**Оглавление**

[Осциллограф на AVR 3](#_Toc124442759)

[Принцип работы 3](#_Toc124442760)

[Устройство собственного протокола передачи данных по UART 3](#_Toc124442761)

[Схема в Proteus 4](#_Toc124442762)

[Описание Аналго-цифрового преобразователя 5](#_Toc124442763)

[Пример кода в редакторе Atmel Studio: 7](#_Toc124442764)

[Пояснение кода 8](#_Toc124442765)

[USART 11](#_Toc124442766)

[Описание USART 11](#_Toc124442767)

[Регистры управления интерфейса USART в контроллере Atmega8 15](#_Toc124442768)

[Программирование интерфейса USART 19](#_Toc124442769)

[Исходный код 19](#_Toc124442770)

[Пояснение кода 19](#_Toc124442771)

[Инициализация модулей Usart в функции main () 21](#_Toc124442772)

[I2C 22](#_Toc124442773)

[Описание I2C 22](#_Toc124442774)

[Описание I2C функции библиотеке 23](#_Toc124442775)

[Пример кода I2C: 24](#_Toc124442776)

[i2c.h 24](#_Toc124442777)

[i2c.cpp 24](#_Toc124442778)

[LCD 26](#_Toc124442779)

[Ножки LCD Дисплея 26](#_Toc124442780)

[Память 28](#_Toc124442781)

[Инициализация 30](#_Toc124442782)

[Пример кода работы LCD: 32](#_Toc124442783)

[lcd.h 32](#_Toc124442784)

[lcd.cpp 33](#_Toc124442785)

[Основная часть программы – main 36](#_Toc124442786)

[main.h 36](#_Toc124442787)

[main.cpp 36](#_Toc124442788)

[Примеры передачи и приема по UART 38](#_Toc124442789)

[Передача синусоидального сигнала (генерация через функции) 38](#_Toc124442790)

[Схема: 38](#_Toc124442791)

[Настройки COMPIM в Proteus 39](#_Toc124442792)

[Код генератора: 39](#_Toc124442793)

[График на принимающей сигнал программе: 40](#_Toc124442794)

[Прием и вывод данных на LCD 40](#_Toc124442795)

[Схема в Proteus (первый запуск) 41](#_Toc124442796)

[Код: 41](#_Toc124442797)

[Отправка данных через Terminal 41](#_Toc124442798)

[Результат 42](#_Toc124442799)

[Приложение 43](#_Toc124442800)

# Осциллограф на AVR

## Принцип работы

АЦП микроконтроллера считывает и преобразует напряжение с одной ножки и полученное значение отправляет на компьютер через интерфейс UART, так же раз в несколько десятков тысяч тактов обновляет значение напряжения на LCD дисплее.

Для инициализации передачи данных по UART используется миниатюрный протокол. Один пакет данных занимает 2 байта, это число в диапазоне от 0 до 1023, где 0 соответствует 0В а 1023 соответствует 5В – опорное напряжение АЦП. Таким образом минимальный шаг напряжения равен 5В / 1023 = 0,005В.

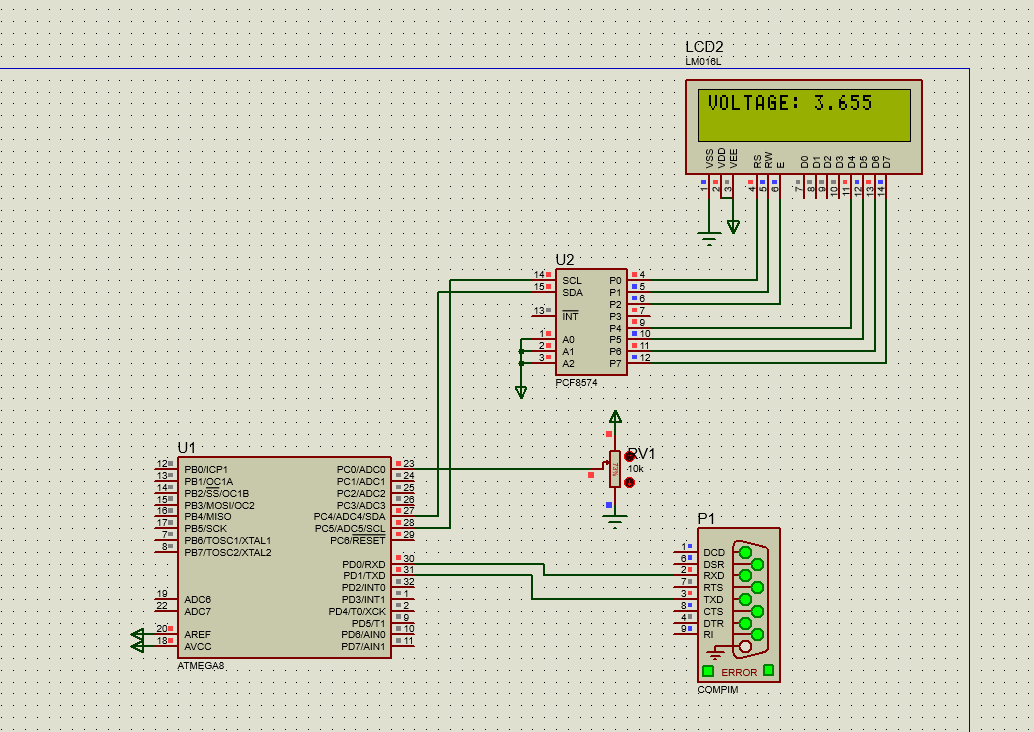
Программа на компьютере принимает пакет байтов с микроконтроллера и преобразует их в значение вольтажа запоминает и рисует график.

## Устройство собственного протокола передачи данных по UART

Для того, чтобы компьютер и микроконтроллер выполнили синхронизировались и МК начал отправлять данные, компьютер сначала должен отправить 1 байт данных, это число 0, после чего МК отправляет ответный байт, число 255. После чего микроконтроллер начинает отправлять поток данных, полученных с АЦП.

Без этого протокола МК данные отправлять не начнет!

## Схема в Proteus



С ножки C0 на МК снимается входное напряжение от 0 до 5В.

К AREF и подключается опорное напряжение АЦП 5В.

Ножки PD0 и PD1 отвечают за передачу данных по UART. PD0 – RXD, PD1 – TXD соответственно.

Ножки PC4 и PC5 отправляют данные на LCD через I2C преобразователь.

# Описание Аналго-цифрового преобразователя

**Аналго-цифровой преобразователь** или как его называют **АЦП**. В английской аббревиатуре, гораздо чаще встречающейся в технической документации — **ADC** (**Analog-to-Digital Converter**). Это устройство, которая преобразует величину электрического сигнала в цифровой код. Затем данный код уже используется для обработки или для отображения тем или иным образом данной электрической величины. Это очень распространённая периферия или технология. АЦП активно используется в звукозаписи, измерительной технике, видеозаписи и во многих других случаях. Поэтому нас обойти данную вещь стороной никак не получится, тем более АЦП поддерживается аппаратно в контроллерах **AVR**.

В контроллере Atmega8 АЦП имеет следующие характеристики

* Разрешение 10 бит
* Время преобразования одного показания от 13 до 250 микросекунд в зависимости от битности измерения, а также от тактовой частоты генератора, тактирующего контроллер,
* Поддержка запуска по прерываниям,

Как работает цифровое преобразование?

Берётся опорное напряжение и сравнивается с измеряемым. Соответственно, опорное напряжение всегда должно быть больше измеряемого. Если это не так, то нужно будет применять делители напряжения.

Изобразим схематично процесс измерения

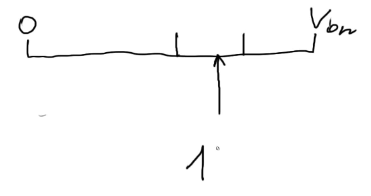


Отрезок — это диапазон измерений. Он находится между нулём и опорным напряжением. А стрелка — это измеряемое напряжение.

