**AVR Урок 3. Пишем код на СИ. Зажигаем светодиод**

**Урок 3**

**Пишем код на СИ. Зажигаем светодиод**

Сегодня мы научимся писать код на C в среде Atmel Studio, для примера мы возьмём тот же проект, который мы создали на прошлом занятии.

Прежде чем мы начнём непосредственно заниматься написанием кода, мы изучим те строки кода, которые нам уже сгенерировала студия в нашем файле Test01.c.

В самом начале кода мы видим строку в виде следующей директивы

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

Посмотрим определение данной директивы

*Директива****#include****просит препроцессор (компилятор) включить файл, объявленный после нее в исходный код. Имя файла может заключаться либо в треугольные скобки****<>****либо в кавычки****””****. Треугольные скобки используются в случае включения в код файла из самой среды разработки, а кавычки – пользовательского файла.*

***#include <имя файла>***

В нашем случае в текст кода включается файл**io.h**. Если мы откроем данный файл, то мы увидим, что в зависимости от каких-то условий в наш код включаются ещё определённые заголовочные файлы. В нашем частном случае условием является использование нами определённого микроконтроллера. Так как мы используем Atmega8a, то соответственно включится следующий файл:

#elif defined (\_\_AVR\_ATmega8A\_\_)  
# include <avr/**iom8a.h**>

В данном файле находятся макросы для векторов прерываний, для определённых шин. Что такое макросы и как их использовать, мы разберём немного позднее. Движемся дальше по файлу **io.h** и видим, что в него также включены ещё 3 файла, но нам будет интересен следующий файл:

#include <avr/**portpins.h**>

В данном файле находятся также макроподстановки для наших лапок портов и прочей периферии, чтобы нам было удобнее писать и читать наш код. Откроем данный файл и изучим в нём данные строки

/\* Port Data Register (generic) \*/  
**#define PORT7  7**  
**#define PORT6  6**  
**#define PORT5  5**  
**#define PORT4  4**  
**#define PORT3  3**  
**#define PORT2  2**  
**#define PORT1  1**  
**#define PORT0  0**

Данные строки нам говорят о том, что, например, если при компиляции проекта препроцессор (интерпретатор команд) встретит в коде слово **PORT4**, то он его сразу заменит на цифру 4.

Тем самым мы постепенно с вами подошли к изучению ещё одной директивы

*Директива****#define***

*Просит препроцессор (компилятор) в случае появления в тексте кода буквосочетания 1 заменить его на буквосочетание 2.*

*Данная директива используется для удобства чтения кода.*

***#define <буквосочетание 1> <буквосочетание 2>***

Вернёмся в наш код. Далее мы видим следующее

**int main(void)**  
**{**

**…**

**}**

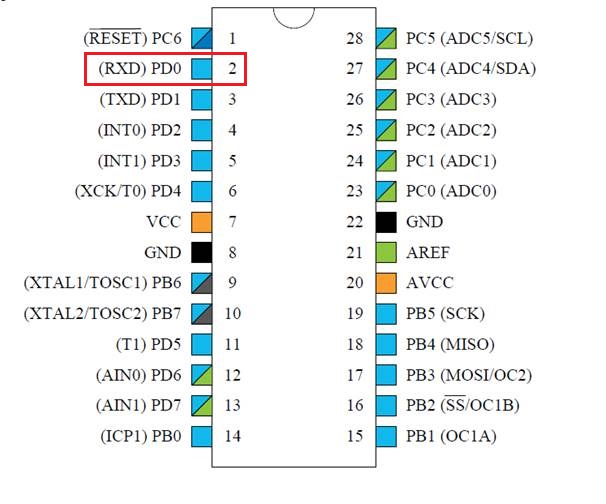
То, что мы с вами наблюдаем в языке C и C++ называется **функция**. Функциями мы будем пользоваться постоянно. Функция — это такая подпрограмма, которая вызывается из какого-нибудь участка кода. Самое важное у функции — это её имя (в нашем случае **main**). По имени мы будем вызывать функцию из другого участка кода. Также у функции существуют входные аргументы, возвращаемые аргументы, а также тело. Входные аргументы находятся сразу после имени в скобках и перечисляются один за другим, а разделяются запятыми. В нашем случае стоит один тип «void», обозначающий, что у нашей функции вообще нет входных аргументов. Поэтому если мы подобную функцию будем вызывать в другом участке кода, то мы в скобках вообще ничего не ставим. Возвращаемый аргумент у функции один. Если нам потребуется больше чем один тип переменных, то мы будем пользоваться глобальными переменными, о которых мы узнаем позже. Изучение переменных вообще не входит в рамки наших уроков, как правило это объясняется непосредственно в уроках и литературе по языкам программирования. Тип возвращаемого аргумента указывается перед именем функции. В нашем случае — это int (целочисленная знаковая переменная). Также у функции существует тело — это участок кода, находящийся между открывающей и закрывающей фигурными скобками. Вот этот участок кода и будет выполняться в случае вызова функции.

