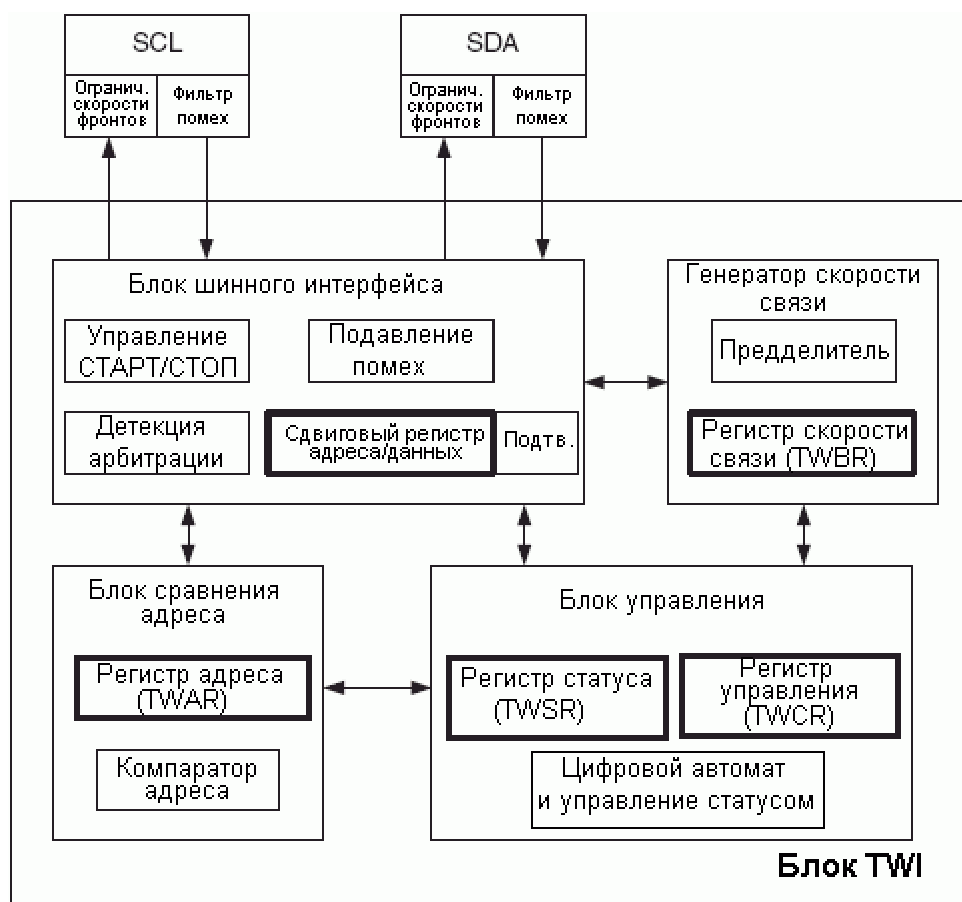
AVR: РАБОТАЕМ С ВНЕШНЕЙ ПАМЯТЬЮ I2C EEPROM типа 24CXX

Для того чтобы полностью разобраться с Two-Wire Interface (TWI) , пишем с нуля в AVR STUDIO процедуры инициализации, чтения и записи. Подробно останавливаемся на каждом шаге и разбираемся. Затем промоделируем все в Proteus.

I. Кратко теория

Аппаратный модуль TWI и протокол I2C

В микроконтроллеры серии MEGA входит модуль TWI, с помощью которого мы будем работать с шиной I2C. Модуль TWI по сути является посредником между нашей программой и подключаемым устройством (память I2C EEPROM, например).



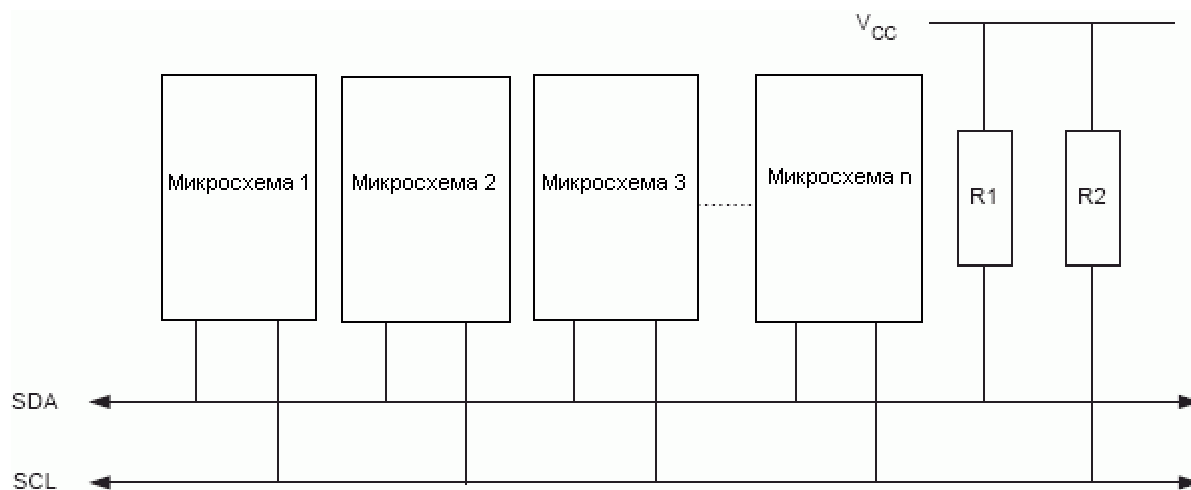
Структура модуля TWI в микроконтроллерах AVR

Шина I2C состоит из двух проводов:

SCL (Serial Clock Line) – линия последовательной передачи тактовых импульсов.

SDA (Serial Data Line) – линия последовательной передачи данных.

К этой шине мы можем одновременно подключить до 128 микросхем.



Подключения к шине TWI

Наш контроллер будем называть ведущим, а все подключаемые микросхемы ведомыми. Каждый ведомый имеет определенный адрес, зашитый на заводе в ходе изготовления (кстати, граница в 128 микросхем как раз и определяется диапазоном возможных адресов). С помощью этого адреса мы и можем работать с каждый ведомым индивидуально, хотя все они подключены к одной шине одинаковым образом. Из нашей программы мы управляем модулем TWI, а этот модуль берет на себя дергание ножками SCL и SDA.

Как видно на блок-схеме, TWI условно состоит из четырех блоков. Вначале нам нужно будет настроить Генератор скорости связи, и мы сразу забудем про него. Основная работа – с Блоком управления.

Блок шинного интерфейса управляется Блоком управления, т.е. непосредственно с ним мы контактировать не будем. А Блок сравнения адреса нужен, чтобы задать адрес на шине I2C, если тока наш контроллер будет подчиненным устройством(ведомым) от другого какого-то контроллера или будет приемником от другого ведущего (как в статье **Налаживаем обмен данными между двумя контроллерами по шине I²C** на моем блоге). В этой статье он нам не нужен. Если хотите, почитайте про него в любом даташите контроллера серии MEGA.

Итак, наша задача сейчас разобраться с регистрами, входящими в Генератор скорости связи и Блок управления:

- Регистр скорости связи TWBR
- Регистр управления TWCR
- Регистр статуса(состояния) TWSR
- Регистр данных TWDR

И все, разобравшись всего лишь с 4-мя регистрами, мы сможем полноценно работать с внешней памятью EEPROM и вообще, обмениваться данными по I2C с любым другим устройством.

Генератор скорости связи и Регистр скорости связи TWBR

Блок генератора скорости связи управляет линией SCL, а именно периодом тактовых импульсов. Управлять линией SCL может только ведущий. Период SCL управляется путем установки регистра скорости связи TWI (TWBR) и бит предделителя в регистре статуса TWI (TWSR).

