**AVR Урок 12. LCD индикатор 16×2. Часть 1**

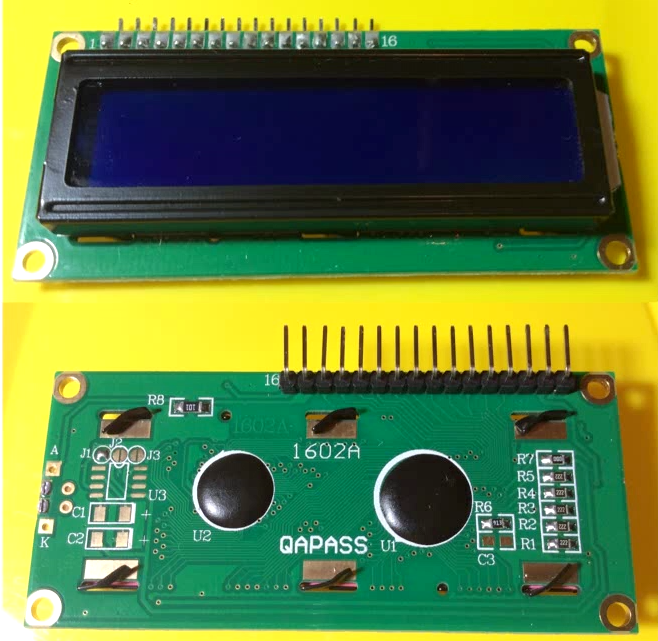
**Урок 12**

**Часть 1**

**LCD индикатор 16×2**

Сегодня мы начнём изучение **жидкокристаллического индикатора символьного**, который способен выводить определённые символы в две строки по 16 символов в каждую. Изучать мы данный индикатор будем с целью его подключения к микроконтроллеру AVR и управления им.

Выглядит индикатор, с которым мы будем работать вот таким вот образом



То есть индикатор выполнен в виде модуля, в котором установлен контроллер **HD44780**, предназначенный для управления дисплеем. Также, как мы видим, в данном модуле существуют 16 контактов, которые я припаял самостоятельно и к которым мы и будем подсоединяться с контроллера с помощью проводов. Поставляется данный модуль без контактов. На их месте существуют контактные технологические отверстия. Скорее всего, сделано это для того, чтобы мы сами могли выбрать тип контактов, которые нам будут удобны для работы с данным модулем.

Вроде и кажется, что 16 контактов — это много, но на самом деле не так уж и много. За счет контроллера **HD44780**, установленного в модуле, нам не придется подводить по 8 и более контактов к каждому из 32 символов, этим как раз и займётся данный контроллер. Мы лишь только будем давать ему определённые команды и посылать определённые данные. Хотя это также сопряжено с определёнными трудностями программирования данного устройства, но на это и существует урок, чтобы данные трудности как-то преодолеть.

Также следует отметить, что типов таких модулей существует несколько и распиновка контактов может незначительно отличаться. Поэтому я и показываю в приемлемом качестве именно тот дисплей, который будем исользовать мы во избежание путаницы из за неправильного подключения, а в последствии и выхода из строя дисплея. Данные случаи уже имели место, нам этого совершенно не нужно.

Например, модуль компании Winstar на 1 ножке имеет анод, а наш китайский дисплей, который стоит дешевле, в отличие от Winstar, на данной ножке имеет катод.

Поэтому будьте внимательны и читайте документацию на тот именно модуль, который будет у вас.

Иногда ножки подключения индикаторов нумеруются на плате, как, например в том модуле, который мы будем сегодня использовать



Ну раз уж мы увидели надписи на ножках модуля, давайте тогда разберёмся, какая ножка для чего служит. Пойдём слева направо.

Ножка первая —  VSS — это общий провод или "земля"

2 —  VDD — питание.

3 — V0 — это ножка, с помощью которой регулируется контрастность дисплея. То есть контрастность дисплея будет зависеть от поданного напряжения на данную ножку. Как правило берётся переменный резистор на 10 килоом, подключенный крайними ножками на общий провод и на питание, а с центральной ножки данного резистора провод идёт как раз на ножку V0 и посредством регулировки движка резистора мы и регулируем контрастность дисплея в модуле.

4 — RS — это такая хитрая ножка, с помощью которой контроллер дисплея будет "знать", какие именно данные нахдятся на шине данных. Если мы подадим на данную ножку логический 0, то значит будет команда, если 1 — то это данные.

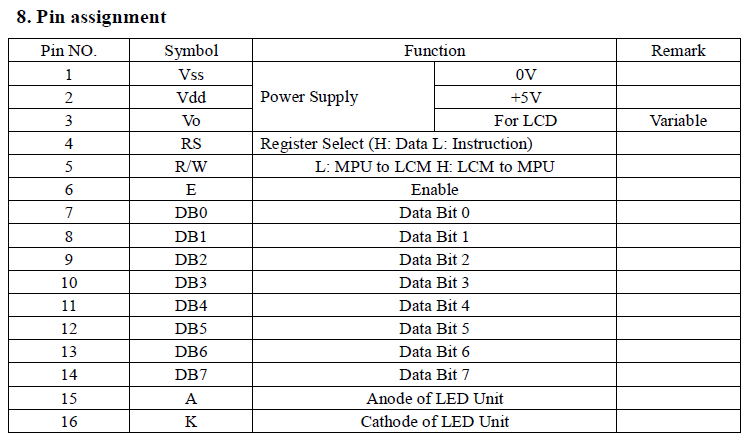
5 — RW — данная ножка в зависимости от логического состояния на ней говорит контроллеру дисплея, будем мы с него читать или будем мы в него писать данные. Если будет 0 — то мы в контроллер дисплея будем писать, а если 1 — то будем читать данные из контроллера дисплея. Данная функция используется редко. Как правило мы всегда только пишем данные в дисплей. Чтение обычно требуется для того, чтобы определить, что дисплей принял наши данные, либо чтобы определить состояние. Но существуют определённые тайминги, позволяющие нам на слово "верить" котнроллеру дисплея, что он наши данные принял и обработал. Также читать мы можем из памяти дисплея, что, в принципе, незачем. Поэтому мы обычно соединяем данный контакт с общим проводом.