Функцию **main** мы явно нигде не вызываем. Это главная функция нашего приложения, недаром она и называется main, что по английски значит главный. Встретив данное имя, компилятор и начинает выполнение программы с данного места. То есть это своего рода входная точка нашей программы. Отсюда всё и начинается. Сюда мы и начинаем писать свой код.

Давайте же что-нибудь сюда и напишем. Мы пока не будем обращать внимание на строки, уже содержащиеся в теле данной функции.

У программистов, которые пишут программы под ПК, начинать занятия принято с вывода строки «Hello World!», а у тех программистов, которые пишут под мелкие чипы, принято начинать работу с подключения и управления свечением светодиодами. Затем они учат их мигать просто, мигать по очереди, а уже после этого приступать к программированию каких-то более серьёзных вещей. Мы также не будем отступать от данного правила.

Давайте сначала подключим светодиод к какой-нибудь ножке контроллера, например к ножке 0 порта D



У порта D, как мы видим из данной распиновки, существует как раз 8 ножек, что соответствует байту (8 бит). Также как биты в байты, ножки портов отсчитываются от 0.

Напишем мы сначала следующую строку

int main(void)  
{

**DDRD = 0xFF;**

Впредь я буду выделять жирным шрифтом то, что мы добавляем в код или изменяем в коде. Так как я кроме этого использую в написании блогов ещё кусок кода, находящийся или до или после написанного кода для того, чтобы читателю было понятно, куда именно мы пишем код. То есть то что не обозначено жирным шрифтом, это уже есть в коде.

**DDRD** — это команда, которая устанавливает состояние лапок порта D. Состояние лапки порта — это то, в каком направлении данная лапка будет работать — на выход или на вход, что соответствует установке определённого бита в **0** или в **1**. Но так как мы будем зажигать светодиод, мы должны на определённой ножке порта выставить высокий логический уровень (в нашем случае 5 вольт). А чтобы управлять уровнями ножки, она должна быть определена, как работающая на выход или на вывод. То есть состоянием лапки мы будем управлять из контроллера, а не из внешнего источника уровня. Так как у нас лапка нулевая, то и бит мы и должны выставить в ноль нулевой бит нашего байта, соответствующего нашему порту D. Так как мы не пользуемся сегодня остальными лапками порта, то нам их состояние будет не важно и мы выставляем все лапки портов на вывод. Поэтому мы присваиваем переменной **DDRD** значение **0xFF**. Данное значение мы написали в шестнадцатиричном виде. Этот вид очень удобен для программистов, так как визуально о многом говорит. FF — это в десятичной системе 255, а в двоичной — 11111111, что говорит о том, что все биты в данном числе выставлены в единицу. Также мы видим, что наша строка заканчивается точкой с запятой (**;**). Данный оператор — это разделитель команд, так как в одной строке мы можем писать не обязательно одну только команду, а можем и несколько, если они небольшие. Также в данной строке мы видим оператор «=» (знак равенства). Данный оператор заставляет наш препроцессор присвоить значение, находящееся справа возле него переменной, находящейся слева.

Ну, вообщем, переключили мы весь наш порт в состояние вывода данных. Теперь осталось включить на лапке **PD0** высокий логический уровень. Сделать это мы можем следующей командой:

DDRD = 0xFF;

**PORTD = 0b00000001;**

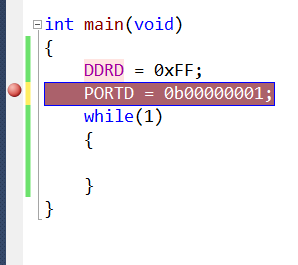
Данная команда или переменная PORTD управляет записью или считыванием значений в порт или из порта в зависимости от состояния. То есть данной командой мы включили нулевую лапку в высокое логическое состояние (в единицу). Здесь мы с вами уже попробуем использовать написание значения в двоичном виде. Чтобы писать значения в данном виде, мы используем префикс 0b. Данный вид удобен тем, что мы здесь видим полностью, как выглядит наш байт побитно. Лапки портов в байте, также как и биты считаются справа налево. То есть данной командой мы выставили в высокое состояние нулевую лапку порта D, а остальные мы выставили в низкое. Вообщем, арифметическо-логическое устройство микроконтроллера сначала включит все ножки порта на выход, а затем установит на нулевой ножке высокое логическое состояние, и после этого у нас должен будет зажечься светодиод, так как через токоограничивающий резистор мы его анодом подключим к данной ножке, а катодом к общему проводу. Тем самым на контактах светодиода появится разность потенциалов, которая заставит его светиться. Кроме написанных нами двух строк далее в коде присутствует команда **while**. Данная команда является командой условного цикла.