Данный отрезок делится пополам, и АЦП оценивает, в какой половине находится приложенное напряжение

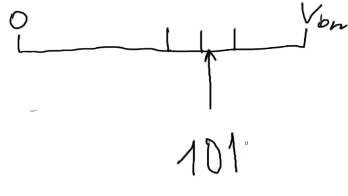


Если оно находится в стороне нуля, то в самый старший бит результата записывается 0, а если в стороне максимального напряжения, то единица. У нас будет единица. Затем та половина отрезка, на которой находится измеряемое напряжение делится ещё пополам, и АЦП опять измеряет, в какой половинке уже данного отрезка у нас находится измеряемое напряжение



Оценка идёт по тому же принципу — в какой стороне отрезок. В нашем случае будет 0. И этот ноль записывается в следующий более младший бит

Затем та четвертинка опять делится пополам и АЦП опять оценивает, где находится отрезок



И АЦП так и продолжает такой процесс до тех пор, пока не кончатся ячейки для битов. Atmega8 имеет 10-битный АЦП с возможностью подачи напряжения на один из 5 аналоговых входов (порт С, мы используем С0), следовательно будет 10 подобных измерений и заполнятся 10 бит величины. Чем больше бит, тем точнее результат, но уже потребуется на это больше времени и более точный АЦП. Имея данный результат, несложно будет посчитать величину измеренного напряжения. Мы знаем. что если у нас АЦП 10-битный, то данный результат у нас лежит в промежутке от 0 до 1024, получается всего у нас 1023 отрезка. И мы затем наш результат делим на 1023 и умножаем на величину опорного напряжения, в нашем случае равное 5 вольтам.

## Пример кода в редакторе Atmel Studio:

1. **adc.h**

#ifndef ADC\_H\_

#define ADC\_H\_

#include "main.h"

void ADC\_Init(void);

unsigned int ADC\_convert (void);

#endif /\* ADC\_H\_ \*/

1. **adc.cpp**

#include "adc.h"

//----------------------------------------

void ADC\_Init(void)

{

ADMUX = 0x00;

ADCSRA = 0x8D;

}

//----------------------------------------

unsigned int ADC\_convert ()

{

ADMUX= 0 | (ADMUX & 0xF0);

//задержка для стабилизации входного напряжения

*\_delay\_us*(10);

//начинаем преобразование (ADSC = 1)

ADCSRA |= 0x40;

while((ADCSRA & 0x10)==0); //ждем, пока АЦП закончит преобразование (ADIF = 0)

ADCSRA|=0x10;//устанавливаем ADIF

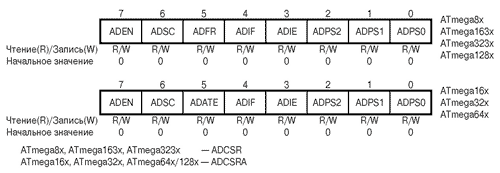
return ADCW;//ADCW - содержит ADCH и ADCL как нам нужно

}

## Пояснение кода

[#include](https://vk.com/im?sel=99356403&st=%23include) "adc.h"  
//--------------------------------------—  
void ADC\_Init(void)  
{  
ADCSRA |= (1«ADEN);//Вкл АЦП

Регистр **ADCSRA** — управляющий и статусный регистр



**ADEN** — данный бит включает АЦП.

ADCSRA |= 1«ADPS2 | 1«ADPS1 | 1«ADPS0; // предделитель на 128

**ADPS2-ADPS0** — биты, от комбинации которых зависит величина делителя

|  |  |  |  |
| --- | --- | --- | --- |
| **ADPS2** | **ADPS1** | **ADPS0** | **Коэффициент деления** |
| 0 | 0 | 0 | 2 |
| 0 | 0 | 1 | 2 |
| 0 | 1 | 0 | 4 |
| 0 | 1 | 1 | 8 |
| 1 | 0 | 0 | 16 |
| 1 | 0 | 1 | 32 |
| 1 | 1 | 0 | 64 |
| 1 | 1 | 1 | 128 |

ADCSRA |= (1«ADIE);// разрешаем прерывание

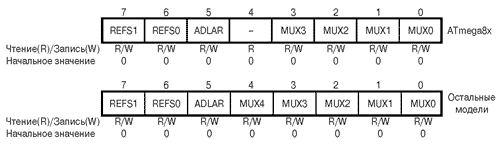
**ADIE** — бит, включающий режим прерываний.

ADCSRA |= 1«ADFR;

**ADFR** — используется в режиме с использованием прерываний. При установке в 1 включает круговой режим, при котором измерения автоматически следуют одно за другим.

ADMUX |= (0«REFS1) | (0«REFS0); //AVCC with external capacitor at AREF pin

**ADMUX** – управляющий и статусный регистр, отвечающий за выбор входа.

****

**REFS1-REFS0** — биты, включающие определённый режим использования опорного напряжения.

ADMUX |= (0«ADLAR); //лев выравнивание

**ADLAR** — это бит организации расположения измеренных 10 битов в двух байтах регистровой пары данных.

ADMUX |= (1«MUX1) | (1«MUX0); // PC3 запуск по 1 каналу

**MUX3-MUX0** — биты, включающие определённый канал мультиплексора

|  |  |
| --- | --- |
| **MUX3-MUX0** | **Несимметричный вход** |
| 0000 | ADC0 |
| 0001 | ADC1 |
| 0010 | ADC2 |
| 0011 | ADC3 |
| 0100 | ADC4\* |
| 0101 | ADC5\* |
| 0110 | ADC6\*\* |
| 0111 | ADC7\*\* |
| 1000-1101 | Зарезервировано |
| 1110 | 1,22V |
| 1111 | 0V(GND) |
| \*8-ми разрядное преобразование  \*\*Имеются только в корпусах TQFP-32 и MLF-32. | |

ADMUX = (ADMUX & 0x00);  
ADCSRA |= 1«ADSC;

**ADSC** — при установке в 1 заставляет АЦП начинать преобразование.  
}  
//--------------------------------------—  
unsigned int ADC\_convert ()  
{  
ADMUX = (ADMUX & 0xF0);  
//задержка для стабилизации входного напряжения  
\_delay\_us(10);  
//начинаем преобразование (ADSC = 1)  
ADCSRA |= 0x40;  
while((ADCSRA & 0x10)==0); //ждем, пока АЦП закончит преобразование (ADIF = 0)  
ADCSRA|=0x10;//устанавливаем ADIF  
return ADCW;//ADCW - содержит ADCH и ADCL как нам нужно  
}

Эти функции используются в main().

# USART

## Описание USART

USART – это метод передачи и приема информации, используемый при работе с АЦП. Как правило используется для обмена данными между устройствами на контроллерах и ПК. Но вполне подходит и для связи между двумя микроконтроллерами, а также для связи любых устройств, где данная шина поддерживается.

Также **USART** называют **UART (Universal Asynchronous Receiver-Transmitter (универсальный асинхронный приемопередатчик))**. Однако во второй аббревиатуре отсутствует синхронизация. До появления синхронного UARTа шел обмен только по двум проводам, а потом в данный интерфейс добавили ещё шину синхронизации. Но до сих пор она не часто используется, Данные и так нормально доходят. Достигается это равенством скоростей на приёмном и передающим устройстве, а также некоторыми определёнными битами в протоколе.

**Организация интерфейса USART в Atmega8**

Данный интерфейс организован в микроконтроллере Atmega8 таким образом:

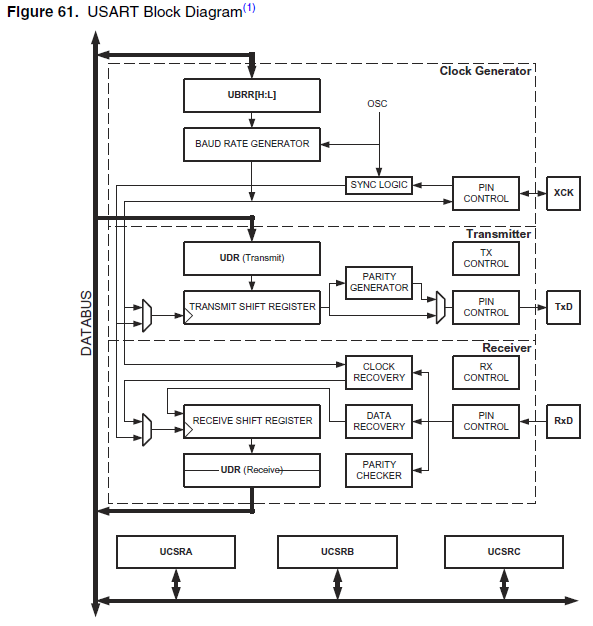


Рисунок 1 - Организация в микроконтроллере Atmega8

Как видно по схеме, в USART используются три основных выхода:

RxD – ножка приема информации.

TxD – ножка передачи информации.

XCK – ножка синхронизации. (Так как синхронный режим не используется, то эта ножка не подключается)

В контроллере Atmega8 за каждый выход USART отвечает две ножки: PD0 и PD1.

