светодиод будет светиться на половину своей яркости. При этом, не имеет значения с какой стороны подключен резистор, со стороны катода или анода. В схеме, показанной на Рис. 3.3, резистор подключен со стороны катода.

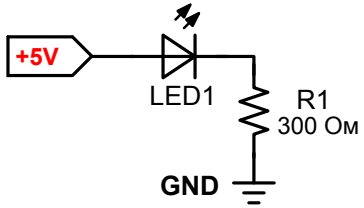


Рис. 3.3 Схема включения светодиода

Практика:

Светодиод, включенный согласно рассмотренной схеме, будет гореть все время, пока на него подается питание. Если необходимо усложнить схему, например сделать так, чтобы светодиод мигал или реагировал на нажатие кнопки, тогда не обойтись без контроллера.
Давайте соберем схему, но вместо светодиода с резистором воспользуемся готовым модулем, который позволяет сразу перейти к программированию.

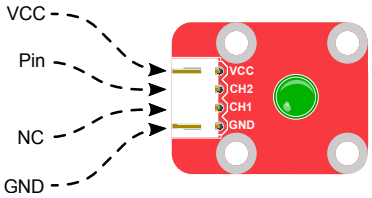


Рис. 3.4 Светодиод, назначение пинов

На Рис. 3.4 показано, как выглядит модуль со светодиодом.
В целях унификации, модуль подключается с помощью стандартного разъема с четырьмя контактами. Но, в данном случае, только два контакта оказываются задействованы. Пины обозначенные как NC (общепринятое сокращение от английского Not Connected), не используются. Контакты разъема имеют следующее назначение:

VCC — не задействован (NC)
CH2 — питание светодиода (+5V) от пина контроллера
CH1 — не задействован (NC)
GND — земля (общий)

Подключать светодиод к контроллеру будем через шилд QuatroPort A050. Перед тем как приступить к сборке, соедините контроллер DaVinci с шилдом QuatroPort A050 как показано на Рис. 3.5.

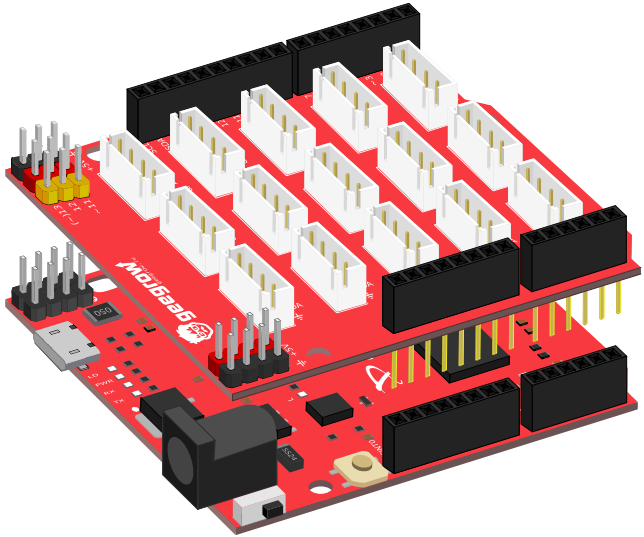


Рис. 3.5 Схема соединения QuatroPort A050 и контроллера DaVinci

Задача 3.1. Маячок

Собрать мигающий маячок согласно, схеме и написать программу так,чтобы светодиод мигал 1 раз в секунду.

Светодиодный модуль подключается к контроллеру через шилд, разъем номер 9 согласно схеме Рис. 3.6. При этом управляющий пин светодиода (CH2), соединяется с портом контроллера №9 .

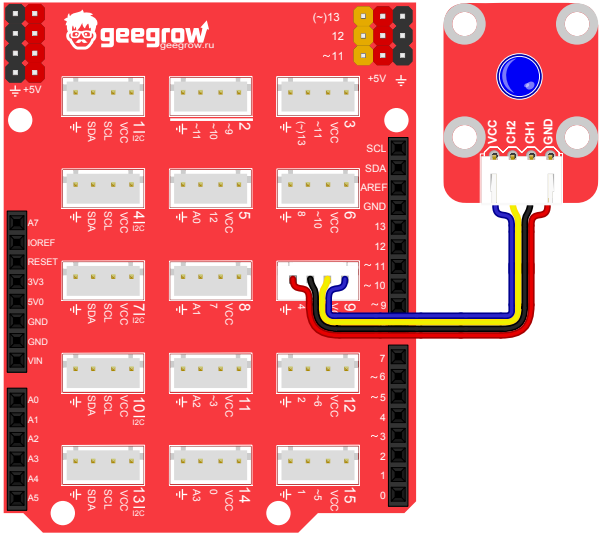
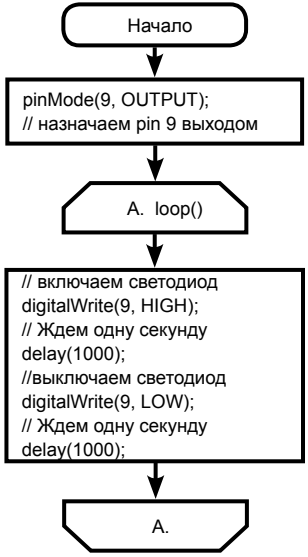


Рис. 3.6 Схема макета для задачи 3.1, Маячок

Начнем с составления блок-схемы алгоритма, суть которого сводится к выполнению бесконечного цикла, в ходе которого программа включает светодиод, ждет одну секунду, а затем выключает светодиод и снова ждет одну секунду. После этого цикл повторяется.



Напишем программу в соответствии с блок-схемой.

```
1  void setup(){
2    //Инициализируем вывод 9 как выход
3    pinMode(9, OUTPUT);
4  }
5
6  void loop(){
7    //Включаем светодиод
8    digitalWrite(9, HIGH);
9    //Ждем одну секунду
10   delay(1000);
11   //Выключаем светодиод
12   digitalWrite(9, LOW);
13   //Ждем одну секунду
14   delay(1000);
```

```
15 }
```

Программа готова, осталось загрузить ее в контроллер. Для этого откройте среду Arduino IDE и введите текст программы в поле редактирования кода.*(см. пункт 8, Рис. 1.3)*.
Вы так же можете скачать готовые примеры из раздела, посвященного данному набору на сайте geegrow.ru.

После того, как текст программы введен, правильно выбран тип контроллера и порт подключения, остается лишь нажать на кнопку загрузки в левом верхнем углу *(Кнопка со стрелкой, см. пункт 2, Рис. 1.3)*. Если все было сделано верно, светодиод сразу начнет мигать.
Теперь подробнее остановимся на самой программе и разберемся, как она работает.

Начнем с комментариев — строк в начале которых стоят символы «//». Все что следует за двойным слешем «//» воспринимается компилятором, как комментарий и пропускается. Такие строки служат лишь для пояснений к тексту программы, предназначенных для человека. Обратите внимание, что следующая за комментарием строка уже не расценивается как комментарий, если она так же не выделена символом «//».

Компилятор — служебная программа, переводящая код программы на языке C++ в машинный код, понятный контроллеру. Компилятор вызывается каждый раз, после того как вы нажимаете кнопку загрузки.

Если необходимо «закомментировать» больше одной строки, можно использовать символы многострочного комментирования:
/* - открывающий знак многострочного комментария
*/ - закрывающий знак многострочного комментария

Функция **`void setup(){}`** — выполняется в самом начале и только один раз при старте микроконтроллера. Она используется для инициализации переменных, определения режимов работы выводов микроконтроллера, подключения библиотек и т.д.
`void` — это ключевое слово, которое означает что функция ничего не возвращает программе. Если бы функция должна была вернуть переменную, на месте **`void`** следовало бы указать тип переменной (**`int`**, **`char`** и т.д.).

`setup` — имя функции. Имена функций и переменных лучше всегда делать осмысленными. Поскольку функция всегда выполняет определенные действия, общепринятой практикой является именование функций с помощью глаголов английского языка, характеризующих это действие. Так же стоит поступать с выбором имен переменных. Понятные имена функций и переменных облегчают чтение программы.

В круглых скобках **`()`** — указывается список входных параметров, которые называют аргументами функции. У функции **`void setup(){}`**, входные параметры отсутствуют, поэтому скобки оставляем пустыми либо вписываем ключевое слово **`void`**, что означает функцию без входных параметров. Например, **`void setup (void){}`**.

Скобки **`{}`** — являются телом функции, в котором содержится исполняемый код. В данном примере, функция setup содержит единственную строку **`pinMode(9, OUTPUT).`**

Команда **`pinMode(9, OUTPUT)`** устанавливает ножку микроконтроллера под номером **9** в состояние ВЫХОД (**`OUTPUT`**).

Вывод микроконтроллера может иметь два состояния ВХОД (**`INPUT`**) или ВЫХОД (**`OUTPUT`**).

Если установить вывод контроллера в состояние OUTPUT, то с помощью данного вывода можно будет управлять внешним устройством, то есть нагрузкой. В роли нагрузки может выступать светодиод, двигатель, реле и т.д. Во время исполнения программы, контроллер может подать на вывод **`OUTPUT`** напряжение +5В, и таким образом зажечь светодиод, запустить мотор или переключить реле.

Если вывод микроконтроллера установлен в состояние ВХОД (**`INPUT`**), это значит, что данный вывод может быть использован

для получения информации из внешнего мира. К такой ножке можно подключать различные датчики, например, датчик температуры, давления, освещенности и т.д. На самом деле есть еще одно состояние вывода микроконтроллера, которое называется Hi-Z, но оно используется очень редко и в рамках данного курса мы его рассматривать не будем.

Ниже по коду, объявлена функция **void loop()**. В теле этой функции находится постоянно исполняемый код. Код выполняется сверху вниз, циклически и с очень высокой скоростью, до нескольких тысяч раз в секунду.

Функция **digitalWrite(9, HIGH)** устанавливает на 9-ом выводе уровень 5 вольт. Обычно 5В или напряжение питания микроконтроллера VCC еще называют высоким уровнем или HIGH Level.

Функция **digitalWrite(9, LOW);** устанавливает на 9-м выводе 0 вольт или низкий уровень или LOW Level.

Функция **delay(1000);** — заставляет микроконтроллер ждать 1000 миллисекунд, то есть ровно 1 секунду. Аргумент функции **delay();** может принимать любое значение в диапазоне от 0 до 4294967295 миллисекунд, что составляет 1193 часа.

Снова обратимся к тексту программы. Не смотря на то, что программа работает правильно, она составлена не оптимально и ее можно улучшить.
Предположим мы решим переключить светодиод в соседний разъем шилда №12. В этом случае придется изменить все строки в программе, в которых встречается обращение к порту №9 разъема №9 на порт №6. Если код состоит всего из 10 строк, как у нас, то это еще не столь критично, но реальная программа может содержать несколько сотен строк и больше. В этом случае постоянно вносить подобные изменения станет не только тяжело, но и опасно. Ведь можно случайно заменить номер пина там где это не нужно или наоборот, пропустить и не изменить пин там где это нужно.

Для того, чтобы избежать ошибок и упростить отладку программы, перепишем ее следующим образом:

```
1  //Определяем константу LED
2  #define LED 9
3  void setup(){
4    //Инициализируем вывод 9 как выход
5    pinMode(LED, OUTPUT);
6  }
7
8  void loop(){
9    //Включаем светодиод
10   digitalWrite(LED, HIGH);
11   //Ждем секунду
12   delay(1000);
13   //Выключаем светодиод
14   digitalWrite(LED, LOW);
15   //Ждем секунду
16   delay(1000);
17 }
```

Что изменилось? Появилась единственная строка **#define LED 9**. Здесь **#define** — директива (команда) которая объявляет константу **LED** и присваивает ей значение **9**. Теперь в программе вместо числа **9** можно везде записать **LED**. При компиляции программы, компилятор сам подставит вместо **LED** значение **9**.

Константа, в данном случае **LED**, так же называется макросом. А процесс замены, который мы совершили, называется макроподстановкой.
Второй недостаток в программе это использование функции задержки **delay(1000)** в теле программы. Но, почему это является серьезным недостатком и как его исправить обсудим в следующем уроке.

Урок 4. Подключение кнопки

В этом уроке вы научитесь правильно работать с тактовой кнопкой, управлять с ее помощью другими устройствами, а так же узнаете, что такое эффект дребезга контактов и как его избежать.

Теория:

Для начала, рассмотрим, как устроена тактовая кнопка. Как можно догадаться, внутри пластикового корпуса кнопки находится пара металлических контактов, которые разомкнуты в нормальном состоянии и замыкаются при нажатии. На Рис. 4.1 показан внешний вид кнопки и ее обозначение на электрических схемах.

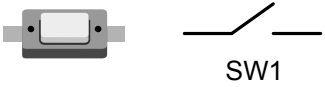


Рис. 4.1 Тактовая кнопка

С точки зрения контроллера, тактовая кнопка представляет собой простейший датчик с двумя состояниями замкнут/разомкнут.

Общаясь с внешним миром, контроллер оперирует, прежде всего, понятиями высокого HIGH (логическая единица — 5В) и низкого LOW (логический ноль — 0В) уровней. Значит нужно подключить кнопку таким образом, чтобы ее нажатие вызывало изменение логического уровня на ножке микроконтроллера.
Не будем усложнять схему, а просто подключим кнопку с одной стороны к ножке микроконтроллера, а с другой стороны к «земле», как показано на Рис. 4.2 и разберемся, как она будет работать.

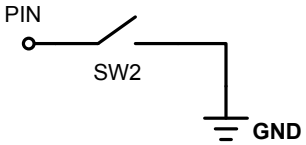


Рис. 4.2 Подключение кнопки к контроллеру (вариант 1)

При нажатии на кнопку, потенциал вывода контроллера изменится на низкий (LOW), так как вывод напрямую соединен с «землей» GND. Но, если затем отпустить кнопку, то ничего не произойдет — ножка микроконтроллера останется «висеть в воздухе», так как ножка не подключена ни к +5В, ни к 0В. В этом случае ее состояние может самопроизвольно измениться, например, из-за статического электричества. В любом случае, результат будет непредсказуемым и схема будет работать неправильно.

В таких случаях, чтобы ножка не висела в воздухе, ее с помощью резистора «подтягивают» к высокому уровню питающего напряжения (HIGH) Рис. 4.3.

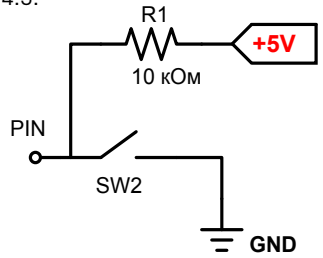


Рис. 4.3 Подключение кнопки к контроллеру (вариант 2)

Сопротивление резистора выбирают достаточно большим, порядка 10кОм, чтобы уменьшить протекающий через него ток.

При нажатии на кнопку, заряд стекает на «землю» и потенциал вывода контроллера меняется на LOW (0В). Это происходит потому, что ток текущий через подтягивающий резистор на много меньше и пока кнопка нажата, его не хватает для того, чтобы компенсировать стекание заряда на «землю». После отпускания кнопки, подтягивающий резистор быстро возвращает потенциал вывода к состоянию HIGH (5В).

Для простоты этот процесс можно представить себе на примере сосуда с большим отверстием в дне, в который втекает тонкая струйка воды. Пока отверстие в дне открыто (кнопка нажата), тонкая струйка не может наполнить сосуд и вся вода сразу же вытекает. Когда отверстие закрывается (кнопка отпущена), сосуд наполняется и уровень воды в нем поднимается.

Итак, мы исправили схему, но она все еще может работать неправильно из-за явления, называемого «дребезгом контактов».

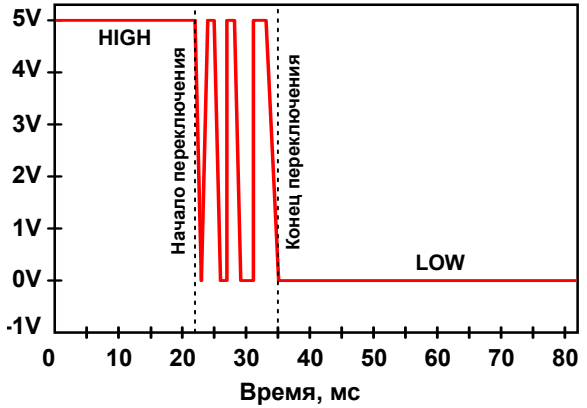


Рис. 4.4. Эффект дребезга контактов при нажатии кнопки

Дело в том, что переключение с высокого уровня на низкий происходит не мгновенно, а занимает несколько миллисекунд. В течение этого времени, из-за несовершенства контактов, они несколько раз размыкаются и замыкаются, а напряжение, несколько раз, изменяется от HIGH до LOW и обратно, как показано на Рис. 4.4

Чтобы устранить эффект дребезга контактов, параллельно кнопке включают конденсатор, сглаживающий скачки напряжения и делающий процесс переключения более плавным. Обычно номинал конденсатора делают не слишком большим, порядка 0.1 мкФ (микрофарад), иначе он будет заряжаться слишком долго и время переключения кнопки станет заметно больше.

Сглаживающий конденсатор заряжается до уровня HIGH почти мгновенно, а во время дребезга контактов разряжается, повышая уровень напряжения в цепи. Таким образом скачки напряжения значительно уменьшаются.

Теперь кнопка будет работать правильно, но схема все еще нуждается в доработке. Предположим, что во время написания программы, мы ошиблись и случайно определили вывод контроллера не как INPUT, а как OUTPUT. В этом случае через ножку контроллера потечет ток короткого замыкания, превышающий предельно допустимый и контроллер выйдет из строя.
Чтобы защитить контроллер, последовательно с кнопкой добавим токоограничительный резистор 10 кОм, как показано на Рис. 4.5
Используя Закон Ома, можно вычислить, что резистор ограничит протекающий через него ток на уровне 0.5мА.

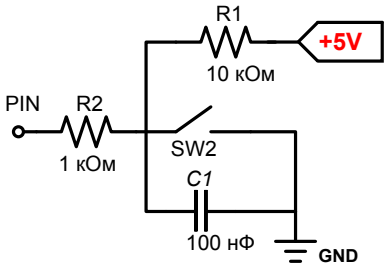


Рис. 4.5. Подключение кнопки к контроллеру (вариант 3)

Практика:

Рассмотренная схема подключения реализована в конструкции модуля с тактовой кнопкой. В этом и будущих уроках этот модуль будет использоваться вместо отдельной кнопки для облегчения подключения и проведения экспериментов.
На Рис. 4.6 показано, как выглядит модуль с тактовой кнопкой, а так же назначение пинов разъема.

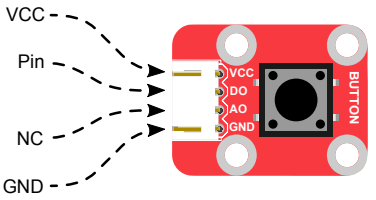


Рис. 4.6. Кнопка, назначение пинов

Модуль подключается с помощью стандартного разъема с четырьмя контактами. Задействованы три пина из четырех. Как видно из рисунка Рис. 4.6., контакты разъема имеют следующее назначение:
VCC — питание +5V
DO — подключается к порту контроллера
AO — не задействован (NC)
GND — земля (общий)

Теперь попробуем решить практическую задачу с кнопкой.

Задача 4.1. Простой фонарик

Написать программу управления светодиодом при помощи кнопки. Когда кнопка нажата, светодиод должен гореть. Когда же кнопка отпущена, светодиод должен гаснуть.

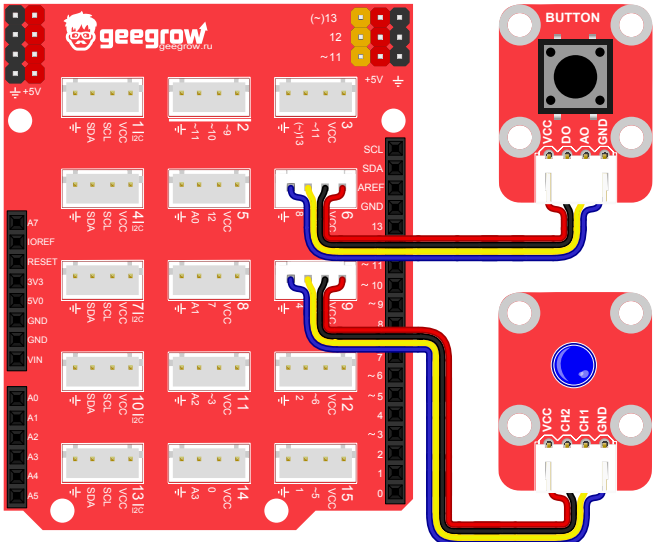
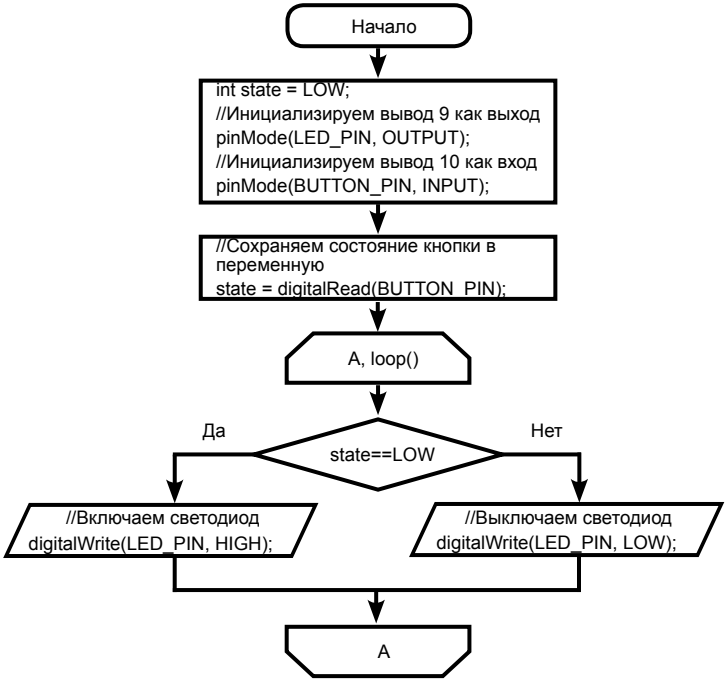


Рис. 4.7. Схема макета для задачи 4.1

Соберем макет согласно схеме Рис. 4.7. Светодиодный модуль включается в разъем №9, при этом управление им осуществляется с помощью порта контроллера №9. Модуль с тактовой кнопкой подключается к разъему №6, задействуя порт №10.