6 — E — это так называемая стробирующая шина, по спадающему фронту (когда 1 меняется в 0) на которой контроллер дисплея понимает, что именно сейчас наступил момент чтения данных на ножках данных D0 — D7, либо передачи данных из модуля в зависимости также от состояния ножки RW.

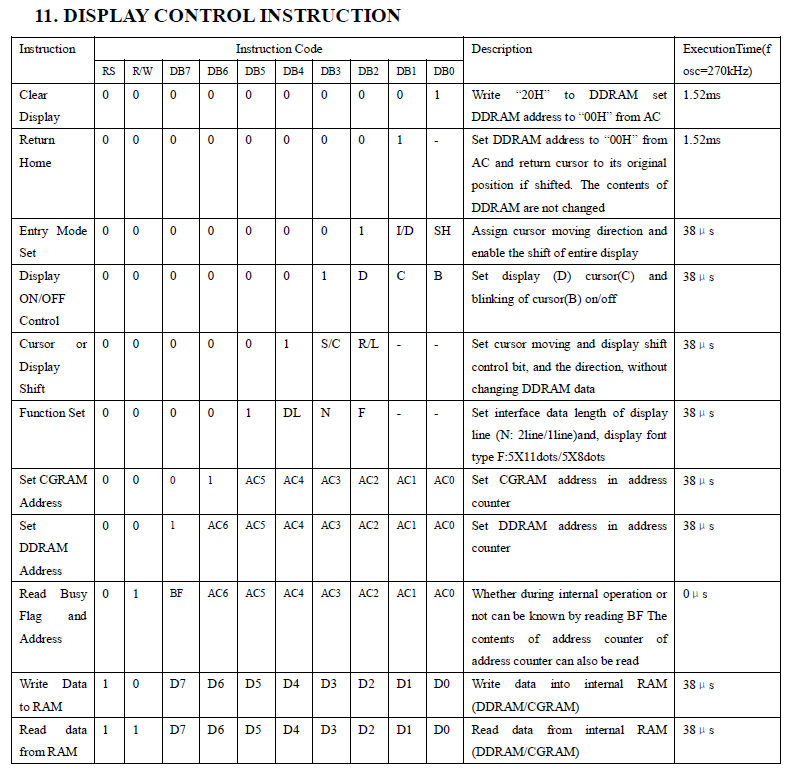
Ножки D0 — D7 — это параллельная восьмибитная шина данных, через которую и передаются или принимаются данные. Номера 0 — 7 соответствуют одноименным битам в байте данных. Но также есть ещё 4-битный способ передачи данных в контроллер и из контроллера дисплея, когда используются только ножки данных D4 — D7, а ножки D0 — D3 уже не используются. Как правило такой способ используется в целях экономии ножек порта и именно такой способ мы и будем сегодня использовать, так как мы теряем скорость вдвое, но у нас дисплей символьный и спешить нам некуда. В 4-битном режиме мы передаём или принимаем байт в 2 приёма по половинке, сначала старшую часть байта, затем младшую.

Ножки A и K — это анод и катод для подачи напряжения для питания светодиодной подсветки дисплея. Как правило можно питать от 5 вольт, но желательно поставить токоограничивающий резистор на 100 ом и скорее всего тогда подсветка дисплея "проживёт" дольше. Всё это обычно указывается в технической документации на дисплей.

Также данную информацию мы видим в технической документации на дисплей



Теперь немного ознакомимся, каким образом мы будем организовывать процесс общения с дисплеем. То есть каким образом мы будем этим процессом управлять. Ведь контроллер дисплея не "знает" что именно мы от него хотим. Для этого в даташите существует вот такая таблица, представляющая собой перечень команд и способы их реализации



В самой первой колонке данной таблице находятся сами команды. Следующие 2 колонки — это то, в каком состоянии должны в момент команды находиться ножки RS и RW. Дальнейшие 8 колонок показывают нам состояние ножек шины данных, затем идёт колонка с пояснениями к командам, то есть что именно с помощью данной команды мы достигнем. А затем в последней колонке находятся тайминги или временные интервалы, необходимые для того, чтобы та или иная команда или инструкция выполнилась. Причем оговорено, при какой именно частоте генератора это достигается. То есть, я так понимаю, что в модуле существует генератор, тактирующий работу контроллера дисплея, который настроен на определённую частоту и данная частота может быть разной. Поэтому желательно пробовать сначала с большим интервалом, то есть всё достигается на практике. Поэтому некоторые модули других моделей могут не работать с тем же кодом, который мы в данном уроке напишем. Опять же читайте внимательно техническую документацию именно к своему модулю.

