# Add a DS1307 RTC clock to your AVR microcontroller

Bruce E. Hall, W8BH

Having a real-time clock (RTC) on your microcontroller can be very handy, especially for data logging operations. The Maxim DS1307 is a common and inexpensive real-time clock. It requires only two I/O lines for data communication. If you want to add a clock to your AVR microcontroller, or if you want to learn more about "two-wire" (I^{2}C) interfaces, please read on.

2) THE I^{2}C INTERFACE

Atmel calls their version of I2C the "two-wire" interface, or TWI. It is a serial-data protocol which uses two data lines for communication: a data line (SDA) and a clock (SCL). Devices on the I2C bus can either be masters or slaves. Masters initiate data transfers, and slaves react only to master requests. In this article, the AVRmega328 is the master, and the RTC is always the slave. Slaves are specified by a 7-bit address, plus a read/write bit. The device address for the DS1307 is fixed at 0xd0.
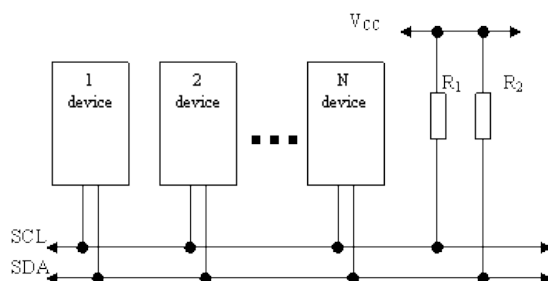
The interface circuit is "open collector", which means that the data lines are passively kept high by resistors to Vcc. Any device on the bus can actively pull a data line low. Up to 128 devices can be put on the same data bus.

There are plenty of good articles on TWI/I2C programming for AVR microcontrollers. Check out the following for a good start:

1. Non-GNU.org: http://www.nongnu.org/avr-libc/user-manual/group__twi__demo.html
2. AVR beginners: http://www.avrbeginners.net/architecture/twi/twi.html
3. ATMEL AVR315: http://www.atmel.com/Images/doc2564.pdf

3) I²C CODING

It is possible to bit-bang the protocol using any two data lines on your microcontroller. However, the ATmega328 has a dedicated TWI interface which simplifies the process. The first job is to set the frequency of the serial data clock. Typically, the clock frequency is 10 (slow mode), 100 (standard mode), or 400 (fast mode) kHz. The maximum clock rate is determined by the slowest device on the bus, as well as bus capacitance. As a practical matter, most I2C devices run at 100 kHz. The DS1307 runs at 100 kHz.

Before going further, keep in mind there are already libraries available for using I2C with your AVR or arduino. You do not need to do this yourself. A search for 'I2C master library' will turn up a few alternatives. Skip this section if you have no interest in learning how to code I2C.

There are two special registers on the ATmega which control the SCL frequency: TWSR and TWBR. TWSR is the TWI status register, and contains prescalar bits used to divide the CPU clock frequency. We do not need a prescalar, so we can ignore these bits. The TWBR is the bit-rate register. The SCL frequency is a function of the CPU frequency and this register, according to the following formula: F_SCL in MHz = $F\_CPU/(16+2(TWBR))$. Kinda complicated, isn't it? To determine the value of TWBR we can rewrite it like this: $TWBR = ((F\_CPU/F\_SCL)-16)/2$. My CPU has a 16 MHz clock, and I want to run the interface in standard 100 kHz mode. So the value of TWBR must be $((16/0.1)-16)/2 = (160-16)/2 = 72$.

```
#define F_CPU        16000000L          // CPU clock speed 16 MHz
#define F_SCL         100000L           // I2C clock speed 100 kHz

void I2C_Init()
// at 16 MHz, the SCL frequency will be 16/(16+2(TWBR)), assuming prescalar of 0.
// so for 100KHz SCL, TWBR = ((F_CPU/F_SCL)-16)/2 = ((16/0.1)-16)/2 = 144/2 = 72.
{
    TWSR = 0;                           // set prescalar to zero
    TWBR = ((F_CPU/F_SCL)-16)/2;        // set SCL frequency in TWI bit register
}
```

Here is the protocol for sending data from master to slave: "MT" (master transmit) mode

- Master generates Start Condition, status code 0x08 is returned
- Master sends slave address (0xd0), DS1307 returns ACK, status code 0x18
- Master sends one or more data bytes, DS1307 returns ACK, status code 0x28
- Master generates Stop Condition, no status code returned