PORTD = 0b00000001;  
**while(1)**  
**{**

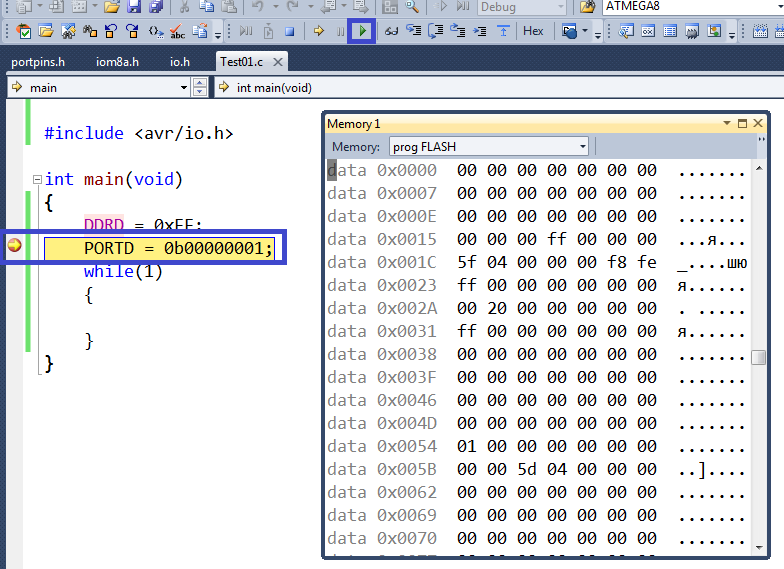
**}**

В скобочках указывается условие, которое должно либо выполняться либо не выполняться. Также как у функции есть тело, то у условия также есть тело, также заключенное в фигурные скобки. И код, находящийся в теле цикла, будет выполняться бесконечно, пока условие, находящееся в скобках будет выполняться, то есть будет истинным. Как только условие перестанет выполняться, а проверяется это тогда, когда код выполнится до конца (до закрывающей фигурной скобки), то мы выходим из цикла и код продолжает выполняться уже дальше тот, который находится уже не в теле цикла, а после закрывающей фигурной скобки. А истина в информатике — это TRUE или 1. Поэтому в данном случае цикл будет бесконечным, так как там стоит единице, а единица всегда равна единице. Бесконечный цикл организован для того, чтобы код, написанный для контроллера, выполнялся постоянно, то есть чтобы наш контроллер постоянно работал и не останавливался. В нашем случае тело пустое, и наш контроллер, вернее его АЛУ, будет всё время висеть в данном цикле и ничего не делать до тех пор, пока мы не отключим питание, либо нам его не отключат в розетке, либо, не дай Бог, сгорит контроллер. То есть светодиод наш будет светиться постоянно.

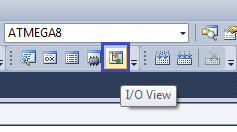
Сегодня мы не будем пробовать нашу программу прошивать в микроконтроллер, и даже не будем пробовать ещё в виртуальном контроллере, то есть в программе симуляции, а попробуем симуляцию запустить в самой студии, так как на прошлом занятии мы в качестве отладчика и выбрали симулятор. Двойным щелчком мыши либо клавишей F9 мы установим точку останова на команде PORTD = 0b00000001; и, когда мы запустим отладку, то отладчик, как только увидит данную точку, должен будет в этом месте остановить выполнение программы, и мы сможем посмотреть, какие уровни и где у нас установились.



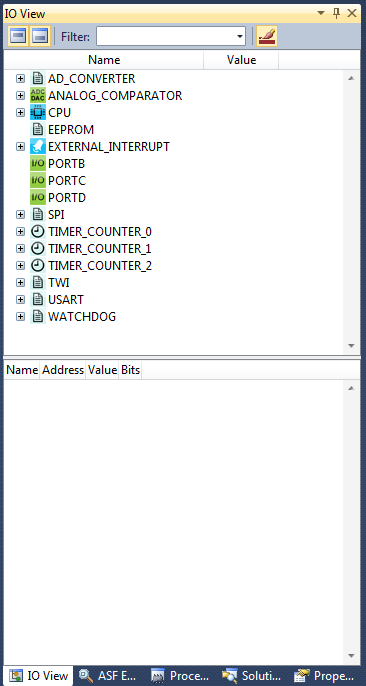
Чтобы запустить отладку, мы нажмём кнопку в виде зелёного треугольника и дождёмся, когда отладчик остановится на нашей красной точке