В будущем, почувствовав себя более уверенно в обращении с контроллером, вы можете подключать модули и к другим разъемам шилда. Но, до тех пор, лучше пользоваться таблицей совместимости, приведенной в разделе знакомства с конструктором и не забывать изменять в программе номера используемых портов.

Начнем с составления блок-схемы.



Теперь, напишем программу в соответствии с блок-схемой.

```
1  //Номер вывода светодиода
2  #define LED_PIN 9
3  //Номер вывода кнопки
4  #define BUTTON_PIN 10
5  //Переменная для хранения состояния кнопки
6  int state = LOW;
7
8  void setup() {
9    //Инициализируем вывод 9 как выход
10   pinMode(LED_PIN, OUTPUT);
11   //Инициализируем вывод 10 как вход
12   pinMode(BUTTON_PIN, INPUT);
13 }
14
15 void loop() {
16   //Сохраняем состояние кнопки
17   //в переменную
18   state = digitalRead(BUTTON_PIN);
19
20   //Если кнопка нажата
21   if (state == LOW) {
22     //Включаем светодиод
23     digitalWrite(LED_PIN, HIGH);
24   } else {
25     //Выключаем светодиод
26     digitalWrite(LED_PIN, LOW);
27   }
28 }
```

В программе были использованы две новые языковые конструкции.

`int state = LOW;` — объявление переменной типа **`int`**. Тип **`int`** — (от английского integer) — целочисленный тип данных, может принимать значение от -32768 до 32767.
При объявлении переменной на первом месте указывается ее тип, в нашем случае **`int`**. На втором месте указывается имя переменной **`state`**.
Если необходимо задать значение переменной, то после знака равно «=» указывается требуемое значение. В будущем значение переменной может изменяться.
Если нет необходимости задавать значение по умолчанию, то знак равно опускается и создается пустая переменная заданного типа;

`digitalRead(BUTTON_PIN);` — возвращает состояние ножки контроллера. В качестве аргумента функция принимает номер входа, в нашем случае константу **`BUTTON_PIN`**. Если на ножке был высокий уровень (+5В), то команда вернет **`HIGH`** или «1». Если на ножке низкий уровень (0В), то **`LOW`** или «0».
Важно помнить, что когда мы говорим об электрической цепи, то подразумеваем, что высокий уровень **`HIGH`** — это +5 вольт, а низкий **`LOW`** — 0 вольт. Однако в контексте программирования, мы оперируем не напряжениями, а константами **`HIGH`** и **`LOW`**. Константа **`HIGH`** определена в среде, как единица «1», а **`LOW`**, как ноль «0». По сути это просто флаг ДА или НЕТ.

`if (state == LOW) {...} else {...}` — условный оператор, предназначенный для проверки выполнения условия. В нашем случае он сравнивает значение переменной state с константой **`LOW`**, которая равно «0». Проще можно переписать выражение следующим образом: **`if (state == 0) {...} else {...}`**.
Если условие выполняется и переменная **`state`** равна нулю, выполняется блок команд в первых фигурных скобках. В противном случае выполняется блок команд во вторых фигурных скобках, после оператора **`else`**.

Условный оператор является одной из самых важных конструкций языка, позволяющий реализовывать различное поведение программы, в зависимости от входных данных, поэтому мы рассмотрим их более детально.

Условные операторы

Условный оператор **`if(...)`**

Условный оператор **`if(...)`**. В общем виде записывается следующим образом:

```
if(логическое выражение) {
        оператор_1;
} else {
        оператор_2;
}
```

Если логическое выражение принимает значение **TRUE** (истина), выполняется **оператор_1**, если ложно **FALSE**, выполняется **оператор_2**.

Оператор может иметь более одной условной части:

```
if(логическое выражение_1){
        оператор_1;
} else if(логическое выражение_2) {
        оператор_2;
} else if(логическое выражение_3) {
        оператор_3;
} else {
        оператор_4;
}
```

В этом случае, по порядку, сверху вниз, будет проверяться каждое логическое выражение, до тех пор, пока одно из них не будет истинно. В этом случае выполнится блок операторов соответствующий именно этому условному оператору **if**.
Если все логические выражения окажутся ложными, будет выполнен **оператор_4**.
Если, будут истинны сразу несколько логических выражений, исполнится только блок операторов соответствующий первому истинному логическому выражению, а остальные будут проигнорированы.

Важно помнить еще одну деталь — абсолютно любое числовое значение в условном операторе **if(...)** является истиной **TRUE**, даже отрицательные числа. И только значение 0, ложно **FALSE**.

Так же, в рамках конструкции условного оператора, нам впервые встретился символ «==». Это оператор сравнения.

Операторы сравнения

- Оператор (==), читается, как «равно». Проверяет на истинность выполнение условия: левая часть равна правой.
- Оператор (!=), читается, как «не равно». Проверяет на истинность выполнение условия: левая часть не равна правой.
- Оператор (>), читается, как «больше». Проверяет на истинность выполнение условия: левая часть больше правой.
- Оператор (<), читается, как «меньше». Проверяет на истинность выполнение условия: левая часть меньше правой.
- Оператор (>=), читается, как «больше либо равно». Проверяет на истинность выполнение условия: левая часть больше либо равна правой.
- Оператор (<=), читается, как «меньше либо равно». Проверяет на истинность выполнение условия: левая часть меньше либо равна правой.

Задание для самостоятельного выполнения 4.1 Управляемый маячок.

Используя схему из данного урока, измените логику работы программы так, чтобы при нажатии на кнопку, светодиод не горел непрерывно, а мигал через каждые 100 миллисекунд.

Ответ к заданию 4.1

```
1  //Номер вывода светодиода
2  #define LED_PIN 9
3  //Номер вывода кнопки
4  #define BUTTON_PIN 10
5  //Переменная для хранения состояния
6  //светодиода
7  bool ledState = false;
8
9  void setup() {
10   //Инициализируем вывод 9 как выход
11   pinMode(LED_PIN, OUTPUT);
12   //Инициализируем вывод 10 как вход
13   pinMode(BUTTON_PIN, INPUT);
14 }
15
```

```
void loop() {
  //Если кнопка нажата
  if (digitalRead(BUTTON_PIN) == LOW) {
    //Изменяем состояние переменной
    //на противоположное
    if (ledState == true) {
      ledState = false;
    } else {
      ledState = true;
    }
  } else {
    ledState = false;
  }

  if (ledState) {
    //Включаем светодиод
    digitalWrite(LED_PIN, HIGH);
  } else {
    //Выключаем светодиод
    digitalWrite(LED_PIN, LOW);
  }

  //Ждем 100 миллисекунд
  delay(100);
}
```

Задание для самостоятельного выполнения 4.2
Маячок с изменяющейся частотой миганий.

Используя схему из данного урока, измените программу так, чтобы, при удержании кнопки скорость миганий возрастала до 50 раз в минуту, а при отпускании снижалась до 1 раза в секунду.

Ответ к заданию 4.2

```
//Номер вывода светодиода
#define LED_PIN 9
//Номер вывода кнопки
#define BUTTON_PIN 10
//Переменная для хранения состояния
//светодиода
bool ledState = false;
//Переменная для хранения времени
//задержки между миганиями
int delayTime = 1000;

void setup() {
  //Инициализируем вывод 9 как выход
  pinMode(LED_PIN, OUTPUT);
  //Инициализируем вывод 10 как вход
  pinMode(BUTTON_PIN, INPUT);
}

void loop() {
  //Если кнопка нажата
  if (digitalRead(BUTTON_PIN) == LOW) {
    //Уменьшаем задержку, если текущее
    //значение больше 50 миллисекунд
    if (delayTime > 50) {
      delayTime = delayTime - 50;
    }
  } else {
    //Увеличиваем задержку, если текущее
    //значение менее 1000 миллисекунд
    //(1 сек)
    if (delayTime < 1000) {
      delayTime = delayTime + 50;
    }
  }

  if (ledState == true) {
    //Включаем светодиод
    digitalWrite(LED_PIN, HIGH);
    //Изменяем значение на противоположное
    ledState = false;
  } else {
    //Выключаем светодиод
    digitalWrite(LED_PIN, LOW);
    //Изменяем значение на противоположное
```

```
    ledState = true;
  }

  //Ждем заданное время
  delay(delayTime);
}
```

Урок 5. Многозадачность, прерывания

В предыдущих уроках, мы писали достаточно простые программы. Их простота заключалась в линейности логики алгоритма. В главном цикле проверялось состояние кнопки и сразу же изменялось состояние светодиода. Задержка, если необходимо, вносилась с помощью функции **delay()**. Этот пример отлично работает, с небольшими схемами и примитивной логикой. Но если хочется чего-то большего, то старыми средствами уже не обойтись и от использования **delay()** придется отказаться.

Теория:

Чтобы лучше понять в каких ситуациях линейно выполняющиеся алгоритмы могут стать проблемой, попробуем решить гипотетическую задачу.

Предположим, что к контроллеру подключено два светодиода LED1, LED2 и кнопка BUTTON. И мы хотим чтобы светодиод LED1 мигал 1 раз в секунду, а LED2 мигал, только когда нажата кнопка.

Так как задача гипотетическая, не будем собирать схему, а просто напишем программу:

```
#define LED_PIN1 10
#define LED_PIN2 9
#define BUTTON_PIN 6

void setup() {
  //Инициализируем вывод 10 как выход
  pinMode(LED_PIN1, OUTPUT);
  //Инициализируем вывод 9 как выход
  pinMode(LED_PIN2, OUTPUT);
  //Инициализируем вывод 6 как вход
  pinMode(BUTTON_PIN,INPUT);
}

void loop() {
  //Включаем светодиод
  digitalWrite(LED_PIN1, HIGH);
  //Ждем 1 секунду
  delay(1000);
  //Выключаем светодиод
  digitalWrite(LED_PIN1, LOW);
  //Ждем 1 секунду
  delay(1000);
  //Проверяем нажата ли кнопка
  if(digitalRead(BUTTON_PIN)==LOW) {
    //Включаем светодиод
    digitalWrite(LED_PIN2,HIGH);
  } else {
    //Выключаем светодиод
    digitalWrite(LED_PIN2,LOW);
  }
}
```

Такая программа не всегда будет работать правильно. Чтобы контроллер гарантированно успел опросить кнопку и «понять», что она нажата, придется удерживать ее не менее 2 сек. В противном случае нажатие кнопки может остаться незамеченным.

Так происходит потому, что микроконтроллер обрабатывает команды последовательно, строка за строкой. Пока контроллер занят выполнением функции **delay(1000)**, он не может перейти к выполнению следующей строки и проверить состояние кнопки. В то время, как остальной код в цикле исполняется очень быстро, функция **delay()** заставляет программу «замирать» на длительные промежутки времени. Очевидно, что от функции **delay()**

придется отказаться и использовать другой инструмент, с которым вы сейчас познакомитесь.

Прерывания

В компьютерах, многозадачность является привычным делом. Работая сразу в нескольких программах, вам не приходится заботиться о распределении ресурсов процессора, за вас это делает операционная система. Она быстро переключает ресурсы процессора между разными задачами и сама выстраивает приоритеты. В контроллерах операционной системы нет, но есть похожий инструмент, который называется прерыванием.

Прерывания — это сигнал (событие), который заставляет контроллер прекратить выполнение текущей задачи и приступить к исполнению другой, имеющей более высокий приоритет. После выполнения высокоприоритетной задачи, контроллер возвращается к той, которой был занят до прерывания.

Прерывание назначается командой:
attachInterrupt(interrupt, function, mode).

interrupt — это номер прерывания. Внешние прерывания связаны с портами контроллера и их можно увидеть на схеме (pinout) контроллера DaVinci. Порты способные работать с прерываниями обозначены как int0..int4 (Рис. 1.1).

Список портов с внешними прерываниями:

int0 — порт 3
int1 — порт 2
int2 — порт 0
int3 — порт 1
int4 — порт 7

function — это задача, которая должна быть выполнена в результате прерывания, оформленная в виде отдельной функции без входных параметров. Такая функция не должна возвращать никаких значений, а только обрабатывать данные и изменять значение внешних переменных, если это требуется.
mode — задает тип события по которому вызывается прерывание.

В качестве **mode** можно использовать следующие константы:
LOW — вызывает прерывание, когда на выводе контроллера установлен уровень **LOW**
CHANGE — прерывание вызывается при ЛЮБОЙ смене значения на выводе, с **LOW** на **HIGH** и наоборот
RISING прерывание вызывается только при смене значения на выводе с **LOW** на **HIGH**
FALLING — прерывание вызывается только при смене значения на выводе с **HIGH** на **LOW**

Замечание

Внутри функции обработки прерывания не работает **delay()**, значения возвращаемые **millis()** не изменяются. Переменные, изменяемые в функции, должным быть объявлены как **volatile**.

Квалификатор **volatile**, указывается во время объявления переменной и ставится перед типом. Применяется для того, чтобы изменить доступ к переменной компилятором и в программе.

volatile заставляет компилятор размещать переменную в ОЗУ, а не во временных регистрах, где хранятся все остальные переменные. При определенных условиях значения переменных, хранящихся в регистрах, могут оказаться неточными.

Переменные нужно объявлять как **volatile** в тех случаях, когда ее значение может быть изменено чем-либо, не зависящим от того участка кода, в котором она фигурирует (например, параллельно выполняющимся потоком). Применительно к Arduino, единственное место, где подобное может случиться, это участки кода, связанные с прерываниями (также называемые процедурами обработки прерываний).

Практика:

Задача 5.1. Фонарик с памятью

Написать программу управления светодиодом с использованием прерываний. Пусть при каждом нажатии на кнопку, состояние светодиода изменяется и остается таким же до следующего нажатия.

Начнем со сборки схемы. Чтобы схема работала правильно, необходимо подключить кнопку к разъему, в котором присутствует один из портов, работающий с прерываниями (3, 2, 0, 1, 7). Таких разъемов пять: разъем 11 (порт №3), разъем 12 (порт №2), разъем 14 (порт №0), разъем 15 (порт №1) и разъем 8 (порт №7).

Если внимательно посмотреть на разъемы, то видно, что порты с прерываниями расположены по разному — где-то на втором пине а где-то на третьем. Обратившись к схеме расположения пинов модуля с кнопкой, видим, что нам подходят только те разъемы, в которых порт прерываний находится рядом с пином питания VCC. Таких разъемов только три: разъем 11 (порт №3), разъем 14 (порт №0) и разъем 8 (порт №7). Мы выберем разъем 14 (порт №0).

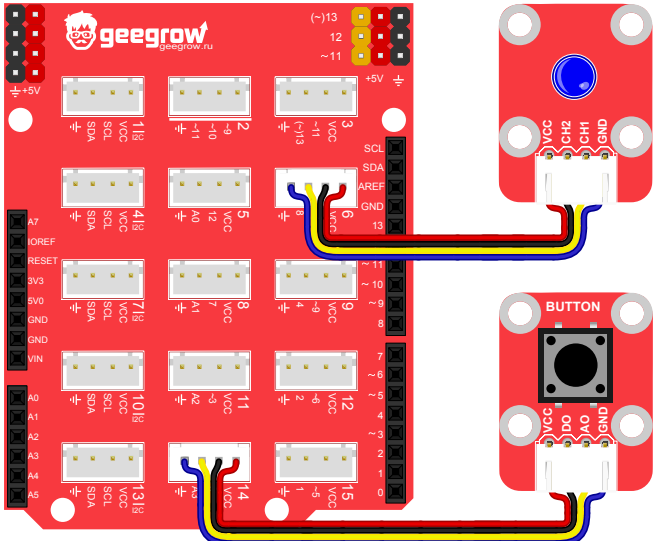
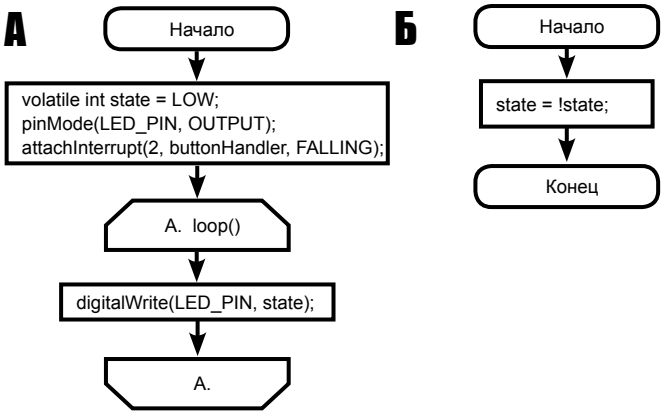


Рис. 5.1. Схема макета для задачи 5.1

Так как в этом уроке используется новый прием (работа с прерываниями), сначала рассмотрим блок-схему, а затем составим программу.



Из приведенной блок-схемы видно, что программа состоит из двух независимых частей. Основной цикл А, выполняется постоянно, раз за разом, а процедура обработки прерываний Б, выполняется однократно, каждый раз после нажатия на кнопку.

Напишем программу в соответствии с блок-схемой.

```
1  //Вывод ножки светодиода
2  int LED_PIN = 10;
3  //Переменная используемая в функции
4  //обработчике прерывания.
5  //Хранит состояние светодиода
6  volatile int state = LOW;
7
8  void setup() {
9    //Инициализируем вывод 5 как выход
10   pinMode(LED_PIN, OUTPUT);
11   //Настраиваем прерывание. Аргументы:
12   //2 - номер прерывания на порте 0,
```

```
13    //buttonHandler - название функции
14    //обработчика,
15    //FALLING - режим, по изменению с
16    //HIGH на LOW (задний фронт импульса)
17    attachInterrupt(2, buttonHandler, FALLING);
18  }
19
20  void loop() {
21    //Задаем светодиоду состояние,
22    //хранящееся в переменной state
23    digitalWrite(LED_PIN, state);
24  }
25
26  //Функция - обработчик прерывания
27  void buttonHandler() {
28    //Меняем значение переменной на
29    //противоположное
30    state = !state;
31  }
```

Подробно рассмотрим текст программы.

attachInterrupt(0, buttonHandler, FALLING) — настраивает прерывание в момент изменения уровня на ножке порта №0 с **HIGH** на **LOW**.

Почему порт №0? Потому, что мы использовали прерывание «2», а оно привязано к порту №0. Если бы мы выбрали, например, прерывание «4», то кнопку следовало бы подключить к выводу №7. Логика работы от этого не изменилась бы.

Почему мы использовали тип события **FALLING**?
Потому что мы хотели, чтобы прерывание срабатывало один раз за нажатие. Для этого подходят два типа FALLING и RISING. Если использовать RISING, то может возникнуть ситуация, когда кнопку нажали и удерживали. При этом прерывание не произойдет пока кнопка не будет отпущена.

LOW — не подходит нам потому, что пока кнопка нажата, прерывания будут происходить бесконечно с очень большой частотой.
CHANGE — не подходит потому, что вызывает два прерывания за один цикл нажатия. Первое прерывание происходит в момент нажатия кнопки, а второе в момент ее отпускания. То есть режим **CHANGE** работает как **FALLING** и **RISING** вместе.

В качестве второго аргумента передается **buttonHandler()**, функция определенная в конце программы. Каждый раз, когда, по событию смены уровня напряжения на порте №0, срабатывает прерывание, контроллер исполняет код записанный в функции **buttonHandler()**.
Практически во всех языках программирования, работающих с внешними событиями, такими как прерывания, существует традиция называть функции-обработчики этих событий используя окончание **handler** (перехватчик) или **listener** (слушатель). Это позволяет любому программисту читающему программу понять, что такая функция является обработчиком внешних событий

state = !state — команда изменяет значение хранящееся в переменной **state** на противоположное, с помощью унарного логического оператора отрицания. Эту строку можно прочитать как: **state = «не»state**. То есть если до этого значение **state** было равно **LOW**, то в результате оно изменится на **HIGH**, потому что **«не» LOW** это **HIGH**. И наоборот, если значение **state** было равно **HIGH**, то в результате оно изменится на **LOW**, потому что **«не» HIGH** это **LOW**.