After each operation, the 'ready' bit in TWCR will go to logic 0, and return to logic 1 when the operation is completed. Byte-sized data is sent/received via the special TWDR register. The start, stop, and data transfer conditions are specified by the TWCR control register. And the status codes are put in the TWSR register. Let's look at the code and compare it to the protocol. Here is how to generate a start condition:

```
#define TW_START      0xA4              // send start condition (TWINT,TWSTA,TWEN)
#define TW_READY    (TWCR & 0x80)       // ready when TWINT returns to logic 1.
#define TW_STATUS   (TWSR & 0xF8)       // returns value of status register

byte I2C_Start()
//  generate a TW start condition
{
    TWCR = TW_START;                    // send start condition
    while (!TW_READY);                  // wait
    return (TW_STATUS==0x08);           // return 1 if found; 0 otherwise
}
```

To generate a start, load TWCR with 0xA4 and wait. That's all there is too it. Why 0xA4? If you really must know, 0xA4 is binary 10100100. The three '1' values correspond to the TWINT, TWSTA, and TWEN bits of the control register. These bits enable the TWI interrupt, the start-condition, and the whole TWI module. You will see many people write it like this: TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN). Most think that this 'self-documenting' style of coding is preferable, so please use it if you like. For me, start is simply code 0xA4.

The next thing to do is send the bus address of the slave we are communicating with. For the DS1307, this value will be 0xd0. Here is our code to do that:

```
#define DS1307      0xD0               // I2C bus address of DS1307 RTC
#define TW_SEND     0x84               // send data (TWINT,TWEN)

byte I2C_SendAddr(addr)
//  send bus address of slave
{
    TWDR = addr;                       // load device's bus address
    TWCR = TW_SEND;                    // and send it
    while (!TW_READY);                 // wait
    return (TW_STATUS==0x18);          // return 1 if found; 0 otherwise
}
```

Put the DS1307 address into TWDR, put the send command in TWCR, and wait. The next operation, sending a data byte, looks almost exactly the same. Notice that the returned status code will be different, however:

```
byte I2C_Write (byte data)             // sends a data byte to slave
{
    TWDR = data;                       // load data to be sent
    TWCR = TW_SEND;                    // and send it
    while (!TW_READY);                 // wait
    return (TW_STATUS!=0x28);          // return 1 if found; 0 otherwise
}
```

For the DS1307 we will do this Write operation twice: once to set the address pointer on the RTC, and again to supply the data for that address.

The last step is the send the Stop condition. Here we just set the command register to 0x94, the value for TW_STOP. Again, this value sets the TW enable, TW interrupt, and TW stop bits. Go ahead, use (1<<TWINT) | (1<<TWEN) | (1<<TWSTO) if you prefer. We do not have to wait or check for status codes, so it is just a one-line command. Instead of writing a routine I made a macro instead:

```
#define TW_STOP      0x94              // send stop condition (TWINT,TWSTO,TWEN)
#define I2C_Stop()   TWCR = TW_STOP    // inline macro for stop condition
```

Just a quick note on the status codes: I've written my routines to check the status, but I ignore the results. In my simple setup this works OK. You may want to check each code and show error messages when appropriate.

Reading data is little trickier: we have to write to the device first, to set its internal address pointer, then read to get the data at that address. Here is the protocol for receiving data from the slave.

- Master generates Start Condition, status code 0x08 is returned
- Master sends slave bus address (0xd0), DS1307 returns ACK, status code 0x18
- Master sends address pointer, DS1307 returns ACK, status code 0x28
- Master generates another Start Condition = restart, status code 0x10 returned
- Master sends slave bus address + read bit (0xd1), DS1307 returns ACK, status code 0x40
- Master requests data byte with NACK, DS1307 returns byte, status code 0x58
- Master sends Stop condition, no status code returned

The only new code required for reading is the read operation in the next to last step. It looks very similar to the write operation. NACK is used to a request of a single (or last) byte of data.

```
#define TW_NACK      0x84                // read data with NACK (last byte)
#define READ         1

byte I2C_ReadNACK ()                     // reads a data byte from slave
{
    TWCR = TW_NACK;                      // nack = not reading more data
    while (!TW_READY);                   // wait
    return TWDR;
}
```