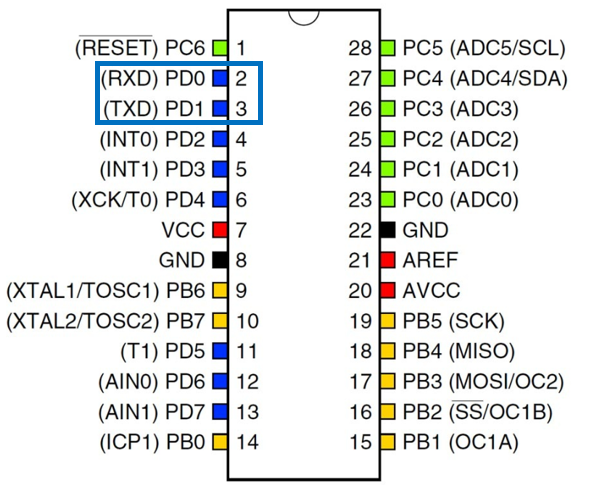


Рисунок 2 - Схема ножек Atmega8

Два устройства связываются посредством прямого подключения ножки TxD (PD0) одного устройства к ножке RxD другого и, наоборот, Ножку RxD (PD1) одного устройства мы подключаем к ножке TxD другого (см.рис.3).

Пример подключения двух устройств:

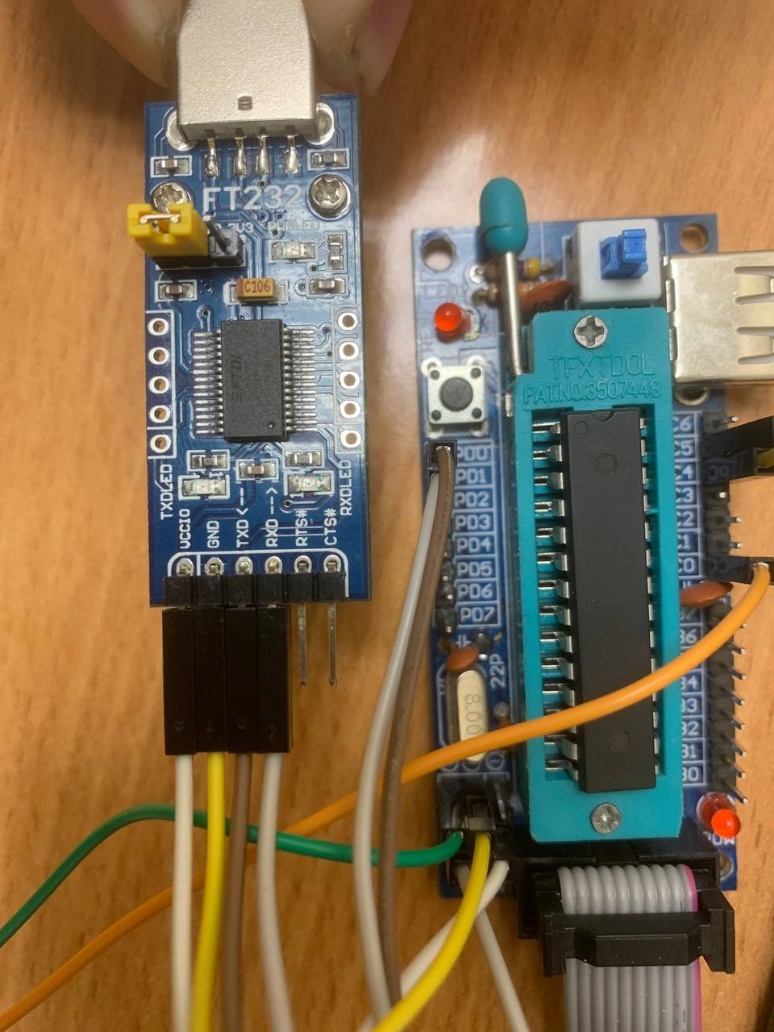


Рисунок 3 - Atmega8 – преобразователь USB-UART (FT232)

Красный провод подключен от ножки Atmega8 PD0 к ножке переходника USB TxD.

Белый провод подключен от ножки Atmega8 PD1 к ножке переходника USB RxD.

**Описание протокола интерфейса USART.**

Для начала работы необходимо разобраться каким образом осуществляется передача данных по шине USART.

В начале, когда шина не используется и данные не передаются, то контакт находится в высоком состоянии. Затем идет обязательный стартовый бит, в момент которого контакт переходит из высокого состояния в низкое. После того как его время истечет, контроллер считывает следующие биты информации. Их может быть от 5 до 9. Чаще всего используется 8. В дальнейшей практической части также будет использоваться 8 бит информации. Состояние ножки во время передачи диктуется самим информационным битом. Если передается 1 - то высокое, если 0 - то низкое.

Следующий бит — это бит чётности. Данный бит осуществляет проверку данных данных. В зависимости от режима он может быть, а также его может и не быть. То есть, когда мы передаём один блок, он будет выставлен, а следующий — сброшен, для ещё большей синхронизации.

Затем идут стоповые биты, гласящие о том, что передача посылки закончена. Данных битов может быть один или два также в зависимости от режима, однако это количество говорит не о весе данного бита, а о его длительности. Данные биты передаются с помощью организации высокого логического состояния на ножке. И затем процесс повторяется сначала, то есть передаём следующую посылку.

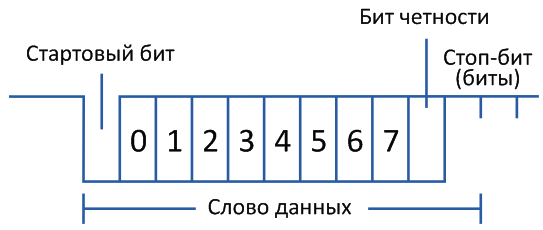


Рисунок 4 - Протокол интерфейса

## Регистры управления интерфейса USART в контроллере Atmega8

Существует три регистра управления – UCSRA, UCSRB, UCSRC. Данные регистры расшифровываются как USART Control and Status Register. То есть они не только управления, но и состояния.

* Регистр UCSRA

Бит RXC — это бит состояния приемной шины. С помощью него можно отслеживать путь пакета в шину, если пришел, то устанавливается определенный флаг. Используется только для чтения, поэтому возможность самостоятельного управления отсутствует

Бит TXC — бит состояния шины передачи. Назначение то же, что у бита RXC, но этот бит может изменяться программно, так как действует как на запись, так и на чтение

Бит UDRE — бит освобождения регистра данных.

Бит FE — бит флаговый и устанавливается в случае ошибки передачи пакета.

Бит DOR – Флаг переполнения. В USART флаг устанавливается в 1, если в момент обнаружения нового старт-бита в сдвиговом регистре приемника находится последнее принятое слово, а буфер приемника полон (два значения). В UART флаг устанавливается в 1, если новый кадр будет помещен в сдвиговый регистр приемника до того, как из регистра данных будет считано предыдущее слово. Флаг сбрасывается при пересылке принятых данных из сдвигового регистра приемника в буфер.

Бит PE – Флаг ошибки контроля четности. Флаг устанавливается в 1, если в данных, находящихся в буфере приемника, выявлена ошибка контроля четности. При отключенном контроле четности этот разряд постоянно читается как 0.

Бит U2X — данный бит используется для удвоения скорости приёма-передачи данных по шине.

Бит MPCM – режим мультипроцессорного обмена. Разряд MPCM используется в режиме мультипроцессорного обмена. Если он установлен в 1, ведомый микроконтроллер ожидает приема кадра, содержащего адрес. Кадры, не содержащие адреса устройства, игнорируются.

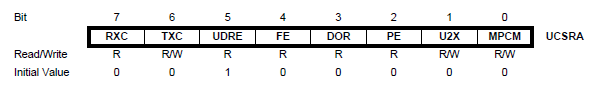


Рисунок 5 - Регистр UCSRA

* Регистр UCSRB

Бит RXCIE – бит для разрешения прерывания по завершению приема. При установленном в состояние 1 бите RXCIE и установленном разрешении глобального прерывания установка бита RXC в регистре UCSRA приведет к выполнению прерывания USART, RXC по завершению приема.

Бит TXCIE – бит для разрешения прерывания по завершению передачи. При установленном в состояние 1 бите TXCIE и установленном разрешении глобального прерывания установка бита ТХС в регистре UCSRA приведет к выполнению прерывания по завершению передачи.

Бит UDRIE – бит для разрешения прерывания при опустошении регистра данных. При установленном бите UDRIE и установленном разрешении глобального прерывания установка бита UDRE в регистре USR приведет к выполнению прерывания по пустому регистру данных UART.

Бит RXEN *–* бит для разрешения приема. Установленный бит RXEN разрешает прием данных в USART. Если прием запрещен, то флаги статуса ТХС, DOR и FE установить невозможно. Если эти флаги установлены, то очистка бита RXEN не приведет к очистке этих флагов.

Бит TXEN – бит для разрешения передачи. Установленный бит TXEN разрешает передачу данных USART. Если разряд сбрасывается во время передачи, выключение передатчика произойдет только после завершения передачи данных, находящихся в сдвиговом регистре и буфере передатчика.