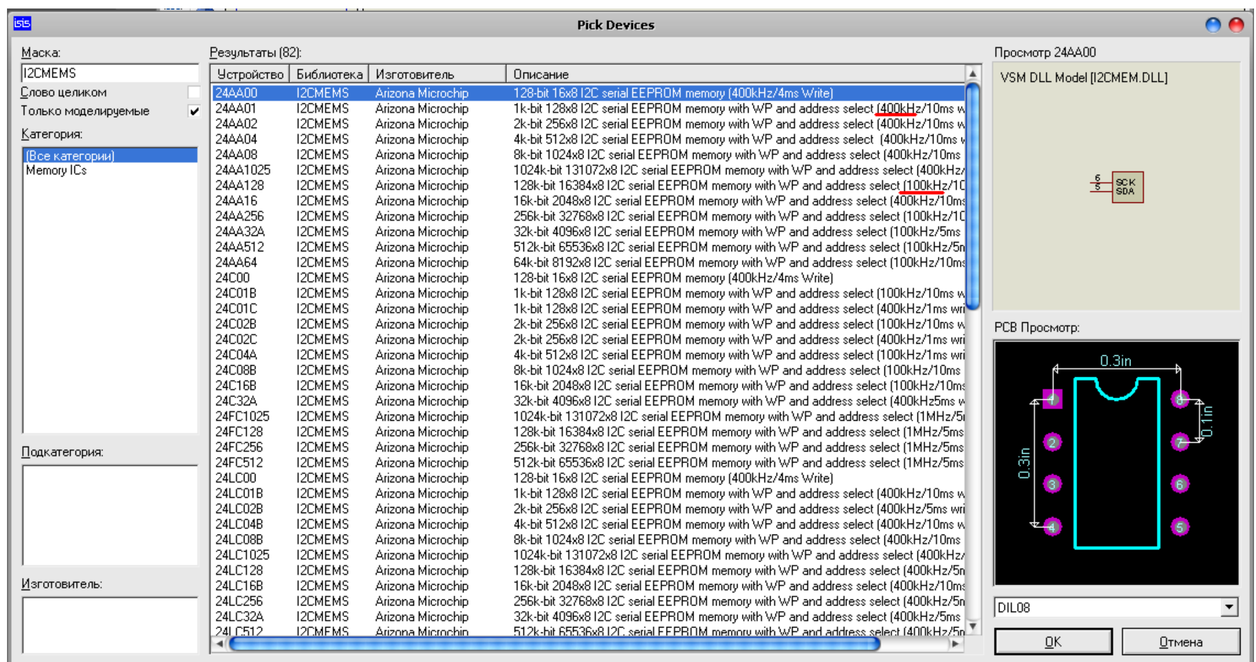
Частота SCL генерируется по следующей формуле:

$$F_{SCL}= F_{цпу}/[16+2(TWBR) \cdot 4^{TWPS}],$$

где

TWBR - значение в регистре скорости связи TWI;
TWPS - значение бит предделителя в регистре статуса TWI (TWSR).
$F_{цпу}$ – тактовая частота ведущего
F_{SCL} – частота тактовых импульсов линии SCL

Настройка TWBR нужна, т.к. ведомая микросхема обучена обмениваться данными на определенной частоте. Например, в Proteus, введите в поиск I2CMEM, увидите обилие микросхем памяти, в основном у них указаны частоты 100 и 400Khz.



Ну вот, подставляя в формулу частоты $F_{цпу}$ и F_{SCL}, мы сможем найти оптимальное значение для регистра TWBR.

TWPS – это 2-битное число [TWPS1: TWPS0], первый бит – это разряд TWPS0, второй – TWPS1 в регистре статуса TWSR. Задавая Предделитель скорости связи 4^{TWPS}, мы можем понижать значение TWBR (т.к. TWBR – это байт, максимальное значение 255). Но обычно это не требуется, поэтому TWPS обычно задается 0 и 4^{TWPS} = 1. Гораздо чаще, наоборот, мы упираемся в нижний диапазон регистра TWBR. Если TWI работает в ведущем режиме, то значение TWBR должно быть не менее 10. Если значение TWBR меньше 10, то ведущее устройство шины может генерировать некорректные сигналы на линиях SDA и SCL во время передачи байта.

TWPS1	TWPS0	$TWPS_{10}$	Значение предделителя 4^{TWPS}
0	0	0	1
0	1	1	4
1	0	2	16
1	1	3	64

Частота ЦПУ, МГц	TWBR	TWPS	Частота SCL, КГц

16.0	12	0	400
16.0	72	0	100
14.4	10	0	400
14.4	64	0	100
12.0	52	0	100
8.0	32	0	100
4.0	12	0	100
3.6	10	0	100

Ну вот, настройка TWI в этом и заключается :

- задание значения предделителя ([TWPS1: TWPS0] в регистре статуса TWSR)

Регистр состояния TWI - TWSR

Разряд	7	6	5	4	3	2	1	0
	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0
Чтение/запись	Чт.	Чт.	Чт.	Чт.	Чт.	Чт.	Чт./Зп.	Чт./Зп.
Исх. значение	1	1	1	1	1	0	0	0

- задание скорости связи (TWBR, Регистр скорости связи).

Регистр скорости связи шины TWI - TWBR

Разряд	7	6	5	4	3	2	1	0
	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
Чтение/запись	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.
Исх. значение	0	0	0	0	0	0	0	0

Блок управления

Регистр управления шиной TWI - TWCR

Регистр TWCR предназначен для управления работой TWI. Он используется для разрешения работы TWI, для инициации сеанса связи ведущего путем генерации условия СТАРТ на шине, для генерации подтверждения приема, для генерации условия СТОП и для останова шины во время записи в регистр TWDR. Он также сигнализирует о попытке ошибочной записи в регистр TWDR, когда доступ к нему был запрещен.

Регистр управления шиной TWI - TWCR

Разряд	7	6	5	4	3	2	1	0
	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
Чтение/запись	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт.	Чт./Зп.	Чт.	Чт./Зп.
Исх. значение	0	0	0	0	0	0	0	0

- Разряд 7 - TWINT: Флаг прерывания TWI

Этот флаг устанавливается аппаратно, если TWI завершает текущее задание (к примеру, передачу, принятие данных) и ожидает реакции программы. Линия SCL остается в низком состоянии, пока установлен флаг TWINT. Флаг TWINT сбрасывается программно путем записи в него логической 1. Очистка данного флага приводит к возобновлению работы TWI, т.е. программный сброс данного флага необходимо выполнить после завершения опроса регистров статуса TWSR и данных TWDR.

- Разряд 6 - TWEA: Бит разрешения подтверждения

Бит TWEA управляет генерацией импульса подтверждения. Как видно в таблице, по умолчанию он сброшен. Останавливаться на нем не буду, он в данной статье не пригодится.

- Разряд 5 - TWSTA: Бит условия СТАРТ

Мы должны установить данный бит при необходимости стать ведущим на шине I2C. TWI аппаратно проверяет доступность шины и генерирует условие СТАРТ, если шина свободна. Мы проверяем это условие (по регистру статуса, будет далее) и если шина свободна, то мы можем начинать с ней работать. Иначе нужно будет подождать, пока шина освободится.

- Разряд 4 - TWSTO: Бит условия СТОП

Установка бита TWSTO в режиме ведущего приводит к генерации условия СТОП на шине I2C. Если на шине выполняется условие СТОП, то бит TWSTO сбрасывается автоматически и шина освобождается.

- Разряд 3 - TWWC: Флаг ошибочной записи

Бит TWWC устанавливается при попытке записи в регистр данных TWDR, когда TWINT имеет низкий уровень. Флаг сбрасывается при записи регистра TWDR, когда TWINT = 1.