Putting it all together, here are the routines for reading and writing registers on the DS1307:

```
void I2C_WriteRegister(byte deviceRegister, byte data)
{
    I2C_Start():
    I2C_SendAddr(DS1307);               // send bus address
    I2C_Write(deviceRegister);          // first byte = device register address
    I2C_Write(data);                    // second byte = data for device register
    I2C_Stop();
}


byte I2C_ReadRegister(byte deviceRegister)
{
    byte data = 0;
    I2C_Start();
    I2C_SendAddr(DS1307);               // send device bus address
    I2C_Write(deviceRegister);          // set register pointer
    I2C_Start();
    I2C_SendAddr(DS1307+READ);          // restart as a read operation
    data = I2C_ReadNACK();              // read the register data
    I2C_Stop();                         // stop
    return data;
}
```

4) DS1307-SPECIFIC CODING

The RTC is pretty straightforward. It contains data registers that specify the seconds, minutes, hours, days, months and years. You write these registers to set the time, and read these registers to get the time. Here are the data register addresses.

```
#define SECONDS_REGISTER   0x00
#define MINUTES_REGISTER   0x01
#define HOURS_REGISTER     0x02
#define DAYOFWK_REGISTER   0x03
#define DAYS_REGISTER      0x04
#define MONTHS_REGISTER    0x05
#define YEARS_REGISTER     0x06
```

There are a few special cases. The seconds register contains a flag to start/stop the clock. And the hours register has flags for 12/24 hour format and AM/PM. Otherwise, getting the time is just a matter of reading the appropriate registers.

```
void DS1307_GetTime(byte *hours, byte *minutes, byte *seconds)
// returns hours, minutes, and seconds in BCD format
{
    *hours = I2C_ReadRegister(DS1307,HOURS_REGISTER);
    *minutes = I2C_ReadRegister(DS1307,MINUTES_REGISTER);
    *seconds = I2C_ReadRegister(DS1307,SECONDS_REGISTER);
    if (*hours & 0x40)                   // 12hr mode:
    *hours &= 0x1F;                      // use bottom 5 bits (pm bit = temp & 0x20)
    else *hours &= 0x3F;                 // 24hr mode: use bottom 6 bits
}

void DS1307_GetDate(byte *months, byte *days, byte *years)
// returns months, days, and years in BCD format
{
    *months = I2C_ReadRegister(DS1307,MONTHS_REGISTER);
    *days = I2C_ReadRegister(DS1307,DAYS_REGISTER);
    *years = I2C_ReadRegister(DS1307,YEARS_REGISTER);
}

void SetTimeDate()
// simple, hard-coded way to set the date 8/13/21013 at 8:51 PM
{
    I2C_WriteRegister(DS1307,MONTHS_REGISTER,  0x08);
    I2C_WriteRegister(DS1307,DAYS_REGISTER,    0x31);
    I2C_WriteRegister(DS1307,YEARS_REGISTER,   0x13);
    I2C_WriteRegister(DS1307,HOURS_REGISTER,   0x08+0x40);  // add 0x40 for PM
    I2C_WriteRegister(DS1307,MINUTES_REGISTER, 0x51);
    I2C_WriteRegister(DS1307,SECONDS_REGISTER, 0x00);
}
```

There are more efficient ways of reading and writing the time. For example, using sequential mode access, we can begin an I2C read operation with the seconds register at 0x00.   The address pointer on the DS1307 auto-increments after each read operation. We can read in all seven time registers before stopping, saving time and code space. I chose to read each register, individually, to be a little more readable and generic.

Data stored in each register is in Binary Coded Decimal (BCD) format. Generally, this means that each byte contains two digits. The most significant digit is stored in the upper four bits, and the least significant digit is stored in the lower four bits. For example, consider the decimal number '36'. Ordinarily, we would code this as 0x24 hexadecimal or 0010.0100 binary. But in BCD, it is stored as 0011.0100. Notice that the upper four bits are 0011 (decimal 3) and the lower four bits are 0100

(decimal 6). Displaying the BCD values is not difficult, since each digit is separately coded. Here is an example for an LCD display, using the LCD_Char() routine:

```
void TwoDigits(byte data)
// helper function for WriteDate() and WriteTime()
// input is two digits in BCD format
// output is to LCD display at current cursor position
{
    byte temp = data>>4;                // get upper 4 bits
    LCD_Char(temp+'0');                 // display upper digit
    data &= 0x0F;                       // get lower 4 bits
    LCD_Char(data+'0');                 // display lower digit
}

void WriteDate()
{
    byte months, days, years;
    DS1307_GetDate(&months,&days,&years);
    TwoDigits(months);
    LCD_Char('/');
    TwoDigits(days);
    LCD_Char('/');
    TwoDigits(years);
}


void WriteTime()
{
    byte hours, minutes, seconds;
    DS1307_GetTime(&hours,&minutes,&seconds);
    TwoDigits(hours);
    LCD_Char(':');
    TwoDigits(minutes);
    LCD_Char(':');
    TwoDigits(seconds);
}
```

