**AVR Урок 5. Мигающий светодиод**

**Урок 5**

**Мигающий светодиод**

В данном уроке мы научимся программировать в цикле, а также использовать задержку исполнения кода, в результате чего наш светодиод станет более живым. Также мы поработаем с некоторыми логическими операциями и с операциями сдвига бита в байте.

Для начала мы создадим новый проект. Создадим мы его приблизительно по той же схеме, как и напрошлом занятии, за исключением только лишь, того, что назовём мы его **Test02**.

Также после запуска настроим в качестве отладчика симулятор. Также, чтобы нам не мучиться с кодом и не вводить его заново, мы его скопируем с прошлого занятия, открыв в проекте с прошлого занятия файл *main.c* в любом редакторе и скопировав весь код в буфер обмена. Удалим весь код из файла main.c в новом проекте и вставим из буфера обмена код.

Теперь начнем исправления. То есть мы создадим новый код, а старый у нас будет нетронутым в прошлом проекте. Порт D мы полностью весь установим в 0, изменив 1 на 0 в следующей строке

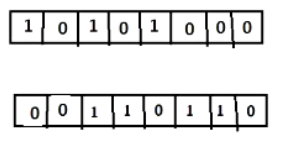
PORTD = 0b00000000;

В принципе, у нас уже все ножки порта по умолчанию установлены в ноль, но все равно сделаем это для порядка.

Дальнейший код мы уже будем писать в теле бесконечного цикла и весь этот код, который мы там напишем будет выполняться постоянно и циклично.

Но прежде, чем мы туда начнем писать код, я хотел бы немного рассказать вам о логических операциях, без которых в наш век никуда. Также они нам пригодятся и в коде. Первым делом мы расмотрим четыре основные операции.

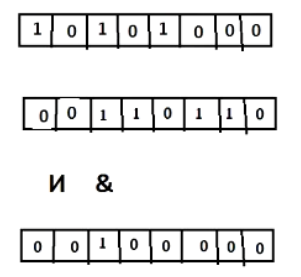
Логические операции мы будем производить над восьмибитовыми числами или байтами, которые мы представим в виде прямоугольников, состоящих из восьми ячеек. Первые три операции будут выполняться над двумя числами. Возьмём, к примеру вот такие:



Все три операции делаются побитно, то есть логически сравниваются одноимённые биты — нулевой с нулевым, первый с первым и так далее.

Первая — это операция "**И**", обозначаемая в языке C/C++ знаком "**&**".  Любой бит в результате будет установлен в**единицу**только в том случае, если оба соответствующих ему бита в операндах будут **единицами**. В любом другом случае мы получаем **ноль**.

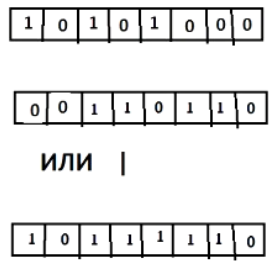
Поэтому результат у нас будет следующий



То есть в единицу у нас установлен в нашем случае только пятый бит, потому что пятый бит установлен в единицу как в первом операнде, так и во втором.

Сделующая логическая операция — это операция "**ИЛИ**", обозначаемая вертикальной чертой "**|**". В результате данной операции мы получим **единицу**, если хотя бы один бит в сравниваемых одноименных битах будет установлен в **единицу**. Таким образом, **ноль** мы получим только тогда, когда в обеих сравниваемых битах будет **ноль**.

Поэтому здесь будет уже вот такой результат



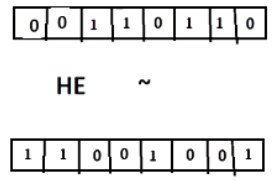
Третий вид операции — операция "**Исключающее ИЛИ**". Эта операция обозначается знаком "**^**". Здесь шансы у бита стать нулём или единицей уравниваются. Единицей результирующий бит станет тогда, когда сравниваемые биты обязательно будут разными — один будет **нулём**, другой — **единицей**. А нулём будет результирующий бит тогда, когда сравниваемые биты будут одинаковыми.