- Разряд 2 - TWEN: Бит разрешения работы TWI

Бит TWEN разрешает работу TWI и активизирует интерфейс TWI. Если бит TWEN установлен, то TWI берет на себя функции управления линиями ввода-вывода SCL и SDA. Если данный бит равен нулю, то TWI отключается и все передачи прекращаются независимо от состояния работы.

- Разряд 1 - Резервный бит
- Разряд 0 - TWIE: Разрешение прерывания TWI

Если в данный бит записана лог. 1 и установлен бит I в регистре SREG (прерывания разрешены глобально), то разрешается прерывание от модуля TWI (*ISR (TWI_vect)*) при любом изменении регистра статуса.

Регистр состояния TWI – TWSR

Регистр состояния TWI - TWSR

Разряд	7	6	5	4	3	2	1	0
	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0

Чтение/запись	Чт.	Чт.	Чт.	Чт.	Чт.	Чт.	Чт./Зп.	Чт./Зп.
Исх. значение	1	1	1	1	1	0	0	0

- Разряды 7..3 - TWS: Состояние TWI

Данные 5 бит отражают состояние логики блока TWI и шины I2C. Диапазон кодов состояния 0b0000 0000 – 0b1111 1000, или с 0x00 до 0xF8. Привожу их ниже:

```
// TWSR values (not bits)
// (taken from avr-libc twi.h - thank you Marek Michalkiewicz)
// Master (Ведущий)
#define TW_START                         0x08  //условие СТАРТ
#define TW_REP_START                     0x10  //условие ПОВТОРНЫЙ СТАРТ (повтор условия начала передачи)
// Master Transmitter (Ведущий в роли передающего)
#define TW_MT_SLA_ACK                    0x18  //Ведущий отправил адрес и бит записи. Ведомый подтвердил свой адрес
#define TW_MT_SLA_NACK                   0x20  //Ведущий отправил адрес и бит записи. Нет подтверждения приема (ведомый с таким адресом не найден)
#define TW_MT_DATA_ACK                   0x28  //Ведущий передал данные и ведомый подтвердил прием.
#define TW_MT_DATA_NACK                  0x30  //Ведущий передал данные. Нет подтверждения приема
#define TW_MT_ARB_LOST                   0x38  //Потеря приоритета
// Master Receiver (Ведущий в роли принимающего)
#define TW_MR_ARB_LOST                   0x38  //Потеря приоритета
#define TW_MR_SLA_ACK                    0x40  // Ведущий отправил адрес и бит чтения. Ведомый подтвердил свой адрес
#define TW_MR_SLA_NACK                   0x48  //Ведущий отправил адрес и бит чтения. Нет подтверждения приема (ведомый с таким адресом не найден)
#define TW_MR_DATA_ACK                   0x50  //Ведущий принял данные и передал подтверждение
#define TW_MR_DATA_NACK                  0x58  //Ведущий принял данные и не передал подтверждение
// Slave Transmitter (Ведомый в роли передающего)
#define TW_ST_SLA_ACK                    0xA8  //Получение адреса и бита чтения, возвращение подтверждения
#define TW_ST_ARB_LOST_SLA_ACK           0xB0  //Потеря приоритета, получение адреса и бита чтения, возвращение подтверждения
#define TW_ST_DATA_ACK                   0xB8  //Данные переданы, подтверждение от ведущего принято
#define TW_ST_DATA_NACK                  0xC0  //Данные переданы. Нет подтверждения приема от ведущего.
#define TW_ST_LAST_DATA                  0xC8  //Последний переданный байт данных, получение подтверждения
// Slave Receiver (Ведомый в роли принимающего)
#define TW_SR_SLA_ACK                    0x60  //Получение адреса и бита записи, возвращение подтверждения
#define TW_SR_ARB_LOST_SLA_ACK           0x68  //Потеря приоритета, получение адреса и бита записи, возвращение подтверждения
#define TW_SR_GCALL_ACK                  0x70  //Прием общего запроса, возвращение подтверждения
#define TW_SR_ARB_LOST_GCALL_ACK         0x78  //Потеря приоритета, прием общего запроса, возвращение подтверждения
#define TW_SR_DATA_ACK                   0x80  // Прием данных, возвращение подтверждения
#define TW_SR_DATA_NACK                  0x88  //Данные переданы без подтверждения.
#define TW_SR_GCALL_DATA_ACK             0x90  //Прием данных при общем запросе, возвращение подтверждения
#define TW_SR_GCALL_DATA_NACK            0x98  // Прием данных при общем запросе, без подтверждения
#define TW_SR_STOP                       0xA0  //Условие СТОП
// Misc (Ошибки интерфейса)
#define TW_NO_INFO                       0xF8  //Информация о состоянии отсутствует
#define TW_BUS_ERROR                     0x00  //Ошибка шины
```

Если мы работаем с пассивным ведомым (т.е. это не другой контроллер AVR, а микросхема памяти или например, часы RTC), то коды состояний из разделов Slave Transmitter (Ведомый в роли передающего) и Slave Receiver (Ведомый в роли принимающего) нам не понадобятся, поскольку единственная «разумная» микросхема у нас – это Ведущий (наш контроллер).

Проверяя регистр статуса после каждой выполненной операции с шиной, мы можем контролировать обмен информацией. Например, будем точно знать – записались ли данные во внешнюю память или нет.

Регистр данных шины TWI - TWDR

Разряд	7	6	5	4	3	2	1	0
	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0
Чтение/запись	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.	Чт./Зп.
Исх. значение	1	1	1	1	1	1	1	1

В режиме передатчика регистр TWDR содержит байт для передачи. В режиме приемника регистр TWDR содержит последний принятый байт. Будьте внимательны, после аппаратной установки флага TWINT, регистр TWDR не содержит ничего определенного.

Ну вот, этих знаний более чем достаточно, чтобы работать с I2C EEPROM. Теперь переходим непосредственно к программной части. Я решил пояснения писать прямо в коде в виде комментариев.

II. Программа

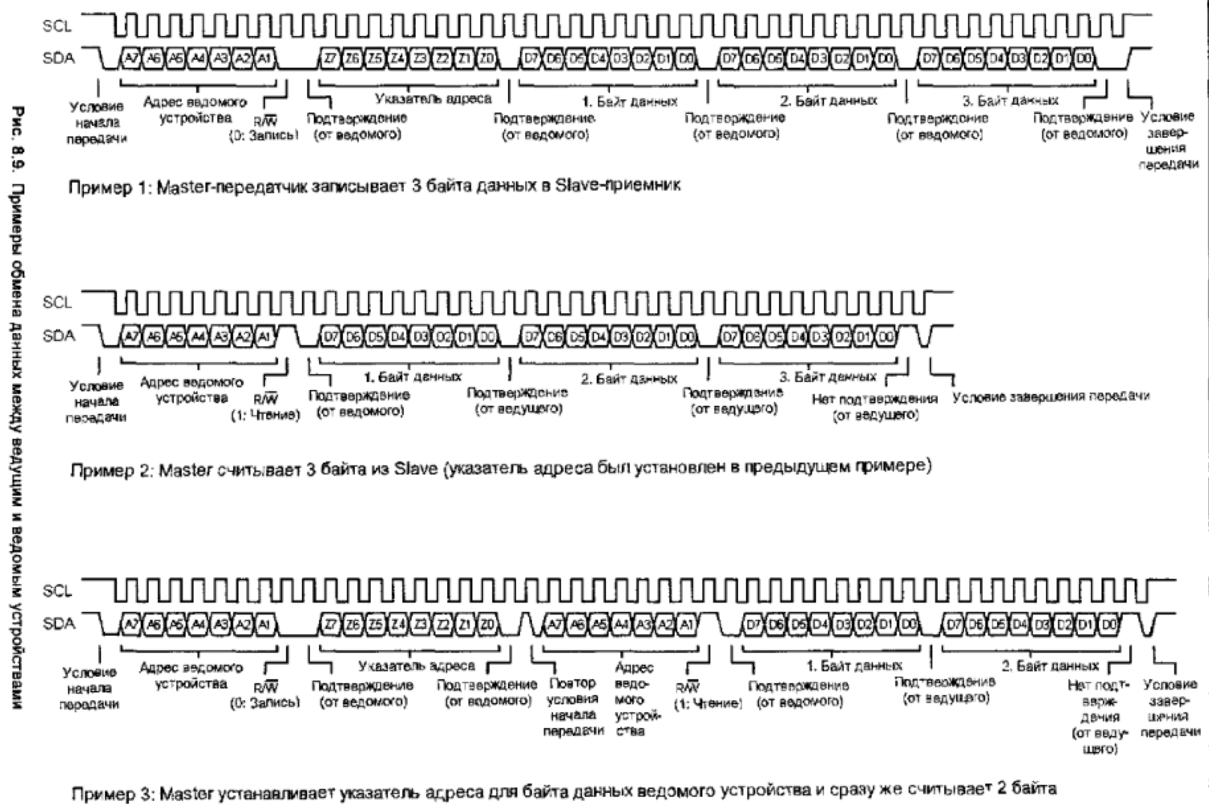
Все функции (инициализация TWI, чтение, запись внешней памяти) я вынесу в отдельные файлы, как это и принято делать, i2c_eeprom.c и i2c_eeprom.h.

