

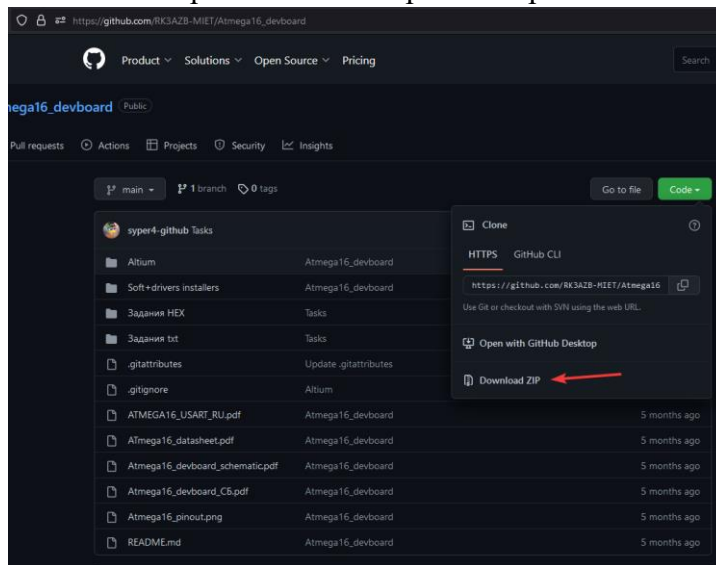
Установка Microchip Studio

Microchip Studio – это среда разработки (IDE) в которой вы будете разрабатывать и отлаживать код для микроконтроллеров AVR.

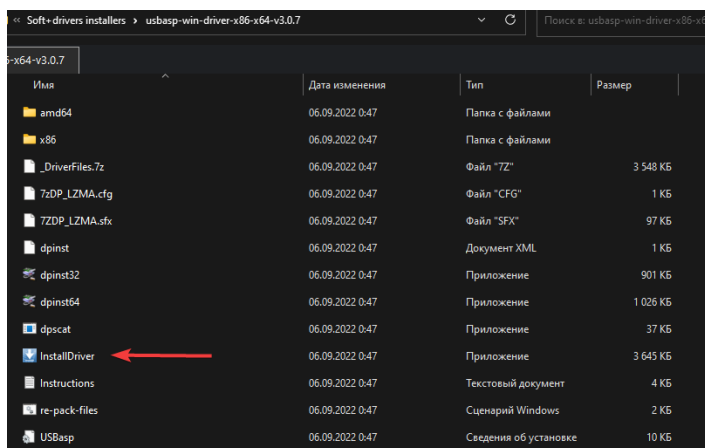
1. В гитхабе радиоклуба (<https://github.com/RK3AZB-MIET>) переходим по пути [Atmega16_devboard](#) >> [README.md](#)
2. Скачиваем установщик Microchip Studio и проходим стандартный процесс установки. Галочки оставляем по умолчанию.

Установка драйверов и загрузчика

1. Скачиваем и распаковываем репозиторий.



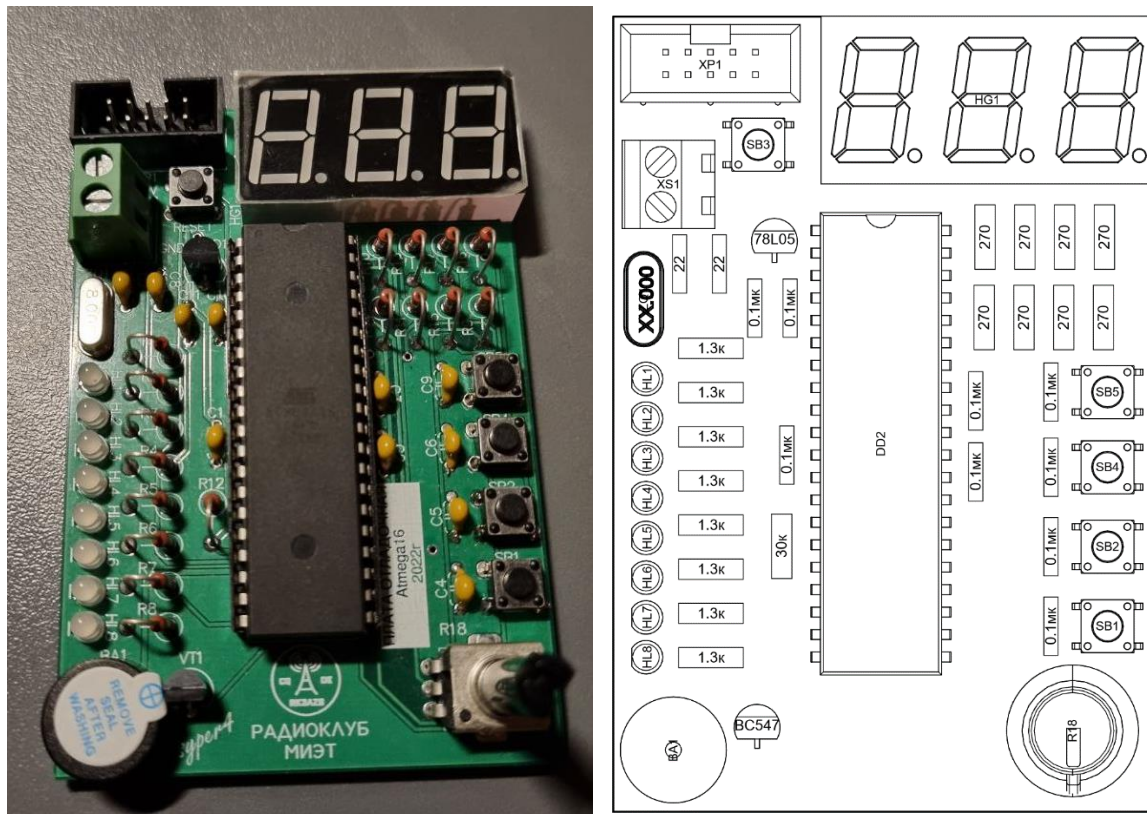
2. Из папки «Soft+drivers installers» устанавливаем драйвер для программатора usbasp



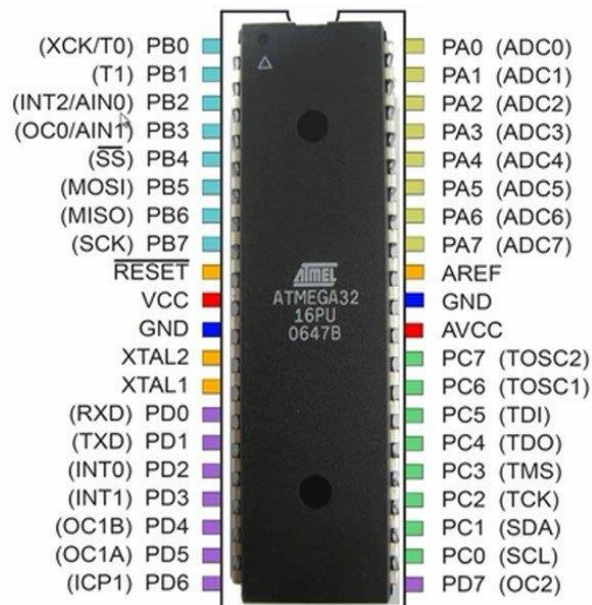
3. Программу avrdudeprog33 советуем скопировать в любое удобное место, потому что запускать её придётся часто. Именно с помощью неё код, написанный в Microchip Studio, будет преобразовываться в файл .hex и загружаться в микроконтроллер.