Итак, получим следующий результат



А четвертая операция — операция "**НЕ**", обозначаемая знаком тильда — "**~**", проделывается над одним байтом. В результате данной операции все биты меняются обратный. То есть **ноль**становится **единицей**, а **единица**— **нулём**. Данную операцию ещё называют операцией **инвертирования**.

Вот такой вот получим результат



Теперь вернёмся к коду. Напишем следующую очень непонятную на первый взгляд команду

while(1)

{

**PORTD |= (1<<(PORTD0));**

Первый непонятный оператор в данной команде — это сдвоенный оператор "**|=**". Такого рода операторов существует несколько. В данном случае сначала результат, который получится в правой части, сразу не присваивается переменной. которая находится в правой части, а складывается логически по "**ИЛИ**" со значением, хранящемся до этого в перменной, которая находится справа. Затем уже результат, полученный после применения данной логической операции, присваивается переменной, находящейся в правой части. Также существуют подобные операции типа "**+=**и "**-=**", которыми мы в дальнейшем будем очень часто пользоваться.

Теперь нам осталось этот результат получить. А мы даже не знаем, что за оператор у нас в правом выражении в виде стрелочек. А это один из двух вариантов битовых сдвигов, которые могут осуществляться над числами.

Битовые сдвиги бывают двух видов:

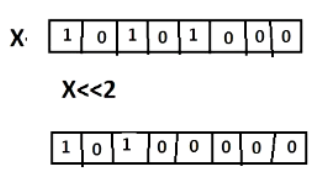
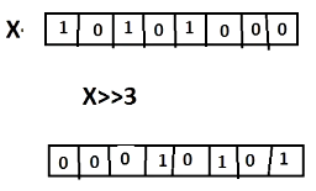
"**<<**" — сдвиг влево,

"**>>**" — сдвиг враво.

Давайте остановимся на них немного поподробнее.

Битовый сдвиг сдвигает все биты числа вправо или влево на количество положений, находящемся в числе справа. Число может быть разное. В результате сдвига самые крайние биты, находящиеся в стороне, в которую сдвигаем исчезают вникуда, а биты, находящиеся в противоположной стороне, в которые сдвигаться нечему, то есть они самые крайние, заполняются нулями.

Давайте проделаем данные операции над определённым байтом

Ну, я думаю, со сдвигом всё ясно. Давайте проделаем данную операцию над нашим примером в коде

Единичка, стоящая слева — это число, над которым проводится операция, а не число, на которое мы сдвигаем байт побитно, как очень многие путают. Запомните это накрепко! То есть в нашем случае — это единица или в двоичном выражении **0b00000001**. А **PORTD0** — это константа, которая определена в макросе в файле **io.h**. Вот разновидности данных констант

#define PORTD7 7

#define PORTD6 6

#define PORTD5 5

#define PORTD4 4

#define PORTD3 3

#define PORTD2 2

#define PORTD1 1

#define PORTD0 0

То есть в нашем случае — это ноль. Вот и получается, что мы единицу сдвигаем влево на ноль положений. В принципе, в результате операции ничего с нашей единицей не произойдёт и она также останется единицей, но зато код станет наглядным и мы будем видеть, что мы нулевую ножку порта установили в единицу.

Дальше мы делаем операцию "ИЛИ" между значением, находящимся в переменной **PORTD** и данным результатом, равным единице. Так как у нас в порте D в данный момент находится ноль на всех ножках, то мы только установим единицу на нулевой ножке, так как **0b00000000**  и **0b00000001**в результате данной операции дадут на **0b00000001**. Но если бы в регистре порта D находилось бы какое-нибудь другое число, то мы также бы установили бы единицу в нулевом бите, а остальные биты бы не тронулись. То есть мы в результате команды получили и лояльную к остальным лапкам операцию, и в то же время эта операция у нас имеет очень наглядный и читабельный вид.