Логические операторы

«!» — Логическое отрицание, **НЕ**. Записывается как «!a» и читается, как «НЕ a». Оператор инвертирует смысл логического выражения к которому применяется. То есть если **a=TRUE**, то **!a=FALSE**. И наоборот, если **a=FALSE**, то **!a=TRUE.**

«&&» — логическое **И**. Возвращает **TRUE** если все операнды истинны, иначе **FALSE**. Пример: пусть **a=TRUE**, **b=TRUE**, **c=TRUE**, тогда логическое выражение **(a && b && c)** будет равно **TRUE**. Если хотя бы одно условие не выполнится, все логическое выражение будет ложным. Пример: пусть **a=TRUE**, **b=TRUE**, **c=FALSE**, тогда логическое выражение **(a && b && c)** будет равно **FALSE**.

«||» — логическое **ИЛИ**. Возвращает **TRUE**, если хотя бы один из операндов имеет значение **TRUE**, иначе возвращает **FALSE**. Пример: пусть **a=TRUE**, **b=FALSE**, тогда **(a || b)** вернет **TRUE**. Если же **a=FALSE**, **b=FALSE**, то **(a || b)** вернет **FALSE**

Таким образом применение обработчика внешнего прерывания решило проблему некорректного определения нажатий на кнопку. Даже если основной цикл будет содержать многократные вызовы функции **delay()**, программа все равно будет работать верно.

Кроме внешних прерываний, существуют и другие. Причем, в ряде случаев, их использование может быть на много удобнее. Одним их таких прерываний являются прерывания таймера, которые мы рассмотрим в следующем уроке.

Урок 6. Таймеры

В этом уроке вы познакомитесь с таймерами — еще одним инструментом, позволяющим решать другую, часто встречающуюся на практике задачу — выполнять команды по расписанию.

Теория:

Упрощенно можно представить себе таймер, как отдельное устройство, счетчик, в составе микроконтроллера, которое может считать от 0 до 65535 с различной частотой. Частота работы таймера определяется частотой работы микроконтроллера и предделителя.

Частота — это единица измерения, показывающая, сколько раз в секунду происходит событие. Например, если светодиод мигает со скоростью 10 раз в секунду, то мы говорим, что частота миганий равна 10Гц *(Герц, сокращ. Гц)*.

Микроконтроллер Atmega32U4, установленный на плате DaVinci имеет 4 таймера: Timer0, Timer1, Timer3 и Timer4. Причем Timer0 — 8-битный таймер/счетчик, то есть его счётный диапазон от 0 до 255. Timer1 и Timer3 — 16-битные таймеры/счетчики. Их счётный диапазон от 0 до 65535. А Timer4 — высокоскоростной 10-битный таймер/счетчик со счетным диапазоном от 0 до 1024.

Диапазон счета ограничен емкостью регистров в которых хранится текущее значение таймера. Как можно догадаться, 8-битный таймер хранит значение в 8-битном регистре, а 16-битный в 16-битном регистре. Максимальное число в двоичной форме, которое можно записать в один 8-битный регистр соответствует $2^8 = 256$. А в 16-битный регистр можно записать число $2^{16} = 65536$.

Регистр — группа быстродействующих ячеек памяти, предназначенных для хранения данных в двоичном виде. Каждая ячейка регистра может хранить 1 бит информации — 0 или 1.

Для удобства работы с таймером мы будем использовать библиотеку **TimerOne**, которую вы можете скачать с нашего сайта, либо загрузить самую последнюю версию с сайта https://github.com/PaulStoffregen/TimerOne.
Библиотека работает с 16-битным таймером Timer1 и предоставляет ряд функций, для более удобного взаимодействия с ним. Некоторые из этих функций выходят за пределы изучаемой темы, поэтому они будут рассмотрены в последующих уроках.

Рассмотрим, как подключить библиотеку, инициализировать таймер для управления процессами привязанными ко времени.

Для подключения библиотеки (любой, а не только TimerOne), необходимо включить в начало программы строку с именем заголовочного файла: **#include <TimerOne.h>**.
Предварительно необходимо импортировать файл библиотеки в среду Arduino IDE. Для этого зайдите в пункт меню Эскиз>Импорт библиотек>Добавить библиотеку (или, если вы используете англоязычную версию IDE Sketch > Import Library .> Add Library).

Инициализация таймера осуществляется при помощи вызова функции **Timer1.initialize(period)**, которая в качестве аргумента принимает значение периода между тактами в микро-

секундах. Если инициализировать таймер не указав период, то по умолчанию он будет равен 1000000 микросекунд (1сек). Период обязательно должен быть выражен целым числом, ни в коем случае не дробным.

Момент, когда таймер досчитает до числа, заданного периодом, называется переполнением таймера. Переполнение таймера и есть то событие, которое вызывает прерывание.

`Timer1.setPeriod(period)` — изменяет значение периода у инициализированного таймера и принимает в качестве аргумента новое значение периода в микросекундах. Функция полезна, если в процессе работы программы вы решили изменить период.

`Timer1.attachInterrupt(void(*isr)(), long period=-1)` — в качестве первого параметра передается функция обработчик прерывания. Второй аргумент, период можно не указывать, если вы не собираетесь его изменять.

Внимание! Изменяя период, вы меняете его для всего таймера. Поэтому, будут затронуты все блоки кода работающие с таймером.

Внимание! Не стоит делать period слишком маленьким и размещать слишком много кода в обработчике прерывания. Иначе может сложиться ситуация, когда контроллер все время будет работать в прерываниях и не будет успевать исполнять основной блок программы — программа окажется заперта в прерывании.
`Timer1.detachInterrupt()` — отключает прикрепленное прерывание.

Практика:

Перейдем к решению практической задачи. Так как алгоритм будет предельно прост, в данном случае откажемся от составления блок-схемы. В будущем так же будем сопровождать блок-схемами только сложные алгоритмы.

Задача 6.1. Сигнальные огни

Пусть синий светодиод мигает 10 раз в секунду в основном цикле. А красный светодиод будет управляться таймером и мигать с частотой 1 раз секунду. Соберем схему согласно Рис. 6.1 и напишем программу.

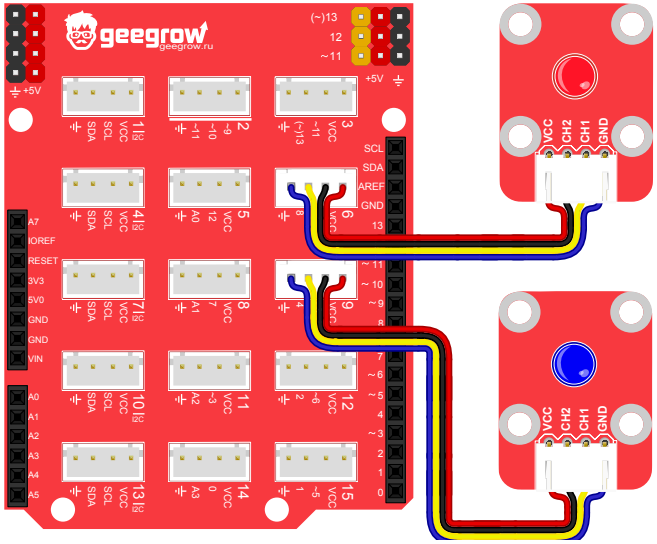


Рис. 6.1. Схема макета для задачи 6.1

```
1  //Подключаем библиотеку TimerOne
2  #include <TimerOne.h>
3  //Синий светодиод
4  #define LED1 9
5  //Красный светодиод
6  #define LED2 10
7
8  void setup() {
9    //Инициализация входов/выходов
10   pinMode(LED1, OUTPUT);
11   pinMode(LED2, OUTPUT);
```

```
12    //Переполнение таймера в микросекундах
13    Timer1.initialize(500000);
14    Timer1.attachInterrupt(timerHandler);
15  }
16
17  void loop() {
18    //Переключаем состояние светодиода на
19    //противоположное с помощью оператора
20    //логического отрицания «!»
21    digitalWrite(LED1, !digitalRead(LED1));
22    //Ждем 0.1 сек
23    delay(100);
24  }
25
26  void timerHandler(void){
27    //Переключаем состояние светодиода на
28    //противоположное с помощью оператора
29    //логического отрицания «!»
30    digitalWrite(LED2, !digitalRead(LED2));
31  }
```

Timer1.initialize(500000) — инициализирует таймер с периодом срабатывания 0.5 сек.

Timer1.attachInterrupt(timerHandler) — задает функцию обработчик прерывания.

Рассмотренный пример демонстрирует использование таймера для решения задач строго привязанных ко времени. Причем, даже использование в основном цикле функции **delay()**, не мешает нормальной работе программы.
Как и в случае с внешними прерываниями, программа представляет собой два параллельно выполняющихся алгоритма. Первый выполняется в главном цикле, а второй в прерывании.
При необходимости, можно даже расположить в функции **timerHandler()** код для управления обоими светодиодами и полностью освободить главный цикл. Подобный подход позволяет упростить любую, даже сильно запутанную программу и заставить множество устройств работать вместе, не мешая друг другу.

Задание для самостоятельного выполнения 6.1

Используя схему к задаче 6.1, измените логику работы программы так, чтобы, красный светодиод (LED2) мигал только 3 раза, после чего отключался.
При решении задачи воспользуйтесь квантификатором **volatile**.

Ответ к заданию 6.1

```
1   //Подключаем библиотеку TimerOne
2   #include <TimerOne.h>
3
4   #define LED1 9
5   #define LED2 10
6
7   volatile int iterations = 0;
8
9   void setup() {
10    //Инициализация входов/выходов
11    pinMode(LED1, OUTPUT);
12    pinMode(LED2, OUTPUT);
13    //Переполнение таймера в микросекундах
14    Timer1.initialize(1000000);
15    Timer1.attachInterrupt(timerHandler);
16  }
17
18  void loop() {
19    //Переключаем состояние светодиода на
20    //противоположное с помощью оператора
21    //логического отрицания !
22    digitalWrite(LED1,!digitalRead(LED1));
23    //Ждем 0.1 сек
24    delay(100);
25  }
26
27  void timerHandler(void){
28    //Переключаем состояние светодиода на
29    //противоположное с помощью оператора
30    //логического отрицания !
31    digitalWrite(
```

```
    LED2,
    !digitalRead(LED2)
  );

  iterations++;

  if (iterations == 6) {
    Timer1.detachInterrupt();
  }
}
```

Задание для самостоятельного выполнения 6.2

Используя схему к задаче 6.1, измените логику работы программы так, чтобы оба светодиода управлялись таймером.

Ответ к заданию 6.2

```
//Подключаем библиотеку TimerOne
#include <TimerOne.h>

#define LED1 9
#define LED2 10
//Переменная — счетчик итераций
volatile int iterations = 0;

void setup() {
  //Инициализация входов/выходов
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  //Переполнение таймера
  //100000мкс = 0.1сек
  Timer1.initialize(100000);
  Timer1.attachInterrupt(timerHandler);
}

void loop() {
  //В основном цикле пусто
}
```

```
//Сюда попадаем 10 раз в секунду
void timerHandler(void){
  //Меняем состояние LED1 каждый раз
  digitalWrite(
    LED1,
    !digitalRead(LED1)
  );

  //Если досчитали до 10,  значит
  //значит прошла 1 секунда.
  if (iterations == 10) {
    //Обнуляем счетчик
    iterations = 0;

    //Меняем состояние LED2
    digitalWrite(
      LED2,
      !digitalRead(LED2)
    );
  } else {
    //Если не досчитали до 10,
    //увеличиваем счетчик на 1
    iterations++;
  }
}
```

Урок 7. Управление светодиодом с помощью ШИМ

В этом уроке мы расскажем что такое ШИМ и как с его помощью можно плавно управлять аналоговыми устройствами.

Теория:

В предыдущих уроках мы научились работать со светодиодом. Мы просто включали и выключали его, подавая высокий HIGH либо низкий LOW уровень напряжения. Но что, если нам необходимо плавно изменять яркость светодиода? Очевидно, что для этого необходимо так же плавно регулировать уровень напряжения на светодиоде и в этом нам поможет ШИМ.

ШИМ (Широтно Импульсная Модуляция)

В английском языке эта аббревиатура пишется как PWM (Pulse Width Modulation). ШИМ, это метод, позволяющий получить плавно меняющееся напряжение с помощью устройства, умеющего генерировать только цифровые импульсы.

Чтобы лучше разобраться с ШИМ модуляцией, проведем мысленный эксперимент.

Предположим, что у нас есть лодка с электрическим мотором, скорость которого не регулируется, но обеспечить плавную регулировку скорости лодки необходимо.

Мы можем включить двигатель и тогда лодка постепенно наберет максимальную скорость или выключить двигатель и тогда лодка так же постепенно остановится. Но для нормального управления нам надо научиться регулировать скорость плавно. Что если мы станем периодически включать и выключать двигатель с некоторой частотой, допустим, 1 раз в секунду. Тогда за счет сопротивления воды и инерционности лодки, ее ход станет плавным, будет обеспечена плавность движения.

Если потребуется набрать скорость, надо увеличить длительность периодов работы двигателя или же наоборот уменьшить, если скорость потребуется сбросить.

Вернемся к светодиоду. Следуя аналогии с корабликом, чтобы регулировать яркость светодиода, мы должны подавать на светодиод не постоянное напряжение а периодические импульсы, следующие друг за другом с некоторой частотой и коэффициентом заполнения, как показано на Рис. 7.1.

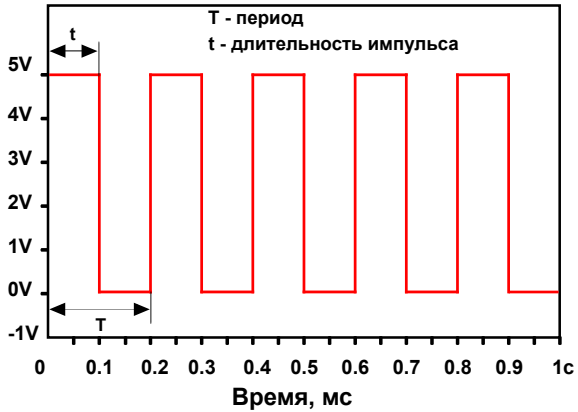


Рис. 7.1. ШИМ модуляция

Чтобы увеличить яркость светодиода, надо изменить длительность импульса t — увеличить коэффициент заполнения, Рис. 7.2.

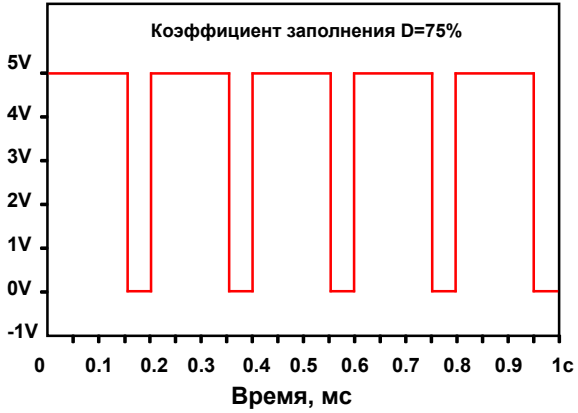


Рис. 7.2. ШИМ модуляция. Коэф. заполнения 75%

Чтобы уменьшить яркость, надо уменьшить длительность импульса — уменьшить коэффициент заполнения Рис. 7.3.

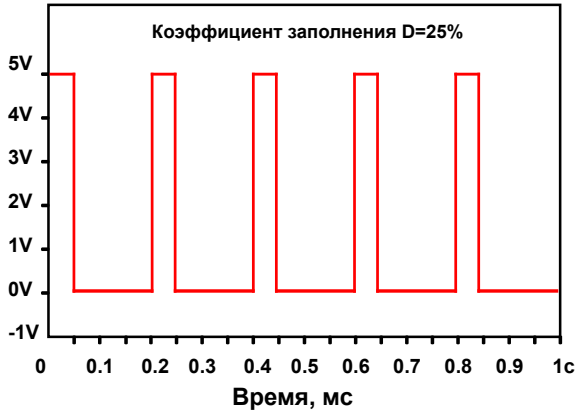


Рис. 7.3. ШИМ модуляция. Коэф. заполнения 25%

Очевидно, что чем больше длительность импульса по отношению к периоду, тем больше энергии передается светодиоду. Объем передаваемой энергии эквивалентен площади под графиком.

Терминология

Ранее было введено понятие частоты. Теперь введем еще несколько, связанных с ней, понятий.
Итак, напомним что, *частота* — это количество импульсов в секунду. Если светодиод мигает 1 раз в секунду, можно сказать, что он мигает с частотой 1Гц. Если 10 раз в секунду то 10Гц и т.д.

Период — величина обратная частоте и обозначающая время между двумя последовательными импульсами. Если частота F=10Гц, то период T=1/F = 0.1 сек.

Коэффициент заполнения — безразмерная величина характеризующая отношение длительности импульса к длительности периода. Например, если период T = 0.1сек, а длительность импульса t = 0.05сек, то коэф. заполнения определяется как D = t/T и равен 0.5. Коэффициент заполнения удобнее выражать в процентах, то есть, в нашем случае, вместо 0.5 запишем 50%.
В литературе часто используют термин «Скважность», это величина обратная коэффициенту заполнения. Но коэффициент заполнения более интуитивно понятная единица измерения, поэтому мы пока будем использовать ее.
Рис. 7.1, 7.2 и 7.3 иллюстрируют ШИМ сигнал одинаковой частоты но с коэффициентом заполнения 50%, 75% и 25% соответственно.

Практика:

Несмотря на то, что метод ШИМ подразумевает изменение только длительности импульсов, важную роль играет выбор частоты. Если частота будет меньше 25 Гц, то глаз будет видеть просто мигающий светодиод и создать эффект плавности регулировки, не получится. Но при работе с контроллером DaVinci, как и с другими платами Arduino, об этом можно не беспокоиться, потому что функция analogWrite() генерирует ШИМ с частотой 470 Гц.

Контроллеры DaVinci имеют 7 аппаратных выводов шим, отмеченных знаком «~». Это выводы: 3, 5, 6, 9, 10, 11, 13. Но, при необходимости ШИМ можно вывести и на другие выводы.

ШИМ сигнал генерируется с помощью таймеров:

Таймер 0 — отвечает за системное время, ШИМ на выводах 3 и 11.
Таймер 1 — отвечает за генерацию ШИМ на выводах 9 и 10
Таймер 3 — отвечает за генерацию ШИМ на выводе 5.
Таймер 4 — отвечает за генерацию ШИМ на выводах 6 и 13.

Это означает, что изменение режима работы любого из этих таймеров приведет к неправильной работе ШИМ на соответствующих выводах. Очень Важно помнить об этом, когда вы собираетесь использовать таймер.

Задача 7.1. Пульсар

Напишем программу, имитирующую поведение квазара, так чтобы яркость светодиода плавно увеличивалась до 100%, а затем

уменьшалась до 0%. После чего цикл будет повторяться. Для проведения эксперимента, соберем схему согласно Рис. 7.4 и напишем программу.