1. Знакомство с отладочной платой.

Вашему вниманию предоставляется отладочная плата, разработанная в радиоклубе МИЭТ на базе микроконтроллера Atmega16.



Непосредственно в центре располагается сам микроконтроллер. Atmega16 – это 8-разрядный микроконтроллер с 16 Кбайтами памяти. Каждой ноге микроконтроллера отведена определённая функция. Узнать эти функции можно посмотрев на специальную схему (Pinout). Схему можно найти на просторах интернета или в файле Datasheet (файл, в котором написана вся информация о микроконтроллере). Файл можно найти в репозитории Atmega16_devboard на GitHub.



Микроконтроллер ATmega16 имеет 40 выводов. Почти каждый вывод может выполнять несколько функций. Сокращения названий функций приводятся в скобках рядом с выводами. Рассмотрим некоторые из них.

Для работы микроконтроллера, впрочем, как и любой другой микросхемы, необходимо напряжение. Во избежание ложных срабатываний МК нужно питать только стабильным напряжением от 4,5 В до 5,5 В. Этот диапазон напряжений строго регламентирован и приведен в даташите.

Плюс («+») источника питания подсоединяется к ножке, обозначенной VCC. Минус («-») – к любой из тех, которые имеют обозначение GND (GND – сокращенно от ground – «земля»).

Остальные ножки позволяют микроконтроллеру взаимодействовать с внешними устройствами. Они объединены в отдельные группы и называются порты ввода-вывода микроконтроллера. С помощью них можно как подавать сигналы на вход МК, например с различных датчиков, так и выдавать сигналы для управления другими устройствами или для отображения информации на различных индикаторах и дисплеях.

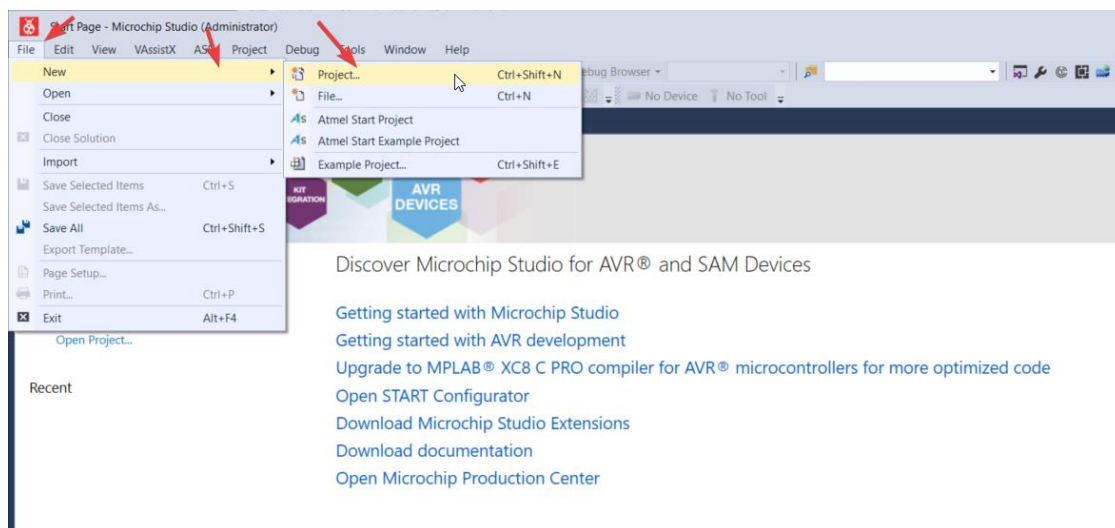
Микроконтроллер ATmega16 имеет четыре порта ввода-вывода: A, B, C и D. Порты могут быть полными и неполными.

Нумерация портов начинается с нуля, например PB0, PB1, PB2...

Знакомство с интерфейсом программы

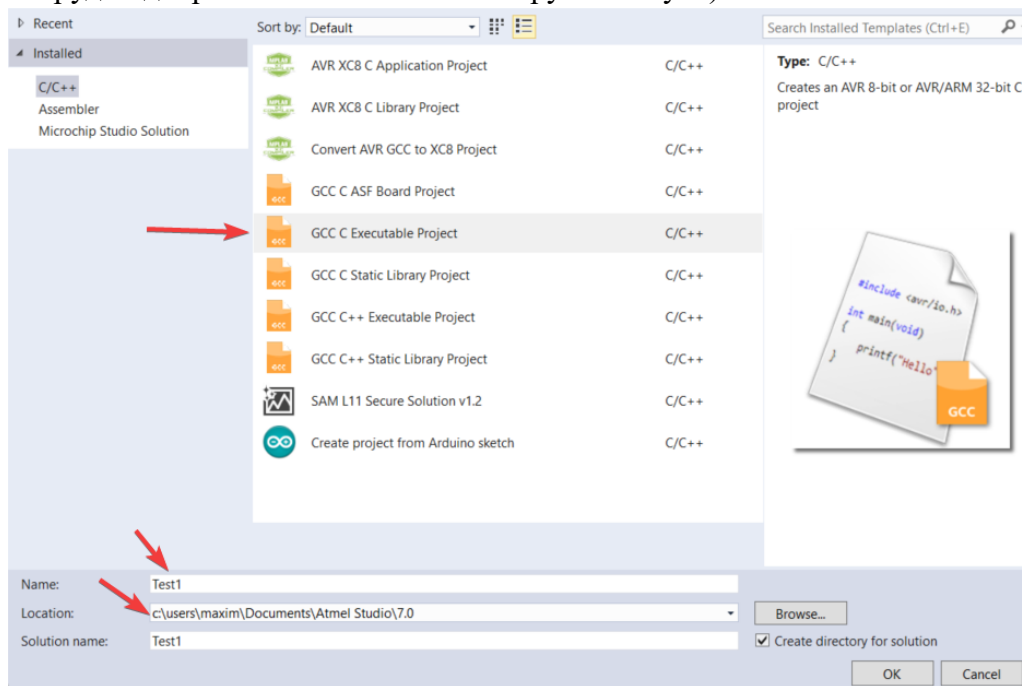
При первом запуске Microchip Studio вас встречает стартовое окно программы.

Для того чтобы начать работу с программой, создайте новый проект.