5) CONSTRUCTION:

Instead of breadboarding an ATmega328 directly, I use the DC boarduino by Adafruit: it is breadboard friendly, and puts a DC power supply, microprocessor, external oscillator, ISP programming header, status LED, and reset switch all on a very small circuit board.
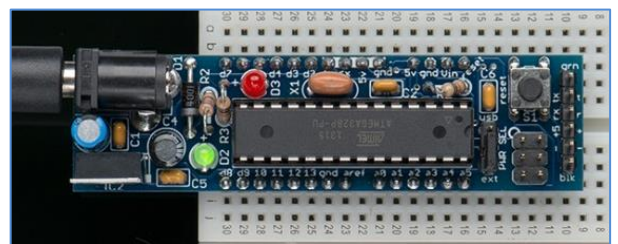
Next, you need a DS1307. Again I used a small circuit module rather than the chip. The module I used is the $15 RTC kit by Smiley Micros. The module adds the required external oscillator and battery backup. Other good ones are available from SparkFun and Adafruit.

First, connect both devices to +5V and ground. Then, connect the I^{2}C data lines as follows:

DC Boarduino/328	DS1307 module
A4 (PC4)	SDA
A5 (PC5)	SCL

Remember than each data line needs a pull-up resistor. Check your RTC module for these resistors. Mine uses a pair of 2.2K (red/red/red) resistors. If your module does not include these resistors, install them on your breadboard between +5V and SDA/SCL. I also tried a pair of 4.7K resistors and they worked fine.
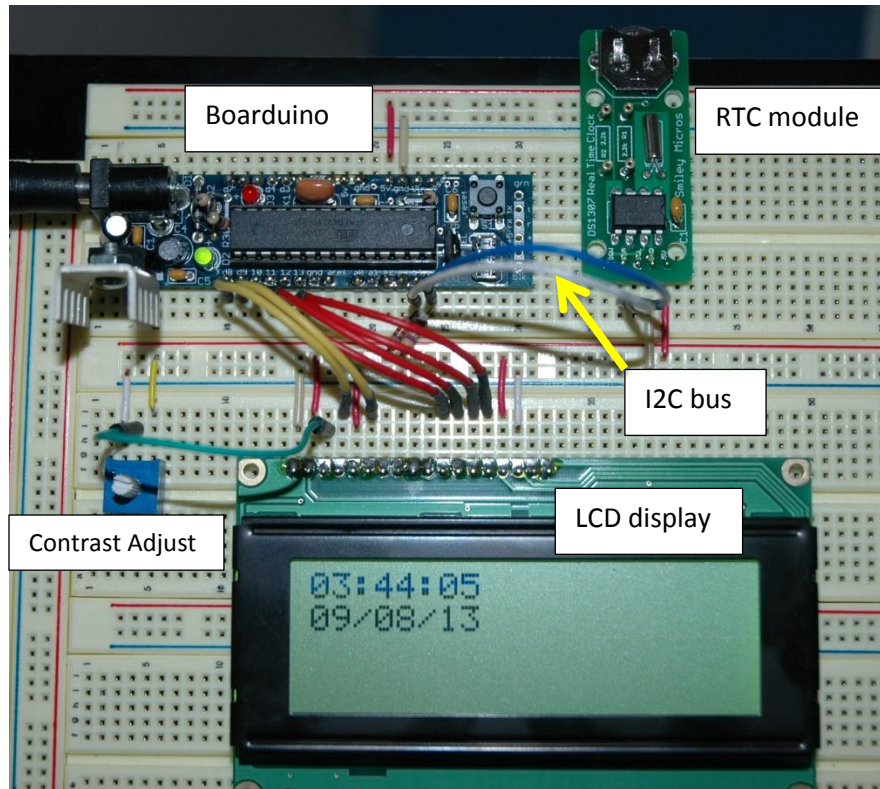
Here is the breadboard layout. The I^2C bus is represented by the white (SDA) and blue (SCL) wires. There are two 4.7K pull-up resistors on the bus, which are partially hidden by the four red data lines. The LCD is a 20x4 character HD44780-compatible display. See my LCD article for more information on this interface. A 10K potentiometer controls the display contrast.

That's it. In my next article I'll show how to interface a more clock-like LED display over I^2C.