Содержание i2c_eeprom.h следующее:

```
#define false 0
#define true 1

//#define slaveF_SCL 100000 //100 Khz
#define slaveF_SCL 400000 //400 Khz

#define slaveAddressConst 0b1010 //Постоянная часть адреса ведомого
устройства
#define slaveAddressVar 0b000 //Переменная часть адреса ведомого устройства

//Разряды направления передачи данных
#define READFLAG 1 //Чтение
#define WRITEFLAG 0 //Запись

void eeInit(void); //Начальная настройка TWI
uint8_t eeWriteByte(uint16_t address,uint8_t data); //Запись байта в модуль
памяти EEPROM
uint8_t eeReadByte(uint16_t address); //Чтение байта из модуля памяти EEPROM

// TWSR values (not bits)
// (taken from avr-libc twi.h - thank you Marek Michalkiewicz)
// Master
#define TW_START                    0x08
#define TW_REP_START                0x10
// Master Transmitter
#define TW_MT_SLA_ACK               0x18
#define TW_MT_SLA_NACK              0x20
#define TW_MT_DATA_ACK              0x28
#define TW_MT_DATA_NACK             0x30
#define TW_MT_ARB_LOST              0x38
// Master Receiver
#define TW_MR_ARB_LOST              0x38
#define TW_MR_SLA_ACK               0x40
#define TW_MR_SLA_NACK              0x48
#define TW_MR_DATA_ACK              0x50
#define TW_MR_DATA_NACK             0x58
// Slave Transmitter
```

```
#define TW_ST_SLA_ACK               0xA8
#define TW_ST_ARB_LOST_SLA_ACK      0xB0
#define TW_ST_DATA_ACK              0xB8
#define TW_ST_DATA_NACK             0xC0
#define TW_ST_LAST_DATA             0xC8
// Slave Receiver
#define TW_SR_SLA_ACK               0x60
#define TW_SR_ARB_LOST_SLA_ACK      0x68
#define TW_SR_GCALL_ACK             0x70
#define TW_SR_ARB_LOST_GCALL_ACK    0x78
#define TW_SR_DATA_ACK              0x80
#define TW_SR_DATA_NACK             0x88
#define TW_SR_GCALL_DATA_ACK        0x90
#define TW_SR_GCALL_DATA_NACK       0x98
#define TW_SR_STOP                  0xA0
// Misc
#define TW_NO_INFO                  0xF8
#define TW_BUS_ERROR                0x00
```

Чтобы было легче воспринимать нижеследующий код, приведу здесь теоретические примеры осциллограмм при обмене данными между Ведущим и Ведомым по шине I2C (взял в книжке [1], в более высоком разрешении найдете по адресу, указанном в конце статьи):



Далее привожу листинг файла i2c_eeprom.c с подробными комментариями:

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

#include "i2c_eeprom.h"

void eeInit(void)
{
```

```
    /*Настраиваем Генератор скорости связи*/
    TWBR = (F_CPU/slaveF_SCL - 16)/(2 * /* TWI_Prescaler= 4^TWPS */1);

    /*
    Если TWI работает в ведущем режиме, то значение TWBR должно быть не менее
    10. Если значение TWBR меньше 10, то ведущее устройство шины может
    генерировать некорректные сигналы на линиях SDA и SCL во время передачи
    байта.
    */
    if(TWBR < 10)
        TWBR = 10;

    /*
    Настройка предделителя в регистре статуса Блока управления.
    Очищаются биты TWPS0 и TWPS1 регистра статуса, устанавливая тем самым,
    значение предделителя = 1.
    */
    TWSR &= (~((1<<TWPS1)|(1<<TWPS0)));
}

uint8_t eeWriteByte(uint16_t address,uint8_t data)
{

    /*****УСТАНАВЛИВАЕМ СВЯЗЬ С ВЕДОМЫМ********/

    do
    {
        //Инициализация Регистра управления шиной в Блоке управления
        /*Перед началом передачи данных необходимо сформировать т.н. условие
        начала. В состоянии покоя линии SCL и SDA находятся на высоком уровне.
        Ведущее устройство (Контроллер AVR в нашем примере), которое хочет
        начать передачу данных, изменяет состояние линии SDA к низкому уровню.
        Это и есть условие начала передачи данных.*/

        /*
        а)Сброс флага прерывания TWINT (Флаг TWINT сбрасывается программно
        путем записи в него логической 1) для разрешения начала новой передачи
        данных
        б)Уст. бит условия СТАРТ
        в)Уст. бит разрешения работы TWI
        */
        TWCR=(1<<TWINT)|(1<<TWSTA)|(1<<TWEN);

        //Ждем, пока шина даст добро (возможно, линия пока еще занята, ждем)
        //TWINT бит устанавливается аппаратно, если TWI завершает текущее
        задание и ожидает реакции программы
        while(!(TWCR & (1<<TWINT)));

        /*Проверяем регистр статуса, а точнее биты TWS3-7,
        которые доступны только для чтения. Эти пять битов
        отражают состояние шины. TWS2-0 «отсекаем» с помощью операции «И
        0xF8». Если TWS7-3 = 0x08, то СТАРТ был успешным.*/
        if((TWSR & 0xF8) != TW_START)
            return false;

        /*К шине I2C может быть подключено множество подчиненных устройств (к
        примеру, много микросхем внешней памяти EEPROM). Для того, чтобы все
        микросхемы и контроллер знали, от кого и кому передается информация, в
        протоколе реализована Адресация ведомых устройств. В каждой
        микросхеме, предназначенной для работы с I2C, на заводе "зашит"
        определенный адрес. Мы этот адрес передаем по всей шине, т.е. всем
        ведомым. Каждый ведомый получает этот адрес и смотрит, типа мой это
        или чужой. Если мой, то О КРУТО, со мной хочет работать контроллер
        AVR. Так вот и происходит "рукопожатие" между ведущим и ведомым.*/
```

```
    /*Так вот, мы хотим работать с микросхемой памяти 24LC64, поэтому по
    шине нам надо передать ее адрес. Она узнает свой адрес, и будет знать,
    что данные на запись адресуются именно ей. А остальные микросхемы,
    если они есть, эти данные будут просто игнорировать.*/