Бит UCSZ2 – это бит формат посылок; используется совместно с разрядами UCSZ0(1) регистра UCSRC для задания размера слов данных, передаваемых по последовательному каналу.

Бит RXB8 – это бит для приема 8-разрядных данных. При использовании 9-разрядных слов данных этот разряд содержит значение старшего разряда принятого слова. Содержимое этого разряда должно быть считано до прочтения регистра данных UDR.

Бит ТХВ8 – это бит для передачи 8-разрядных данных. При использовании 9-разрядных слов данных содержимое этого разряда является старшим разрядом передаваемого слова. Требуемое значение должно быть занесено в этот разряд до загрузки байта данных в регистр UDR.

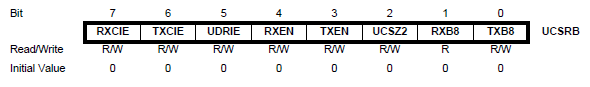


Рисунок 6 - Регистр UCSRB

* Регистр UCSRC

Бит URSEL – бит для выбора регистра. Этот бит определяет, в какой из регистров модуля производится запись. Если бит установлен, обращение производится к регистру UCSRC; если бит сброшен, обращение производится к регистру UBRRH.

Бит UMSEL – бит для определения режима работы USART. Если бит сброшен, модуль USART работает в асинхронном режиме (режим UART); если бит установлен, то модуль USART работает в синхронном режиме.

Биты UPM1 и UPM0 – это биты для определения режима работы схемы контроля и формирования четности.

Таблица 1 - Режим работы схемы контроля

|  |  |  |
| --- | --- | --- |
| UPM1 | UPM0 | Режим работы |
| 0 | 0 | Выключено |
| 0 | 1 | Резерв |
| 1 | 0 | Проверка на четность |
| 1 | 1 | Проверка на нечетность |

Бит USBS - количество стоп-битов, посылаемых передатчиком. Если разряд сброшен, передатчик посылает 1 стоп-бит, если установлен, то 2 стоп бита. Для приемника содержимое этого разряда безразлично.

Таблица 2 - Определение количества стоповых битов

|  |  |
| --- | --- |
| USBS | Количество стоп-битов |
| 0 | 1 бит |
| 1 | 2 бита |

Биты UCSZ1 и UCSZ0 – это биты, которые определяют формат посылок. Совместно с разрядом UCSZ2 регистра UCSRB эти разряды определяют количество разрядов данных в посылках

Таблица 3 - Регулирование количества полезных информационных битов для приема-передачи

|  |  |  |  |
| --- | --- | --- | --- |
| UCSZ2 | UCSZ1 | UCSZ0 | Количество бит |
| 0 | 0 | 0 | 5 |
| 0 | 0 | 1 | 6 |
| 0 | 1 | 0 | 7 |
| 0 | 1 | 1 | 8 |
| 1 | 0 | 0 | резерв |
| 1 | 0 | 1 | резерв |
| 1 | 1 | 0 | резерв |
| 1 | 1 | 1 | 9 |

Бит UCPOL – это бит для определения полярности тактового сигнала. Значение этого разряда определяет момент выдачи и считывания данных на выводах модуля. Разряд используется только при работе в синхронном режиме. При работе в асинхронном режиме он должен быть сброшен.

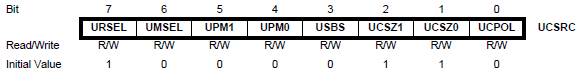


Рисунок 7 - Регистр UCSRC

* Регистры скорости передачи UBRRL и UBRRH

URSEL – Выбор регистра. Этот разряд определяет, в какой из регистров модуля производится запись. Если разряд установлен в «1», обращение производится к регистру UCSRC. Если же разряд сброшен в «0», обращение производится к регистру UBRRH.

UBRR11 – UBRR0 - Значение скорости передачи в бодах.

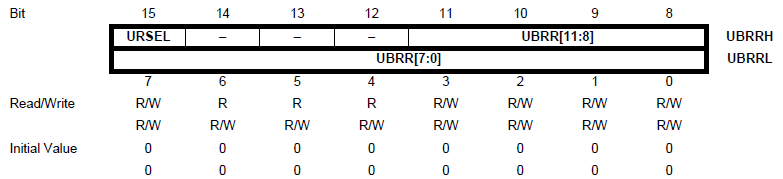
[](http://www.doneathome.ru/wp-content/uploads/2014/04/UBRRHL.bmp)

Рисунок 8 - Регистры скорости передачи UBRRL и UBRRH

## Программирование интерфейса USART

### Исходный код

#include "usart.h"

void USART\_Init( unsigned int speed)//Инициализация модуля USART

{

UBRRH = (unsigned char)(speed>>8);

UBRRL = (unsigned char)speed;

UCSRB=(1<<RXEN)|( 1<<TXEN); //Включаем прием и передачу по USART

UCSRB |= (1<<RXCIE); //Разрешаем прерывание при передаче

UCSRA |= (1<<U2X); // Для 8 мгц

UCSRC = (1<<URSEL)|(1<<USBS)|(1<<UCSZ1)|(1<<UCSZ0);// Обращаемся именно к регистру UCSRC (URSEL=1),

//ассинхронный режим (UMSEL=0), без контроля четности (UPM1=0 и UPM0=0),

//1 стоп-бит (USBS=0), 8-бит посылка (UCSZ1=1 и UCSZ0=1)

//UCSRC |= (1<<UPM1);//четность

}

void USART\_Transmit( unsigned char data ) //Функция отправки данных

{

while ( !(UCSRA & (1<<UDRE)) ); //Ожидание опустошения буфера приема

UDR = data; //Начало передачи данных

}

### Пояснение кода

#include "usart.h"

void USART\_Init (unsigned int speed) // Инициализация модуля USART

{

UBRRH = (unsigned char) (speed>>8);

UBRRL = (unsigned char) speed;

// Так как регистровая пара UBRR 16-битная и представляет собой два регистра, то по ним и распределяется число. Старшая часть идет в регистр UBRRH, младшая – в UBRRL. Старший бит в регистре UBBRH изначально отвечает за переключение регистров, но в регистровую пару будут передаваться не слишком большие значения, следовательно обращение будет идти именно к регистру UBRR.

UCSRB=(1<<RXEN) | (1<<TXEN); // Биты RXEN и TXEN включает USART на прием и передачу, так как мы используем оба направления, то и включено оба.

UCSRB |= (1<<RXCIE); // Бит RXCIE разрешает прерывания на приёмной шине. При помощи прерывания обработчик дает понять, когда данные будут в регистре

UCSRA |= (1<<U2X); // Бит U2X отвечает за удвоение скорости приема-передачи данных по шине.

UCSRC = (1<<URSEL)|(1<<USBS)|(1<<UCSZ1) | (1<<UCSZ0); // Бит URSEL совмещен с таким же битом в регистровой паре UBBR, поэтому если данный бит равен нулю то запись будет в UBRR, а если равен единице, то в UCSRC. Необходимо обратиться именно к регистру UCSRC, поэтому URSEL=1. Бит USBS отвечает за количество стоповых битов. Нам нужно 2 стоповых бита, поэтому значение равное 1. Биты UCSZ1 и UCSZ0 отвечают за количество полезных информационных битов для приема передачи. Значения установлены равные единице, что значит, что установленное количество - 8 бит.

}

void USART\_Transmit (unsigned char data) //Функция для передачи данных по шине. В данной функции отправляем данные побайтно

{

while (! (UCSRA & (1<<UDRE))); //Ожидание опустошения буфера приема. Прежде чем что-то передать, необходимо убедиться, что шина передачи не занята. Делается это проверкой бита UDRE в регистре UCSRA. Данный бит обнуляется, как только освобождается регистр приема-передачи данных и он будет готов к записи в него новых данных. Это помогает защититься от потери данных при передаче.

UDR = data; // Непосредственно присваивание буферу передачи данных байт, который необходимо отправить на ПК.

}

int USART\_Read()

{

return ((UCSRA >> RXC) & 1)? UDR : -1; // возвращает значение буфера приёма

}

## Инициализация модулей Usart в функции main ()

В main необходимо добавить строчку USART\_Init (8) для инициализации, что соответствует частоте передачи данных 115200 бод.

В осциллографе необходимо реализовать две функции для правильного отображения напряжения.