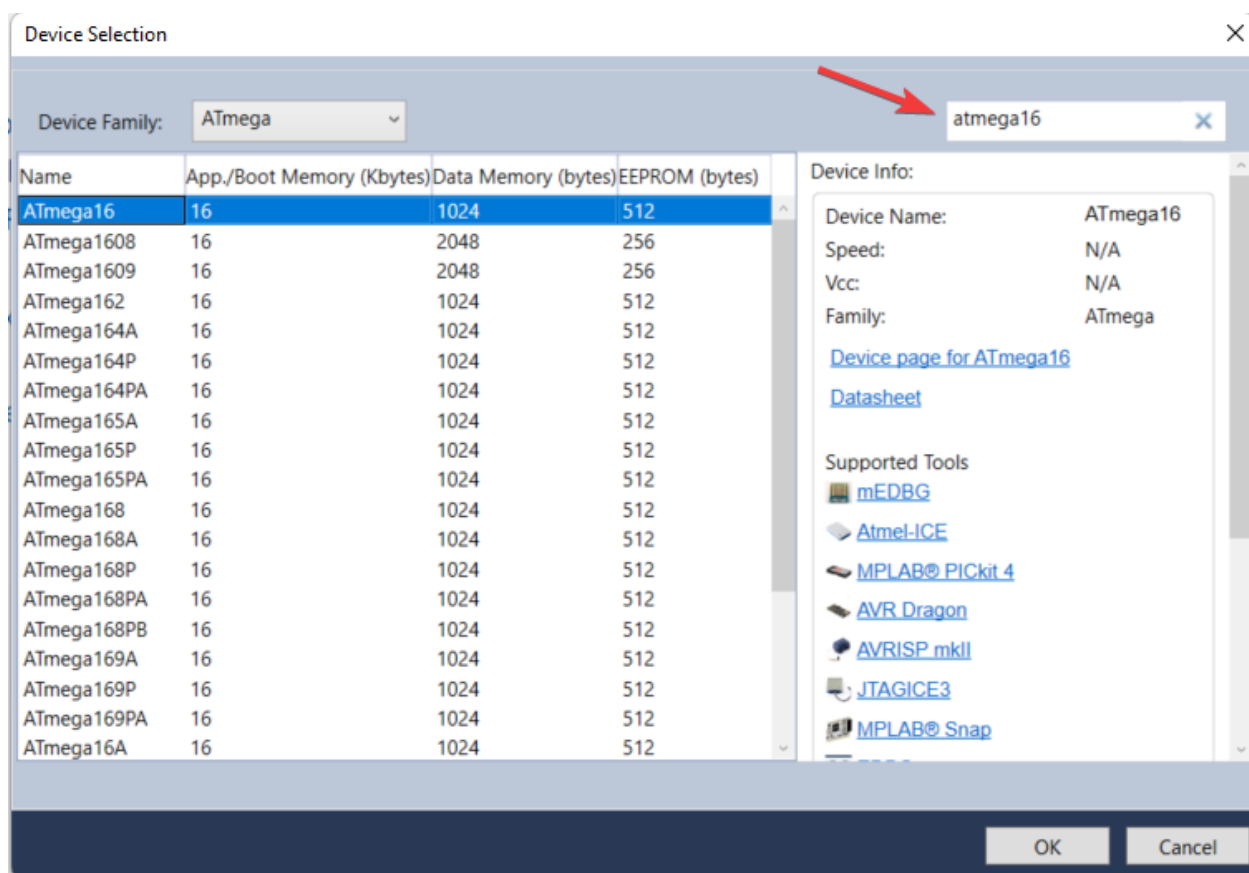


Перед вами откроется окно создания нового проекта. В списке доступных компиляторов выбираем GCC C Executable Project. В поле Name пишем имя нашего проекта. В поле Location выбираем удобное место для сохранения проекта (в ту папку, до которой будет

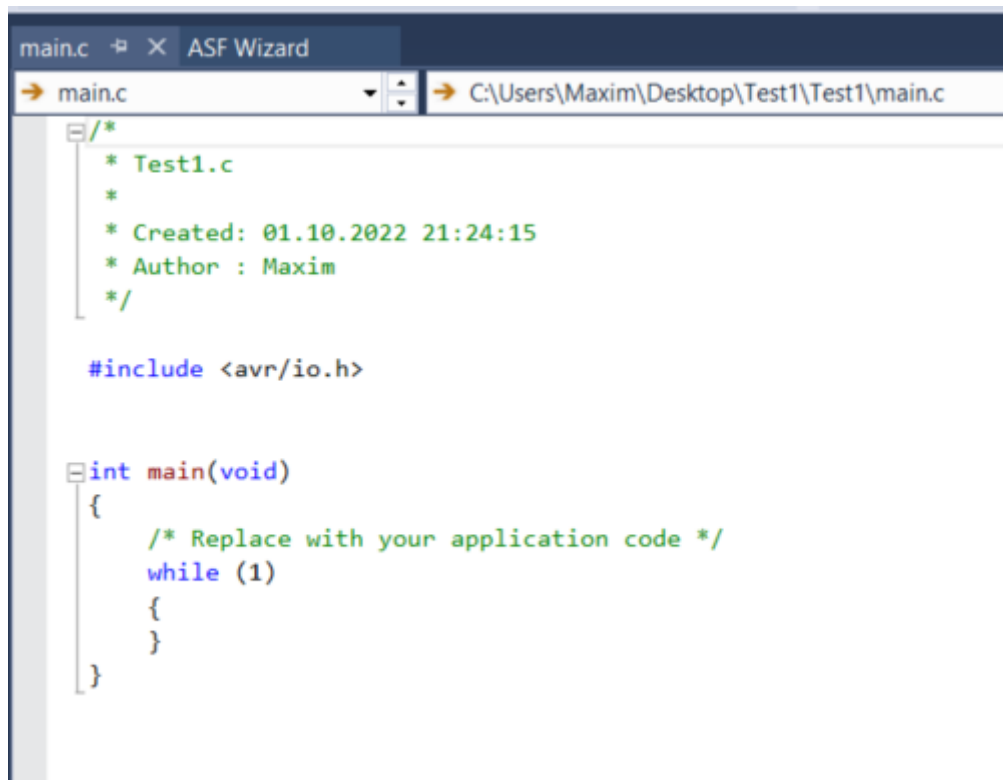
нетрудно добраться и желательно без русских букв).



Далее вы увидите окно выбора микроконтроллера. Чтобы быстро найти нужный вам контроллер в поиске можете вбить его название (на данном курсе вы будете использовать atmega16).



Перед вами откроется файл main.c В нём вы будете писать код программы.



```
main.c X ASF Wizard
main.c C:\Users\Maxim\Desktop\Test1\Test1\main.c
/*
 * Test1.c
 *
 * Created: 01.10.2022 21:24:15
 * Author : Maxim
 */

#include <avr/io.h>

int main(void)
{
    /* Replace with your application code */
    while (1)
    {
    }
}
```

Текст, выделенный зеленым цветом – это комментарии. В данном случае приводится имя проекта и автора, а также время и дата создания проекта. Комментарии не являются программным кодом, поэтому они игнорируются компилятором. Комментарии позволяют программисту улучшить читаемость кода, что особенно помогает при поиске ошибок и отладке программы.

Комментарии бывают однострочные и многострочные. В данном шаблоне программы применяются многострочные комментарии. Они начинаются косой линией со звездочкой, а заканчиваются звездочкой с косой линией.

/ — начало комментария*

.
.
.
.

**/ — конец комментария*

Весь текст, который помещен между */** и **/* полностью пропускается компилятором.

Однострочный комментарий обозначается двумя косыми линиями и действует в пределах одной строки. Текст перед двумя косыми распознается компилятором как код программы, а после – как комментарий.

// Здесь пишется однострочный комментарий

На практике использование комментариев считается хорошим тоном программирования. В дальнейшем мы будем применять оба их типа.

Директива препроцессора

Следующим элементом программы является строка

```
#include <avr/io.h>
```

Эта строка указывает компилятору, что к данному файлу нужно подключить другой файл с именем `io.h`, который находится в папке `avr`. В подключаемом файле находится информация о базовых настройках микроконтроллера.

По сути, можно было бы и не подключать файл `io.h`, а набрать его содержимое вручную, однако это очень неудобно.

Знак `#` означает, что данная команда – это директива препроцессора. Дело в том, что прежде, чем скомпилировать файл компилятору необходимо выполнить предварительную его обработку. Поэтому сначала выполняется некая подготовка файла к компиляции путем добавления некоторых инструкций, прописанных в файле `io.h`

`io` – название файла, которое происходит от сокращения `input/output` – ввод/вывод.

`.h` – расширение файла, название его происходит от слова `header` – заголовок.

Теперь все вместе. `io.h` – это заголовочный файл, в котором записана информация о настройках ввода-вывода микроконтроллера.