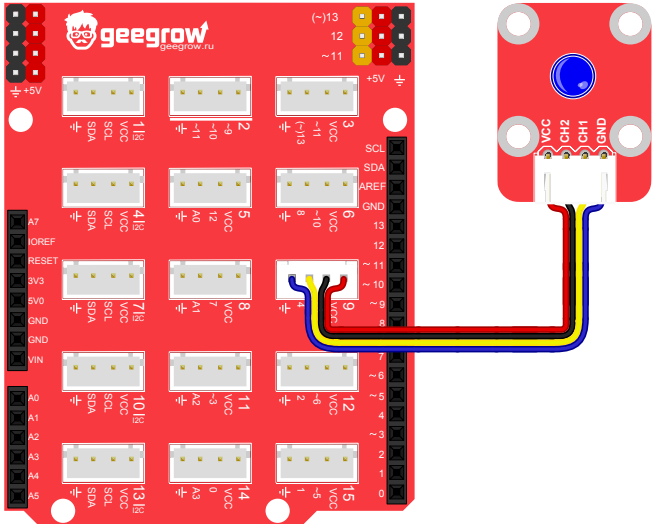


Рис. 7.4. Схема макета для задачи 7.1

```
#define LED_PIN 9

//pwmValue - содержит значение ШИМ, отве-
//чает за яркость изменяется от 0 до 255
byte pwmValue=1;

//Если brightnessUp равна TRUE, то увели-
//чиваем яркость, если нет, то уменьшаем
bool brightnessUp=true;

void setup(){
  pinMode(LED_PIN, OUTPUT);
}
void loop(){
  //Если надо увеличивать яркость
  if (brightnessUp) {
    //Увеличиваем значение ШИМ на единицу
    pwmValue++;

    //Если яркость достигла максимума
    if (pwmValue==255) {
      //Переключаем флаг
      brightnessUp=false;
    }
  } else {
    //Уменьшаем значение ШИМ на единицу
    pwmValue--;

    //Если яркость достигла минимума
    if (pwmValue==0) {
      //Переключаем флаг
      brightnessUp=true;
    }
  }

  //Задаем ШИМ на выводе LED
  analogWrite(LED_PIN, pwmValue);
  //Ждем 5мс
  delay(5);
}
```

«**++**» — оператор инкремента. Увеличивает значение переменной на единицу. Запись **a++** эквивалентна записи **a=a+1**.
«**--**» — оператор декремента. Уменьшает значение переменной на единицу. Запись **a--** эквивалентна записи **a=a-1**.

analogWrite(LED_PIN, pwmValue) — функция, запускающая генерацию сигнала ШИМ на порте **LED_PIN**. Второй аргумент **pwmValue** определяет коэффициент заполнения и может изменяться от 0 до 255. Если написать **analogWrite(LED_PIN, 127)**, то ШИМ будет запущен на порте **LED_PIN**, с коэффициентом

заполнения 50%. Вызов функции **analogWrite(LED_PIN, 0)** — установит на порте №5 низкий уровень **LOW**, а **analogWrite(LED_PIN,255)** установит высокий уровень **HIGH**.
Загрузив скетч, увидим, как светодиод плавно мигает. Это происходит благодаря изменению коэффициента заполнения ШИМ сигнала.
Теперь попробуйте поэкспериментировать со светодиодом и ШИМ.

Задание для самостоятельного выполнения 7.1

Используя схему из данного урока, измените логику работы программы так, чтобы светодиод 1 секунду горел на треть яркости, затем 1 секунду на две трети яркости и затем еще 1 секунду на 100% яркости, после чего цикл повторялся.

Ответ к заданию 7.1

```
#define LED_PIN 9

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

void loop(){
  //Задаем ШИМ на выводе LED 33%
  analogWrite(LED_PIN, 85);
  //Ждем 50мс
  delay(1000);

//Задаем ШИМ на выводе LED 66%
  analogWrite(LED_PIN, 170);
  //Ждем 50мс
  delay(1000);

  //Задаем ШИМ на выводе LED 100%
  analogWrite(LED_PIN, 255);
  //Ждем 50мс
  delay(1000);
}
```

Урок 8. Управление RGB светодиодом

В прошлом уроке был разобран пример управления светодиодом с помощью ШИМ. В этом уроке вы узнаете что такое RGB светодиод и каких интересных эффектов можно добиться с его помощью.

Теория:

RGB светодиод (от английского Red, Green, Blue) представляет собой сборку из трех светодиодов разных цветов красного, зеленого и синего. Внешний вид светодиода изображен на Рис. 8.1.

Рис. 8.1. Внешний вид RGB светодиода

С точки зрения схемотехники, в зависимости от способа соединения диодов внутри корпуса, RGB светодиоды бывают двух типов, с общим катодом (ОК) и с общим анодом (ОА).

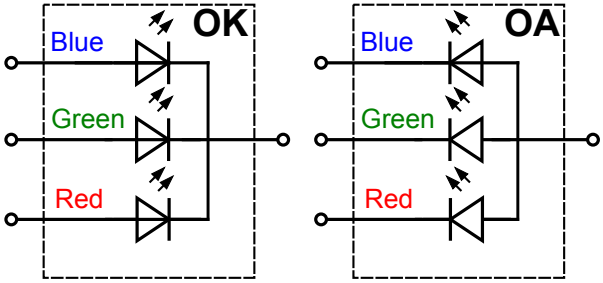


Рис. 8.2. Схемотехника RGB светодиода

RGB светодиод построенный по схеме с общим катодом (ОК) подключается катодом к «земле», а каждый из анодов, при этом, под-

ключается к управляющим портам контроллера.
Напротив, схема с общим анодом (ОА), подключается анодом к положительной шине питания, а катоды подключаются к портам контроллера, каждый из которых играет роль «земли».

Управляя RGB светодиодом дискретно, подавая напряжение на его ножки и отключая их, можно получить лишь три цвета. Но питая его выводы ШИМ сигналом различной скважности и комбинируя три цвета, можно получить практически любые оттенки. Лучше всего это иллюстрирует картинка изображенная на Рис. 8.3.

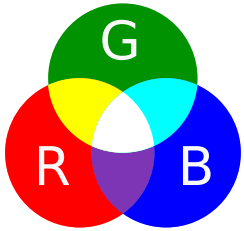


Рис. 8.3. Получение оттенков с помощью комбинации трех цветов.

Чтобы узнать какое значение ШИМ, каждого из трех каналов светодиода соответствует желаемому цвету, достаточно воспользоваться любым графическим редактором. Большинство из них позволяет выбрать нужный оттенок в палитре цветов и сразу увидеть цветовой код в формате RGB. Обычно цветовой код записывается либо цифрами в привычном десятичном формате, например:
Фиолетовый: (102, 0, 255). В скобках, первое число это интенсивность красного, второе зеленого и третье синего цвета. Так же, часто встречается запись в шестнадцатеричном HEX формате:
Фиолетовый: #6600FF. Логика записи в HEX формате точно такая же. Первые два знака после — красный цвет, вторые два — зеленый и последние два — синий.

Практика:

Во всех экспериментах с RGB светодиодом мы будем использовать готовый модуль, содержащий сам RGB светодиод и токоограничительные резисторы цепи питания. Внешний вид RGB модуля, и распиновка разъема показаны на Рис. 8.4.

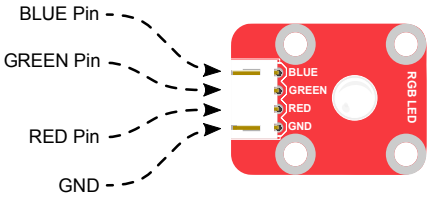


Рис. 8.4. RGB модуль

Модуль подключается с помощью разъема с четырьмя контактами, которые имеют следующее предназначение:
BLUE — к ШИМ порту контроллера
GREEN — к ШИМ порту контроллера
RED — к ШИМ порту контроллера
GND — земля (общий)

Так как в модуле задействованы все пины, причем пин VCC заменен на один из пинов светодиода, данный модуль можно подключать только в разъем номер 2 на шилде, предназначенный специально для него. В то же время, ни один другой модуль в этот разъем подключать нельзя.

Задача 8.1. Гирлянда

Для демонстрации управления RGB светодиодом с помощью ШИМ сигнала, соберем схему согласно Рис. 8.5 и напишем программу моделирующую поведение лампочки из разноцветной гирлянды. Светодиод должен с интервалом в 1 секунду изменять цвета в следующей последовательности: красный (255,0,0), оранжевый (255,102,0), желтый (255,212,42), аквамарин (0,255,255), фиолетовый (102,0,255), розовый (255,42,127).

В процессе написания программы, будем работать с каждым из внутренних светодиодов так, как будто это отдельные одноцвет-

ные светодиоды.

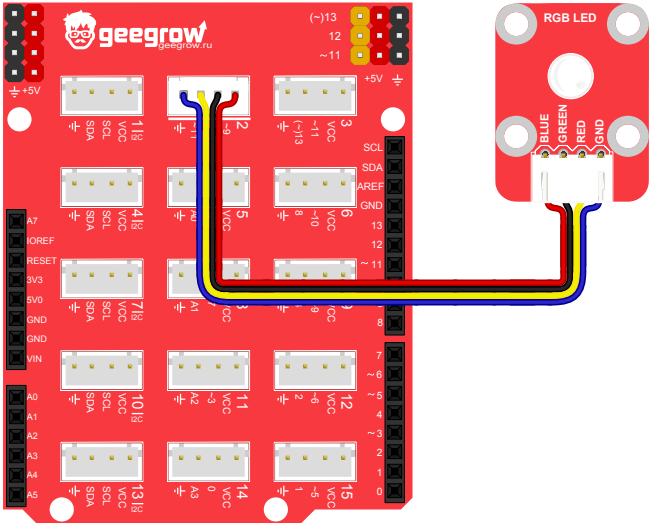


Рис. 8.5. Схема макета для задачи 8.1

Напишем программу.

```
//Красный светодиод
#define RED_PIN 11
//Зеленый светодиод
#define GREEN_PIN 10
//Синий светодиод
#define BLUE_PIN 9
void setColor(char r, char g, char b);

void setup(){
  pinMode(RED_PIN, OUTPUT);
  pinMode(GREEN_PIN, OUTPUT);
  pinMode(BLUE_PIN, OUTPUT);
}

void loop(){
  //Задаем цвет: красный (255,0,0)
  setColor(255, 0, 0);
  //Ждем 1сек
  delay(1000);

  //Задаем цвет: оранжевый (255,102,0)
  setColor(255, 102, 0);
  //Ждем 1сек
  delay(1000);

  //Задаем цвет: желтый (255,212,42)
  setColor(255, 212, 42);
  //Ждем 1сек
  delay(1000);

  //Задаем цвет: аквамарин (0,255,255)
  setColor(0, 255, 255);
  //Ждем 1сек
  delay(1000);

  //Задаем цвет: фиолетовый (102,255,255)
  setColor(102, 255, 255);
  //Ждем 1сек
  delay(1000);

  //Задаем цвет: розовый (255,42,127)
  setColor(255, 42, 127);
  //Ждем 1сек
  delay(1000);
}

void setColor(char r, char g, char b) {
  //Задаем ШИМ на выводе RED_PIN
  analogWrite(RED_PIN, r);
  //Задаем ШИМ на выводе GREEN_PIN
  analogWrite(GREEN_PIN, g);
```

```
  //Задаем ШИМ на выводе BLUE_PIN
  analogWrite(BLUE_PIN, b);
}
```

В ходе составления программы, был применен прием позволяющий упростить код и сделать его более наглядным и читаемым. Вместо того, чтобы каждый раз в момент изменения цвета вызывать команду **analogWrite** для каждого светодиода, мы вынесли эту часть кода в отдельную функцию **setColor**. Она объявлена в начале программы — **void setColor(char r, char g, char b)**, Само тело функции, при этом, описано в конце программы.

Теперь, когда нужно изменить цвет, достаточно вызвать функцию **setColor**, передав ей три аргумента — значения цветов в следующем порядке (красный, зеленый, синий). Тип аргументов объявлен как **char**.
char — целочисленный (символьный) тип данных, хранит число от 0 до 255.

Задание для самостоятельного выполнения 8.1

Используя схему из урока, еще больше упростите программу, переместив команду **delay** в функцию **setColor**. Для хранения переменной, отвечающей за задержку, используйте тип **short int**.

Ответ к заданию 8.1

```
//Красный светодиод
#define RED_PIN 11
//Зеленый светодиод
#define GREEN_PIN 10
//Синий светодиод
#define BLUE_PIN 9

void setColor(
  char r,
  char g,
  char b,
  short int msec
);

void setup(){
  pinMode(RED_PIN, OUTPUT);
  pinMode(GREEN_PIN, OUTPUT);
  pinMode(BLUE_PIN, OUTPUT);
}

void loop(){
  //Задаем цвет: красный(255,0,0)
  setColor(255, 0, 0, 1000);
  //Задаем цвет: оранжевый(255,102,0)
  setColor(255, 102, 0, 1000);
  //Задаем цвет: желтый(255,212,42)
  setColor(255, 212, 42, 1000);
  //Задаем цвет: аквамарин (0,255,255)
  setColor(0, 255, 255, 1000);
  //Задаем цвет: фиолетовый (102,255,255)
  setColor(102, 255, 255, 1000);
  //Задаем цвет: розовый (255,42,127)
  setColor(255, 42, 127, 1000);
}

void setColor(
  char r,
  char g,
  char b,
  short int msec
) {
  //Задаем ШИМ на выводе RED_PIN
  analogWrite(RED_PIN, r);

  //Задаем ШИМ на выводе GREEN_PIN
  analogWrite(GREEN_PIN, g);

  //Задаем ШИМ на выводе BLUE_PIN
  analogWrite(BLUE_PIN, b);

  //Ждем msec секунд
```

```
52    delay(msec);
53  }
```

В ответе к заданию, запись функции **void setColor()** не поместилась в одну строку, поэтому была записана в несколько строк. Смысл от этого не изменился.

Урок 9. Зуммер

В предыдущих двух уроках был рассмотрен метод управления светодиодами с помощью ШИМ, но светодиод, это лишь один из многих приборов, которыми можно управлять таким образом. На месте светодиода может быть электродвигатель, нагреватель и даже звуковой излучатель. В данном уроке мы покажем, как можно добиться различных звуковых эффектов с помощью зуммера.

Теория:

Зуммер представляет собой миниатюрный динамик, предназначенный для излучения сигналов малой мощности. Зуммеры бывают двух типов: активные (со встроенным генератором) и пассивные (управляемые внешним генератором). Как вы, наверное, уже догадались, мы будем использовать пассивный зуммер, так как он, в отличие от активного, меняет тональность вместе с изменением частоты внешнего сигнала.

Из предыдущих уроков вы знаете, что частоту генерации ШИМ сигнала на контроллере можно изменять. Но функция **analogWrite()** этого делать не умеет, поэтому мы воспользуемся командой **tone()**.

tone(pin, frequency, duration) — генерирует ШИМ сигнал заданной частоты. В качестве аргументов она принимает следующие параметры:
pin — номер порта, на котором необходимо запустить ШИМ;
frequency — желаемую частоту в Герцах;
duration — время звучания в миллисекундах (не обязательный аргумент). Если параметр **duration** не передан, сигнал будет звучать пока не будет вызвана функция **noTone()**.

noTone(pin) — выключает генерацию ШИМ на заданном порте. Использование функции **tone()** имеет свои особенности. Например нельзя запустить сигнал сразу на двух портах, а только на одном. Так же, ввиду того, что функция **tone()** использует Timer0, ее вызов может помешать генерации обычного ШИМ сигнала на портах №3 и №11 с помощью функции **analogWrite()**. Об этом важно помнить!

Практика:

Как и любой другой динамик, зуммер является индуктивной нагрузкой, и его лучше подключать к контроллеру через управляющий транзистор, который позволяет исключить протекание критических токов через порт контроллера.
Модуль с зуммером, который мы будем использовать в экспериментах полностью соответствует требованиям и уже содержит схему защищающую порт контроллера от повреждения. Внешний вид и распиновка модуля с зуммером представлены на Рис. 9.1.

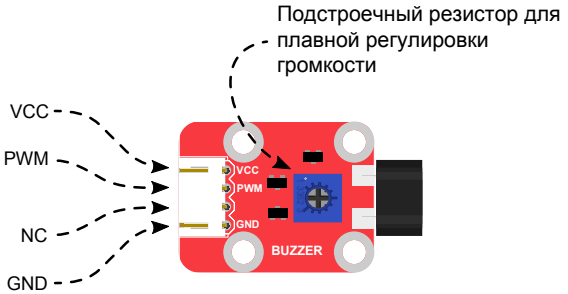


Рис. 9.1. Модуль-Зуммер

Модуль подключается с помощью разъема с четырьмя контакта-

ми, которые имеют следующее предназначение:
VCC — питание +5V
PWM — подключается к порту контроллера с ШИМ
NC — не задействован (NC)
GND — земля (общий)

Задача 9.1. Определитель возраста

Известно, что с возрастом, слух человека ухудшается и ухо теряет способность слышать верхние частоты. Таким образом, о возрасте человека, косвенно, можно судить по его слуху. Напишем программу, которая проверяет возраст по следующему алгоритму:
Если вы слышите частоту:
8 000 Гц — значит вы все еще живы
12 000 Гц — вы младше 50-ти лет
15 000 Гц — вы младше 40-ти лет
16 000 Гц — вы младше 30-ти лет
17 000 Гц — 18 000 Гц – вы младше 24-лет
19 000 Гц — вы младше 20-ти лет.

Теперь соберем макет согласно Рис. 9.2 и напишем программу, которая будет последовательно воспроизводить перечисленные частоты друг за другом с интервалом в 1 секунду. Длительность импульсов звука будет 3 секунды:

```
1  //Пин зуммера
2  #define BUZZER_PIN 9
3
4  void setup(){
5    pinMode(BUZZER_PIN, OUTPUT);
6  }
7
8  void loop(){
9    //Подаем на зуммер 8000Гц на 3 секунды
10   tone(BUZZER_PIN, 8000, 3000);
11   delay(1000);
12   //Подаем на зуммер 12000Гц на 3 секунды
13   tone(BUZZER_PIN, 12000, 3000);
14   delay(1000);
```

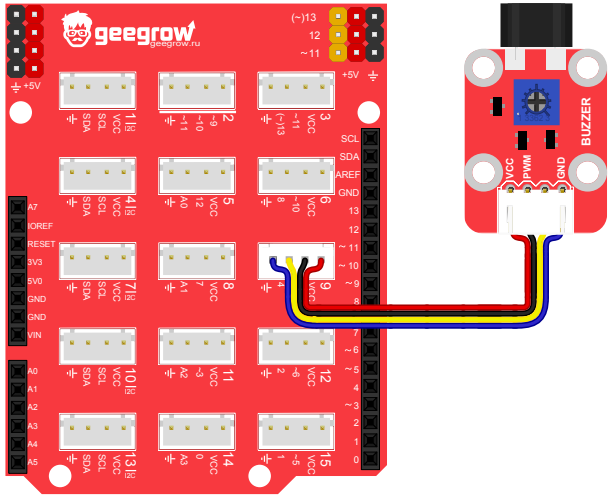


Рис. 9.2. Схема макета для задачи 9.1

```
15   //Подаем на зуммер 15000Гц на 3 секунды
16   tone(BUZZER_PIN, 15000, 3000);
17   delay(1000);
18   //Подаем на зуммер 16000Гц на 3 секунды
19   tone(BUZZER_PIN, 16000, 3000);
20   delay(1000);
21   //Подаем на зуммер 17000Гц на 3 секунды
22   tone(BUZZER_PIN, 17000, 3000);
23   delay(1000);
24   //Подаем на зуммер 19000Гц на 3 секунды
25   tone(BUZZER_PIN, 19000, 3000);
26   delay(1000);
27 }
```

Урок 10. Взаимодействие с компьютером. Терминал

В этом уроке мы расскажем, как с помощью специального инструмента среды Arduino IDE, можно передать данные с контроллера на компьютер и обратно, а также вывести их на экран. Такая возможность может оказаться полезной, если нужно «заглянуть» в контроллер и проверить правильность работы программы или просто управлять контроллером с помощью команд с компьютера.

Теория:

Для обмена данными с платами семейства Arduino, к которым относится и контроллер DaVinci, существует специальный инструмент — *Монитор порта* или *Терминал*. Общение через терминал осуществляется с помощью набора методов класса **Serial**.

В программировании, классом называют часть программы, описывающую некий тип данных и содержащую функции для работы с этими данными. Функции класса называют так же методами.

Serial.begin(speed) — устанавливает соединение с компьютером, а параметр **speed** задает скорость передачи данных в бит/с (бод). В экспериментах будем задавать скорость 9600 бит/с.

Serial.write(str) — передает данные на компьютер в виде бинарного кода. Может принимать второй аргумент **len**, задающий длину строки.

Serial.print(str, format) — передает данные на компьютер, как обычный текст. Переменная **format** может принудительно задавать формат пересылаемых данных. Допустимые значения **BYTE**, **BIN** (двоичный), **OCT** (восьмеричный), **DEC** (десятичный), **HEX** (шестнадцатеричный). Для дробных чисел второй параметр задает количество знаков после запятой. Например:
Serial.print(2.1462, 0) выводит «2»
Serial.print(2.1462, 2) выводит «2.14»
Serial.print(2.1462, 4) выводит «2.1462»

Serial.println(val, format) — работает точно так же, как и **Serial.print()**, но в конце добавляет перенос строки

Serial.available() — получает количество байт(символов) доступных для чтения из последовательного интерфейса связи.

Serial.read() — побайтно считывает принятые данные из буфера последовательного соединения.

Serial.end() — закрывает соединение и освобождает порты RX и TX, после чего они могут быть использованы для ввода/вывода так же как и другие порты контроллера. Чтобы заново открыть соединение, необходимо еще раз вызвать **Serial.begin()**.

Практика:

Чтобы лучше разобраться, как работает обмен данными через последовательное соединение, напишем простую программу.

Задача 10.1. Цифровое эхо

Логика работы программы следующая — пользователь вводит данные в окне терминала и нажимает кнопку «Послать», отправляя их в контроллер. Контроллер, в свою очередь, принимает данные, обрабатывает и передает обратно в терминал.