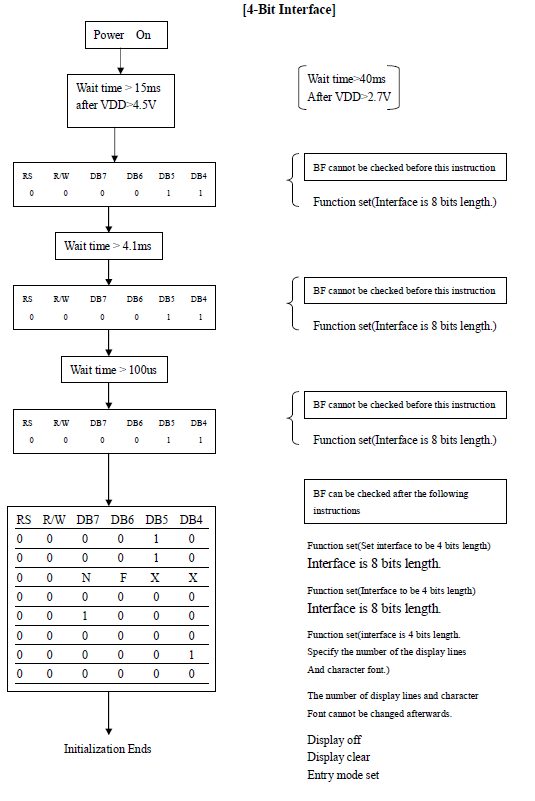
Например, первая команда **Clear Display** говорит сама за себя. Она очищает дисплей. Вообще за отображение на дисплее у нас отвечает оперативная память DDRAM, также существующая в контрроллере дисплея. Вот данную память как раз и очищает данная команда.

Остальных команд мы коснёмся несколько позднее.

Теперь идём дальше по технической документации. Дальше идёт объяснение каждой команды и назначение каждого бита, представленного в таблице. Мы можем заметить, что в таблице каждый используемый бит как-то называется.

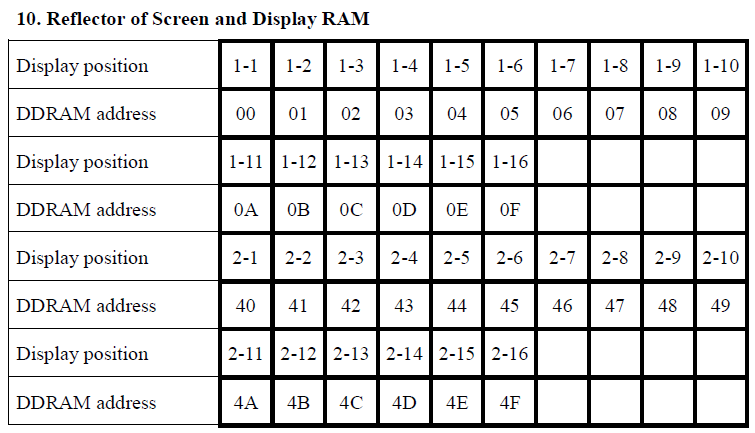
Дальше идёт объяснение процесса инициализации модуля дисплея. Инициализация любого активного устройства — это неотъемлимая часть программирования. Без первичной инициализации не будет работать ни одно устройство.

Сначала показана инициализация 8-битного режима, а затем 4-битного режима. Нам интересен именно последний способ. Поэтому посмотрим данную страничку



Мы видим, что всё здесь очень подробно рассказано и показано. Вот эту диаграмму мы и будем использовать, когда будем писать код инициализации дисплея. Опять же требование — 270 кГц частота работы генератора.

Также посмотрим организацию знакомест дисплея в памяти DDRAM. Это нам будет необходимо для написания функции позиционирования



Как мы видим, вторая строка находится в области видиопамяти через некоторый пропуск после первой. Во-первых, существуют дисплеи разной размерности, например у меня ещё есть дисплей 20х4 на том же контроллере, поэтому и пропуск. Также существует определённая команда, которая передвигает видимую часть памяти, это может быть использовано для подготовки символов в невидимой области, а затем путём передвижения невидимую область мы делаем видимой. Нам это пока не требуется. Если потребуется, то мы обязательно с этим разберёмся, ну либо для какого-то красивого скроллинга дисплея также может это потребоваться.

Подобная документация существует также не только на дисплей, а ещё и на контроллер. Я её также в конце статьи обязательно приложу к страничке.

А теперь наконец-то проект.

В отличие от предыдущих занятий сегодня у нас будет пустой проект. Обычно мы продолжали начатый и копировали код с прошлого занятия. Сегодня мы этим заниматься не будем и создадим пустой проект **Test09**. Создадим мы его таким же образом, как и все предыдущие, выбрав тот же контроллер, поэтому процедуру создания и настройки я показывать не буду. У нас и так урок обещает быть очень большим.

Для протеуса проект мы скопируем как и раньше, переименовав его в Test09, и выбрав путь к прошивке нового проекта, чтобы нам заново не искать контроллер, резисторы и прочие вещи, а также не настраивать контроллер.

Вернемся в наш проект в студию. Верхнюю часть файла, где мы объявляем частоту и подключаем библиотеки, мы, в принципе можем всё-таки их прошлого кода скопировать

**#define F\_CPU 8000000UL**

**#include <avr/io.h>**

**#include <avr/interrupt.h>**

**#include <util/delay.h>**

Скомпилируем код, чтобы у нас была хотя бы какая-то прошивка.