Главная функция `main`

Ниже нам встречается следующая строка:

```
int main(void)
```

В данной строке объявляется функция, носящая имя `main`. С нее начинается выполнение программы. Эта функция является как бы точкой начала всей программы, написанной в текущем файле. Имя `main` зарезервировано в языке Си, поэтому во избежание конфликтов, таким именем нельзя называть другую функцию, находящуюся внутри данной. `main` переводится главный, т. е. данная функция является главной.

В синтаксисе языка Си идентификатором функции служат круглые скобки

`()`

Внутри скобок помещено слово `void`. Оно обозначает пустота. Это указывает на то, что функция `main` ничего не принимает, т. е. не принимает никаких аргументов. По мере написания более сложных программ, мы детальнее остановимся на этом моменте.

`int` – это целочисленный тип данных. В данном случае функция работает с целыми числами: как положительными, так и отрицательными. Существуют и другие типы данных, например с плавающей запятой, символьные и др. Более подробно мы будем рассматривать типы

данных по мере необходимости или в отдельной статье. Однако для функции `main` рекомендуется всегда использовать тип данных `int`, поскольку конструкция `int main(void)` определена стандартом языка Си и распознается любым компилятором.

Область действия функции определяется фигурными скобками

```
{  
  
.  
  
.    → тело функции  
  
.  
  
}
```

Код программы, помещенный между открывающейся и закрывающейся скобками, относится к телу функции.

В общем случае любая функция имеет следующую конструкцию:

```
тип данных имя функции (аргумент)  
  
{  
  
тело функции (код)  
  
}
```

Функция *while*

Внутри функции `main` находится функция `while`:

```
while (1)  
  
    {  
  
    }
```

While переводится с английского «пока». Это говорит о том, что программа, которая находится в теле данной функции, будет выполняться до тех пор, пока условие истинно. Единица в круглых скобках указывает, что условие истинно, поэтому код программы, написанный в данной функции, будет повторяться бесконечное число раз, т.е. программа будет заиклена. Для чего это делается? Дело в том, что микроконтроллер должен непрерывно выполнять записанную программу. Поэтому программа не может просто взять и оборваться. Микроконтроллер всегда опрашивает порты ввода-вывода либо выдает в них сигналы, даже находясь в ждущем режиме.

Теперь, когда мы рассмотрели основные элементы конструкции программы, давайте посмотрим целиком на базовый шаблон. Без комментариев он имеет следующий вид:

```
#include <avr/io.h>
```

```
int main(void)
{
    while (1)
    {
    }
}
```

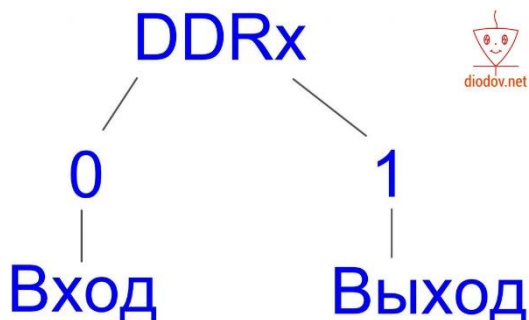
Программирование портов ввода-вывода микроконтроллера ATmega16

Сейчас мы уже можем дополнить программу нужным нам кодом. Первым делом необходимо настроить нулевой бит порта C PD0 на выход.

МК может, как принимать, так и выдавать сигнал, т.е. выводы (порты) его могут работать как *входы* и как *выходы*. Поэтому предварительно нужно настроить вывод МК на соответствующий режим. Для этого в микроконтроллере есть специальный регистр, который называется **DDR** – *direct data register* – регистр направления данных.

У каждого порта есть свой такой регистр. Например, регистр порта C называется **DDRC**, порта B – **DDRB**, порта D – **DDRD**.

Чтобы настроить вывод порта на *вход* в регистр DDR необходимо записать *ноль*, а на *выход* – *единицу*.



Команда настройки нулевого бита порта D выглядит следующим образом

```
DDRD = 0b00000001;
```

Данной командой в регистр DDRD записывается двоичное число равное десятичному 1. Префикс 0b идентифицирует данное число, как двоичное.

Двоичная форма записи очень удачно сочетается с количеством битов порта, так как количество битов соответствует количеству выводов порта, а порядковый номер бита отвечает номеру бита внутри порта.

Также можно записать в регистр шестнадцатеричное число:


```
DDRD = 0x01;
```

Однако двоичная форма записи более наглядна, поэтому ее мы и будем использовать на начальных этапах программирования микроконтроллеров.

Давайте рассмотрим еще один пример. Допустим нам необходимо настроить нулевой, третий и седьмой биты порта D на выход, а остальные биты на вход. Для этого случая код имеет такой вид:

```
DDRD = 0b10001001;
```

Регистр микроконтроллера *PORT*

После того, как мы настроили нулевой бит порта D PD0 на выход, нужно еще выполнить настройку, чтобы на данном выводе появилось напряжение +5 В. Для этого необходимо установить нулевой бит в регистре ***PORT***. Если бит установлен в ***единицу***, то на выводе будет **+5 В** (точнее говоря величина напряжения питания микроконтроллера, которая может находиться в пределах 4,5...5,5 В для микроконтроллера ATmega16). Если бит установлен в ***ноль***, — то на выводе будет напряжение, величина которого близка к ***нулю***.

Каждый порт имеет свой регистр: порт A – ***PORTA***, порт B – ***PORTB***, порт C – ***PORTC***.

И так, чтобы получить на выводе PD0 напряжение +5 В, необходимо записать такую команду:

```
PORTD = 0b00000001;
```

Обратите внимание на то, что каждая команда заканчивается точкой с запятой.

Таким образом, чтобы засветить светодиод, нам необходимы всего лишь две команды:

```
DDRD = 0b00000001;
```

```
PORTD = 0b00000001;
```

Первой командой мы определяем вывод PD0 как выход, а второй устанавливаем на нем напряжение +5 В.

Полный код программы выглядит так:

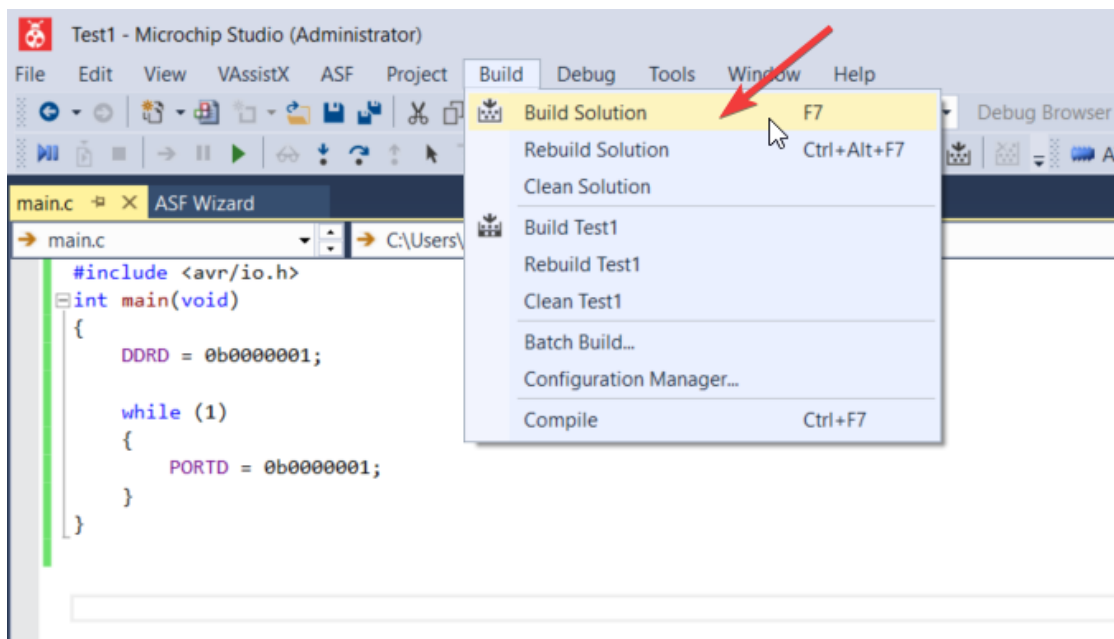
```
#include <avr/io.h>

int main(void)
{
    DDRD = 0b00000001;
    while (1)
    {
        PORTD = 0b00000001;
    }
}
```