6) SOURCE CODE:

```
//------------------------------------------------------------------------------
// i2c01:  Experiments with interfacing ATmega328 to an DS1307 RTC
//
// Author    :  Bruce E. Hall <bhall66@gmail.com>
// Website   :  w8bh.net
// Version   :  1.1
// Date      :  7 Sep 2013
// Target    :  ATTmega328P microcontroller
// Language  :  C, using AVR studio 6
// Size      :  1386 bytes, using -O1 optimization
//
// Fuse settings:  8 MHz osc with 65 ms Delay, SPI enable; *NO* clock/8


//       ------------------------------------------------------------------------
//       GLOBAL DEFINES

#define F_CPU         16000000L  F_CPU   // run CPU at 16 MHz
#define LED           5                  // Boarduino LED on PB5
#define ClearBit(x,y) x &= ~_BV(y)       // equivalent to cbi(x,y)
#define SetBit(x,y)   x |= _BV(y)        // equivalent to sbi(x,y)


//       ------------------------------------------------------------------------
//       INCLUDES

#include <avr/io.h>                  // deal with port registers
#include <util/delay.h>             // used for _delay_ms function
#include <string.h>                 // string manipulation routines
#include <stdlib.h>


//       ------------------------------------------------------------------------
//       TYPEDEFS

typedef uint8_t byte;               // I just like byte & sbyte better
typedef int8_t sbyte;


//       ------------------------------------------------------------------------
//       MISC ROUTINES

void InitAVR()
{
    DDRB = 0x3F;                    // 0011.1111; set B0-B5 as outputs
    DDRC = 0x00;                    // 0000.0000; set PORTC as inputs
}

void msDelay(int delay)            // put into a routine
{                                  // to remove code inlining
    for (int i=0;i<delay;i++)      // at cost of timing accuracy
    _delay_ms(1);
}

void FlashLED()
{
    SetBit(PORTB,LED);
    msDelay(250);
    ClearBit(PORTB,LED);
    msDelay(250);
}



//       ------------------------------------------------------------------------
//       HD44780-LCD DRIVER ROUTINES
//
// Routines:
//       LCD_Init    initializes the LCD controller
//       LCD_Cmd     sends LCD controller command
```

```c
//      LCD_Char     sends single ascii character to display
//      LCD_Clear    clears the LCD display & homes cursor
//      LCD_Home     homes the LCD cursor
//      LCD_Goto     puts cursor at position (x,y)
//      LCD_Line     puts cursor at start of line (x)
//      LCD_Hex      displays a hexadecimal value
//      LCD_Integer  displays an integer value
//      LCD_String   displays a string
//
//  The LCD module requires 6 I/O pins:  2 control lines & 4 data lines.
//  PortB is used for data communications with the HD44780-controlled LCD.
//  The following defines specify which port pins connect to the controller:

#define LCD_RS     0                    // pin for LCD R/S (eg PB0)
#define LCD_E      1                    // pin for LCD enable
#define DAT4       2                    // pin for d4
#define DAT5       3                    // pin for d5
#define DAT6       4                    // pin for d6
#define DAT7       5                    // pin for d7

//  The following defines are HD44780 controller commands
#define CLEARDISPLAY 0x01
#define SETCURSOR    0x80

void PulseEnableLine ()
{
    SetBit(PORTB,LCD_E);                // take LCD enable line high
    _delay_us(40);                      // wait 40 microseconds
    ClearBit(PORTB,LCD_E);              // take LCD enable line low
}

void SendNibble(byte data)
{
    PORTB &= 0xC3;                      // 1100.0011 = clear 4 data lines
    if (data & _BV(4))  SetBit(PORTB,DAT4);
    if (data & _BV(5))  SetBit(PORTB,DAT5);
    if (data & _BV(6))  SetBit(PORTB,DAT6);
    if (data & _BV(7))  SetBit(PORTB,DAT7);
    PulseEnableLine();                  // clock 4 bits into controller
}

void SendByte (byte data)
{
    SendNibble(data);                   // send upper 4 bits
    SendNibble(data<<4);                // send lower 4 bits
    ClearBit(PORTB,5);                  // turn off boarduino LED
}

void LCD_Cmd (byte cmd)
{
    ClearBit(PORTB,LCD_RS);             // R/S line 0 = command data
    SendByte(cmd);                      // send it
}

void LCD_Char (byte ch)
{
    SetBit(PORTB,LCD_RS);               // R/S line 1 = character data
    SendByte(ch);                       // send it
}

void LCD_Init()
{