Напишем функцию port\_ini(). Под все ножки модуля дисплея мы будем использовать порт D. Так как режим у нас 4-битный, то нам вполне хватит ножек, даже останутся

**//—————————————-**

**void port\_ini(void)**

**{**

**PORTD=0x00;**

**DDRD=0xFF;**

**}**

**//—————————————-**

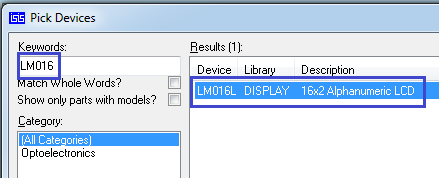
Также давайте данную функцию вызовем в функции main()

int main(void)

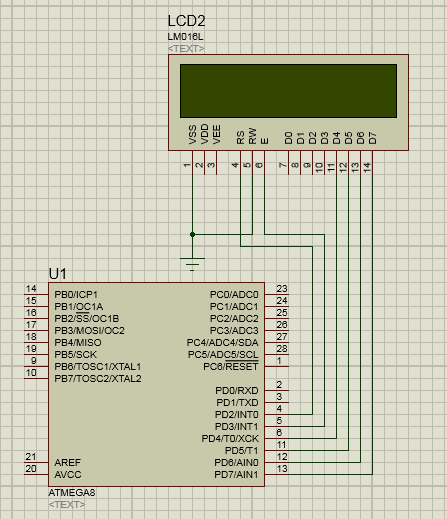
{

**port\_ini(); //Инициализируем порты**

Зайдём теперь в проект в протеус и найдем там дисплей в библиотеке компонентов



Подключим его следующим образом



Если мы запустим проект, то мы увидим всего лишь, что дисплей у нас просо будет светиться и всё.

Дальнейшую работу с данным дисплеем мы продолжим в следующей части.

**AVR Урок 12. LCD индикатор 16×2. Часть 2**

**Урок 12**

**Часть 2**

**LCD индикатор 16×2**

Сегодня мы продолжим изучение **жидкокристаллического индикатора символьного**, который способен выводить определённые символы в две строки по 16 символов в каждую.

Вернёмся в проект Atmel Studio и начнём думать, как нам начать писать инициализацию дисплея. По приведённой выше диаграмме из даташита мы видим, что мы через определённое время должны три раза дисплею передать двоичное число 11, и тогда контроллер дисплея "поймёт", что от него хотят именно того, чтобы он общался с нами в 4-х битном режиме и чтобы он в данный режим как-то переключился. Напишем функцию инициализации дисплея. Код данной функции мы разместим перед функцией main()

**//—————————————-**

**void LCD\_ini(void)**

**{**

**}**

**//—————————————-**

Вызовем данную функцию также в функции main()

port\_ini(); //Инициализируем порты

**LCD\_ini(); //Инициализируем дисплей**

Теперь начнём писать код в теле данной функции. Начнём с того. что судя по диаграмме инициализации из технической докуметации мы должны подождать сначла не менее 15 милисекунд

void LCD\_ini(void)

{

**\_delay\_ms(15); //Ждем 15 мс**

А дальше мы должны как-то передавать контроллеру дисплея байты. Но так как мы байты сразу передавать не можем, ибо у нас даже ножки данных половина не подключены, то передавать мы будем полубайты. Так как это не совсем простой процесс, предлагаю завести под это дело ещё одну функцию и назвать её **sendhalfbyte**. В качестве входного параметра мы ей будем передавать unsigned char, так как у нас 4-битных переменных не бывает, мы просто не будем использовать в данной переменной первые 4 бита. Разместим код данной функции над кодом функции LCD\_ini

**//—————————————-**

**void sendhalfbyte(unsigned char c)**

**{**

**}**

**//—————————————-**

Сначала мы сдвинем наш входной аргумент влево на 4 бита, так как работаем мы со старшими разрядами шины (4-7). И не просто сдвинем, а этому же аргументу и присвоим, написав после операции сдвига знак **равно**.

void sendhalfbyte(unsigned char c)

{

**c<<=4;**

Давайте ещё напишем несколько макроподстановок для удобства работы с линиями E и RS, то есть их включение и выключение. Зная, на каких ножках порта D данные линии находятся и используя багаж знаний предыдущих занятий нам будет это сделать очень легко

#include <util/delay.h>

//—————————————-

**#define e1 PORTD|=0b00001000 // установка линии E в 1**

**#define e0 PORTD&=0b11110111 // установка линии E в 0**

**#define rs1 PORTD|=0b00000100 // установка линии RS в 1 (данные)**

**#define rs0 PORTD&=0b11111011 // установка линии RS в 0 (команда)**

//—————————————-

Вернёмся в нашу функцию передачи полубайта, включим линию E, для того чтобы сказать дисплею, что мы будем передавать командуЗатем мы подождём 50 микросекунд, хотя достаточно судя по таймингам гораздо меньше, затем сотрём информацию на ножках 4-7 порта D, остальные ножки не трогаем, затем установим нужные биты на шине данных из переменной **c**, в которой находятся данные биты. Потом мы отключим линию E, вот тут как раз и получится отрицательный фронт, затем подождём ещё 50 микросекунд на всякий случай, хотя написано 38 для записи в DDRAM или в RAM

  c<<=4;

**e1; //включаем линию Е**

**\_delay\_us(50);**

**PORTD&=0b00001111; //стираем информацию на входах DB4-DB7, остальное не трогаем**

**PORTD|=c;**

**e0; //выключаем линию Е**

**\_delay\_us(50);**

**}**

Использовать данную функцию мы будем уже в [**следующей части**](https://narodstream.ru/avr-urok-12-lcd-indikator-16x2-chast-3/) занятия.