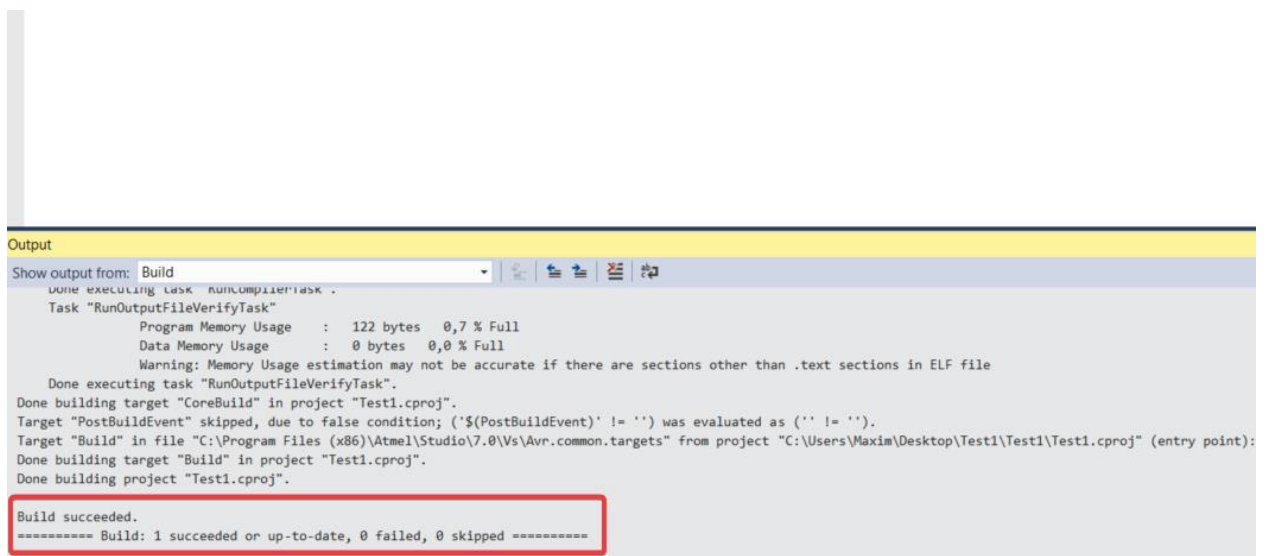
Здесь необходимо заметить следующее: команда `DDRD = 0b00000001;` выполнится всего один раз, а команда `PORTD = 0b00000001;` будет выполняться все время в цикле, поскольку она находится в теле функции `while (1)`. Но даже если мы вынесем команду за пределы функции `while` и поместим ее после `DDRD = 0b00000001;;`, светодиод и в этом случае будет светиться все время. Однако, разместив команду `PORTD = 0b00000001;` в теле `while (1)`, мы делаем явный акцент на том, что светодиод должен светиться все время.

Компиляция файла

Теперь, когда код полностью готов, его нужно скомпилировать. Для этого необходимо нажать клавишу `F7` или кликнуть по кнопке *Build* и в выпавшем меню выбрать *Build Solution*.



Если ошибок в коде нет, то файл скомпилируется, а в нижней части экрана появится запись о том, что проект скомпилирован успешно: *Build succeeded*.

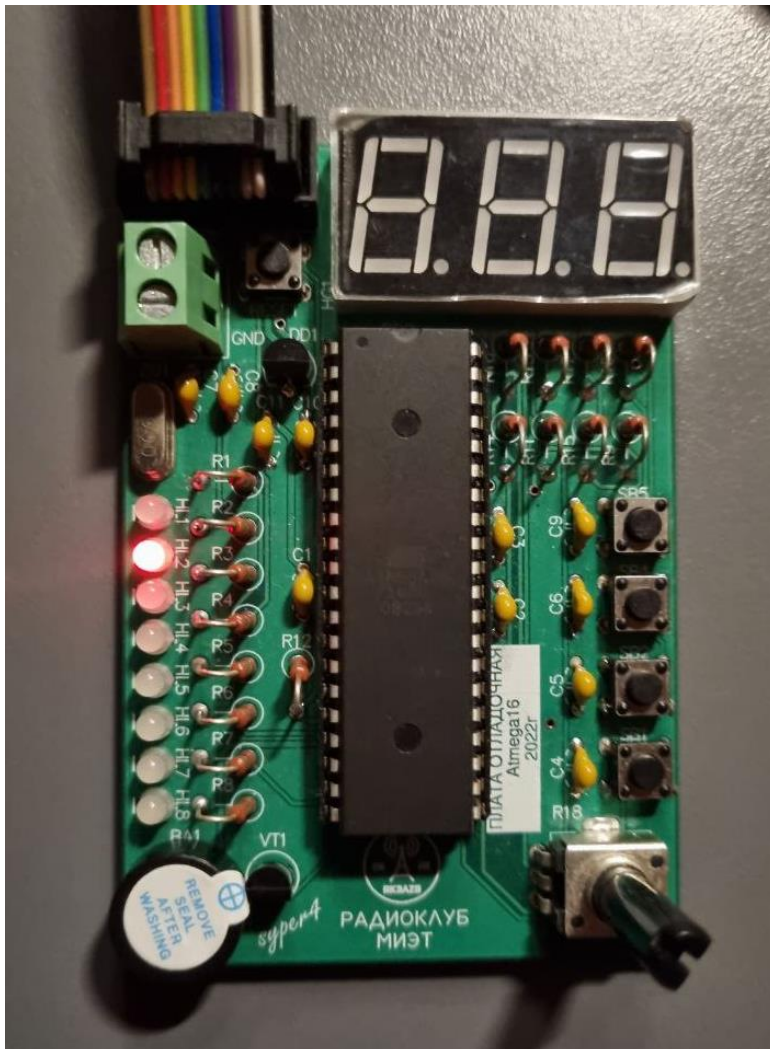


Далее открываем `avrdude_prog`. В выбираем наш микроконтроллер, указываем путь к `.hex` файлу, выбираем нужную версию программатора (`Usbasp_8M`). Чтобы проверить, что программа видит микроконтроллер, в верхней части программы нажимаем на кнопку «Чтение». В результате в поле справа от кнопки будет написана последовательность букв и цифр. Если такого не произошло, то программа не может получить данные с микроконтроллера. Причин этого может быть несколько:

- неправильно выбранная версия программатора или микроконтроллера
- неустановленные драйверы для программатора
- сбой в работе самого микроконтроллера
- неправильно вставленный провод в разъём отладочной платы и др.

В таком случае лучше обратиться к руководителю группы.

Для того чтобы залить программу на микроконтроллер нужно нажать кнопку «Программирование» в разделе Flash. В результате должен загореться светодиод (HL2).