ISR(USART\_RXC\_vect) // Функция проверки инициализации

{

if (usart\_ready) { UDR; return; } // Если USART инициализирован, то ничего не происходит

if (UDR == 0) {

USART\_Transmit(0xFF);

usart\_ready = true;

Clear();

Curs(0, 0);

PrintString("USART CONNECTED"); // Если USART не инициализирован, то функция отправляет 8 единиц, очищает дисплей и пишет о том, что USART подключен

}

}

void TransmitInt(unsigned int num) // Функция передает целое число (напряжение)

{

unsigned char\* ptr = (unsigned char\*)&num; //Получение значения указателя

USART\_Transmit(\*(ptr++)); // смещение на 1 байт

USART\_Transmit(\*(ptr)); // передача следующего байта

}

# I2C

## Описание I2C

I2C / IIC (Inter-Integrated Circuit) – это протокол, изначально создававшийся для связи интегральных микросхем внутри электронного устройства. Разработка принадлежит фирме Philips. В основе I2C протокола является использование 8-битной шины, которая нужна для связи блоков в управляющей электронике, и системе адресации, благодаря которой можно общаться по одним и тем же проводам с несколькими устройствами. Мы просто передаем данные то одному, то другому устройству, добавляя к пакетам данных идентификатор нужного элемента.

В работе I2C можно выделить несколько преимуществ:

Для работы требуется всего 2 линии – SDA (линия данных) и SCL (линия синхронизации).

Подключение большого количества ведущих приборов.

Уменьшение времени разработки.

Для управления всем набором устройств требуется только один микроконтроллер.

Возможное число подключаемых микросхем к одной шине ограничивается только предельной емкостью.

Высокая степень сохранности данных из-за специального фильтра, подавляющего всплески, встроенного в схемы.

Самая простая схема I2C может содержать одно ведущее устройство и несколько ведомых (например, дисплей LCD). Каждое устройство имеет адрес в диапазоне от 7 до 127. Двух устройств с одинаковым адресом в одной схеме быть не должно.

## Описание I2C функции библиотеке

Для использования необходимо подключить библиотеку для работы I2C см. приложение №1.

wire\_set(8000000,100000) — тактовая частота контроллера, частота шины I2C

setInit() — инициализация

Clear() — очистка дисплея

led(1) — включение (1) и отключение (0) подсветки дисплея

Write(а, b) — функция собственных символов (8 символов)

а — номер символа

b — массив символа

Curs(0,0) — номер строки, положение курсора

PrintString() — вывод String

PrintFloat(a, b) — вывод Float

a — число Float

b — кол-во чисел после запятой

PrintInt() — вывод int

PrintChar() — вывод собственного символа

## Пример кода I2C:

### i2c.h

/\*

\* i2c.h

\*

\* Created: 10.11.2022 15:07:23

\* Author: dimas

\*/

#ifndef I2C\_H\_

#define I2C\_H\_

#include "main.h"

void wire\_set(long f\_clock, long i2c\_clock);

void wire\_start\_w(unsigned char i2c\_addr);

void wire\_write(unsigned char i2c\_data);

void wire\_stop();

#endif

### i2c.cpp

/\*

\* i2c.cpp

\*

\* Created: 10.11.2022 14:56:29

\* Author: dimas

\*/

#include "i2c.h"

////// I2C ////////////

void wire\_set(long f\_clock, long i2c\_clock){

TWBR = (((f\_clock)/(i2c\_clock)-16 )/2) ;

TWSR = 0;

}

void wire\_start\_w(unsigned char i2c\_addr){

TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);

while (!(TWCR & (1<<TWINT)));

TWDR = i2c\_addr << 1;

TWCR = (1<<TWINT) | (1<<TWEN);

while (!(TWCR & (1<<TWINT)));

}

void wire\_write(unsigned char i2c\_data){

TWDR = i2c\_data;

TWCR = (1<<TWINT) | (1<<TWEN);

while (!(TWCR & (1<<TWINT)));

}

void wire\_stop(){

TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTO);

}

# LCD

## Ножки LCD Дисплея

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 кОм, 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 —Ножка с помощью которой контроллер дисплея будет "знать", какие именно данные находятся 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. 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 восьми битная 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, но мы теряем скорость вдвое. 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.

Есть несколько способов реализации подключения LCD дисплея

Подключение 8 параллельных шин - для передачи байта за один заход

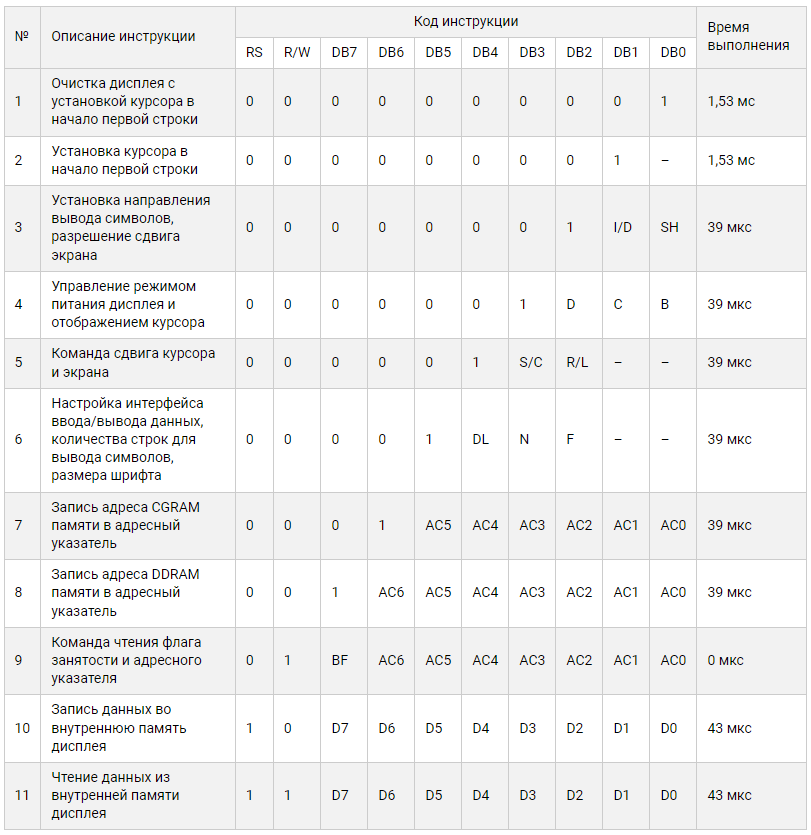
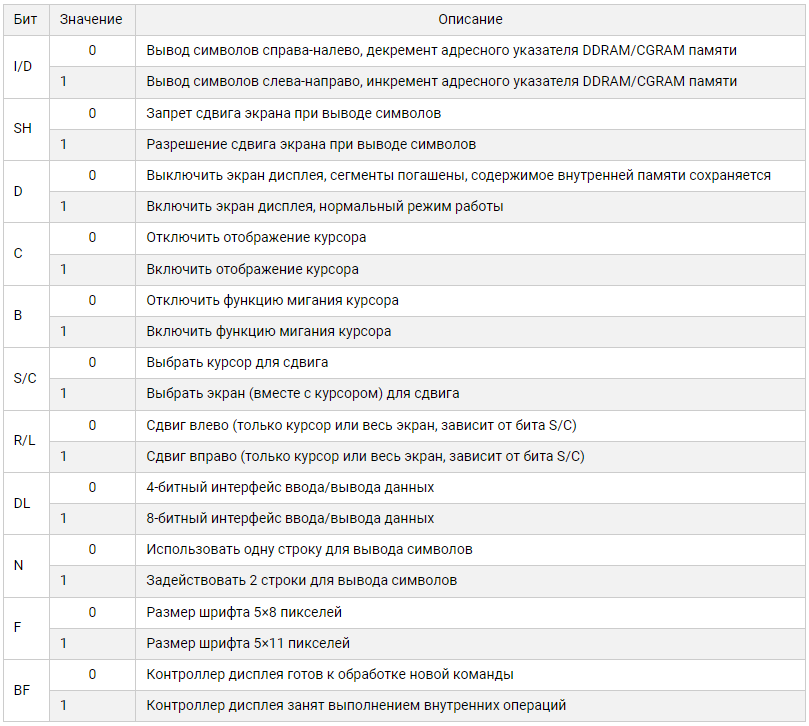
4-битный режим - мы передаём или принимаем байт в 2 приёма по половинке, сначала старшую часть байта, затем младшую

## Память

DDRAM — память дисплея. Все что запишется в DDRAM будет выведено на экран. То есть, например, записали мы туда код 0x31 — на экране выскочит символ «1» т.к. 0х31 это ASCII код цифры 1. Но есть тут одна особенность — DDRAM память гораздо больше чем видимая область экрана. Как правило, DDRAM содержит 80 ячеек — 40 в первой строке и 40 во второй, а на дисплей может двигаться по этой линейке как окошко на логарифмической линейке, высвечивая видимую область. То есть, например, можно поместить в DDRAM сразу пять пунктов меню, а потом просто переключать дисплей по разным областям, показывая по одному пункту. Для перемещения дисплея есть специальная команда. Также есть понятие курсора — это место в которое будет записан следующий символ, т.е. текущее значение счетчика адреса. Курсор не обязательно может быть на экране, он может располагаться и за экраном или быть отключен вовсе.