Для решения задачи, нам даже не потребуется собирать макет. Просто отсоедините все модули от контроллера и подключите его к компьютеру с помощью USB кабеля.

```
1  //Объявляем переменную для хранения данных
2  //полученных из последовательного соединения
3  char inputString;
4
5  void setup() {
6    //Открытие соединения на скорости
7    //9600 бит/сек
8    Serial.begin(9600);
```

```
}

void loop() {
  //Проверяем, есть ли принятые данные
  //в буфере
  if (Serial.available()) {
    //Читаем данные из буфера в переменную
    inputString = Serial.read();

    //Пишем в терминал собщение - You wrote:
    //Строка записывается в кавычках
    Serial.print("You wrote:");
    //Отправляем данные обратно в терминал.
    Serial.println(inputString);
  }
}
```

Осталось загрузить программу в контроллер и открыть терминал (монитор порта). Выберите в меню Инструменты > Монитор последовательного порта (или, если вы используете англоязычную версию программы, выберите Tools > Serial monitor).

Внешний вид Монитора порта изображен на Рис. 10.1.
Поле, рядом с кнопкой «Послать», служит для ввода сообщения, передаваемого контроллеру. Область под этим полем служит для отображения данных полученных от контроллера.

Если ввести в терминале любой символ, например цифру 1 и нажать кнопку «Послать», то контроллер выведет следующий результат:

```
You wrote:1
```

Если теперь ввести несколько символов, например abc, то результат окажется несколько неожиданным:

```
You wrote:a
You wrote:b
You wrote:c
```

Все дело в том, что команда `Serial.read()` читает не всю строку разом, а последовательно, символ за символом.
Чтобы прочитать всю строку целиком, нужно либо условиться, что длина передаваемой строки будет всегда одинакова и читать строго определенное количество символов. Либо определить символ конца строки и читать до тех пор, пока не будет отправлен этот символ.

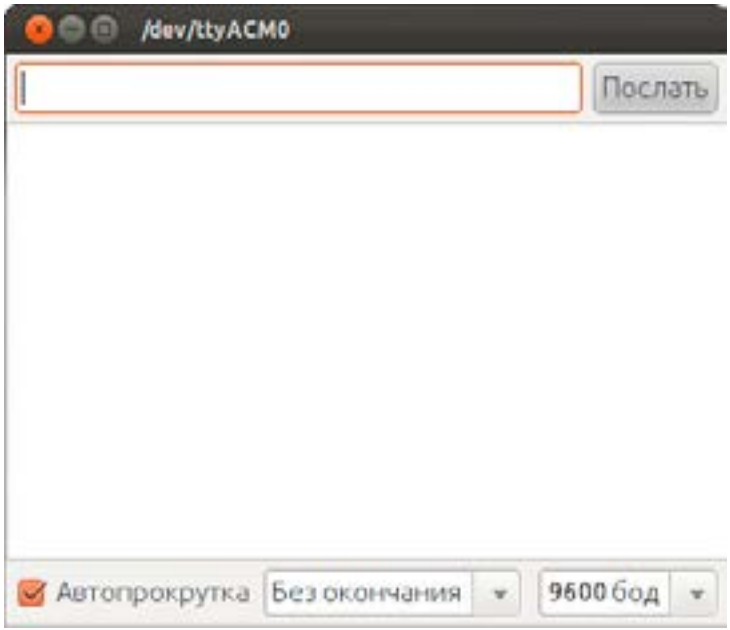


Рис. 10.1. Внешний вид окна Монитора порта (Терминал)

Первый вариант прост, но неудобен, ведь длина передаваемых данных, обычно, не одинакова. Поэтому, мы изменим программу, условившись, что символ точка «`.`» будет означать конец строки.

```
//Объявляем переменную типа String
//для хранения строки символов
String str;
//Объявляем переменную типа char
//для хранения строки символов
char buff;

void setup() {
  //Открытие соединения на скорости
  //9600 бит/сек
  Serial.begin(9600);
}

void loop() {
  //Проверяем, есть ли принятые данные
  if (Serial.available()) {
    //Читаем принятый символ во
    //временную переменную. При этом
    //старое значение buff затирается
    buff = Serial.read();

    //Если прочитанный символ точка
    if (buff == '.') {
      //выводим результат
      Serial.println("You wrote:"+str);
      //Очищаем переменную str
      str="";
    } else {
      //Добавляем содержимое буфера
      //в конец строки str
      str+=buff;
    }
  }
}
```

В примере была использована переменная с новым типом данных **String**. Класс **String** — предназначен специально для работы со строками. Он позволяет объединять и расширять строки, осуществлять поиск и замену подстрок, и многое другое.

Serial.println("You wrote:"+str) — обратите внимание, выражение в скобках можно ошибочно принять за операцию математического сложения двух строк. На самом деле, оператор «+», в данном случае, осуществляет «склеивание» *(конкатенацию)* двух строк, левой и правой.

В строке **str+=buff,** так же осуществляется конкатенация двух строк, но запись осуществлена в более короткой форме. Для наглядности можно было бы переписать ее таким образом: **str=str+buff.** Обе формы записи эквивалентны по смыслу.

Теперь попробуем ввести в терминале «qwerty.», не забыв поставить в конце точку и убедимся, что программа работает именно так, как и было задумано. В терминале будет выведена строка:

```
You wrote:qwerty
```

Теперь, когда мы научились принимать и обрабатывать команды на стороне контроллера, а так же отправлять результат обратно в терминал, давайте попробуем управлять реальным устройством с компьютера.

Задача 10.2. Управляемый синтезатор частоты

Соберем управляемый синтезатор частоты. Желаемую частоту мы будем вводить в терминале, а подключенный к контроллеру зуммер будет издавать звук заданной тональности. Схему возьмем из предыдущего урока, изображенную на Рис. 9.2.

```
//Номер вывода зуммера
#define BUZZER_PIN 9
//Переменная для хранения частоты
String frequency;
//Переменная для хранения текущего
//символа прочитанного из порта
char buff;

void setup() {
  //Открытие соединения на скорости
  //9600 бит/сек
```

```
  Serial.begin(9600);
}

void loop() {
  //Проверяем, есть ли принятые данные
  if (Serial.available()) {
    //Читаем принятый символ во временную
    //переменную.
    buff = Serial.read();

    //Если прочитанный символ точка
    //считаем, что это конец команды и
    //переходим к ее обработке
    if (buff == '.') {
      //Выводим принятый результат в терминал
      Serial.println("You wrote:"+frequency);
      //Включаем зуммер с заданной частотой
      //на 3 секунды
      tone(BUZZER_PIN, frequency.toInt(), 3000);
      //Очищаем переменную frequency
      frequency="";
    } else {
      //Убеждаемся, что  получена цифра
      if (isDigit(buff)) {
        //Добавляем содержимое буфера в
        //конец строки frequency
        frequency+=buff;
      }
    }
  }
}
```

В основном цикле программы, в строке **Serial.available()**, постоянно проверяется наличие новых данных в буфере последовательного соединения.
Очередной, прочитанный из порта символ, сохраняется во временную переменную **buff**. При этом, программа ожидает только цифры или знак точка «.». Если вы случайно введете букву или какой-то другой символ, он будет проигнорирован программой.

Если была получена точка, считаем, что ввод значения частоты закончен и отправляем прочитанное значение, для проверки, в терминал: **Serial.println("You wrote:"+frequency);**
Затем, с помощью функции **tone()** запускаем ШИМ с заданной частотой на выводе **BUZZER_PIN**.

Обратите внимание, функция **tone()** ожидает, что второй аргумент будет иметь тип **int**. Здесь пригодился новый тип данных **String**, благодаря которому мы сначала посимвольно собрали значение задаваемой частоты в одну строку, а затем преобразовали ее в число типа **int** с помощью метода **String.toInt()**.

Задание для самостоятельного выполнения 10.1

Используя схему, изображенную на Рис. 9.2, напишите программу так, чтобы частота звучания зуммера менялась со 100Гц до 10000Гц с шагом 100Гц, через каждые 100 миллисекунд. После каждого изменения, текущее значение частоты должно передаваться в терминал.

Ответ к заданию 10.1

```
//Номер вывода зуммера
#define BUZZER_PIN 9
//Переменная для хранения частоты
int frequency = 100;

void setup() {
  //Переменная для хранения текущего
  //символа прочитанного из порта
  Serial.begin(9600);
}

void loop() {
  //Проверяем, что частота меньше 10000Гц
  if (frequency < 10000) {
    //Увеличиваем частоту на 100Гц
    frequency = frequency + 100;
  } else {
    //Если частота больше 10000Гц,
```

```
19      //изменяем ее снова на 100Гц
20      frequency = 100;
21    }
22
23    //Выводим значение частоты в терминал
24    Serial.println(frequency);
25
26    //Подаем частоту на зуммер
27    tone(BUZZER_PIN, frequency);
28
29    //Ждем 100мс
30    delay(100);
31  }
```

Урок 11. Работа с энкодером

В этом уроке вы узнаете, что такое энкодер, где он используется и для чего нужен. Во второй части урока мы покажем как работать с энкодером и соберем устройство плавного управления яркостью светодиода.

Теория:

Энкодер — устройство преобразующее угол поворота в электрический сигнал. Вы наверняка уже сталкивались с энкодерами. Чаще всего их используют в музыкальных центрах и аудио системах, в качестве ручки регулировки громкости или тембра.

Мы будем использовать готовый модуль с энкодером изображенный на Рис. 11.1. Помимо земли и питания, он, как и любой другой энкодер, имеет два выхода, их называют каналами. На плате они обозначены как CH1 и CH2 (сокращ. от лат. channel1 и channel2). В других учебниках вы можете встретить другое обозначение — канал A и канал B.

.

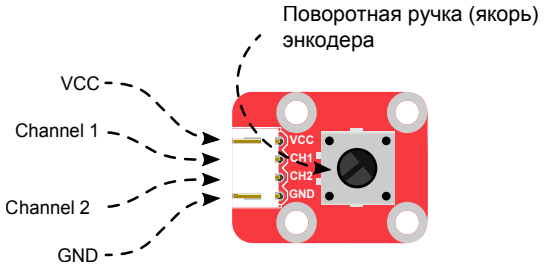


Рис. 11.1. Модуль-Энкодер

Модуль подключается с помощью разъема с четырьмя контактами, которые имеют следующее назначение:
VCC — питание +5V
CH2 — Канал B. Подключается к порту контроллера
CH1 — Канал A. Подключается к порту контроллера
GND — земля (общий)

Вращая ручку энкодера, можно почувствовать характерные щелчки — такты. Чем больше тактов, тем точнее можно определить угол поворота ручки. Точность можно вычислить, разделив 360° на количество тактов. В нашем энкодере 24 такта, значит с его помощью можно определить угол поворота с точностью до 15°.

Двумя основными задачами, которые нужно решить при работе с энкодером, является определение направления вращения ручки и подсчет количества тактов. Чтобы понять, как это делается, давайте разберемся в конструкции энкодера.

На Рис. 11.2. схематично изображена конструкция четырехтактного энкодера. Как видно из рисунка, якорь энкодера соединен с землей (общим проводом), а значит имеет потенциал **LOW**.
Так же в корпусе расположены две группы контактных площадок, изображенные в виде дуг красного (канал A) и синего (канал B) цветов.
Заметьте, что синие и красные контактные площадки смещены друг относительно друга ровно на половину.

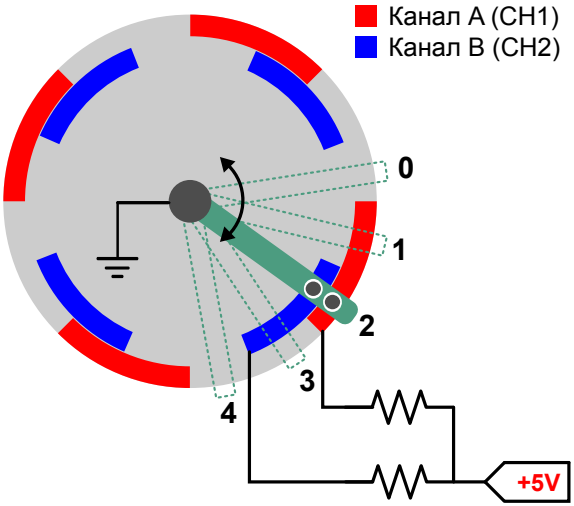


Рис. 11.2. Схематичное изображение конструкции энкодера

Все красные контактные площадки соединены между собой и подключены к шине +5В через подтягивающий резистор. Так же соединены и синие контактные площадки.
Якорь механически соединен с ручкой энкодера и вращается вместе с ней, скользя по контактным площадкам. Когда якорь касается контактной площадки, ее потенциал меняется с **HIGH** на **LOW**. В этом, энкодер похож на кнопку.

Теперь рассмотрим, что произойдет, если мы будем вращать ручку энкодера вместе с якорем. Для большей наглядности, сигналы получаемые на выходах обоих каналов изображены на Рис. 11.3.

Предположим, якорь находится в исходном положении, обозначенном цифрой «0» на Рис. 11.2 и 11.3. На обоих каналах A и B в этот момент установился высокий уровень **HIGH.**

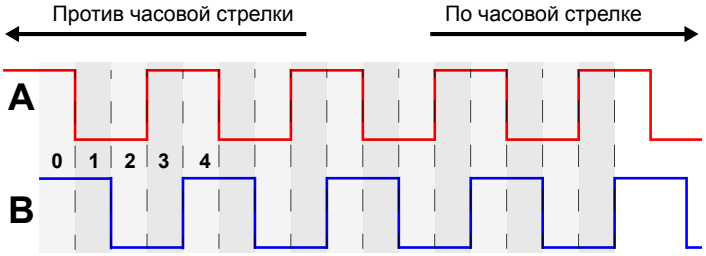


Рис. 11.3. Изменение сигнала на выходах энкодера

Когда ручка вращается по часовой стрелке, соединенный с ней якорь, сначала касается контакта красного цвета, в результате чего, контакты группы A оказываются соединенными с землей и на канале A устанавливается уровень **LOW**. Контакт синего цвета по прежнему имеет потенциал **HIGH**, так как он еще не коснулся якоря. На Рис. 11.2 и 11.3 этот момент обозначен цифрой «1»

Продолжая свое движение, ровно на половине длины красного контакта, якорь «встречается» с контактом синего цвета — точка «2» на Рис. 11.2 и 11.3. Теперь уже на обеих группах контактов установится низкий уровень **LOW**.

При дальнейшем вращении ручки, якорь соскальзывает с красного контакта и продолжает скользить только по синему — точка «3» на Рис. 11.2 и 11.3. В этот момент в канале A снова устанавливается высокий уровень **HIGH**, а в канале B все еще низкий уровень **LOW**.
Двигаясь дальше, якорь завершает цикл, попадая на изолированную площадку — точка «4» на Рис. 11.2 и 11.3. В этом положении на обоих каналах снова устанавливается высокий уровень **HIGH**.
Очевидно, что при вращении против часовой стрелки, те же самые события происходят в обратном порядке.

Снова обратимся к графику на Рис. 11.3. Если посмотреть на него внимательно, можно заметить одну закономерность. При вращении якоря по часовой стрелке, в момент, когда потенциал канала

А меняется с **HIGH** на **LOW**, канал В всегда имеет высокий потенциал — **HIGH**. А при вращении против часовой стрелки, в момент, когда потенциал канала А меняется с **HIGH** на **LOW**, канал В всегда имеет низкий потенциал — **LOW**. Это свойство позволяет легко определить направление вращения якоря.

Практика:

Рассмотрим способ подключение энкодера с помощью прерываний по таймеру.

Задача 11.1. Светильник с регулятором яркости

Соберем макет согласно схеме, изображенной на Рис. 11.4. и напишем программу управления яркостью светодиода при помощи энкодера.

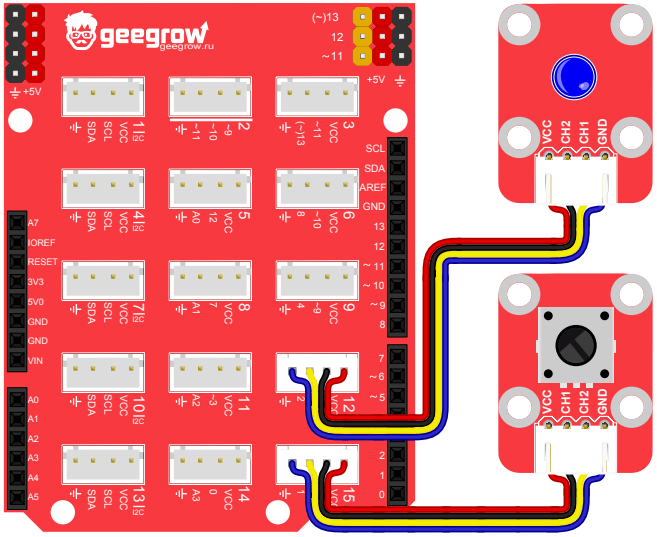


Рис. 11.4. Схема макета для задачи 11.1

Следует заметить, что энкодер является прибором со скользящими контактами. Из-за этого эффект дребезга проявляется значительно сильнее чем в случае с кнопкой. По этой причине использование внешних прерываний может приводить к неправильной обработке данных и регистрации ложных срабатываний.

По этой причине, для отслеживания изменения уровня сигнала в канале А (CH1), мы будем использовать прерывания по таймеру, делая измерения 1 раз в 100мс. Если будет зафиксировано изменение состояния канала А, необходимо будет проверить состояние канала В и соответствующим образом изменить яркость светодиода.

```
1  //Подключаем библиотеку TimerOne
2  #include <TimerOne.h>
3
4  //Используем порт 6, потому что библиотека
5  //TimerOne мешает работе ШИМ на портах 1,2,
6  //9,10
7  #define LED_PIN 6
8  #define ENC_A 1
9  #define ENC_B 5
10 //Предыдущее состояние канала А
11 volatile int aPrevState = -1;
12 //Текущее значение ШИМ
13 volatile int value = 120;
14
15 void setup() {
16   //Инициализация входов/выходов
17   pinMode(LED_PIN, OUTPUT);
18   pinMode(ENC_A, INPUT);
19   pinMode(ENC_B, INPUT);
20
21   //Переполнение таймера в микросекундах
22   //1000 микросек. = 1 миллисек.
23   Timer1.initialize(1000);
24   //Задаем функцию обработки прерываний
25   //таймера
26   Timer1.attachInterrupt(timerHandler);
```

```
}

void loop() {
  //Меняем яркость светодиода
  analogWrite(LED_PIN, value);
}

//Сюда попадаем раз в 1мс
void timerHandler(void) {
  //Запоминаем текущее состояние канала
  //А во временную переменную
  int aCurrentState = digitalRead(ENC_A);

  //Проверяем направление вращения
  if (
      aPrevState > -1
      && aPrevState != aCurrentState
      && aCurrentState == LOW
  ) {
    //Проверяем состояние порта B чтобы
    //определить направление вращения
    if (digitalRead(ENC_B) == HIGH) {
      //Сюда попадаем если по часовой
      //стрелке
      if (value <= 240) {
        value = value+15;
      }
    } else {
      //Сюда попадаем если против
      //часовой стрелки
      if (value >= 15) {
        value = value-15;
      }
    }
  }

  //Сохраняем текущее значение порта A
  aPrevState = aCurrentState;
}
```

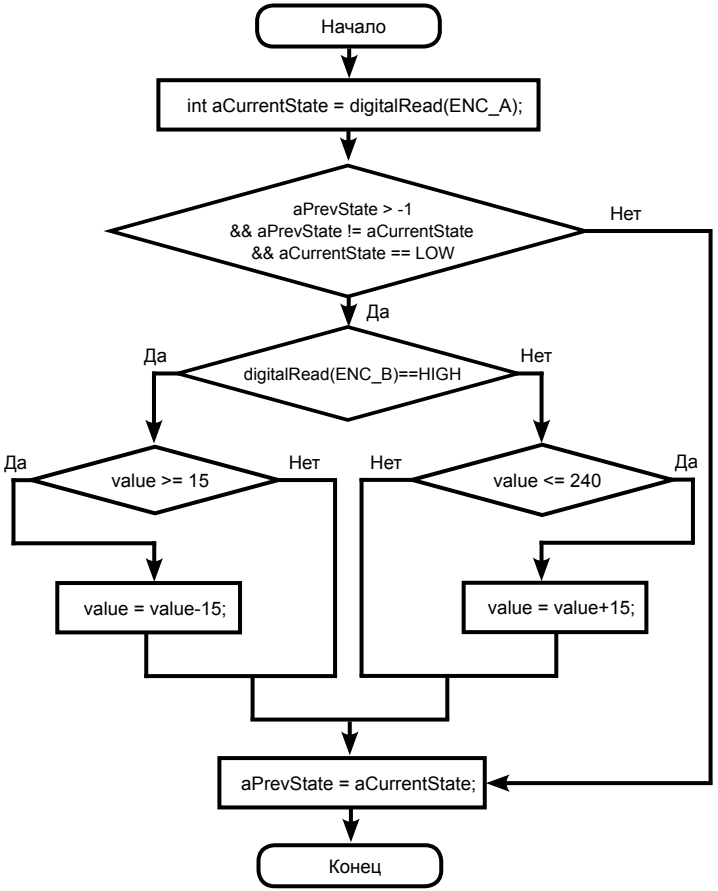


Рис. 11.5. Блок-схема описывающая работу функции timerHandler()

Блок-схема, приведенная на Рис. 11.5 подробно описывает рабо-

ту функции `timerHandler()`.