2. Устройство и работа портов ввода-вывода

С внешним миром микроконтроллер общается через порты ввода-вывода. **PIN, PORT, DDR** это регистры конфигурации порта.

PINx

Это регистр чтения. Из него можно только читать. В регистре **PINx** содержится информация о **реальном текущем логическом уровне** на выводах порта. Вне зависимости от настроек порта. Если хотим узнать, что у нас на входе — читаем соответствующий бит регистра **PINx**. Причем существует две границы: граница гарантированного нуля и граница гарантированной единицы — пороги за которыми мы можем однозначно четко определить текущий логический уровень. Для пятивольтового питания это 1.4 и 1.8 вольт соответственно. То есть при снижении напряжения от максимума до минимума бит в регистре **PIN** переключится с 1 на 0 только при снижении напряжения ниже 1.4 вольт, а вот когда напряжение нарастает от минимума до максимума переключение бита с 0 на 1 будет только по достижении напряжения в 1.8 вольт. То есть возникает гистерезис переключения с 0 на 1, что исключает хаотичные переключения под действием помех и наводок, а также исключает ошибочное считывание логического уровня между порогами переключения.

При снижении напряжения питания, разумеется, эти пороги также снижаются, график зависимости порогов переключения от питающего напряжения можно найти в datasheet.

DDRx — Это регистр направления порта. Порт в конкретный момент времени может быть либо входом, либо выходом (но для состояния битов **PIN** это значения не имеет. Читать из **PIN** реальное значение можно всегда).

PORTx

Режим управления состоянием вывода. Когда мы настраиваем вывод на вход, то от **PORT** зависит тип входа (Hi-Z или PullUp, об этом чуть ниже).

Когда ножка настроена на **выход**, то значение соответствующего бита в регистре **PORTx** определяет состояние вывода. Если **PORTx=1**, то на выводе лог1, если **PORTx=0**, то на выводе лог0.

Когда ножка настроена на **вход**, то если **PORTx=0**, то вывод в режиме **Hi-Z**.

Если **PORTx=1**, то вывод в режиме **PullUp** с подтяжкой резистором в 100к до питания.

Теперь кратко о режимах входа:

Вход Hi-Z — режим высокоимпедансного входа.

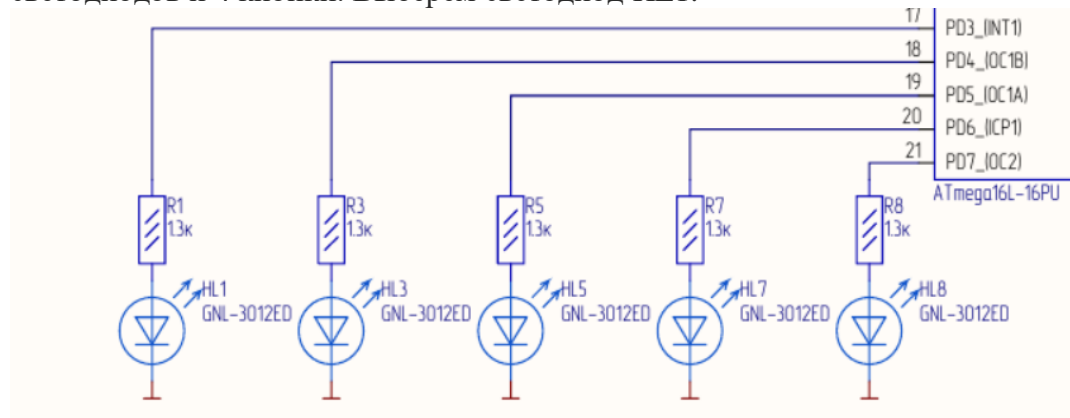
Этот режим включен по умолчанию, сопротивление порта **очень велико**. То есть электрически вывод как бы вообще никуда не подключен и ни на что не влияет. Но! При этом он постоянно считывает свое состояние в регистр **PIN**, и мы всегда можем узнать, что у нас на входе — единица или ноль. Этот режим хорош для прослушивания какой-либо шины данных, т.к. он не оказывает на шину никакого влияния. А что будет если вход висит в воздухе? А в этом случае напряжение будет на нем скакать в зависимости от внешних наводок, электромагнитных помех. Очень часто на порту в этом случае нестабильный синус 50Гц — наводка от сети 220В, а в регистре **PIN** будет меняться 0 и 1 с частотой около 50Гц

Вход PullUp — вход с подтяжкой.

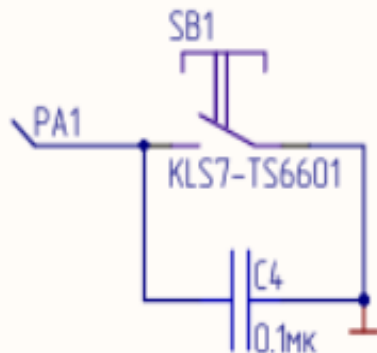
При **DDRx=0** и **PORTx=1** замыкается ключ подтяжки и к линии подключается резистор в 100кОм, что моментально приводит неподключенную никуда линию в состояние лог1.

Цель подтяжки очевидна — не допустить хаотичного изменения состояния на входе под действием наводок. Но если на входе появится логический ноль (замыкание линии на землю кнопкой или другим микроконтроллером/микросхемой), то слабый 100кОмный резистор не сможет удерживать напряжение на линии на уровне лог1 и на входе будет ноль.

Теперь попробуем управлять с помощью кнопки светодиодом. На отладочной плате есть 8 светодиодов и 4 кнопки. Выберем светодиод HL1.



И кнопку SB1, светодиод подключен к выводу PD3, кнопка к PA1.



```
#include <avr/io.h>

int main(void)
{
    DDRD = 0b00001000; //PD3 - выход
    DDRA = 0b00000000; //PA1 - вход
    PORTA = 0b00000010; //Вход с подтяжкой

    while (1)
    {
        /*Используем логическую операцию И, если кнопка разомкнута,
        то на входе лог1 1 * 1 = 1! = 0
        Условие if - ложь, светодиод не горит.
        Когда кнопка замкнута на входе PINA лог0 0 * 1 = 0,
        то условие выполняется и диод загорается. */
        if (!(PINA & 0b00000010))
        {
            PORTD = 0b00001000;
        }
        else
        {
            PORTD = 0b00000000;
        }
    }
}
```

Однако не всегда бывает удобно инициализировать порты в виде двоичного числа. Для этого применяют побитовые операции. Как мы все знаем, любое число можно привести к

двоичному типу. В этом случае один разряд данного числа может принимать только одно из двух значений — **0** или **1**, и этот разряд, соответственно, называется битом. До сих пор, изучая язык C, мы работали с числами целиком, но, оказывается, можно оперировать его битами. Не так, конечно, как при использовании ассемблера. Там вообще существуют инструкции, которые изменяют значение отдельно взятого бита. В C таких команд и операций нет, но есть операции, которые работают с каким-то числом и изменяют значение его битов определённым образом и эти операции называются **побитовые** (или битовые) **операции** и для реализации данных операций существуют соответствующие операторы, которые, в свою очередь, называются **побитовыми** (или битовыми) **операторами**.

Побитовых операторов существует всего шесть. Вот их список:

- **&** — побитовое **И** (AND).
- **|** — побитовое **ИЛИ** (OR).
- **^** — побитовое **исключающее ИЛИ** (XOR).
- **<<** — сдвиг битов влево.
- **>>** — сдвиг битов вправо.
- **~** — побитовое отрицание (**НЕ** или NOT).