**AVR Урок 12. LCD индикатор 16×2. Часть 3**

**Урок 12**

**Часть 3**

**LCD индикатор 16×2**

Сегодня мы продолжим изучение **жидкокристаллического индикатора символьного**, который способен выводить определённые символы в две строки по 16 символов в каждую.

Вернёмся в проект Atmel Studio и, используя написанную в прошлой части функцию, передадим в контроллер дисплея двоичное число 11 три раза в функции LCD\_ini, также применяя соответствующие задержки, взяв их из даташита

\_delay\_ms(15); //Ждем 15 мс (стр 45)

**sendhalfbyte(0b00000011);**

**\_delay\_ms(4);**

**sendhalfbyte(0b00000011);**

**\_delay\_us(100);**

**sendhalfbyte(0b00000011);**

**\_delay\_ms(1);**

Дальше передаём двоичное число 10 таким же образом

\_delay\_ms(1);

**sendhalfbyte(0b00000010);**

**\_delay\_ms(1);**

В некоторых даташитах пишут, что данное число нужно передавать 2 раза, но работает и с 1 передачей. Вы на своих дисплеях можете поэксперементировать и с 2 передачами числа 10.

В принципе, уже контроллер дисплея должен "догадаться", что мы его просим переключиться в 4 битный режим, и следующая команда будет же с полноправным байтом, переданным поочередно и причем в этой команде уже будет конкретная команда перевода в 4-битный режим. Но для полнорпавных команд мы напишем другую функцию **sendbyte**, расположив ее код после функции sendhalfbyte.

/**/—————————————-**

**void sendbyte(unsigned char c, unsigned char mode)**

**{**

**}**

**//—————————————-**

В данную функцию мы будем передавать уже два аргумента, один — это данные, а другой это режим, то есть мы здесь будем говорить, данные мы будем передавать или команду. Начнем писать код функции.

Сначала мы с помощью условия узнаем, команда в нашу функцию пришла или данные, и среагируем на это установки в соответствующее состояние шины **RS**

void sendbyte(unsigned char c, unsigned char mode)

{

**if (mode==0) rs0;**

**else rs1;**

Добавим ещё одну переменную

else rs1;

**unsigned char hc=0;**

Сдвинем вправо на 4 пункта наш байт и отправим результат в данную переменную. Тем самым мы в младшую тетраду байта поместим старшую

unsigned char hc=0;

**hc=c>>4;**

Затем передадим сначала её в функцию sendhalfbyte, а затем и саму нетронутую переменную c. Не важно, что будет в её старшей части, так как функция sendhalfbyte работает только с младшей тетрадой.

  hc=c>>4;

**sendhalfbyte(hc); sendhalfbyte(c);**

}

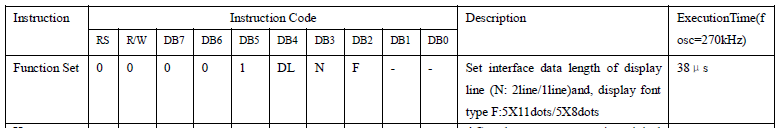
Ну и теперь, применяя вышенаписанную функцию, мы передаём следующий байт, взятый из даташита в функции LCD\_ini

\_delay\_ms(1);

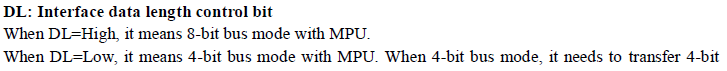
**sendbyte(0b00101000, 0); //4бит-режим (DL=0) и 2 линии (N=1)**

**\_delay\_ms(1);**

Здесь мы используем следующую команду



Единичка в 5 бите нам говорит о том, что мы используем именно эту команду. Дальше идёт у нас бит DL, говорящий контроллеру дисплея о том, какую размерность передачи данных мы используем



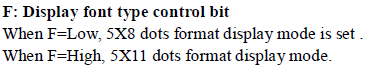
И, так как мы включаем режим 4-битной передачи, то в 4 бите у нас будет 0.

Следующий бит — N, который отвечает за колисество строк

image10

У нас 2 строки, поэтому в 3 бит поставим 1.

F — это размер символа



Здесь также 0.

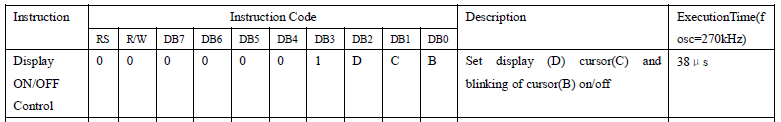
Передадим следующий байт

\_delay\_ms(1);

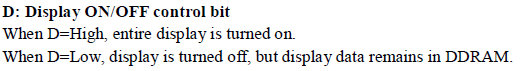
**sendbyte(0b00001100, 0); //включаем изображение на дисплее (D=1), курсоры никакие не включаем (C=0, B=0)**

**\_delay\_ms(1);**

Посмотрим данную команду в документации

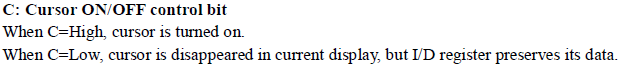


Соответственно, здеь 3 бит, установленный в единицу, говорит о том, что мы применяем именно эту команду, вторая единичка — это параметр D



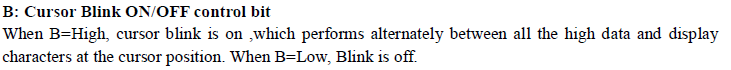
Данной единичкой мы включаем дисплей.