CGROM — таблица символов. Когда мы записываем в ячейку DDRAM байт, то из таблицы берется символ и рисуется на экране. CGROM нельзя изменить, поэтому важно, чтобы она имела на борту русские буквы. Если, конечно, планируется русскоязычный интерфейс.

CGRAM — тоже таблица символов, но ее мы можем менять, создавая свои символы. Адресуется она линейно, то есть вначале идет 8 байт одного символа, построчно, снизу-вверх — один бит равен одной точке на экране. Потом второй символ тем же методом. Поскольку знакоместо у нас 5 на 8 точек, то старшие три бита роли не играют. Всего в CGRAM может быть 8 символов, соответственно CGRAM имеет 64 байта памяти. Эти программируемые символы имеют коды от 0х00 до 0х07. Так что, закинув, например, в первые 8 байт CGRAM (первый символ с кодом 00) какую-нибудь запись, и записав в DDRAM нуль (код первого символа в CGRAM) мы увидим на экране наш символ.



## Инициализация

Инициализация любого активного устройства — это неотъемлемая часть программирования. Без первичной инициализации не будет работать ни одно устройство. Сначала показана инициализация 8-битного режима, а затем 4-битного режима.

**Ожидание готовности, чтение флага занятости.**

1. Порт данных на вход с подтяжкой (DDR=0, PORT=1)
2. RS=0 (команда)
3. RW=1 (чтение)
4. E=1
5. Пауза (14 тактов процессора на 8МГЦ должно хватить)
6. Е=0
7. Читаем из порта. Если бит 7 (Busy flag) установлен, то повторяем все заново, пока не сбросится.

**Запись команды**

1. Ожидание готовности
2. RS=0 (команда)
3. RW=0 (запись)
4. Е=1
5. Порт на выход
6. Вывести в порт код команды
7. Пауза
8. Е=0
9. Орудие на плечо Порт на вход, на всякий случай.

**Запись Данных**

1. Ожидание готовности
2. RS=1 (Данные)
3. RW=0 (запись)
4. Е=1 (почти готово)
5. Порт на выход
6. Вывести в порт код команды
7. Пауза
8. Е=0
9. Порт на вход, на всякий случай.

**Чтение команды**

1. Ожидание готовности
2. Порт данных на вход с подтяжкой (DDR=0, PORT=1)
3. RS=0 (команда)
4. RW=1 (чтение)
5. Е = 1 (В этот момент данные из LCD выводятся на шину)
6. Пауза
7. Считываем данные с порта
8. E=0

**Чтение Данных**

1. Ожидание готовности
2. Порт данных на вход с подтяжкой (DDR=0, PORT=1)
3. RS=1 (Данные)
4. RW=1 (чтение)
5. Е = 1 (В этот момент данные из LCD выводятся на шину)
6. Пауза
7. Считываем данные с порта
8. E=0 (выводим)

## Пример кода работы LCD:

### lcd.h

#ifndef LCD\_H\_

#define LCD\_H\_

/\*

\* lcd.h

\*

\* Created: 10.11.2022 14:59:39

\* Author: dimas

\*/

#define RS 0

#define E 2

#define LED 3

#define ADDR\_LED 0x27

#define F\_CPU 8000000UL

#include "main.h"

extern unsigned char led\_b;

void led(bool led\_on\_off);

void Write(unsigned char addr\_w, unsigned char wr1,unsigned char wr2,unsigned char wr3,unsigned char wr4,unsigned char wr5,unsigned char wr6,unsigned char wr7,unsigned char wr8);

void PrintInt(int data\_int);

void PrintFloat(float data\_float, unsigned char dp);

void LCD\_Init();

void Clear();

void Curs(unsigned char str, unsigned char mesto);

void PrintString(const char\* str);

void PrintChar(const char chr);

void e\_pin();

void lcd(unsigned char sett);

void lcdSend(bool rs, unsigned char data);

void i2c\_write\_1bit(unsigned char i2c\_reg);

#endif

### lcd.cpp

/\*

\* lcd.c

\*

\* Created: 10.11.2022 14:57:43

\* Author: dimas

\*/

#include "lcd.h"

unsigned char led\_b;

void led(bool led\_on\_off){

if(led\_on\_off==1){i2c\_write\_1bit(led\_b |= (1<<LED));}

if(led\_on\_off==0){i2c\_write\_1bit(led\_b &=~ (1<<LED));}

}

void Write(unsigned char addr\_w, unsigned char wr1,unsigned char wr2,unsigned char wr3,unsigned char wr4,unsigned char wr5,unsigned char wr6,unsigned char wr7,unsigned char wr8){

lcd(0b01000000|addr\_w\*8);PrintChar(wr1);PrintChar(wr2);PrintChar(wr3);PrintChar(wr4);PrintChar(wr5);PrintChar(wr6);PrintChar(wr7);PrintChar(wr8);}

void PrintInt(int data\_int){char str[6];PrintString(*itoa*(data\_int, str, 10));}

void PrintFloat(float data\_float, unsigned char dp){

char str[8];

if(data\_float<0){data\_float=-data\_float;PrintChar(0b101101);PrintString(*itoa*((int)data\_float, str, 10));}

else{PrintString(*itoa*((int)data\_float, str, 10));}

int float10 = *round*((data\_float - (int)data\_float)\**pow*(10,dp));

PrintChar(0b101110);

PrintString(*itoa*(float10, str, 10));

}

void LCD\_Init(){

lcd(0x03);*\_delay\_us*(4500);

lcd(0x03);*\_delay\_us*(4500);

lcd(0x03);*\_delay\_us*(200);

lcd(0b00000010);*\_delay\_ms*(5);

lcd(0b00001100);*\_delay\_ms*(5);

lcd(0b00000001);

}

void Clear(){lcd(0b00000001);}

void Curs(unsigned char str, unsigned char mesto){

if(str==0){lcd(0b10000000+mesto);}

if(str==1){lcd(0b11000000+mesto);}

}

void PrintString(const char\* str) {while(\*str != '\0') {*\_delay\_us*(200);PrintChar(\*str);str++;}}

void PrintChar(const char chr) {lcdSend(false, (unsigned char)chr);}

void e\_pin(){

i2c\_write\_1bit(led\_b |= (1<<E));

*\_delay\_us*(200);

i2c\_write\_1bit(led\_b &= ~(1<<E));

}

void lcd(unsigned char sett) {lcdSend(true, sett);}

void lcdSend(bool rs, unsigned char data) {

if(rs==0){led\_b |= (1<<RS);} else {led\_b &= ~(1<<RS);}//RS

*\_delay\_us*(200);

if(((data >> 7) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<7));} else {i2c\_write\_1bit(led\_b &= ~(1<<7));}

if(((data >> 6) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<6));} else {i2c\_write\_1bit(led\_b &= ~(1<<6));}

if(((data >> 5) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<5));} else {i2c\_write\_1bit(led\_b &= ~(1<<5));}

if(((data >> 4) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<4));} else {i2c\_write\_1bit(led\_b &= ~(1<<4));}

e\_pin();

if(((data >> 3) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<7));} else {i2c\_write\_1bit(led\_b &= ~(1<<7));}

if(((data >> 2) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<6));} else {i2c\_write\_1bit(led\_b &= ~(1<<6));}

if(((data >> 1) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<5));} else {i2c\_write\_1bit(led\_b &= ~(1<<5));}

if(((data >> 0) & 1) ==1){i2c\_write\_1bit(led\_b |= (1<<4));} else {i2c\_write\_1bit(led\_b &= ~(1<<4));}

e\_pin();

}

void i2c\_write\_1bit(unsigned char i2c\_reg){

wire\_start\_w(ADDR\_LED);

wire\_write(i2c\_reg);

wire\_stop();

}

# Основная часть программы – main

## main.h

#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>

#include "usart.h"

#include "adc.h"

#include "i2c.h"

#include "lcd.h"

#endif /\* MAIN\_H\_ \*/

## main.cpp

/\*

\* Osciloscope.cpp

\*

\* Created: 10.11.2022 15:15:34

\* Author : dimas

\*/

#include "main.h"

extern unsigned int adc\_value = 5;

bool adc\_value\_updated = false;

bool usart\_ready = false;

char \*ftoa(float f)

{

static char buf[17];

char \* cp = buf;

unsigned long l, rem;

if(f < 0) {

\*cp++ = '-';

f = -f;