Побитовая операция **И** с применением соответствующего оператора **&** является бинарной, поэтому работает с двумя операндами.

Данная операция сравнивает биты обоих операндов с одним индексом и в результате мы получаем **1** только в том случае, если оба сравниваемых бита будут равны **1**. В остальных случаях результат будет **0**.

Для простоты мы не будем брать слишком большие числа, а возьмём числа восьмибитные и далее будем оперировать именно такими значениями, хотя мы можем побитовые операции проводить между значениями любых типов и любых диапазонов.

Произведём нашу операцию, например, между вот такими числами и мы получим в результате вот такое значение

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

&

1	0	0	0	1	0	0	1
---	---	---	---	---	---	---	---

=

1	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---

В результате операции, как и было сказано, единицу мы получили только там, где в обоих операндах также были единицы.

Следующая побитовая операция — это операция **ИЛИ**, которая происходит благодаря использованию оператора **|**. В результате данной операции мы, наоборот, получаем **ноль** только в тех случаях, когда в обоих сравниваемых битах будут **нули**. Произведём такую операцию между теми же числами

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

|

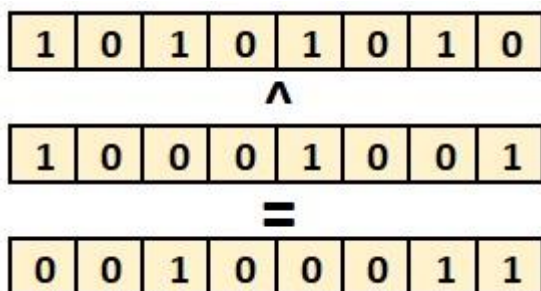
1	0	0	0	1	0	0	1
---	---	---	---	---	---	---	---

=

1	0	1	0	1	0	1	1
---	---	---	---	---	---	---	---

Другими словами, мы получили в результате единицу в том бите, когда хотя бы в одном из наших операндов в соответствующем бите была единица.

Следующая операция — побитовое **исключающее ИЛИ**. Для получения результата такой операции не столько важно то, какие именно будут сравниваемые биты, а важно то, равны они между собой или нет. То есть единицу мы получим только в том случае, если в одном из сравниваемых разрядов будет **ноль**, а в другом — **единица**. А **ноль** в результате, наоборот, будет в том случае, когда сравниваемые биты будут **равны**, то есть они либо оба будут равны **1**, либо оба будут равны **0**. Операнды будут те же. Так как интересно, как при применении одних и тех же чисел, но при разных операциях, меняется результат

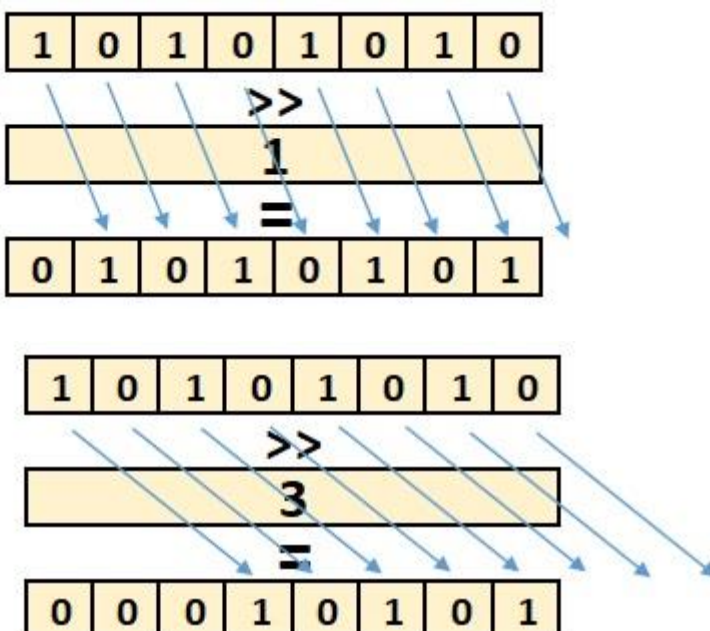


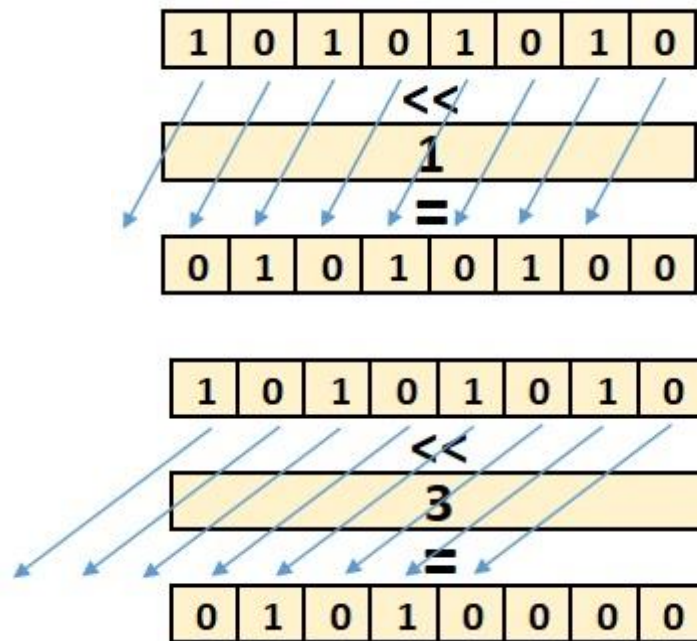
Теперь поговорим о битовых сдвигах. Хотя, здесь и говорить-то, собственно, не о чем. Существует в С всего два типа битовых сдвига — влево << и вправо >>.

В результате таких операций все биты первого операнда смещаются в ту или иную сторону в зависимости от направления сдвига на количество бит, равное значению второго операнда.

Биты в количестве, на которое они сдвигаются, со стороны, противоположной направлению сдвига, заполняются нулями.

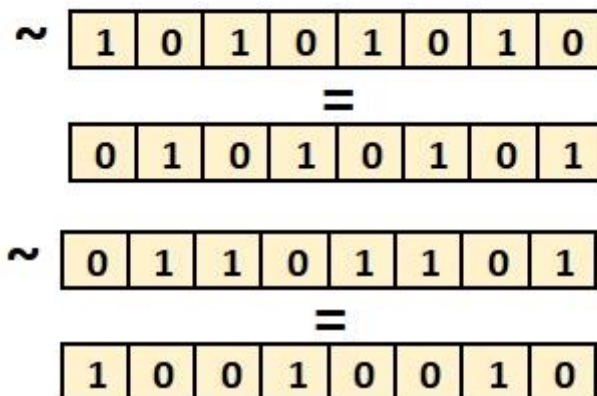
Рассмотрим несколько операций сдвига





Думаю, что с данными операциями также всё понятно.

Остался у нас один оператор, который производит логическое отрицание NOT, обозначаемый тильдой (~). Данный оператор, в отличие от пяти предшествующих, является унарным, и операция с помощью него производится только над одним операндом. В результате данной операции все биты инвертируются, то есть **единицы** превращаются в **нули**, а **нули** — в **единицы**. Данный оператор ставится в коде впереди операнда, над которым он производит операцию побитового инвертирования. Прделаем пару таких операций



Установка отдельного бита

Для установки отдельного бита, например порта D, применяется побитовая операция ИЛИ.

```
PORTD = 0b00011100; // начальное значение
PORTD = PORTD | (1<<0); применяем побитовую ИЛИ
PORTD |= (1<<0); // сокращенная форма записи
PORTD == 0b00011101; // результат
```

Эта команда выполняет установку нулевого разряда, а остальные оставляет без изменений. Для примера установим еще 6-й разряд порта D.

```
PORTD = 0b00011100; // начальное состояние порта
PORTD |= (1<<6);
PORTD == 0b01011100; // результат
```

Чтобы записать единицу сразу в несколько отдельных бит, например нулевой, шестой и седьмой порта B применяется следующая запись.

```
PORTB = 0b00011100; // начальное значение
PORTB |= (1<<0) | (1<<6) | (1<<7);
PORTB == 0b11011101; // результат
```

Сброс (обнуление) отдельных битов

Для сброса отдельного бита применяются сразу три ранее рассмотренные команды: \ll ; $\&$; \sim .