    /*Постоянная часть адреса 24LC64 - 1010 (см. даташит на 24XX64), 3
    бита - переменные (если вдруг мы захотим подключить несколько
    одинаковых микросхем с одинаковыми заводскими адресами, они
    пригодятся; в ином(нашем) случае выставляем нули), далее бит 0 - если
    хотим записывать в память или 1 - если читаем данные из памяти I2C
    EEPROM*/

    //TWDR = 0b1010‘000‘0;
    TWDR = (slaveAddressConst<<4) + (slaveAddressVar<<1) + (WRITEFLAG);

    /*Говорим регистру управления, что мы хотим передать данные,
    содержащиеся в регистре данных TWDR*/
    TWCR=(1<<TWINT)|(1<<TWEN);

    //Ждем окончания передачи данных
    while(!(TWCR & (1<<TWINT)));

    /*Если нет подтверждения от ведомого, делаем все по-новой (либо
    неполадки с линией, либо ведомого с таким адресом нет).
    Если же подтверждение поступило, то регистр статуса установит биты в
    0x18=TW_MT_SLA_ACK (в случае записи) или 0x40=TW_MR_SLA_ACK (в случае
    чтения).
    Грубо говоря, если TW_MT_SLA_ACK, то ведомый "говорит" нам, что его
    адрес как раз 1010’000 и он готов для записи (чтения, если
    TW_MR_SLA_ACK).*/
}while((TWSR & 0xF8) != TW_MT_SLA_ACK);

/*Здесь можем уже уверенно говорить, что ведущий и ведомый друг друга
видят и понимают. Вначале скажем нашей микросхеме памяти, по какому адресу
мы хотим записать байт данных*/


/*****ПЕРЕДАЕМ АДРЕС ЗАПИСИ********/

/*Записываем в регистр данных старший разряд адреса (адрес 16-битный,
uint16_t))..*/
TWDR=(address>>8);

//..и передаем его
TWCR=(1<<TWINT)|(1<<TWEN);

//ждем окончания передачи
while(!(TWCR & (1<<TWINT)));

/*Проверяем регистр статуса, принял ли ведомый данные. Если ведомый данные
принял, то он передает "Подтверждение", устанавливая SDA в низкий уровень.
Блок управления, в свою очередь, принимает подтверждение, и записывает в
регистр статуса 0x28= TW_MT_DATA_ACK. В противном случае 0x30=
TW_MT_DATA_NACK */
if((TWSR & 0xF8) != TW_MT_DATA_ACK)
    return false;

//Далее тоже самое для младшего разряда адреса
TWDR=(address);
TWCR=(1<<TWINT)|(1<<TWEN);
while(!(TWCR & (1<<TWINT)));

if((TWSR & 0xF8) != TW_MT_DATA_ACK)
```

```
        return false;


    /*****ЗАПИСЫВАЕМ БАЙТ ДАННЫХ********/

    //Аналогично, как и передавали адрес, передаем байт данных
    TWDR=(data);
    TWCR=(1<<TWINT)|(1<<TWEN);
    while(!(TWCR & (1<<TWINT)));

    if((TWSR & 0xF8) != TW_MT_DATA_ACK)
        return false;

    /*Устанавливаем условие завершения передачи данных (СТОП)
    (Устанавливаем бит условия СТОП)*/
    TWCR=(1<<TWINT)|(1<<TWEN)|(1<<TWSTO);

    //Ждем установки условия СТОП
    while(TWCR & (1<<TWSTO));

    return true;
}

uint8_t eeReadByte(uint16_t address)
{
    uint8_t data; //Переменная, в которую запишем прочитанный байт

    //Точно такой же кусок кода, как и в eeWriteByte...
    /*****УСТАНАВЛИВАЕМ СВЯЗЬ С ВЕДОМЫМ********/
    do
    {
        TWCR=(1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
        while(!(TWCR & (1<<TWINT)));

        if((TWSR & 0xF8) != TW_START)
            return false;

        TWDR = (slaveAddressConst<<4) + (slaveAddressVar<<1) + WRITEFLAG;
        TWCR=(1<<TWINT)|(1<<TWEN);

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

    }while((TWSR & 0xF8) != TW_MT_SLA_ACK);


    /*****ПЕРЕДАЕМ АДРЕС ЧТЕНИЯ********/
    TWDR=(address>>8);
    TWCR=(1<<TWINT)|(1<<TWEN);
    while(!(TWCR & (1<<TWINT)));

    if((TWSR & 0xF8) != TW_MT_DATA_ACK)
        return false;

    TWDR=(address);
    TWCR=(1<<TWINT)|(1<<TWEN);
    while(!(TWCR & (1<<TWINT)));

    if((TWSR & 0xF8) != TW_MT_DATA_ACK)
        return false;


    /*****ПЕРЕХОД В РЕЖИМ ЧТЕНИЯ********/
    /*Необходимо опять «связаться» с ведомым, т.к. ранее мы отсылали адресный
    пакет (slaveAddressConst<<4) + (slaveAddressVar<<1) + WRITEFLAG, чтобы
```

```
записать адрес чтения байта данных. А теперь нужно перейти в режим чтения
(мы же хотим прочитать байт данных), для этого отсылаем новый пакет
(slaveAddressConst<<4) + (slaveAddressVar<<1) + READFLAG.*/

 //Повтор условия начала передачи
 TWCR=(1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
//ждем выполнения текущей операции
 while(!(TWCR & (1<<TWINT)));

/*Проверяем статус. Условие повтора начала передачи (0x10=TW_REP_START)
должно подтвердиться*/
 if((TWSR & 0xF8) != TW_REP_START)
     return false;

 /*Записываем адрес ведомого (7 битов) и в конце бит чтения (1)*/
 //TWDR=0b1010'000'1;
 TWDR = (slaveAddressConst<<4) + (slaveAddressVar<<1) + READFLAG;

//Отправляем..
 TWCR=(1<<TWINT)|(1<<TWEN);
 while(!(TWCR & (1<<TWINT)));

/*Проверяем, нашелся ли ведомый с адресом 1010'000 и готов ли он работать
на чтение*/
 if((TWSR & 0xF8) != TW_MR_SLA_ACK)
     return false;


/*****СЧИТЫВАЕМ БАЙТ ДАННЫХ********/

/*Начинаем прием данных с помощью очистки флага прерывания TWINT. Читаемый
байт записывается в регистр TWDR.*/
 TWCR=(1<<TWINT)|(1<<TWEN);

 //Ждем окончания приема..
 while(!(TWCR & (1<<TWINT)));

/*Проверяем статус. По протоколу, прием данных должен оканчиваться без
подтверждения со стороны ведущего (TW_MR_DATA_NACK = 0x58)*/
 if((TWSR & 0xF8) != TW_MR_DATA_NACK)
     return false;

 /*Присваиваем переменной data значение, считанное в регистр данных TWDR*/
 data=TWDR;

 /*Устанавливаем условие завершения передачи данных (СТОП)*/
 TWCR=(1<<TWINT)|(1<<TWEN)|(1<<TWSTO);

 //Ждем установки условия СТОП
 while(TWCR & (1<<TWSTO));

 //Возвращаем считанный байт
 return data;
}
```

Ну вот, самые главные функции мы написали. На этой базе можно написать функции для чтения\записи массива байтов. Также можно добавить прерывание ISR(TWI_INT), которое будет срабатывать при каждом изменении регистра статуса. Я скажу только пару слов об этом, поскольку разобравшись в вышеизложенном, вам не составит труда реализовать их самому.

Итак, пару слов о чтении\записи массива байтов. Очень просто взять и вогнать в цикл функции `eeReadByte\eeWriteByte`. Это даже будет работать ☺. Но, посмотрите, TWI каждый раз будет формировать условие СТАРТ, устанавливать связь с ведомым, отсылать адрес чтения\записи... Словом, огромная потеря времени. Вы же не покупаете продукты в магазине по частям ☺, нет - вы складываете все продукты (байты) в кулек (в массив) и несете домой. Дак давайте также сложим все наши байты в кулек! Изменения в новой функции `eeWriteBytes` будут следующими, часть кода затаскиваем в цикл:

```
    /*****ЗАПИСЫВАЕМ БАЙТЫ ДАННЫХ********/

    //Аналогично, как и передавали адрес, передаем байты данных
    for(i=0;i<sizeof(ArrayBytes);i++)
    {
        TWDR=(ArrayBytes[i]);
        TWCR=(1<<TWINT)|(1<<TWEN);
        while(!(TWCR & (1<<TWINT)));

        if((TWSR & 0xF8) != TW_MT_DATA_ACK)
            return false;
    }
```

В процедуре чтения изменения будут чуточку сложнее, поскольку между считыванием байтов данных должно быть подтверждение от ведущего, а после считывания последнего байта подтверждения быть не должно, далее идет условие завершения передачи (условие СТОП).

Про прерывание ISR(TWI_INT) говорить ничего не буду, просто приведу пример использования (обычно этого достаточно, сразу становится все понятно):

```
ISR(TWI)
{
    switch(TWSR & 0xF8)
    {
        case TW_START:
            lcd_puts("TW_START\n");
            break;
        case TW_REP_START:
            lcd_puts("TW_REP_START\n");
            break;

        case TW_MT_SLA_ACK:
            lcd_puts("TW_MT_SLA_ACK\n");
            break;
        case TW_MT_DATA_ACK:
            lcd_puts("TW_MT_DATA_ACK\n");
            break;

        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        //и т.д.
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    }
}
```

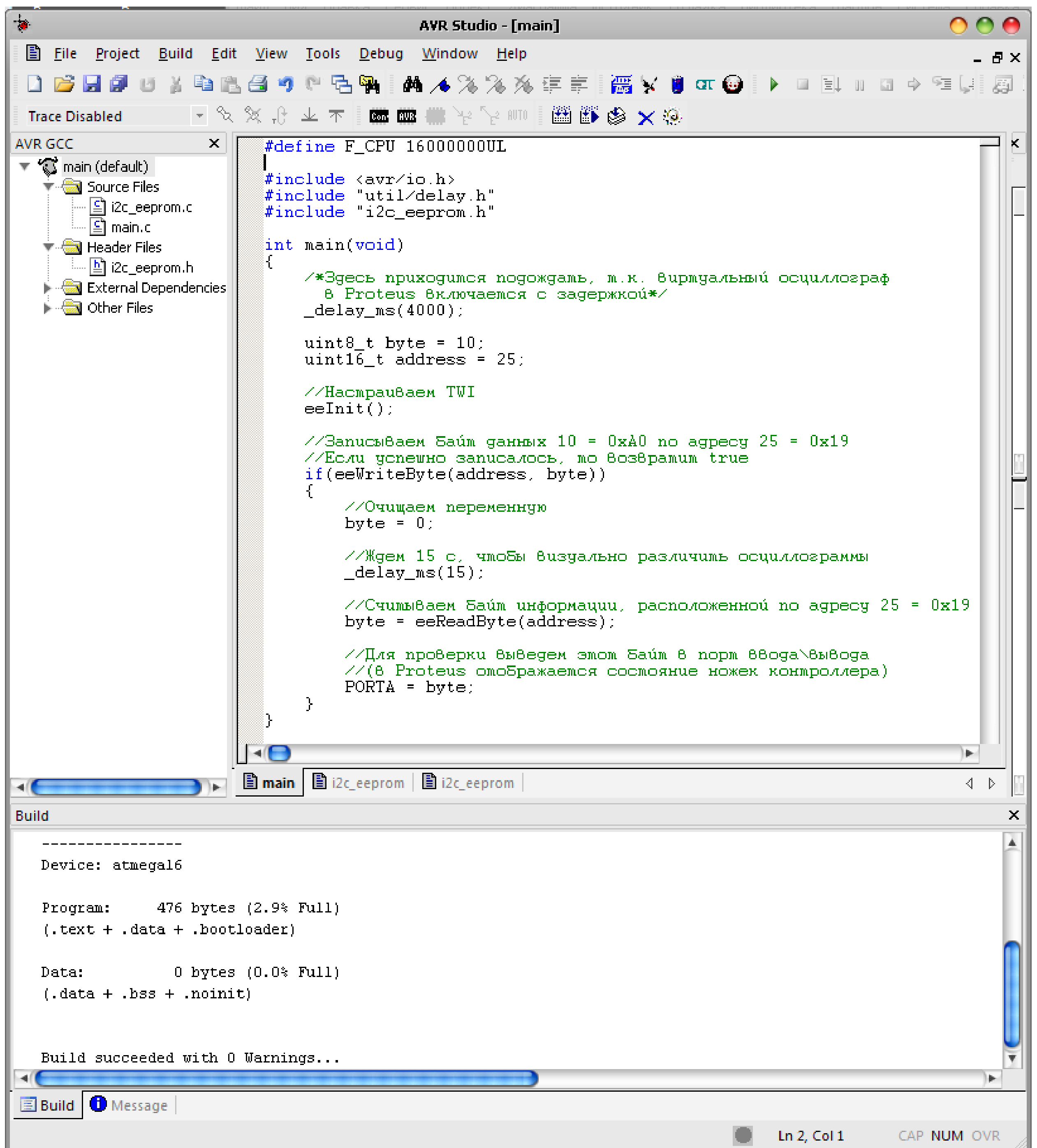
Все, остается создать проект в AVR STUDIO:

В настройках проекта укажите какой-нибудь MEGA (atmega16 например), подключите файлы i2c_eeprom.c и i2c_eeprom.h.

```
#define F_CPU 16000000UL

#include <avr/io.h>
#include "util/delay.h"
#include "i2c_eeprom.h"

int main(void)
{
    _delay_ms(300); /*Здесь приходится подождать, т.к. виртуальный осциллограф
    в Proteus включается с задержкой*/

    uint8_t byte = 10;
    uint16_t address = 25;

    //Настраиваем TWI
    eeInit();

    //Записываем байт данных 10 = 0xA0 по адресу 25 = 0x19
    //Если успешно записалось, то возвратит true
    if(eeWriteByte(address, byte))
    {
        //Очищаем переменную
        byte = 0;

        //Ждем 15 с, чтобы визуально различить осциллограммы
        _delay_ms(15);

        //Считываем байт информации, расположенной по адресу 25 = 0x19
        byte = eeReadByte(address);