В самом начале, мы определили переменную **aPrevState** в которой будет храниться состояние канала А энкодера из предыдущей итерации. Это позволит определить, что состояние изменилось. Но, сразу после старта программы, предыдущее состояние канала А не известно. Чтобы избежать неправильного срабатывания, зададим этой переменной значение «-1».
Так же определим переменную **value** для хранения текущего значения ШИМ.

Таймер выставлен на переполнение раз в миллисекунду, а в качестве обработчика задана функция **timerHandler()**.

В первый раз в функции **timerHandler()** программа определит, что переменная **aPrevState** равна «-1» и запишет в нее текущее значение порта А. Затем, каждую миллисекунду переменная **aPrevState** будет сравниваться с текущим состоянием порта А. Когда оно изменится, как и в предыдущем примере, определим направление вращения ручки энкодера и перезапишем значение ШИМ в переменной **value**.

Рассмотренный способ работы с энкодером обладает лишь одним недостатком — в функции **timerHandler()** достаточно много операций. Но учитывая, что период срабатывания таймера достаточно велик, это не создаст проблем.

Задание для самостоятельного выполнения 11.1

Маячок управляемый энкодером. Используя схему из задачи 11.2 напишите программу так, чтобы светодиод мигал с частотой 12 Гц. При повороте энкодера по часовой стрелке, скорость мигания должна увеличиваться до 24 Гц с шагом 1 Гц. При повороте против часовой стрелки частота должна уменьшаться до 1 Гц с таким же шагом.

Будьте осторожны, длительное наблюдение за пульсирующим источником света может вызвать головную боль и приступы тошноты.

Ответ к заданию 11.1

```
1  //Подключаем библиотеку TimerOne
2  #include <TimerOne.h>
3
4  //Используем порт 6, потому что библиотека
5  //TimerOne мешает работе ШИМ на портах 1,2,
6  //9,10
7  #define LED_PIN 6
8  #define ENC_A 1
9  #define ENC_B 5
10 //Предыдущее состояние канала А
11 volatile int aPrevState = -1;
12 //Текущее значение ШИМ
13 volatile int value = 120;
14 //Частота мигания светодиода
15 volatile int frequency = 12;
16
17 void setup() {
18   //Инициализация входов/выходов
19   pinMode(LED_PIN, OUTPUT);
20   pinMode(ENC_A, INPUT);
21   pinMode(ENC_B, INPUT);
22
23   //Переполнение таймера в микросекундах
24   //1000 микросек. = 1 миллисек.
25   Timer1.initialize(1000);
26   //Задаем функцию обработки прерываний
27   //таймера
28   Timer1.attachInterrupt(timerHandler);
29 }
30
31 void loop() {
32     //Меняем состояние светодиода
33   digitalWrite(LED_PIN, !digitalRead(LED_PIN));
34   //Одно мигание происходит за два цикла:
35   //выкл и вкл. Поэтому в знаменателе частоту
36   //умножаем на два
37   float delayValue = 1000/(frequency*2);
```

```
  //Ждем сколько нужно
  delay(delayValue);
}

//Сюда попадаем раз в 1мс
void timerHandler(void) {
  //Запоминаем текущее состояние канала
  //А во временную переменную
  int aCurrentState = digitalRead(ENC_A);

  //Проверяем направление вращения
  if (
      aPrevState > -1
      && aPrevState != aCurrentState
      && aCurrentState == LOW
  ) {
    //Проверяем состояние порта B чтобы
    //определить направление вращения
    if (digitalRead(ENC_B) == HIGH) {
      //Сюда попадаем если по часовой
      //стрелке
      if (frequency < 24) {
        frequency = frequency+1;
      }
    } else {
      //Сюда попадаем если против
      //часовой стрелки
      if (frequency > 1) {
        frequency = frequency-1;
      }
    }
  }

  //Сохраняем текущее значение порта A
  aPrevState = aCurrentState;
}
```

Урок 12. Сервопривод

В этом уроке мы познакомимся с сервоприводом. Узнаем о том какие сервоприводы бывают, в чем их особенности и для чего они применяются. И, конечно же, научимся управлять сервоприводом с помощью контроллера.

Теория:

Сервопривод или *серводвигатель* — это мотор-редуктор с обратной связью, позволяющий точно управлять положением его вала. Иначе говоря, это «точный исполнитель», которому можно сказать на какой угол повернут вал, и он своими силами будет стремиться поддерживать заданное положение, вне зависимости от внешнего воздействия.

К сервоприводам относится много различных типов устройств, но мы в данном уроке будем рассматривать именно электрический сервопривод. Его внешний вид представлен на Рис. 12.1

Рис. 12.1. Внешний вид сервопривода

Конструктивно, электрический сервопривод состоит из следующих блоков (см. Рис. 12.2):
1. Мотор
2. Редуктор
3. Потенциометр
4. Управляющая плата

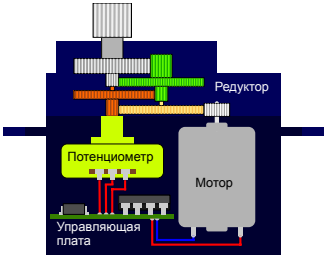


Рис. 12.2. Внешний вид сервопривода

Работает серводвигатель следующим образом. Внешний сигнал поступает на управляющую плату, которая контролирует скорость вращения мотора, преобразуя, с его помощью, электрическую энергию в механическую и создавая крутящий момент.

Как правило, электродвигатели имеют достаточно большую скорость вращения и относительно небольшой крутящий момент. Чтобы снизить скорость и увеличить крутящий момент, используют редуктор.

Для контроля текущего положения, якорь сервопривода соединен с потенциометром, сопротивление которого зависит от угла поворота. Отклонение якоря от заданного положения, приводит к изменению сопротивления потенциометра. Управляющая плата следит за его сопротивлением и в случае изменения, включает электромотор, возвращая якорь в исходное состояние.

Обычно, сервоприводы могут поворачивать якорь, или как его еще называют — качалку, от 0° до 180°, как показано на Рис. 12.3.

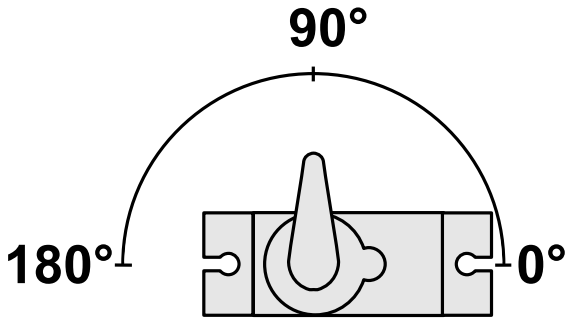


Рис. 12.3. Ориентация сервопривода

Виды сервоприводов

Условно, сервоприводы делятся на несколько типов. Прежде всего по величине и производимому усилию. А так же по конструкции редуктора и материалу из которого изготовлены его шестерни.

Обычно в более мощных сервоприводах используются подшипники качения и металлические шестерни. Если крутящий момент не имеет определяющего значения, шестерни делаются из нейлона. Этот материал выдерживает меньшее усилие, но значительно медленнее изнашивается. Таким образом, если не превышать предельные параметры, такой сервопривод прослужит дольше.

Еще одним важным отличием является тип управляющего блока, который может быть цифровым и аналоговым. Как следует из названия, цифровые сервоприводы управляются цифровым сигналом, что позволяет более точно задавать значение угла поворота. Минусом этого типа является высокая стоимость. По этой причине в хобби-технике обычно применяются сервоприводы с аналоговым управлением, которые мы и рассматриваем.

Управление сервоприводами

Для управления сервоприводом используется разновидность ШИМ модуляции, называемая в некоторых источниках PDM (от лат. Pulse Duration Modulation).

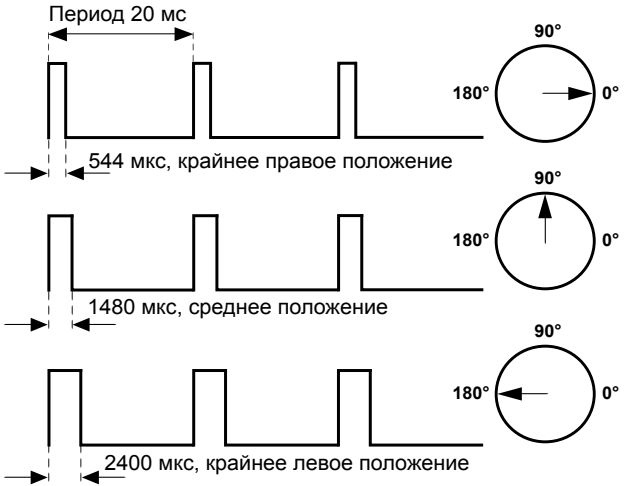


Рис. 12.4. Ориентация сервопривода

Отличие заключается лишь в том, что частота сигнала должна быть постоянной (50Гц), а управление сервоприводом осуществляется с помощью изменения длительности импульсов. Таким образом, длительность импульсов кодирует угол поворота. Чем больше длительность, тем больше угол поворота и наоборот, чем меньше длительность, тем меньше угол поворота. Для наглядности процесс кодирования импульсов изображен на Рис. 12.4.

Крайнее правое положение якоря соответствует длительности импульса 544 мкс, среднее положение 1480 мкс, а крайнее левое положение 2400 мкс. Импульсы следуют друг за другом с интервалом 20 мс, что соответствует частоте 50 Гц.

Стоит заметить, что даже параметры двух сервоприводов одного и того же производителя могут быть различны. То есть при подаче одного и того же управляющего импульса, угол поворота якоря будет немного отличаться. Поэтому, если важна точность, то обычно подбирают значения отклонения под каждый отдельный сервопривод. Кроме того в положениях близких к крайним (0-10° и 170-180°) сервоприводы могут работать некорректно, издавать рычащий звук и быстро вибрировать. Такие режимы работы могут стать причиной быстрого выхода из строя и их надо избегать. Так же стоит избегать чрезмерных механических нагрузок на якорь.

Подключается сервопривод с помощью трех проводов, которые, как правило, имеют идентичную цветовую маркировку, вне зависимости от производителя. Иногда цвет может немного отличаться, но принцип маркировки остается неизменным:

Земля — коричневый/черный
Питание +5 вольт — красный
Сигнальный — оранжевый/желтый/белый

Управлять сервоприводом можно, подключив сигнальный провод напрямую к выводу микроконтроллера. При этом можно не переживать об ограничении тока, потому что сигнальный провод используется сервоприводом лишь для получения управляющих импульсов. Питающие токи текут по черному и красному проводам. Подключать сервопривод к контроллеру DaVinci удобнее с использованием шилда, на котором предусмотрены три пары контактов в верхней части платы.

Практика:

Специально для взаимодействия с сервоприводами существует библиотека **Servo**, которая предоставляет богатый функционал, с лихвой удовлетворяющий нашим запросам. Библиотека позволяет одновременно подключать и использовать до 12-ти сервоприводов, но нам в ходе экспериментов потребуется лишь один.

Чтобы получить доступ к функционалу библиотеки, необходимо подключить заголовочный файл:

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

и определить специальную переменную типа **Servo**:

```
Servo myservo;
```

Для подключения нового сервопривода используется команда **Servo.attach(pin)** или **Servo.attach(pin, min, max)**, где:
pin — номер вывода к которому присоединен сервопривод;
min и **max** — минимальная и максимальная длина импульса соответствующая углу от 0° до 180° соответственно.

Servo.write(angle) — поворачивает якорь на угол переданный в параметре **angle**.

Servo.writeMicroseconds(value) — используется если необходимо вручную сформировать импульс определенной длины, и принимает на вход значение в микросекундах.

Servo.read() — определяет угол отклонения сервопривода и возвращает значение в градусах от 0 до 180.

Servo.attached() — проверяет подключен ли сервопривод к определенному пину и возвращает **TRUE** или **FALSE**.

Servo.detach() — отключает сервопривод.

Задача 12.1. Регулировщик

Впервые подключая сервопривод, надо определить его крайние положения, чтобы корректно установить качалку. Давайте напишем программу, которая переключает сервопривод сначала в 0°, затем в 90° и 180°, делая задержку между переключениями 1 сек. Подключим сервопривод к пину номер 11, как на Рис. 12.5.

Будьте внимательны при подключении, проверьте соответствие сигнального пина и пинов питания. Если перепутать сигнальный пин с одним из пинов питания, то это может привести к выходу контроллера из строя.

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

void setup(){
  myservo.attach(11);
}

void loop(){
  //Устанавливаем угол 0 градусов
  myservo.write(0);
  delay(1000);

  //Устанавливаем угол 90 градусов
  myservo.write(90);
  delay(1000);

  //Устанавливаем угол 180 градусов
  myservo.write(180);
  delay(1000);
}
```

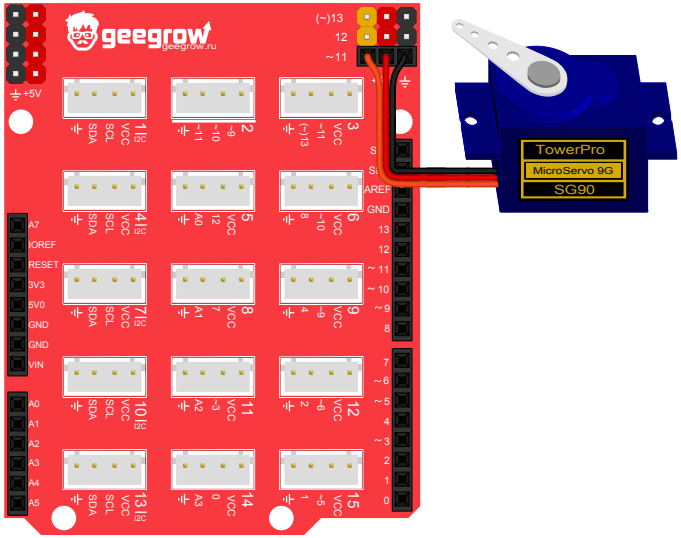


Рис. 12.5. Схема макета для задачи 12.1

После запуска программы понаблюдайте за работой серводвигателя. Часто он может вибрировать в крайних положениях 0° и 180° — это говорит о выходе за границы допустимых углов работы. Если такое происходит, нужно просто подобрать правильную длительность сигнала управляющего сигнала для угла 0° и для угла 180°. Для этого, достаточно явно указать минимальную и максимальную длительность при подключении сервопривода, например так: **`myservo.attach(11, 544, 2400);`**
Попробуйте изменять значения длительности до тех пор, пока не убедитесь, что сервопривод работает штатно в крайних положениях. Всегда используйте подобранные значения при подключении именно этого сервопривода.

Задача 12.2. Автомобильный дворник

Написать программу управления сервоприводом так, чтобы качалка плавно поворачивалась на угол от 0° до 180°, а затем обратно, как автомобильный дворник.

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

void setup(){
  myservo.attach(11, 700, 2650);
}

void loop(){
  //Считаем от 0 до 180 градусов
  for(int angle=0; angle<=180; angle++) {
    //Устанавливаем требуемый угол
    myservo.write(angle);
    //Ждем 20мс
    delay(20);
  }

  //Возвращаемся в 0 градусов
  myservo.write(0);
  //Ждем 100мс
  delay(100);
}
```

При инициализации сервопривода мы использовали подобранные для нашего сервопривода значения длительности сигнала. У вас могут получиться немного другие значения.

В основной части программы появилась новая языковая конструкция — оператор цикла **`for()`**. Операторы циклов являются очень важной частью любого языка программирования и, что очень удобно, почти в каждом языке они реализуются одинаково.

Цикл — это блок кода, который выполняется от начала до конца снова и снова заданное количество раз либо пока не будет нарушено условие выполнения цикла.
Обобщенная запись оператора **`for`** выглядит так:

```
for (инициализация; условие; изменение){
  блок операторов выполняемых в цикле
}
```

Рассмотрим простой пример:

```
for (int i = 0; i < 10; i++) {
   блок операторов выполняемых в  цикле
}
```

Разобьем приведенный код на отдельные команды:
`int  i=0;` — инициализация переменной цикла. Переменная цикла, это переменная изменяющая свое значение от начального до определенного в условии завершения цикла.
`i<10;` — условие завершения цикла. Цикл будет повторяться, пока выполняется условие **`i < 10`**. Когда значение **`i`** станет равно **`10`** или превысит его, цикл закончится.
`i++` — выражение описывающее изменение переменной цикла. Запись **`i++`** означает, что в конце каждого цикла переменная **`i`** будет увеличиваться на единицу. Вспомним, что запись **`i++`** равносильна записи **`i = i+1`**.
Изменение переменной цикла может быть описано любым другим выражением. Например можно записать **`i--`**, что равносильно **`i=i-1`**.
Отдельно стоит упомянуть о том, что переменная цикла объявленная в самом цикле, видима только внутри этого цикла. Как только

цикл завершается, переменная цикла исчезает из памяти.
Для того чтобы переменная цикла была видима за его пределами, она должна быть объявлена заранее, либо как глобальная, либо как локальная переменная.

Вернемся к нашей программе. Теперь, зная как работает цикл, легко увидеть, что в первом блоке **for** угол последовательно возрастает от **0°** до **180°** с шагом **1°**. Новое значение угла передается сервоприводу.
Так как контроллер способен выполнить цикл гораздо быстрее, чем сервопривод повернется на заданный угол, искусственно была внесена задержка 20мс в конце каждой итерации.
В конце качалка возвращается в 0° и сервопривод ждет 100мс.

Задание для самостоятельного выполнения 12.1

Измените программу из задачи 12.2 так, чтобы сервопривод возвращался в исходное положение так же плавно. Используйте второй цикл, не забудьте про задержку 20мс между итерациями.

Ответ к заданию 12.1

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

void setup(){
  myservo.attach(11, 700, 2650);
}

void loop(){
  //Считаем от 0 до 180 градусов
  for(int angle=0; angle<=180; angle++) {
    myservo.write(angle);
    delay(20);
  }