Следующий бит — это параметр C



Здесь мы включаем видимость курсора. Смотреть на курсор нам ни к чему, поэтому 0.

Далее бит B



Этот бит отвечает за мигание дисплея. Также ставим в 0.

Передаём последнюю команду в функции инициализации дисплея

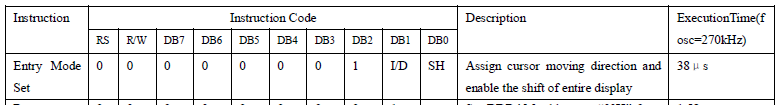
  \_delay\_ms(1);

**sendbyte(0b00000110, 0); //курсор (хоть он у нас и невидимый) будет двигаться влево**

**\_delay\_ms(1);**

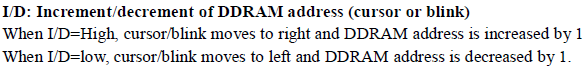
}

Посмотрим данную команду в документации



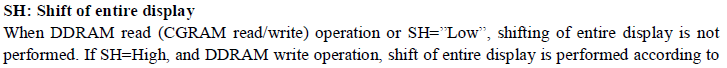
Бит 2 говорит о том, что мы используем именно данную команду.

Следующий бит — I/D



Данный бит отвечает за движение курсора в ту или иную сторону при вводе данных. Хотя у нас курсор и невидимый, двигаемся мы влево, поэтому поставим 1.

Следующий бит — SH



Данный бит отвечает за смещение дисплея. Так как мы это не применяем, то 0.

На этом инициализация закончена.

В сдедующей части урока мы уже попробуем поработать с выводом данных на экран дисплея.

**AVR Урок 12. LCD индикатор 16×2. Часть 4**

**Урок 12**

**Часть 4**

**LCD индикатор 16×2**

Сегодня мы продолжим изучение **жидкокристаллического индикатора символьного**, который способен выводить определённые символы в две строки по 16 символов в каждую.

Вернёмся в проект Atmel Studio.

В прошлой части мы завершили инициализацию дисплея.

Теперь хотелось бы что-нибудь увидеть на дисплее. Для этого нам нужна будет ещё одна функция, которая будет передавать только данные, а не команду

**//—————————————-**

**void sendchar(unsigned char c)**

**{**

**sendbyte(c,1);**

**}**

**//—————————————-**

В данной функции мы просто вызываем другую с единичкой, чтобы каждый раз не писать единички. Данную функцию мы напишем после функции sendbyte.

Ну и давайте попробуем в дисплей передать пару каких-нибудь символов в функции main()

LCD\_ini(); //Инициализируем дисплей

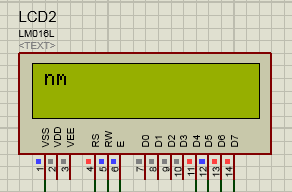
**sendchar('n');**

**sendchar('m');**

while(1)

Соберём код и запустим проект в протеусе.

Всё работает и мы видим следующую картину. В видеоуроке была ошибка, но потом она была исправлена, хотя я думал, что не было изображения из-за того, что не написана была функция позиционирования. Вот так работает дисплей



Собственно, теперь давайте всё-таки напишем функцию позиционирования, чтобы мы могли выводить символы не только подряд, но и в любое место диспелея. Назовём ее **setpos**. Вообще данная функция будет устанавливать указатель в определенное место DDRAM. Выше была показана таблица организации данной памяти и соответствия её ячеек определённым знакоместам на дисплее. Будем этого и придерживаться при написании кода. Напишем нашу функцию после функции sendchar

**//—————————————-**

**void setpos(unsigned char x, unsigned y)**

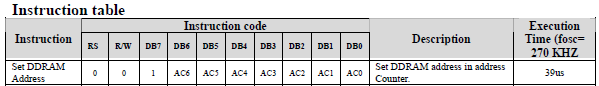
**{**

**}**

**//—————————————-**

Ну, соответственно, у функции будет два входных аргумента — позиция по горизонтали и позиция по вертикали.

Добавим переменную типа char и вычислим её значение в зависимости от значений x и y. Значения входных аргументов мы будем начинать от нуля, а не от единицы. Поэтому чтобы вычислить адрес по y в памяти DDRAM, нам достаточно умножить значение y на 0x40, так как именно с данного адреса начинается 2 строка, а у нас она будет выглядеть как 1. Затем прибавляем x, тем самым получим смещение по горизонтали в данной памяти. И ещё по операции "ИЛИ" вычисленное значение мы сложим с двоичным числом 0b10000000, так как команда передачи адреса памяти DDRAM выглядит вот так



То есть в 6 бите мы передаём единицу для того, чтобы контроллер дисплея "понял" что мы даём именно такую команду — передача адреса памяти DDRAM, чтобы контроллер дисплея установил указатель именно туда, какую позицию мы даём ему в оставшихся семи младших битах — DB6-DB0.