        //Для проверки выведем этот байт в порт ввода\вывода
        //(в Proteus отображается состояние ножек контроллера)
        PORTA = byte;
    }
}
```

В программе я явно указал частоту тактирования моего контроллера 16 Mhz. Далее, в Proteus, мы выберем какую-нибудь микросхему внешней памяти I2C EEPROM. Не забудьте после этого сравнить настройки в i2c_eeprom.h с параметрами выбранной микросхемы (*slaveF_SCL*, *slaveAddressConst* – достоверную информацию всегда можно узнать из даташитов).

Итак, остается собрать проект и переходим к моделированию..



III. Моделируем работу с I2C EEPROM в Proteus

Добавляем на схему ATmega16, 2 резистора для подтяжки шины I2C (см. схему в начале статьи). Из вкладки виртуальные инструменты возьмите Осциллограф и I2C-Отладчик. Для выбора памяти введите в поиск I2CMEMS в окне выбора устройств:

Маска:

I2CMEMS

Слово целиком ☐

Только моделируемые ☑

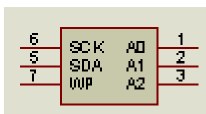
Категория:

(Все категории)
Memory ICs

Из всего списка я выбрал 24LC64 с объемом памяти 64 КБ и частотой работы с шиной I2C 400 КГц.

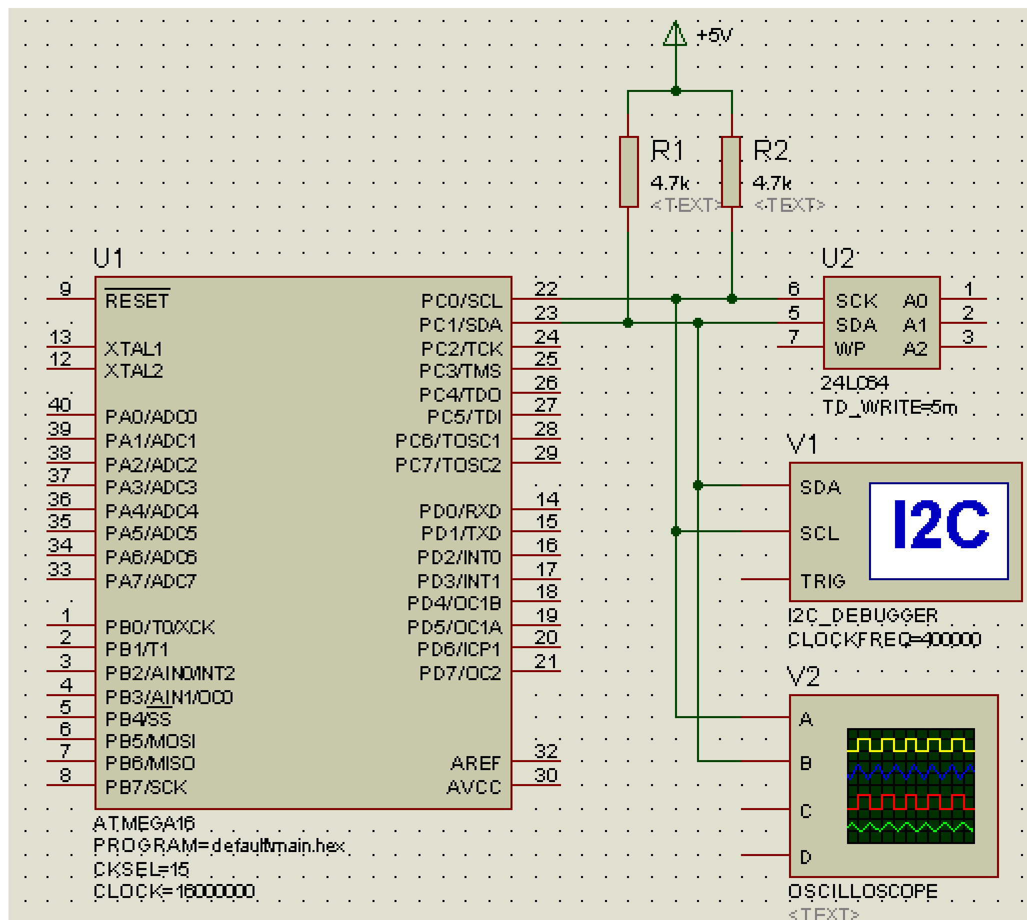
Результаты (82):

Устройство	Изготовитель	Описание
24LC00	Arizona Microchip	128-bit 16x8 I2C serial EEPROM memory (400kHz/4ms Write)
24LC01B	Arizona Microchip	1k-bit 128x8 I2C serial EEPROM memory with WP and address select (400kHz/10ms write)
24LC02B	Arizona Microchip	2k-bit 256x8 I2C serial EEPROM memory with WP and address select (400kHz/5ms write)
24LC04B	Arizona Microchip	4k-bit 512x8 I2C serial EEPROM memory with WP and address select (400kHz/10ms write)
24LC08B	Arizona Microchip	8k-bit 1024x8 I2C serial EEPROM memory with WP and address select (400kHz/10ms write)
24LC1025	Arizona Microchip	1024k-bit 131072x8 I2C serial EEPROM memory with WP and address select (400kHz/5ms write)
24LC128	Arizona Microchip	128k-bit 16384x8 I2C serial EEPROM memory with WP and address select (400kHz/5ms write)
24LC16B	Arizona Microchip	16k-bit 2048x8 I2C serial EEPROM memory with WP and address select (400kHz/10ms write)
24LC256	Arizona Microchip	256k-bit 32768x8 I2C serial EEPROM memory with WP and address select (400kHz/5ms write)
24LC32A	Arizona Microchip	32k-bit 4096x8 I2C serial EEPROM memory with WP and address select (400kHz/5ms write)
24LC512	Arizona Microchip	512k-bit 65536x8 I2C serial EEPROM memory with WP and address select (400kHz/5ms write)
24LC64	Arizona Microchip	64k-bit 8192x8 I2C serial EEPROM memory with WP and address select (400kHz/5ms write)
AT24C1024	ATMEL	1M-bit 131 072x8 I2C serial EEPROM memory with WPT and address select (400kHz @ 2.7V/10ms)

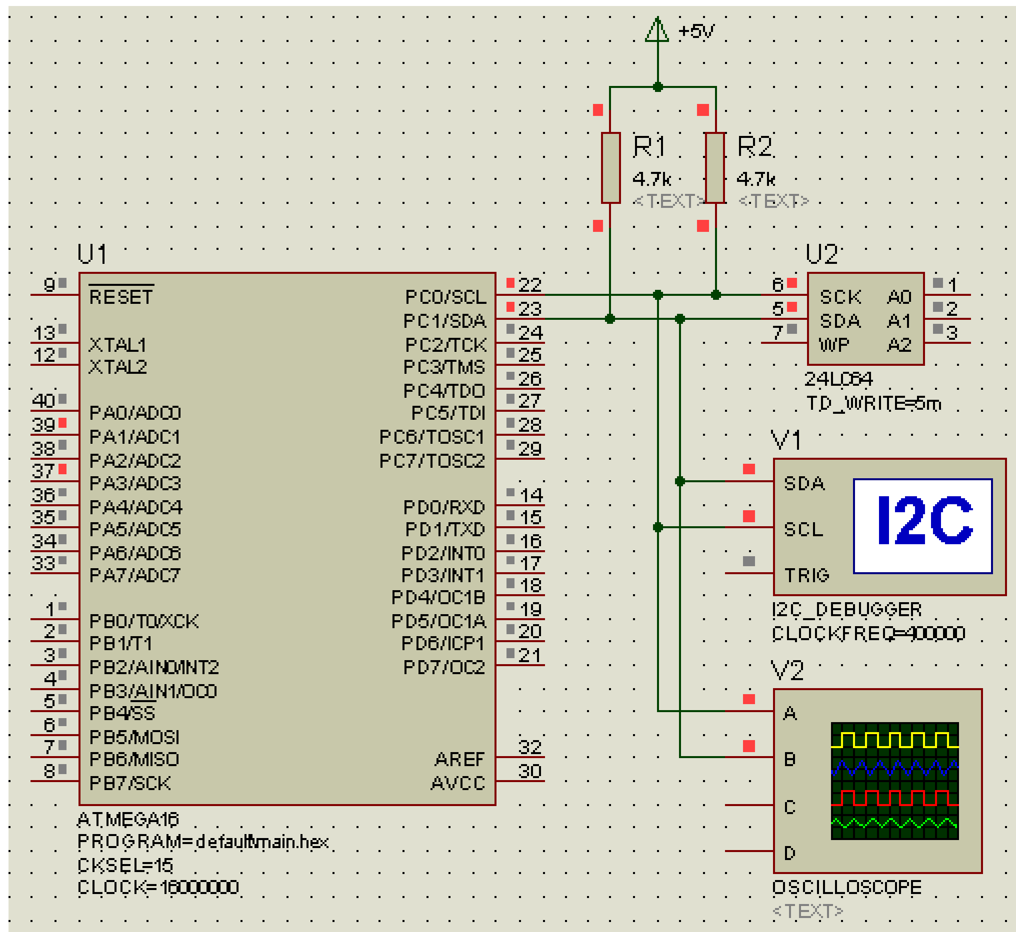


У микросхемы 8 ножек. Все, кроме питания и земли, отображены на скриншоте слева. С SCK и SDA мы знакомы, WP – Write Protect (защита от записи), A0:A2 используются для выбора переменной части адреса ведомой микросхемы (для чего нужны, уже писал где-то выше). Даташит на эту микросхему можете найти по запросу 24XX64 в гугле. Там можете проверить постоянную часть адреса и частоту, а также посмотреть, как использовать ножки WP, A0:A2.

В настройках контроллера укажите прошивку, частоту тактирования 16 МГц и не забудьте выставить CKSEL-фьюзы для кристалла. В настройках I2C-отладчика укажите, что тактовая частота импульсов шины 400 КГц.



Запускаем моделирование..



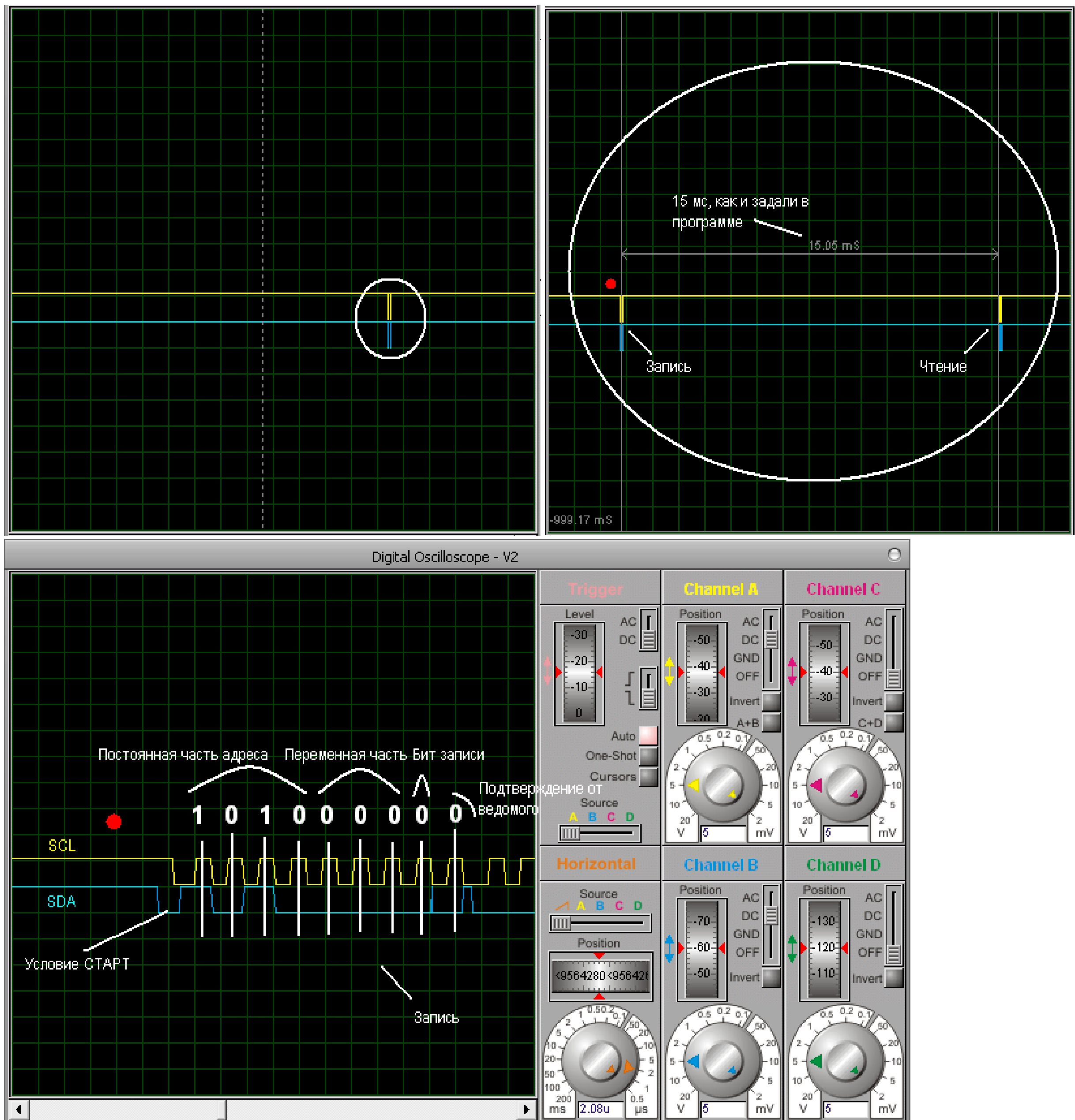
Видно, что в PORTA записалось число 0b0000’1010 или 0x0A или 10, т.е. переменная *uint8_t byte* содержит верное значение после чтения памяти 24LC64 по адресу 0x19. Давайте посмотрим подробнее, что делает наша программа, взглянем на лог I2C-отладчика:

I2C Debug - V1