Вообще, зажгли мы данный светодиод. Но прежде чем нам его погасить, мы обязаны применить соответствующую задержку, так как если нам сразу написать команду, обнуляющую бит, то она его обнулит практически мгновенно. Для задержки имеются две команды

\_delay\_ms(число в милисекундах)

\_delay\_us(число в микросекундах)

Также чтобы воспользоваться функциями задержки, нам необходимо подключить ещё один заголовочный файл в начале модуля

#include <avr/io.h>

**#include <util/delay.h>**

Мы воспользуемся первой командой и применим задержку на 500 милисекунд

PORTD |= (1<<(PORTD0));

**\_delay\_ms(500);**

Теперь можно погасить светодиод. Делается это с помощью следующей команды

\_delay\_ms(500);

**PORTD &= ~(1<<(PORTD0));**

Здесь мы видим, что у нас команда похожа на нашу предыдущую, хотя есть некоторые существенные изменения. Во-первых наш логический оператор превратился в "**И**", а также мы применили инверсию результата справа.

Давайте разберём данную команду. Результат справа нам известен — это **0b00000001**, если мы его инвертируем то получим соответственно **0b11111110**. Тепрь его нужно умножить логически на значение в переменной справа. В данный момент там у нас также **0b00000001**. Соответственно получим мы все нули, так как сравниваемые биты в операндах у нас разные. А если бы было какое-то другое чесло в переменной PORTD, то у нас бы сбросился только нулевой бит, остальные бы не тронулись. Поэтому вот такой операцией в будущем мы и будем сбрасывать определенные биты в регистрах и переменных. Также можно сбрасывать сразу несколько битов или устанавливать, но об этом позже, а то и так мы слишком много уже узнали. Осталось нам добавить ещё такую же задержку, чтобы светодиод находился в погасшем состоянии столько же. сколько и в светящемся

PORTD &= ~(1<<(PORTD0));

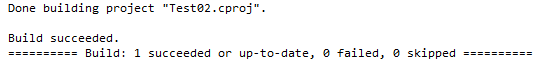
**\_delay\_ms(500);**

Но это ещё не всё. Чтобы задержка корректно работала, мы обязаны ещё в начале модуля или файла main.c написать макрос, говорящий компилятору о том, с какой частотой работает наш контроллер, иначе он неправлиьно применит задержку, и нам даст предупрежедение. Напишем данный макрос

**#define F\_CPU 8000000**

#include <avr/io.h>

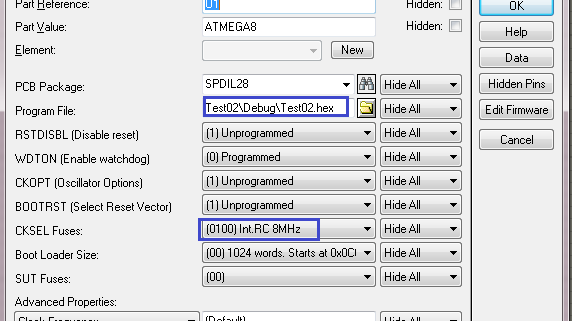
Соберём проект. У нас ошибок и предупреждений нет



Теперь мы можем посмотреть результат работы сначала в протеусе, затем на живом светодиоде и контроллере.

Чтобы нам заново не создавать схему, мы файл с проектом для проетуса из папки с проектом из прошлого урока скопируем в папку с новым проектом и переименуем в **Test02.pdsprj**.

Запустим его. Щёлкнем двойным щелчком по изображению контроллера и выберем там другую прошивку, соответствующую новому проекту, а также выставим в свойствах соответствующую частоту, иначе светодиод будет мигать в 8 раз медленнее



 Запустим проект и увидим, что светодиод у нас мигает.

Затем прошьём настоящий контроллер, и также убедимся, что светодиод мигает и там.