Вот такой мы получим код функции

void setpos(unsigned char x, unsigned y)

{

**char adress;**

**adress=(0x40\*y+x)|0b10000000;**

**sendbyte(adress, 0);**

}

Ну и давайте теперь вызовем данную функцию в функции main для порядка, установив жёстко перед вызовом фунции вывода символа указатель памяти на нулевой адрес

LCD\_ini(); //Инициализируем дисплей

**setpos(0,0);**

sendchar('n');

Проверим на всякий случай сборку кода и запуск в протеусе. Всё работает, но эффекта данной функции мы не прочувствуем, пока не попробуем вывести ещё какие-нибудь симвовлы куда-то в другое место дисплея. Например, вот так

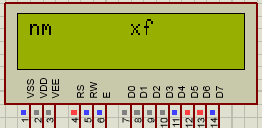
sendchar('m');

**setpos(8,0);**

**sendchar('x');**

**sendchar('f');**

После сборки и запуска мы должны увидеть вот такой результат



Также давайте что-нибудь выведем на другую строку дисплея

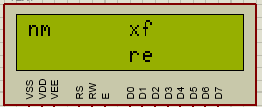
sendchar('f');

**setpos(8,1);**

**sendchar('r');**

**sendchar('e');**

А вот и результат



В следующей части нашего урока мы постараемся уже вывести данные на экран не посимвольно, что не всегда удобно, а целыми строками.

Также ещё мы научимся модульному программированию,то есть писать код мы будем не в один файл, а в несколько, что позволит избавиться от лишнего кода в главном файле или модуле, а поместить код, который включает в себя реализацию какой-то отдельной задачи в программе в отдельный модуль или библиотеку.

# AVR Урок 12. LCD индикатор 16×2. Часть 5

**Урок 12**

**Часть 5**

# ****LCD индикатор 16×2****

Сегодня мы продолжим изучение **жидкокристаллического индикатора символьного**, который способен выводить определённые символы в две строки по 16 символов в каждую.

В прошлой части мы завершили и проверили написание кода функции, которая выводит любой символ на экран дисплея.

Теперь пришло время написать функцию вывода на экран целой строки, так как выводить посимвольно не совсем удобно, хотелось бы работать со строками. Добавим данную функцию прямо перед функцией main() и передавать мы ей будем массив символов неопределённой размерности

**//—————————————-**

**void str\_lcd (char str1[])**

**{**

**}**

**//—————————————-**

Вызовем данную функцию в main(), удалив перед этим весь код посимвольного вывода на дисплей

LCD\_ini(); //Инициализируем дисплей

setpos(0,0);

**str\_lcd("Hello World!");**

Дальше начнём писать тело функции вывода строки. Объявим в теле функции переменную для символа. Переменная у нас будет несколько иного типа. Как правило с таким типом лучше распознаются коды символов. Вы можете, конечно, поэксперементировать с другими типами

void str\_lcd (char str1[])

{

**wchar\_t n;**

Далее мы, соответственно, организуем цикл и будем попеременно перебирать все переданные символы в массиве и выводить их на дисплей. Применим также мы вариант представления нулевого симвлова **'n'** и именно до него мы и будем перебирать символы

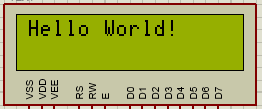
  wchar\_t n;

**for(n=0;str1[n]!='';n++)**

**sendchar(str1[n]);**

}

Соберём код и проверим в протеусе работу кода



Теперь можно попробовать вывести строку ещё и в другое место экрана. Напишем код в main()

str\_lcd("Hello World!");

**setpos(2,1);**

**str\_lcd("String 2");**

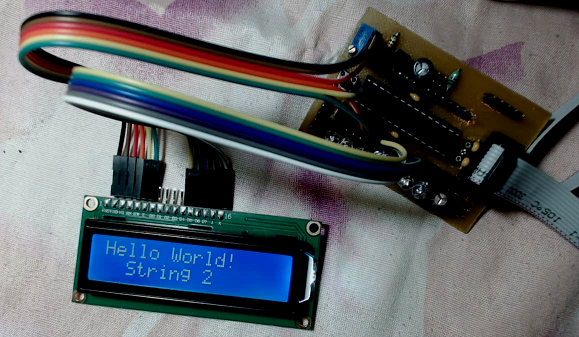
while(1)

Соберём код и посмотрим результат



Всё работает! Отлично!

Ну конечно нужно ещё помотреть, как будет код работать на живом дисплее с живым контроллером. Для этого мы прошьём контроллер

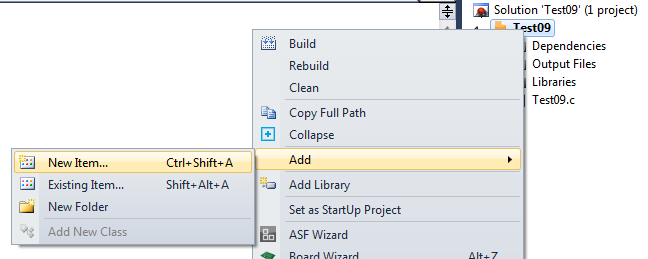


## **Оформляем функции в отдельный модуль**

Дошли мы с кодом до такого состояния, что наш главный и единственный файл с кодом переполнился до того, что в нём теперь тяжело уже вообще что-то найти. Как же мы с этим можем боротья? Бороться с этим мы будем путём оформления кода функций для отдельно взятого устройства или шины или какой-то технологии в отдельный модуль. Грамотный модуль состоит как правило из заголовочного файла и файла реализации функций. Поэтому давайте для нашего LCD дисплея мы так и поступим. Также это всё дело нужно для того, что если мы будем писать новый проект, то мы данные файлы будем просто к нему подключать, если нам потребуется воспользоваться LCD дисплеем. Это будет нашей так называемой библиотекой для дисплея. Конечно, библиотеки обычно пишутся и компилируются в отдельный файл lib, но в этом случае обычно нет исходного кода и данные библиотеки не могут быть подправлены. А наша библиотека будет вполне исправимой и нам ещё ой как пригодится в будущем.