}

l = (unsigned long)f;

f -= (float)l;

rem = (unsigned long)(f \* 1e3);

*sprintf*(cp, "%lu.%3.3lu", l, rem);

return buf;

}

ISR(USART\_RXC\_vect)

{

if(usart\_ready) { UDR; return;}

if(UDR == 0){

USART\_Transmit(0xFF);

usart\_ready = true;

Clear();

Curs(0,0);

PrintString("USART CONNECTED");

}

}

ISR(ADC\_vect)

{

char high\_adc=0, low\_adc=0;

low\_adc = ADCL;

high\_adc = ADCH;//Верхняя часть регистра ADC должна быть считана последней, иначе не продолжится преобразование

adc\_value = high\_adc \* 256 + low\_adc;

ADCSRA |= (1 << ADSC);

adc\_value\_updated = true;

}

void TransmitInt(unsigned int num){

unsigned char \*ptr = (unsigned char \*)&num;

USART\_Transmit(\*(ptr++));

USART\_Transmit(\*(ptr));

}

void loop(){

unsigned int i = 0;

sei();

while(1)

{

//Передаем при включении

if(i == 60){

// Clear(); // очистка экрана

cli();

Curs(1,0);

PrintString("VOLTAGE: ");

PrintString(ftoa(adc\_value / 1023.f \* 5.f));

sei();

i = 0;

}

if(usart\_ready && adc\_value\_updated){

adc\_value\_updated = false;

TransmitInt((unsigned int) adc\_value);

}

*\_delay\_ms*(10);

i++;

}

}

int main()

{

*\_delay\_ms*(1000);

wire\_set(8000000, 100000); // тактовая частота контроллера, частота шины I2C

USART\_Init(8); //12 - 9600 8 - 115200

ADC\_Init(); //Инициализируем АЦП

LCD\_Init();

Clear(); // очистка экрана

led(1); // включение и отключение подсветки экрана

Curs(0,0);

PrintString("WAITING...");

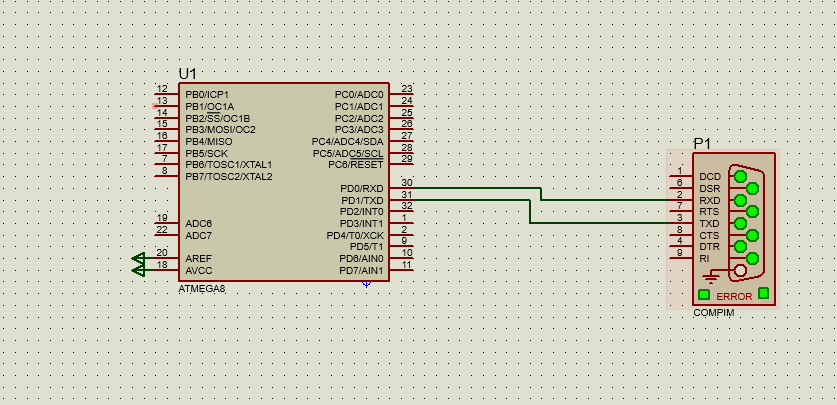
loop();

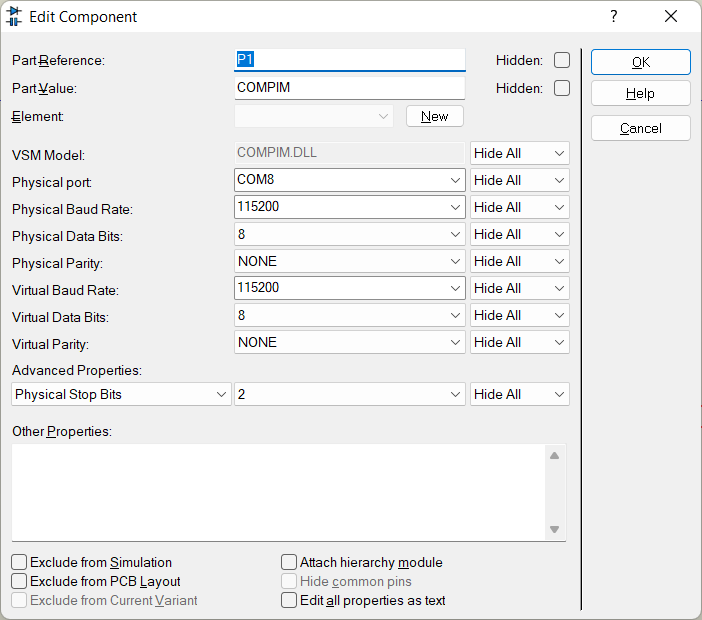
}

# Примеры передачи и приема по UART

## Передача синусоидального сигнала (генерация через функции)

### Схема:



Настройки COMPIM в Proteus

Порт COM 8 соединён с COM 7, через программу Com0Com.

### Код генератора:

void TransmitInt(unsigned int num){

unsigned char \*ptr = (unsigned char \*)&num;

USART\_Transmit(\*(ptr++));

USART\_Transmit(\*(ptr));

}

int main()

{

USART\_Init(8); //12 - 9600 8 - 115200

int i = 0;

while(true){

float sinValue = *sin*(i / 3.14 / 2);

TransmitInt((unsigned int)(sinValue \* 1023) + 2048);

// передаем число от 1024 до 4071, программа будет переводить это в значение вольтажа по формуле value / 1023 \* 5

// приподнимаем синусойду

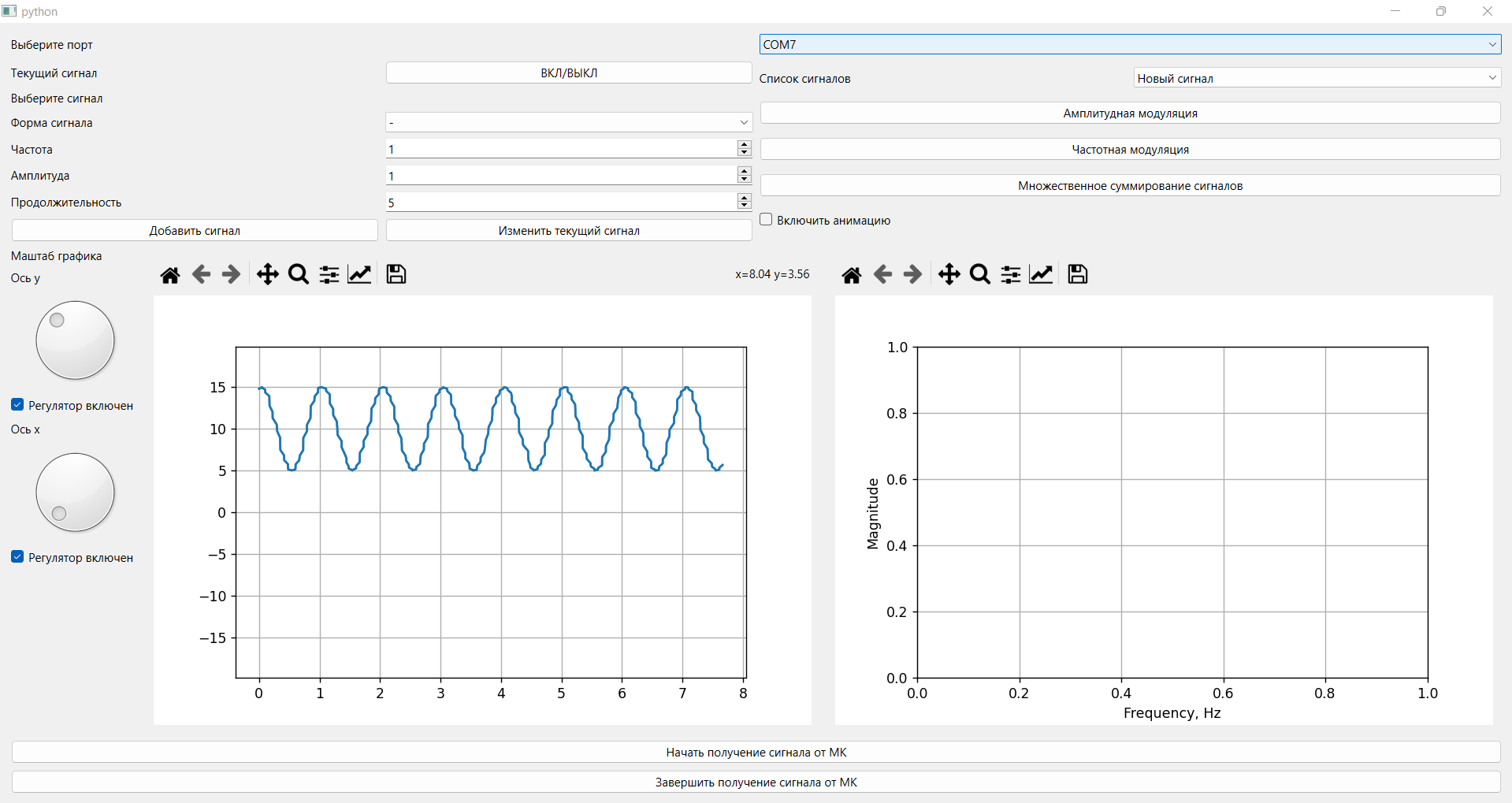
i++;