```
4.010 s   4.010 s  S A0 A 00 A 19 A 0A A P   Записываем байт данных 10 = 0x0A по адресу 25 = 0x19
    4.010 s            S      - Условие начала передачи
    4.010 s   4.010 s  A0 A  - Отсылаем адрес ведомого устройства 1010 + переменный 000 + бит записи (0) = 0xA0
    4.010 s   4.010 s  00 A |
    4.010 s   4.010 s  19 A | - Запись 16-битного адреса 25 = 0x19
    4.010 s   4.010 s  0A A  - Запись байта данных 10 = 0x0A
    4.010 s   4.010 s  P      - Условие завершения передачи
4.025 s   4.025 s  S A0 A 00 A 19 A Sr A1 A 0A N P   Читаем байт данных по адресу 25 = 0x19
    4.025 s            S      - Условие начала передачи
    4.025 s   4.025 s  A0 A  - Отсылаем адрес ведомого устройства 1010 + переменный 000 + бит записи (0) = 0xA0
    4.025 s   4.025 s  00 A |
    4.025 s   4.025 s  19 A | - Запись 16-битного адреса 25 = 0x19
    4.025 s   4.025 s  Sr     - Повтор условия начала передачи
    4.025 s   4.025 s  A1 A  - Отсылаем адрес ведомого устройства 1010 + переменный 000 + бит чтения (1) = 0xA1
    4.025 s   4.025 s  0A N  - Чтение байта данных
    4.025 s   4.025 s  P      - Условие завершения передачи
```

A - подтверждение от ведущего или ведомого, N - нет подтверждения

Теперь можно опуститься еще на уровень ниже и посмотреть на осциллограмму:



Логика, думаю, понятна. Так можно просмотреть и всю осциллограмму, проверить работу I2C в реальной ситуации, а не в режиме симуляции.

Проект и другие файлы к статье можете найти в *"Моих документах\AVR\TWI_EEPROM"* на моем блоге http://nagits.wordpress.com или по адресу http://www.box.net/nagitsdocs.

Литература

[1] Вольфганг Трампперт, AVR-RISC Микроконтроллеры, 2006

[2] AVR315, Использование модуля TWI в качестве ведущего интерфейса I2C http://www.gaw.ru/html.cgi/txt/app/micros/avr/AVR315.htm

[3] ATmega128 Описание регистров TWI, http://www.gaw.ru/html.cgi/txt/doc/micros/avr/arh128/18_4.htm

[4] Microchip 24AA64/24LC64 datasheet

Алексей Наговицын, http:\\nagits.wordpress.com