Но прежде, чем создать данную библиотеку, мы создадим главный заголовочный файл и назовём его main.h, чтобы убрать в данный файл все подключенные библиотеки, различные глобальные переменные и макроподставновки

Для этого мы в дереве проектов щёлкаем правой кнопкой по нашему проекту **Test09** и выбираем в контекстном меню субменю **Add**, а в нём уже выбираем пункт **New Item**



И в открывшемся диалоге выбираем тип файла, который мы будем создавать, "**Include File**" И внизу в имени файла меняем IncFile1 на **main**, затем жмём кнопку **Add.**

Соответственно, у нас создастся файл main.c вот с таким содержимым

**#ifndef MAIN\_H\_**

**#define MAIN\_H\_**

**#endif /\* MAIN\_H\_ \*/**

Это очень хорошо. Например. в Keil, когда мы занимается программированием контроллеров STM, мы должны это всё писать в ручную. Здесь данная директива говорит о том, что, если файл уже подключался в исполняемый код, чтобы прероцессор его повторно не включал.

В данный файл мы посместим подключения всех заголовочных файлов библиотек и все макроподстановки, а в файле Test09.c всё это, конечно, мы удалим

#ifndef MAIN\_H\_

#define MAIN\_H\_

**#define F\_CPU 8000000UL**

**#include <avr/io.h>**

**#include <avr/interrupt.h>**

**#include <util/delay.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**//—————————————-**

**#define e1 PORTD|=0b00001000 // установка линии E в 1**

**#define e0 PORTD&=0b11110111 // установка линии E в 0**

**#define rs1 PORTD|=0b00000100 // установка линии RS в 1 (данные)**

**#define rs0 PORTD&=0b11111011 // установка линии RS в 0 (команда)**

**//—————————————-**

#endif /\* MAIN\_H\_ \*/

Но недостаточно данный заголовочный файл подключить в Solution Explorer, его также мы должны в файле **Test09.c** подключить в самом начале в код

**#include "main.h"**

//—————————————-

Соберём проект, ещё раз проверим его работоспособность.

Теперь начнём создавать нашу библиотеку для дисплея.

Для этого мы таким же образом, как и main.h, создадим заголовочный файл **lcd.h**

Содержание получится аналогичным, только вместо MAIN везде будет LCD. Подключим данный файл в файле main.h

#include <stdlib.h>

**#include "lcd.h"**

И наоборот, в файл **lcd.h** мы подключим файл **main.h**

#ifndef LCD\_H\_

#define LCD\_H\_

**#include "main.h"**

Насчет того, что получится какое-то перекрёстное зацикливание, можно не беспокоиться — директивы не дадут такому случиться.

Также все макроподстановки из файла main.h мы заберём в файл lcd,h, а в main.h удалим

#include "main.h"

//—————————————-

**void LCD\_ini(void);**

**void setpos(unsigned char x, unsigned y);**

**void str\_lcd (char str1[]);**

**void sendchar(unsigned char c);**

//—————————————-

А, чтобы забрать все функции по работе с дисплеем из файла Test09.c, мы создадим теперь уже другой файл — **lcd.с**. В нём и будет код реализации всех функций

Создаётся файл точно таким же образом, только вместо "Include File" мы выбираем тип файла "C File".

Файл lcd.c создан. В нём уже не будет никаких директив, единственное, будет авторский комментарий, который мы удалим, чтоб не мешался.

В данном файле мы также подключим заголовочный файл lcd.h

**#include "lcd.h"**

**//—————————————-**

Теперь мы в данный файл перенесём полностью все функции, предназначенные для работы с дисплеем, из файла Test09.c. В нём останутся только две фунции — **port\_ini** и **main()**.

Тем самым мы очень серьёзно разгрузим главный файл приложения, сделав его удобочитаемым.

Но этого нам недостаточно. Ни одна функция теперь не будет "видна" в файле Test09.c. Поэтому те функции, которые мы будем использовать в других файлах, мы обязаны объявить, или, как говорят в народе, создать на них прототипы. Делается это обычно в заголовочном файле. Поэтому мы создадим прототипы в заголовочном файле lcd.h. Прототип делается очень легко. Пишется, или обычно копируется заголовок функции со всеми аргументами (всё кроме тела) и в конце ставится точка с запятой. Нам нужны будут функции инициализации дисплея, позиционирования на дисплее и вывода строки на дисплее. Символы мы отдельно пока выводить не будем, поэтому на соответствующую функцию мы прототип не создаём. Вот наши прототипы

#include "main.h"

//—————————————-

**void LCD\_ini(void);**

**void setpos(unsigned char x, unsigned y);**

**void str\_lcd (char str1[]);**

//—————————————-

#define e1 PORTD|=0b00001000 // установка линии E в 1

Теперь соберём файл, запустим его в протеусе, и проверим его работоспособность. Также проверим на практике. Если всё работает, то мы всё сделали правильно. Проект на весь урок приложен внизу и доступен по ссылке "Исходный код".

Таким образом, в сегодняшнем уроке мы много чему научились. Мы научились работать с символьным дисплеем и подключать его к контроллеру AVR. Также мы в рамках данного урока научились грамотному оформлению кода и использованию модульного программирования.