  //Считаем от 180 до 0 градусов
  for(int angle=180; angle>=0; angle--) {
    myservo.write(angle);
    delay(20);
  }
}
```

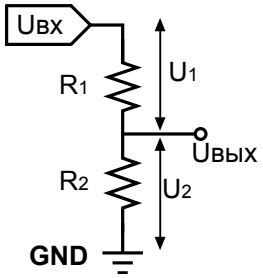
Урок 13. Потенциометр

В этом уроке мы узнаем, что такое делитель напряжения и как его рассчитать, а так же познакомимся с потенциометром, рассмотрим его конструкцию и способ подключения к контроллеру.

Теория:

Одной из самых важных схем в электронике является делитель напряжения. Для чего же он нужен?
Предположим, у вас есть источник напряжения $+5В$, а вам хотелось бы получить нестандартное значение $3.66В$. Проще всего это сделать с помощью делителя напряжения.

Классическая схема делителя напряжения, представляет собой два соединенных последовательно резистора. С одной стороны они подключены к шине питания, а с другой стороны к земле. Место соединения резисторов — это выход схемы, откуда мы снимаем интересующее нас напряжение $U2$.
Чтобы лучше понять, как работает делитель напряжения, давайте рассмотрим следующую схему:



$$\frac{R1}{R2} = \frac{U1}{U2}; \quad (1)$$

$$Uвх = U1 + U2; \quad (2)$$

Из формул 1 и 2 получаем:

$$U2 = Uвх \frac{R2}{R1 + R2}; \quad (3)$$

Пользуясь формулой (3), легко рассчитать выходное напряжение, если даны номиналы резисторов в Омах.
Если же требуется, наоборот, рассчитать номиналы резисторов, чтобы получить заданное напряжение, поступают следующим образом. Сначала выбирают номинал резистора *R1*. Затем, по следующей формуле рассчитывают номинал резистора *R2*:

$$R2 = \frac{U2 * R1}{Uвх - U2}; \quad (4)$$

Подбирать номиналы резисторов нужно так, чтобы ток протекающий через суммарное сопротивление *R1+R2* оставался небольшим и не превышал *10мА*.
Например, необходимо рассчитать делитель напряжения, позволяющий получить напряжение *3В* на выходе. Пусть *Uвх=5В*.
Сначала подберем резистор *R1*, воспользовавшись Законом Ома:

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

Подставив в формулу напряжение *5В* и ток *0.01А (10мА)*, рассчитаем номинал резистора: *R1 = 5/0.01 = 500 Ом*. Для удобства, выберем номинал *1кОм*. Затем рассчитаем номинал *R2* :

$$Uрез=5В;\ Uвых=3В;\ R1=1000Ом;$$

$$Тогда:\ R = \frac{3В * 1000Ом}{5В - 3В} = 1500Ом;$$

Разобравшись с делителем напряжения, можно перейти к рассмотрению потенциометра.

Потенциометр — это регулируемый резистор с подвижным контактом, который может перемещаться по телу резистора.

Внутреннее устройство потенциометра и его обозначение на схеме приведены на рисунке 13.1, взглянув на который, можно заметить, что он представляет собой регулируемый делитель напряжения.
Сектор окружности между выводами 1 и 3 выполнен из резистивного материала. При вращении ротора, ползунок, соединенный с выводом 2, скользит по поверхности полосы из резистивного материала. Таким образом, если вывод 1 потенциометра подключить к источнику питания Uвх, а вывод 3 к земле, то напряжение на выводе 2 будет определяться положением ротора.
Это свойство потенциометра, позволяет использовать его в качестве датчика угла поворота. Чтобы узнать на какой угол повернут ротор потенциометра, достаточно измерить напряжение на выводе 2 при помощи Аналого-цифрового Преобразователя или АЦП.

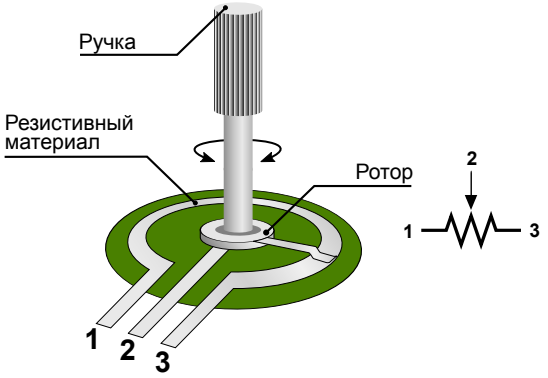


Рис. 13.1. Модель потенциометра

АЦП — это устройство преобразующее физическую величину, напряжение, в цифровой вид.

АЦП может принимать на вход сигнал, уровень которого изменяется в диапазоне от 0В до напряжения питания Vcc. На выходе АЦП выдает число от 0 до 1024. Что означает этот результат и, как его интерпретировать?
Все очень просто, АЦП разбивает весь диапазон входных напряжений от 0В до 5В на 1024 отсчета. Каждый отсчет равен 5В/1024=0.00488В. Для удобства можно округлить до 5мВ. Если, на выходе АЦП получили, например, число 250, значит измеряемое напряжение равно 250*5мВ =1.25В.

Практика

Контроллер DaVinci, имеет 12 портов АЦП, но чаще всего используются порты с A0 по A5. Чтобы измерить уровень сигнала, необходимо подать его на один из этих выводов.
В экспериментах мы будем подключать потенциометр к контроллеру не напрямую, а через шилд сопряжения, так же как мы поступали в предыдущих уроках. Выводы контроллера с A0 по A3 продублированы на шилде и выведены в разъемах 5, 8, 11, и 14.
Вместо отдельного потенциометра будем использовать готовый модуль с доработанной схемой включения Рис. 13.2 .

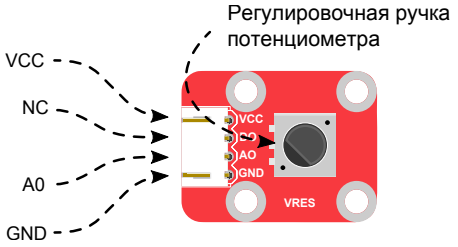


Рис. 13.2. Распиновка модуля с потенциометром

Пины модуля с потенциометром имеют следующее назначение:
VCC — питание +5V
D0 — не используется
A0 — подключается к аналоговому входу контроллера (A0...A5)
GND — земля (общий)

Схема и внешний вид модуля представлены на Рис. 13.3. Конденсатор C1 добавлен для борьбы с эффектом дребезга. Он сглаживает скачки напряжения во время перемещения бегунка по поверхности полосы из резистивного материала.

Защитный резистор R2 не позволит контроллеру выйти из строя, если вы случайно инициализируете порт контроллера, как выход и повернете ручку потенциометра в крайнее положение, при котором его сопротивление будет равно нулю. Номинал резистора R2 равен 470 Ом и он ограничивает ток до 10мА.

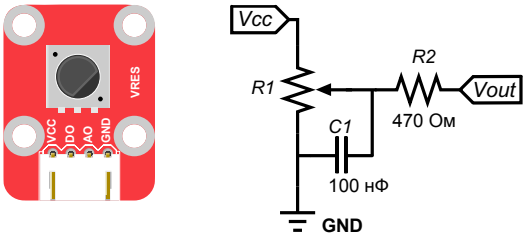


Рис. 13.3. Схема и внешний вид модуля потенциометра

Задача 13.1. Диммер

Соберем макет согласно Рис. 13.4 и напишем программу плавной регулировки яркости светодиода с помощью потенциометра.

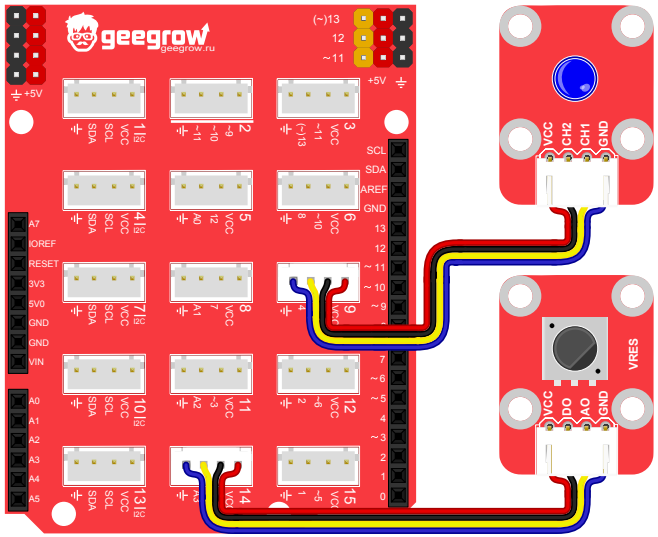


Рис. 13.4. Схема макета для задачи 13.1

```
#define LED_PIN 9
#define VAR_RES_PIN A3

//Переменная хранящая результат полученный
//от АЦП. Изменяется от 0 до 1024
int value = 0;
//Яркость светодиода для ШИМ от 0 до 255
int brightness = 0;

void setup(){
  //Настройка входов/выходов
  pinMode(LED_PIN, OUTPUT);
  pinMode(VAR_RES_PIN, INPUT);
}

void loop(){
  //Читаем результат измерения. АЦП
  //возвращает число от 0 до 1024
  value = analogRead(VAR_RES_PIN);

  //Для использования ШИМ нам
  //нужно число в диапазоне от 0 до 255.
  //Поэтому делим value на 4
  brightness = value/4;
  //Подаем ШИМ на светодиод
  analogWrite(LED_PIN, brightness);
  //Ждем 200 миллисекунд
  delay(200);
}
```

Подробно разберем работу программы.

В заголовке файла мы определили две переменные. Первая из них **value**, предназначена для хранения результата полученного от АЦП. В ней может содержаться число от 0 до 1024. Во второй переменной **brightness** мы будем хранить число для задания ШИМ сигнала, который и будет определять яркость светодиода.

В функции **setup()** настраиваем порты контроллера. Останавливаться на этом не будем, заметим только, что порты АЦП настраиваются так же как и остальные.

В начале главного цикла **loop()** читаем результат работы АЦП с помощью функции **analogRead()** и помещаем в переменную **value** для дальнейшей обработки.

Функция **analogRead()** принимает в качестве аргумента номер вывода и возвращает значение от 0 до 1024, соответствующее напряжению поданному на этот вывод. Почему только до 1024? Потому что АЦП имеет разрядность 10 бит. Это значит что результат хранится в регистре емкостью 10 бит. Максимальное число, которое может храниться в таком регистре $2^{10} = 1024$.

Как вы могли узнать из предыдущих уроков, функция **analogRead()**, формирующая ШИМ сигнал, ожидает в качестве аргумента число от 0 до 255. Это в 4 раза меньше чем максимальное значение переменной value. Значит нам надо всего лишь разделить value на 4, пропорция при этом сохранится и функция **analogWrite()** будет работать корректно.

В дальнейшем цикл повторяется снова и снова с интервалом 200мс.

Таким образом используя потенциометр мы решили задачу схожую с той, что решали с помощью энкодера. Очевидно, что у каждого решения есть свои плюсы и минусы. Энкодер дает возможность достигать большей точности при увеличении количества оборотов, но его использование вынуждает нас задействовать внешнее прерывание или таймер, которые могли бы понадобиться нам для чего-то еще. С другой стороны, потенциометр на много проще в подключении. Программа с ним получается более компактной. Но потенциометр, как правило, имеет только один оборот, что не всегда удобно и сказывается на точности. Кроме того он быстрее изнашивается, причем часто не равномерно.

Задание для самостоятельного выполнения 13.1

В схеме из задачи 13.1, замените светодиод на зуммер. Напишите программу так, чтобы частота плавно изменялась в диапазоне от 100 до 1000Гц, в зависимости от угла поворота ручки потенциометра. Чтобы звук не дрожал, в главном цикле используйте задержку 100мс.

Ответ к заданию 13.1

```
#define BUZZER_PIN 9
#define VAR_RES_PIN A3

//Переменная хранящая результат полученный
//от АЦП. Изменяется от 0 до 1024
int value = 0;
//Частота от 100Гц до 10000Гц
int frequency = 0;

void setup(){
  pinMode(BUZZER_PIN, OUTPUT);
  pinMode(VAR_RES_PIN, INPUT);
}

void loop(){
  //Читаем результат измерения. АЦП
  value = analogRead(VAR_RES_PIN);

  //Ширина диапазона изменения частоты
  //10000Гц - 100Гц = 9900Гц. А АЦП выдает
  //заначение от 0 до 1024. То есть одному
  //отсчету АЦП соответствует изменение
  //частоты на 9900/1024 = 9.66Гц. Составив
  //пропорцию получим выражение для
  //нахождения частоты. Кроме того, перемен-
  //ная frequency имеет тип int, поэтому
  //дробное значение частоты будет
  //прабразовано к целому.
  frequency =  100 + (9900/1024)*value;
  //Активируем tone()
  tone(BUZZER_PIN, frequency);
  //Ждем 100 миллисекунд
  delay(100);
}
```

Урок 14. Аналоговые датчики: фоторезистор

В прошлом уроке, вы познакомились с простейшим механическим датчиком, работающим по принципу делителя напряжения. Но по такому же принципу работает множество других датчиков, в том числе и бесконтактных, например фоторезистор, с помощью которого можно собирать устройства реагирующие на свет.

Теория:

Фоторезистор — электронный элемент, изменяющий свое сопротивление в зависимости от количества падающего на него света.

В экспериментах будем использовать модуль на основе фоторезистора VT90N. Его сопротивление в полной темноте, в зависимости от модификации, может изменяться в диапазоне 12...36кОм, а в хорошо освещенной комнате сопротивление снижается до 1..3кОм. Внешний вид фоторезистора и его обозначение на схеме показано на Рис. 14.1.

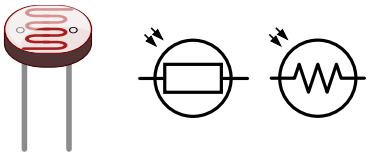


Рис. 14.1. Внешний вид фоторезистора и его обозначение на схеме

Чтобы подключить фоторезистор к контроллеру, его нужно включить последовательно с резистором по схеме делителя напряжения Рис. 14.2. Номинал резистора для верхнего плеча выбран близким к сопротивлению фоторезистора, 10кОм. Дополнительно, для уменьшения шума АЦП, параллельно фоторезистору включается конденсатор 0.1нФ. Приведенная схема полностью повторяет схему модуля, который мы будем использовать в экспериментах.

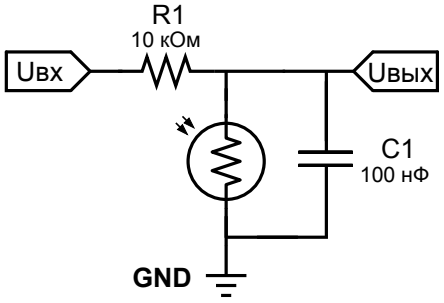


Рис. 14.2. Схема включения фоторезистора

Практика

Как и все прочие аналоговые модули, используемый в наборе модуль с фоторезистором можно подключать к разъемам 5, 8, 11 или 14, на которые выведены порты АЦП. Распиновка модуля показана на Рис. 14.3.

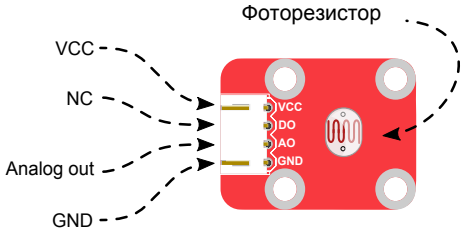


Рис. 14.3. Распиновка и внешний вид модуля с фоторезистором

VCC — питание +5V
D0 — не используется
A0 — подключается к аналоговому входу контроллера (A0...A5)
GND — земля (общий)

В этом уроке мы спроектируем и соберем с использованием фотодиода и зуммера настоящий музыкальный инструмент — Терменвокс.

Задача 14.1. Терменвокс

С тех пор, как Терменвокс был изобретен нашим соотечественником, Львом Сергеевичем Терменом в далеком 1920-ом году, он получил широкую известность во всем мире и его успели собрать тысячи человек во множестве различных вариантов.

В общем случае для сборки Терменвокса нужен бесконтактный сенсор, показания которого используются для изменения частоты звучания динамика. В оригинальном инструменте, построенном Львом Сергеевичем, в качестве сенсора использовался колебательный контур с емкостью. Величина емкости изменялась, если поднести руку, в результате чего менялась и частота звучания инструмента.

В нашем эксперименте, вместо колебательного контура, в качестве сенсора будет использован фоторезистор. Поднося руку ближе или убирая ее дальше от фоторезистора, можно регулировать количество падающего на него света. В свою очередь, изменение степени освещенности фоторезистора приведет к изменению сопротивления, которое будет зафиксировано АЦП. Данные полученные с АЦП используются для управления частотой звучания зуммера.

Для того, чтобы все работало как задумано, необходимо будет откалибровать фоторезистор, замерив значение измеренное АЦП в момент, когда рука находится в ближней и дальней точке.

Соберем макет согласно Рис. 14.4 и напишем программу для калибровки освещенности фоторезистора. Программа будет выводить данные, полученные от АЦП в терминал.

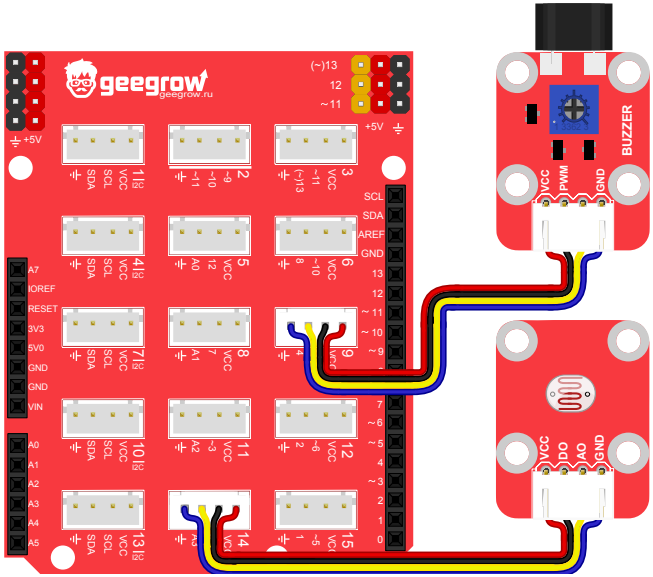


Рис. 14.4. Схема макета для задачи 14.1

```
#define PROTORESISTOR_PIN A3
//Переменная для хранения значения АЦП
int value = 0;

void setup(){
  pinMode(PROTORESISTOR_PIN, INPUT);
  //Открытие соединения на скорости 9600
  Serial.begin(9600);
}

void loop(){
  //Читаем результат полученный от АЦП
  value = analogRead(PROTORESISTOR_PIN);
  //Отправляем данные в терминал.
  Serial.println(value);
  delay(200);
}
```

Загрузите программу в контроллер и, открыв терминал, проведите калибровку, заслоняя фоторезистор рукой от источника света. Наблюдая за результатами измерений можно заметить, что значение получаемое от АЦП максимально в нижней точке (у нас получилось примерно 890). По мере увеличения расстояния между фоторезистором и рукой, значение АЦП будет уменьшаться, пока рука не перестанет влиять на освещенность фоторезистора. Очевидно, что минимальное значение АЦП так же зависит от уровня освещенности помещения и распределения источников света. У нас минимальное значение АЦП получилось равным 400.

Теперь зная диапазон изменения данных от фоторезистора, напишем программу так, чтобы частота звучания зуммера изменялась от 3000Гц до 7000Гц. Изменение частоты должно быть пропорционально значению АЦП, характеризующему сопротивление фоторезистора.

```
#define PROTORESISTOR_PIN A3
#define BUZZER_PIN 9

#define VALUE_MIN 400
#define VALUE_MAX 890
#define FREQ_MIN 3000
#define FREQ_MAX 7000

int value = 0;
//Частота звука от 3000 до 7000
int frequency = 3000;

void setup(){
  pinMode(PROTORESISTOR_PIN, INPUT);
  pinMode(BUZZER_PIN, OUTPUT);
}

void loop(){
```

```
  //Читаем результат измерения. АЦП
  value = analogRead(PROTORESISTOR_PIN);
  frequency = map(
    value,
    VALUE_MIN, VALUE_MAX,
    FREQ_MIN, FREQ_MAX
  );
  //Меняем частоту звучания зуммера
  tone(BUZZER_PIN, frequency);
  delay(50);
}
```

Подробно разберем работу программы.
В начале были определены границы диапазона освещенности:
#define VALUE_MIN 400
#define VALUE_MAX 890
Затем, границы диапазона изменения частоты:
#define FREQ_MIN 3000
#define FREQ_MAX 7000
В функции **setup()** настраиваем порты контроллера. В функции **loop()** читаем значение АЦП в переменную **value**. А для вычисления значения частоты, соответствующего значению полученному от АЦП, используем новую функцию **map()**.

Функция **map(value, from_min, from_max, to_min, to_max)** — переносит значение **value** из диапазона значений **(from_min, from_max)** в диапазон **(to_min, to_max)**, сохраняя пропорции.
Аргументы функции:
value — значение для переноса
from_min — нижняя граница текущего диапазона
from_max — верхняя граница текущего диапазона
to_min — нижняя граница нового диапазона
to_max — верхняя граница нового диапазона
В нашем случае, переносимое значение **value**, это значение АЦП, которое изменяется в диапазоне от 400 до 890. Целью является перенос **value** в область частот звучания зуммера. То есть вычисление значения частоты пропорционального изменяющегося в диапазоне от 3000Гц до 7000Гц. Для наглядности можно вывести результат, возвращаемый функцией **map()** в терминал и понаблюдать за ее изменением.
Обратите внимание, что нижняя граница может быть как меньше, так и больше верхней границы. Это может быть использовано для того чтобы получить обратно пропорциональную зависимость.

Задание для самостоятельного выполнения 14.1

Измените программу так, чтобы частота звучания зуммера уменьшалась когда вы подносите руку к фоторезистору, и увеличивалась, когда убираете.

Ответ к заданию 14.1

```
#define PROTORESISTOR_PIN A3
#define BUZZER_PIN 9
#define VALUE_MIN 400
#define VALUE_MAX 890
#define FREQ_MIN 3000
#define FREQ_MAX 7000

int value = 0;
//Частота звука от 3000 до 7000
int frequency = 3000;

void setup(){
  pinMode(PROTORESISTOR_PIN, INPUT);
  pinMode(BUZZER_PIN, OUTPUT);
}

void loop(){
  //Читаем результат измерения. АЦП
  value = analogRead(PROTORESISTOR_PIN);
  //Меняем местами границы первого диапазона
  frequency = map(
    value,
    VALUE_MAX, VALUE_MIN,
    FREQ_MIN, FREQ_MAX
  );
  //Меняем частоту звучания зуммера
```