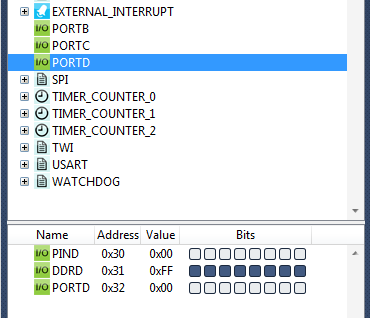
Здесь мы наблюдаем, что ещё у нас открылась карта нашей памяти, в которой нам пока ещё ничего не понятно. Если бы мы писали на ассемблере либо на машинном коде, то нам бы это было понятнее. Поэтому нас интересует другая. Карта. Для этого мы нажмём вот эту кнопочку



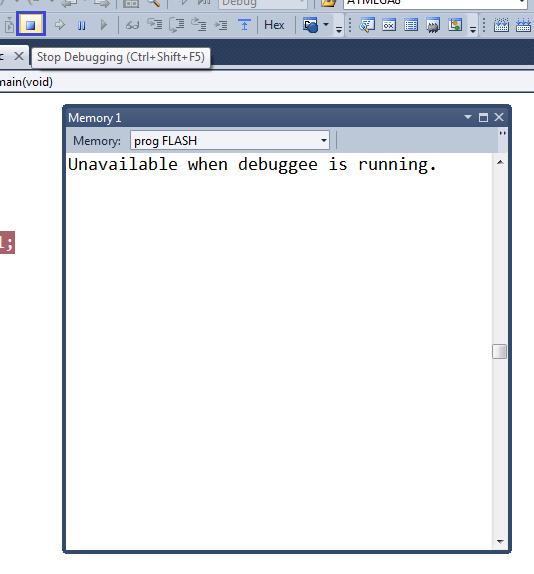
Данная кнопочка (I/O View) откроет нам окно с данными наших портов ввода-вывода и прочей периферии



Нажмем в данном окне на строку PORTD и увидим в нижней половине окна, что весь наш регистр DDRD, отвечающий за направление отдельных ножек порта выставился весь в единички, то есть на выход



А дальше уже проблема. Чтобы нам посмотреть, как сработает следующая команда, которая включит нам нулевую ножку, отладчику необходимо остановиться на следующей строке кода, а у нас её нет, у нас только бесконечный цикл, который для отладчика — не строка. Поэтому мы пока остановим отладчик следующей кнопкой

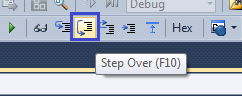


И мы напишем какую-нибудь ещё строку, например, мы захотим, чтобы данный светодиод у нас погас, а зажегся следующий, который присоединен к следующей лапке, чтобы создать эффект бегущего огня

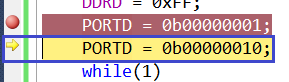
PORTD = 0b00000001;

**PORTD = 0b00000010;**

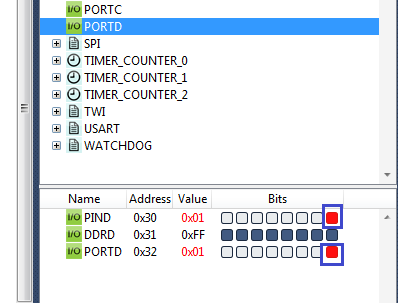
Конечно всё это на практике у нас не сработает, так как мы не успеем заметить свечение предыдущего светодиода. Чтобы задумка заработала практически, мы должны ещё с вами включить между данными командами задержку, а это тема уже других более поздних занятий. Но тем не менее мы данную команду включим, чтобы отладчику было где остановиться. Затем мы запустим заново отладку. Точка останова у нас также находится пока на той строке, на какой и была до этого. Запустим опять отладчик. Собирать проект перед отладкой необязательно, так как отладчик сам его пересоберет. Дожидаемся остановке отладчика на точке. В окошке с вводом-выводом опять нажмём на строке с нашим портом. Дальше начинаем шагать по программе. Для этого нажимаем следующую кнопку или функциональную клавишу F10, о чем нам подсказывает студия, как только мы подносим указатель мыши к данной кнопке



Теперь отладчик остановится на следующей строке



И теперь в окне ввода-вывода мы видим уже следующую картину



Мы видим, что самый левый бит, соответствующий нулевой ножке порта переключился в высокое логическое состояние, причём мы это видим не только в регистре PORTD, но и в регистре PIND, который отвечает за считывание состояния ножек порта D при его работе на ввод. Вот таким вот образом мы и отлаживаем наши программы.

Остановим наш отладчик, уберём все брекпоинты, а, самое главное, не забываем удалить ненужную команду, которая включает другую лапку.

После этого текст кода у нас должен будет остаться вот таким