Давайте сбросим 3-й разряд регистра PORTC и оставим без изменений остальные.

```
PORTC = 0b00011100;
PORTC &= ~(1<<3);
PORTC == 0b00010100;
```

Выполним подобные действия для 2-го и 4-го разрядов:

```
PORTC = 0b00111110;
PORTC &= ~((1<<2) | (1<<4));
PORTC == 0b00101010;
```

Проверка разряда на наличие логического нуля (сброса) с if

```
if ((PIND & (1<<3)) == 0)
{
    Код1;
}
else {
    Код2;
}
```

Если третий разряд порта D сброшен, то выполняется Код1. В противном случае, выполняется Код2.

Аналогичные действия выполняются при и такой форме записи:

```
if (~PIND & (1<<3))  
  
{  
  
Код1;  
  
}  
  
else {  
  
Код2;  
  
}
```

Проверка разряда на наличие логической единицы (установки) с if

```
if ((PIND & (1<<3)) != 0)  
  
{  
  
Код1;  
  
}  
  
else {  
  
Код2;  
  
}
```

Аналог:

```
if (PIND & (1<<3))  
  
{  
  
Код1;  
  
}  
  
else {  
  
Код2;  
  
}
```

Теперь мы можем изменить нашу программу с помощью побитовых операций.

```
#include <avr/io.h>
```

```

int main(void)
{
    DDRD  |= (1<<3); //PD3 - выход
    DDRA  |= (0<<1); //PA1 - вход
    PORTA |= (1<<1); //Вход с подтяжкой

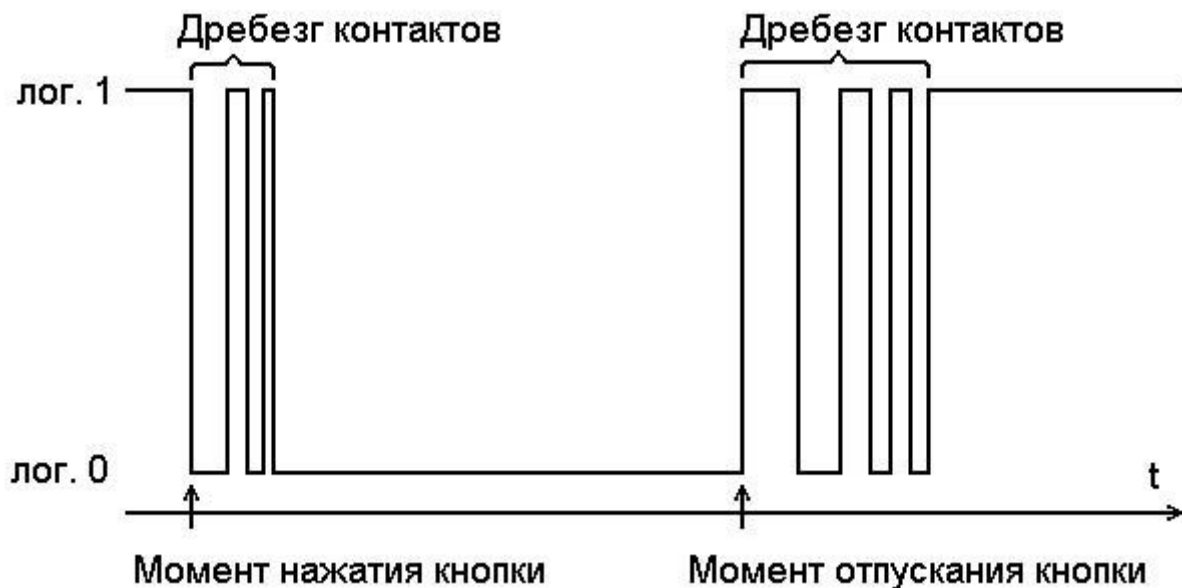
    while (1)
    { /*Используем логическую операцию И, если кнопка разомкнута,
      то на входе лог1 1 * 1 = 1 != 0
      Условие if - ложь, светодиод не горит.
      Когда кнопка замкнута на входе PINA лог0 0 * 1 = 0,
      то условие выполняется и диод загорается. */
        if((PINA & (1<<1)) == 0 )
        {
            PORTD |= (1<<3);
        }
        else
        {
            PORTD &= ~(1<<3);
        }
    }
}

```

Примечание: операция `DDRD |= (1<<3);` аналогична `DDRD |= (1<<PD3);`

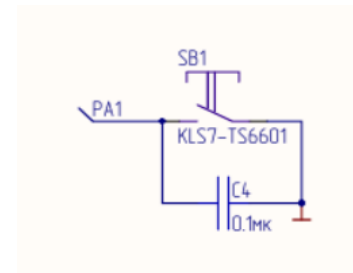
Борьба с дребезгом контактов.

Дребезг контактов – это паразитное явление, которое вносит проблемы преимущественно в электронных схемах. Его суть заключается в повторном многократном и ложном прерывании и подаче сигнала на вход. В результате система, которая его принимает, неверно реагирует. Давайте более подробнее рассмотрим причины дребезга контактов и способы борьбы с ним.



Чтобы устранить дребезг контактов, возможно использовать аппаратное или программное решение. К аппаратным решениям относится:

1. Установка конденсаторов параллельно входу. Тогда может снижаться быстродействие реакции на нажатие при слишком большой ёмкости и неполного устранения дребезга при слишком маленькой.
2. Введение триггеров Шмидта во входную цепь устройства. Более сложное решение, которое затруднительно для реализации в ходе доработки уже готового изделия, но и более технологичное и совершенное.



На нашей отладочной плате параллельно кнопка установлены конденсаторы, поэтому для надёжности нужно реализовать программный антидребезг. Суть алгоритма проста: с помощью переменной счётчика задерживаем выполнение операции после нажатия или отпускания кнопки.

```
#include <avr/io.h>

int main(void)
{
    DDRD  |= (1<<PD3); //PD3 - выход
    DDRA  |= (1<<1); //PA1 - вход
    PORTA |= (1<<1); //Вход с подтяжкой

    unsigned int butCnt = 0; // Переменная-счётчик
    while (1)
    {
        /*
        Используем логическую операцию И, если кнопка разомкнута, то на входе
        лог1 1 * 1 = 1 != 0
        Условие if - ложь, светодиод не горит. Когда кнопка замкнута на входе
        PINA лог0 0 * 1 = 0,
        то условие выполняется и диод загорается.
        */
        if(( PINA & (1<<1)) == 0 )
        {
            // Борьба с дребезгом
            // Обработка нажатия
            if (butCnt < 10)
            {
                butCnt++;
            }
            else
            {
                PORTD |= (1<<3);
            }
        }

        // Обработка отпускания
        else
        {
            if (butCnt > 0)
            {
                butCnt--;
            }
            else
            {
                PORTD &= ~(1<<3);
            }
        }
    }
}
```