```
27    tone(BUZZER_PIN, frequency);
28    delay(50);
29  }
```

В программе изменилась только запись вызова функции **map()**. Обратите внимание на очередность записи границ первого диапазона. Границы диапазона поменялись местами. Было **VALUE_MIN, VALUE_MAX**, стало **VALUE_MAX, VALUE_MIN**. Можно было поменять местами границы второго диапазона (диапазона частот), результат был бы таким же.

Урок 15. Аналоговые датчики: термистор (терморезистор)

В этом уроке мы рассмотрим еще один популярный аналоговый датчик, работающий по той же схеме — термистор или терморезистор.
С его помощью можно легко собрать простейший термометр или пожарную сигнализацию.

Теория:

Терморезистор (термистор) — электронный элемент, изменяющий свое сопротивление в зависимости от температуры.
Основной характеристикой терморезистора является ТКС или температурный коэффициент сопротивления.
ТКС — показывает, на какую величину изменяется сопротивление терморезистора при изменении температуры на 1 градус.

Термисторы бывают двух видов:
PTC-термисторы (позисторы) — получили свое название от сокращения Positive Temperature Coefficient, что в переводе с английского означает "Положительный Коэффициент Сопротивления". Особенность данных термисторов в том, что при нагреве их сопротивление растет.

NTC-термисторы — имеют "Отрицательный Коэффициент Сопротивления" или "Negative Temperature Coefficient". Их сопротивление уменьшается с ростом температуры.

Зависимость сопротивления от температуры у термистора носит нелинейный характер. То есть, чтобы измерить температуру с помощью термистора, его необходимо предварительно откалибровать, составив таблицу соответствия между температурой и сопротивлением.

Внешний вид термистора и обозначение на схеме показано на Рис. 15.1.

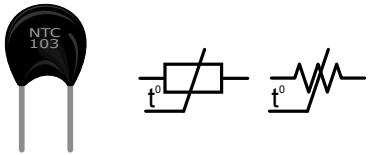


Рис. 15.1. Внешний вид терморезистора и его обозначение на схеме

Термистор включается по схеме делителя напряжения, как и фоторезистор. .

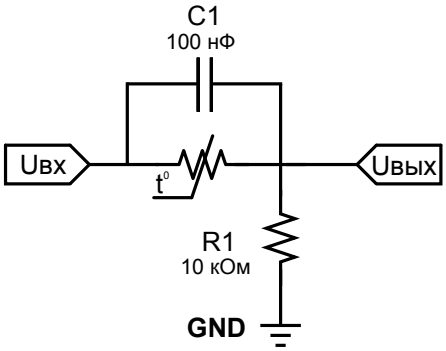


Рис. 15.2. Распиновка и внешний вид модуля с фоторезистором

В схеме на Рис. 15.2, термистор включен в верхнем плече делителя напряжения, но если перенести его в нижнее плечо, то принцип от этого не изменится.

Практика:

Схема используемого в конструкторе модуля полностью соответствует приведенной выше схеме. В нем применен термистор NTC типа. Внешний вид модуля и распиновка разъема показаны на Рис. 15.3.

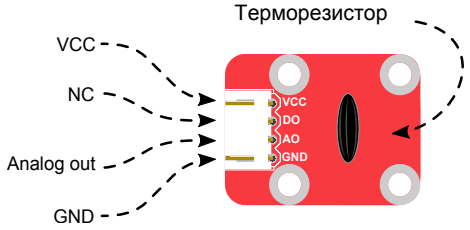


Рис. 15.3. Распиновка и внешний вид модуля с терморезистором

VCC — питание +5V
D0 — не используется
A0 — подключается к аналоговому входу контроллера (A0...A5)
GND — земля (общий)

Модуль с термистором, как и другие аналоговые модули, подключается к одному из разъемов 5, 8, 11 или 14, на которые выведены порты АЦП.

Задача 15.1. Пожарная сигнализация

В этом примере мы соберем пожарную сигнализацию с применением термистора и зуммера.
Контроллер будет отслеживать температуру окружающей среды, измеряя сопротивление термистора, и включать сигнал тревоги в случае превышения установленного предела. В качестве порога примем сопротивление термистора при комнатной температуре.

Используемый термистор принадлежит к NTC типу, значит его сопротивление падает с ростом температуры. Во время пожара воздух в помещении нагревается, а значит нагревается и корпус термистора, при этом его сопротивление уменьшается.
Для нагрева термистора мы не будем устраивать настоящий пожар. Достаточно взять термистор двумя пальцами или подышать на него теплым воздухом. Температура тела примерно на 10 градусов выше комнатной и ее хватит для проведения эксперимента.

Как и в опыте с фоторезистором, необходимо сначала откалибровать датчик. Для этого соберем макет, как показано на Рис. 15.4 и напишем программу для калибровки. Она будет точно такой же, как и программа для калибровки фоторезистора.

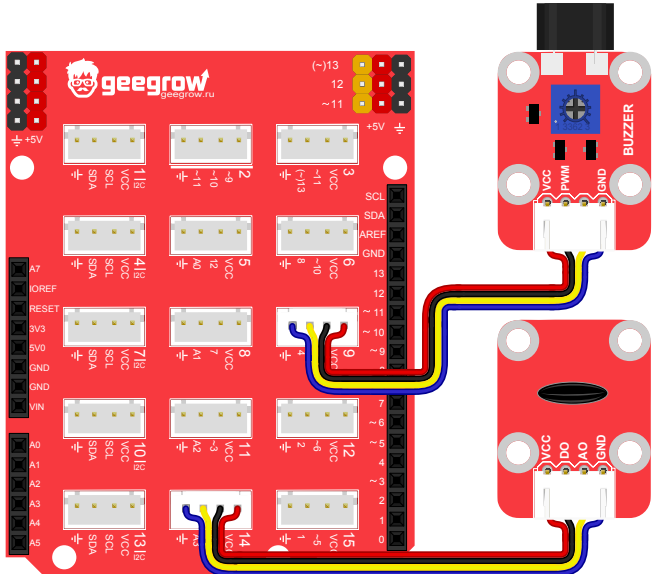


Рис. 15.4. Схема макета для задачи 15.1

```
#define THERMISTOR_PIN A3
//Переменная для хранения значения АЦП
int value = 0;

void setup(){
  pinMode(THERMISTOR_PIN, INPUT);
  //Открытие соединения на скорости 9600
  Serial.begin(9600);
}

void loop(){
  //Читаем результат полученный от АЦП
  value = analogRead(THERMISTOR_PIN);
  //Отправляем данные в терминал.
  Serial.println(value);
  delay(200);
}
```

Загрузите программу в контроллер и, открыв терминал, проведите калибровку, записав показания.
В нашем случае эксперимент показал, что при комнатной температуре показания АЦП равны 478, если взять термистор пальцами, показания постепенно увеличиваются до 530, а если подышать на термистор, значение измеренное АЦП, переваливает за отметку 560. Это объяснимо, ведь температура пальцев всегда ниже, чем температура дыхания. Кроме того, в вашем случае данные могут немного отличаться.

Чтобы не было ложного срабатывания, примем в качестве порогового значения не 478, а немного большее значение, например, 490. Теперь напишем программу и загрузим в контроллер.

```
#define THERMISTOR_PIN A3
#define BUZZER_PIN 9
//Значение АЦП соответствующее пороговой
//температуре
#define VALUE_LIMIT 490
//Переменная для хранения значения АЦП
int value = 0;

void setup(){
  pinMode(THERMISTOR_PIN, INPUT);
  pinMode(BUZZER_PIN, OUTPUT);
}

void loop(){
  //Читаем результат измерения. АЦП
  value = analogRead(THERMISTOR_PIN);
  //Проверяем не превышен ли порог
  if (value > VALUE_LIMIT) {
    tone(BUZZER_PIN, 5000);
  } else {
    noTone(BUZZER_PIN);
  }
  //Ждем 1 секунду
  delay(1000);
}
```

Измерения можно проводить не чаще 1 раза в секунду, потому что изменение температуры происходит достаточно медленно.

Задание для самостоятельного выполнения 15.1

Измените программу так, чтобы сигнализация издавала не непрерывный сигнал, а периодический. Причем, если температура растет, сигнал должен звучать чаще. Если же температура падает, частота звучания сигнала должна уменьшаться, пока сигнализация не отключится. Таким образом по частоте сигналов можно будет судить о том, на сколько сильно превышена максимальная температура.

Для управления периодом звучания сигнала используйте функцию `map()`;

Ответ к заданию 15.1

```
#define THERMISTOR_PIN A3
#define BUZZER_PIN 9

```

```
#define VALUE_LIMIT 490
//Границы диапазона значений АЦП
#define VALUE_MIN 490
#define VALUE_MAX 550
//Границы диапазона изменения периода/100,
//чтобы переключение было ступенчатым. Так
//интереснее
#define MAX_BEEP_TIME 5
#define MIN_BEEP_TIME 1

int value = 0;
//Период включения сигнала
int beepTime = 0;
bool isBeepOn = false;

void setup(){
  pinMode(THERMISTOR_PIN, INPUT);
  pinMode(BUZZER_PIN, OUTPUT);
}

void loop(){
  //Читаем результат измерения АЦП
  value = analogRead(THERMISTOR_PIN);
  //Меняем частоту звучания зуммера
  if (value > VALUE_LIMIT) {
    beepTime = 100 * map(
      value,
      VALUE_MAX, VALUE_MIN,
      MIN_BEEP_TIME, MAX_BEEP_TIME
    );

    if (isBeepOn) {
      noTone(BUZZER_PIN);
    } else {
      tone(BUZZER_PIN, 5000);
    }
    isBeepOn = !isBeepOn;
    delay(beepTime);
  } else {
    noTone(BUZZER_PIN);
    delay(1000);
  }
}
```

Урок 16. Управление ИК-пультом

Пультом дистанционного управления оснащаются многие гаджеты, ведь это удобно. Пришло время добавить немного магии и в наши разработки.

Теория

Для сборки простейшего дистанционно управляемого гаджета необходимы две вещи, ИК-пульт и ИК-приемник. Обмен данными между ними идет по оптическому каналу. Пульт излучает сигнал в инфракрасной части спектра, невидимой для человеческого глаза.

Сигнал, излучаемый пультом, чем-то напоминает азбуку Морзе, но вместо букв, он содержит только нули и единицы, закодированные импульсами модулирующего сигнала.

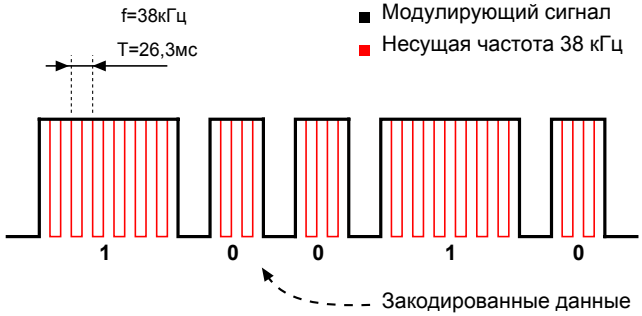


Рис. 16.1. Модулированный сигнал

Изображенный на Рис. 16.1 сигнал называется модулированным. Он состоит из периодического сигнала с постоянной частотой f, называемой несущей и полезного сигнала, который называют модулирующим. Модулирующий сигнал показан черным цветом, а несущая частота красным.

Частота несущей не обязательно равна 38кГц. Встречаются пульты и приемники с частотой модуляции 36 и 40кГц. Для достижения наилучшей чувствительности лучше подбирать пульт и ИК-приемник настроенные на одинаковую несущую частоту.

Чтобы было проще понять принцип получения модулированного сигнала, представьте, что вы пытаетесь передать сообщение, закодированное нулями и единицами, вашему другу, с помощью обычного фонарика. В одной руке вы держите фонарик и мигаете им с одинаковой частотой, а другой рукой прикрываете фонарик, если надо передать ноль или наоборот открываете, если надо передать единицу.

Применение модулированных сигналов в электронике и радиотехнике является повсеместным. Это связано с тем что модулированный сигнал является более помехозащищенным и его удобнее передавать.

Практика

Приемником ИК-сигнала от пульта обычно служат так называемые TSOP датчики. Они принимает инфракрасное излучение и превращают модулированный сигнал в модулирующий. В процессе преобразования происходит фильтрация возможных помех и восстановление огибающей сигнала. Таким образом на выходе TSOP датчика можно получить исходную последовательность нулей и единиц.

TSOP датчик имеет 3 контакта. Один из них служит для подачи питания, другой подключается к земле и еще один предназначен для передачи данных. Схема включения датчика показана на Рис.16.2.

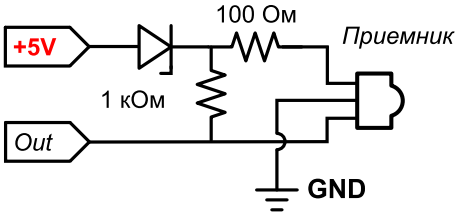


Рис. 16.2. Схема включения TSOP датчика (приемника)

Диод и резистор 100Ом, включенные последовательно с датчиком, выполняют защитную функцию. А резистор 10кОм, включенный между линией данных и портом питания, необходим для подтяжки линии данных к уровню 5В.

Мы будем использовать готовый модуль, собранный по точно такой же схеме.

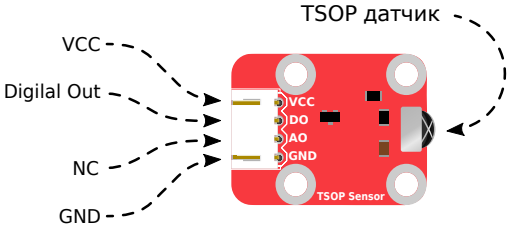


Рис. 16.3. Распиновка и внешний вид ИК-приемника

VCC — питание +5V
D0 — принятые данные
A0 — не используется
GND — земля (общий)

Так же, для приема и декодирования данных мы будем использовать библиотеку **IRemote**. Скачать ее можно с посвященной TSOP датчику страницы на сайте geegrow.ru или с сайта https://github.com/z3t0/Arduino-IRremote.

Перед началом работы нужно импортировать библиотеку в среду Arduino IDE. Для этого зайти в пункт меню Эскиз>Импорт библиотек>Добавить библиотеку (в том случае, если вы используете англоязычную версию Arduino IDE Sketch > Import Library .> Add Library).

Библиотека **IRremote** позволяет использовать ИК-приемник на произвольном порту контроллера. Это означает, что при использовании шилда, ИК-приемник можно включать в любой разъем, кроме I2C разъемов и разъема №2, служащего для работы с RGB светодиодом.

Чтобы получить доступ к функционалу библиотеки, необходимо включить в начало программы строку с именем заголовочного файла: **#include <IRremote.h>**. и инициализировать библиотеку командой **IRrecv irrecv(RecvPin);** где **RecvPin** — номер порта контроллера
Команда создает переменную **irrecv**, которая служит для доступа к методам класса **IRrecv**.

Рассмотрим основные методы библиотеки **IRrecv**.

irrecv.enableIRIn() — запускает прием и накопление данных от ИК-приемника.

irrecv.decode(&results) — записывает принятые данные в переменную **results**. Причем, переменную нужно предварительно объявить следующим образом: **decode_results results**.
В дальнейшем, доступ к данным, записанным в переменную **results** можно получить, обратившись к ней следующим образом **results.value**

irrecv.resume() — возвращает объект **irrecv** в режим приема данных.

Задача 16.1. Дистанционное управление светом

Соберем один из основных элементов умного дома — светильник управляемый пультом дистанционного управления.
Будем включать и выключать свет при нажатии на кнопку OK.

Соберем макет согласно Рис. 16.4, подключив приемник в разъем №6 на шилде, при этом сигнал будет поступать на порт контроллера №10.

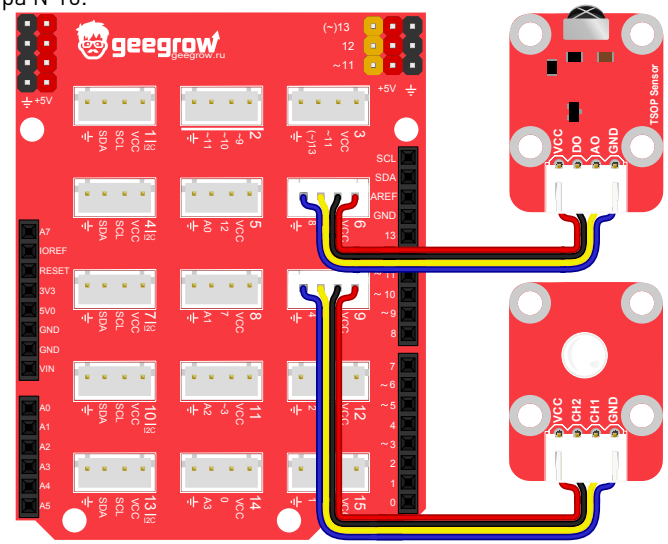


Рис. 16.4. Схема макета для задачи 16.1

Перед тем, как приступить к решению основной задачи, напишем тестовую программу, чтобы выяснить какой код соответствует кнопке OK.

```
1  //Подключаем библиотеку
2  #include <IRremote.h>
3
4  //Задаем порт, к которому подключен приемник
5  IRrecv irrecv(10);
6  decode_results results;
7
8  void setup() {
```

```
  //Выставляем скорость COM порта
  Serial.begin(9600);
  //Запускаем прием
  irrecv.enableIRIn();
}

void loop() {
  //Если данные пришли
  if (irrecv.decode(&results)) {
    //Переводим данные из двоичного
    //в шестнадцатиричный формат
    String code = String(results.value, HEX);
    //Отправляем принятые данные в консоль
    Serial.println(code);
    //Принимаем следующую команду
    irrecv.resume();
  }
}
```

Загрузив программу в контроллер и нажав на кнопку OK, увидим код, соответствующий этой кнопке — **ff38c7**. Если нажать на кнопку и удерживать ее, вы уведите другой код — **ffffffff**. Это код удержания и он одинаков для всех кнопок.

Доступ к принятым данным получаем через свойство **value** переменной **results**. Если вывести полученное значение прямо в консоль, без дополнительных преобразований, то вы увидите число записанное в двоичной форме, а именно **16726215**. Работать с кодом в таком виде можно, но это не очень удобно. Чтобы перевести число в шестнадцатеричный формат, в строке 21 была создана переменная **code** имеющая тип **String**. Во время создания переменной этого типа можно перевести ее значение из двоичной в шестнадцатеричный форму, передав в качестве второго аргумента **HEX**.

Теперь, зная код кнопки OK, напишем программу управления светодиодом.

```
//Подключаем библиотеку
#include <IRremote.h>

#define LED_PIN 9
int state = LOW;

//Задаем порт, к которому подключен приемник
IRrecv irrecv(10);
decode_results results;

void setup() {
  //Настройка входов/выходов
  pinMode(LED_PIN, OUTPUT);
  //Запускаем прием
  irrecv.enableIRIn();
}

void loop() {
  //Если данные пришли
  if (irrecv.decode(&results)) {
    //Переводим данные из двоичного
    //в шестнадцатиричный формат
    String code = String(results.value, HEX);

    //Если нажата кнопка OK
    if (code == "ff38c7") {
      state = digitalRead(LED_PIN);
      //Меняем состояние светодиода на
      //противоположное
      digitalWrite(LED_PIN, !state);
    }

    //Принимаем следующую команду
    irrecv.resume();
  }
  //Ждем 100мс
  delay(100);
}
```

Попробуйте усовершенствовать светильник в домашнем задании.

Задание для самостоятельного выполнения 16.1

Измените программу так, чтобы яркость светодиода плавно регулировалась стрелками вниз/вверх. При составлении программы используйте код удержания кнопки — **ffffffff**.

Перед написанием программы, проверьте, какой код передается пультом при нажатии на кнопки вверх/вниз. В ответе, кнопке вверх соответствует код **ff18e7**, а кнопке вниз **ff4ab5**.

Ответ к заданию 16.1

```
//Подключаем библиотеку
#include <IRremote.h>

#define LED_PIN 9
#define UP_CODE "ff18e7"
#define DOWN_CODE "ff4ab5"
#define REPEAT_CODE "ffffffff"
#define PWM_STEP 15

//Задаем порт, к которому подключен приемник
IRrecv irrecv(10);
decode_results results;
int brightness = 127;
String lastCode;

void setup() {
  //Настройка входов/выходов
  pinMode(LED_PIN, OUTPUT);
  //Запускаем прием
  irrecv.enableIRIn();
}

void loop() {
  //Если данные пришли
  if (irrecv.decode(&results)) {
    //Переводим данные из двоичного
    //в шестнадцатиричный формат
    String code = String(results.value, HEX);

    if (code == REPEAT_CODE) {
      //Код повторяется
      changeBrightness(lastCode);
    } else {
      //Получен новый код
      changeBrightness(code);
      lastCode = code;
    }

    //Принимаем следующую команду
    irrecv.resume();
  }

  analogWrite(LED_PIN, brightness);
  //Ждем 100мс
  delay(100);
}

void changeBrightness(String code) {
  if (code == UP_CODE) {
    //Убедимся, что новое значение
    //будет не больше 255
    if (brightness + PWM_STEP <= 255) {
      brightness = brightness + PWM_STEP;
    } else {
      brightness = 255;
    }
  } else if (code == DOWN_CODE) {
    //Убедимся, что новое значение
    //будет не меньше нуля
    if (brightness - PWM_STEP >= 0) {
      brightness = brightness - PWM_STEP;
    } else {
      brightness = 0;
    }
  }
}
```

Содержание

Проекты

Что дальше?

Поздравляем! Вы только что завершили обучение по программе базового курса начинающего изобретателя и, наверняка, вам не терпится удивить друзей и знакомых своими изобретениями.

Делитесь своими достижениями, находите единомышленников, обменивайтесь идеями новых проектов на сайте geegrow.ru, ищите нас в facebook и Вконтакте.

В поисках идей для новых проектов приходите сюда geegrow.ru/projects.

Обучающее видео и описание новых самоделок ищите на нашем канале в YouTube.

Задавайте вопросы на форуме geegrow.ru/forum.

Изучай
Придумывай
Изобретай