*\_delay\_ms*(20);

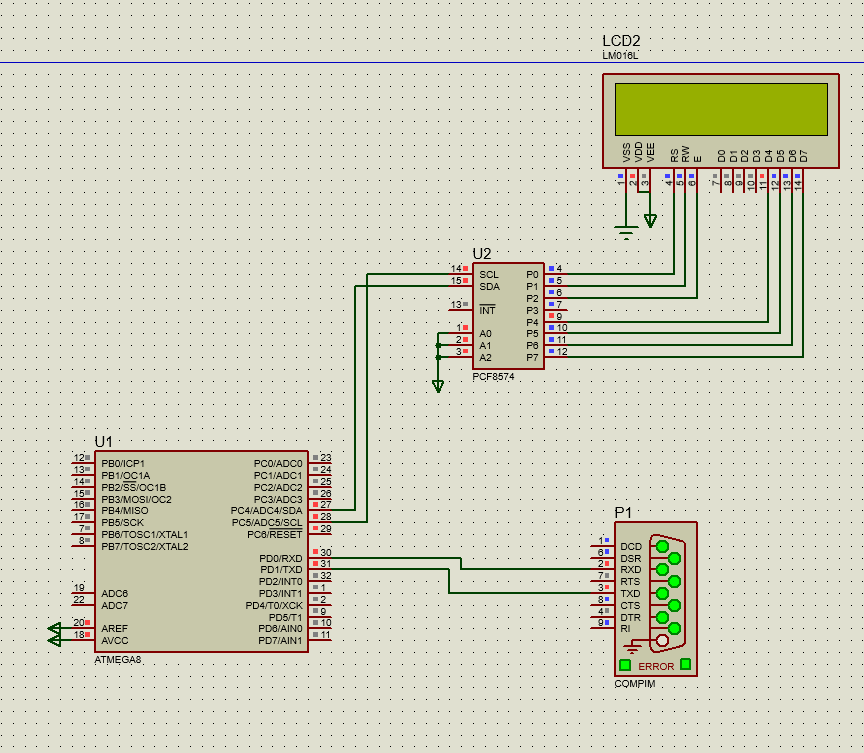
}

}

### График на принимающей сигнал программе:



## Прием и вывод данных на LCD

Схема в Proteus (первый запуск)

### Код:

int main()

{

LCD\_Init();

USART\_Init(8); //12 - 9600 8 - 115200

while(true){

int c = USART\_Read(); //принимаем данные с UART если ничего нет приходит -1

if(c != -1){

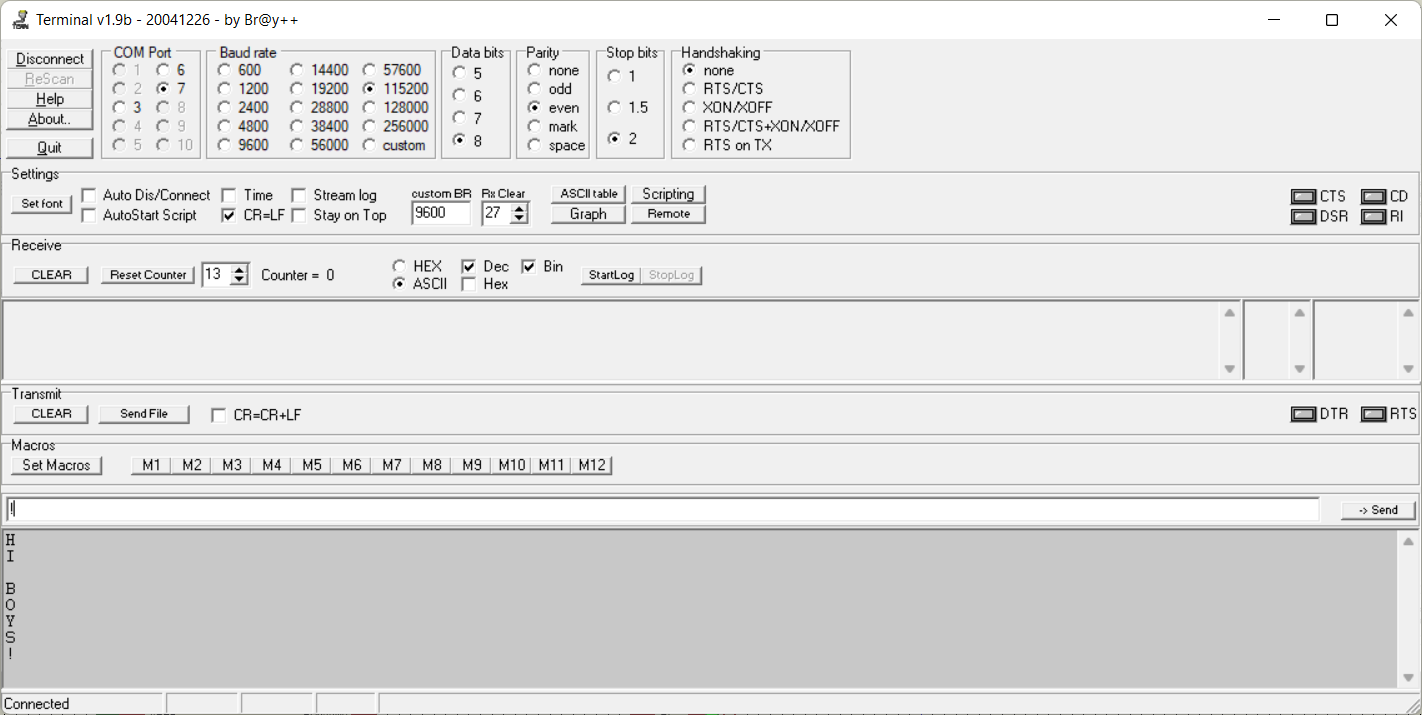
PrintChar(c); // выводим принятый символ на LCD

}

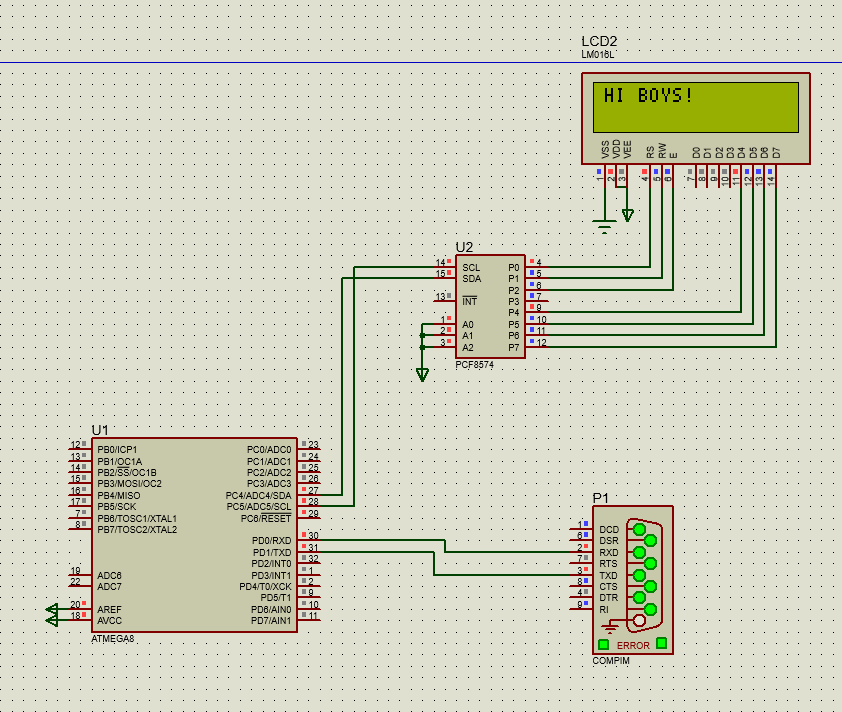
}

}

### Отправка данных через Terminal



### Результат



# Приложение

1. Библиотека для работы LCD и I2C: <https://github.com/TVGneRd/I2C-LCD_lib>
2. Исходный код проекта Osciloscope: <https://github.com/TVGneRd/Osciloscope/tree/master>