    LCD_Cmd(0x33);                      // initialize controller
    LCD_Cmd(0x32);                      // set to 4-bit input mode
    LCD_Cmd(0x28);                      // 2 line, 5x7 matrix
    LCD_Cmd(0x0C);                      // turn cursor off (0x0E to enable)
    LCD_Cmd(0x06);                      // cursor direction = right
    LCD_Cmd(0x01);                      // start with clear display
    msDelay(3);                         // wait for LCD to initialize
```

```c
}

void LCD_Clear()                        // clear the LCD display
{
    LCD_Cmd(CLEARDISPLAY);
    msDelay(3);                         // wait for LCD to process command
}

void LCD_Home()                         // home LCD cursor (without clearing)
{
    LCD_Cmd(SETCURSOR);
}

void LCD_Goto(byte x, byte y)           // put LCD cursor on specified line
{
    byte addr = 0;                      // line 0 begins at addr 0x00
    switch (y)
    {
        case 1: addr = 0x40;  break;    // line 1 begins at addr 0x40
        case 2: addr = 0x14;  break;
        case 3: addr = 0x54;  break;
    }
    LCD_Cmd(SETCURSOR+addr+x);          // update cursor with x,y position
}

void LCD_Line(byte row)                 // put cursor on specified line
{
    LCD_Goto(0,row);
}

void LCD_String(const char *text)       // display string on LCD
{
    while (*text)                       // do until /0 character
        LCD_Char(*text++);              // send char & update char pointer
}

void LCD_Hex(int data)
// displays the hex value of DATA at current LCD cursor position
{
    char st[8] = "";                    // save enough space for result
    itoa(data,st,16);                   // convert to ascii hex
    //LCD_Message("0x");                // add prefix "0x" if desired
    LCD_String(st);                     // display it on LCD
}

void LCD_Integer(int data)
// displays the integer value of DATA at current LCD cursor position
{
    char st[8] = "";                    // save enough space for result
    itoa(data,st,10);                   // convert to ascii
    LCD_String(st);                     // display in on LCD
}


//      --------------------------------------------------------------------------
//      I2C (TWI) ROUTINES
//
//  On the AVRmega series, PA4 is the data line (SDA) and PA5 is the clock (SCL
//  The standard clock rate is 100 KHz, and set by I2C_Init.  It depends on the AVR osc. freq.

#define F_SCL           100000L         // I2C clock speed 100 KHz
#define READ            1
#define TW_START        0xA4            // send start condition (TWINT,TWSTA,TWEN)
#define TW_STOP         0x94            // send stop condition (TWINT,TWSTO,TWEN)
#define TW_ACK          0xC4            // return ACK to slave
#define TW_NACK         0x84            // don't return ACK to slave
#define TW_SEND         0x84            // send data (TWINT,TWEN)
#define TW_READY        (TWCR & 0x80)   // ready when TWINT returns to logic 1.
#define TW_STATUS       (TWSR & 0xF8)   // returns value of status register
```

```c
#define I2C_Stop()     TWCR = TW_STOP     // inline macro for stop condition

void I2C_Init()
// at 16 MHz, the SCL frequency will be 16/(16+2(TWBR)), assuming prescalar of 0.
// so for 100KHz SCL, TWBR = ((F_CPU/F_SCL)-16)/2 = ((16/0.1)-16)/2 = 144/2 = 72.
{
    TWSR = 0;                              // set prescalar to zero
    TWBR = ((F_CPU/F_SCL)-16)/2;           // set SCL frequency in TWI bit register
}

byte I2C_Detect(byte addr)
//  look for device at specified address; return 1=found, 0=not found
{
    TWCR = TW_START;                       // send start condition
    while (!TW_READY);                     // wait
    TWDR = addr;                           // load device's bus address
    TWCR = TW_SEND;                        // and send it
    while (!TW_READY);                     // wait
    return (TW_STATUS==0x18);              // return 1 if found; 0 otherwise
}

byte I2C_FindDevice(byte start)
// returns with address of first device found; 0=not found
{
    for (byte addr=start;addr<0xFF;addr++)  // search all 256 addresses
    {
        if (I2C_Detect(addr))              // I2C detected?
            return addr;                   // leave as soon as one is found
    }
    return 0;                              // none detected, so return 0.
}

void I2C_Start (byte slaveAddr)
{
    I2C_Detect(slaveAddr);
}

byte I2C_Write (byte data)                 // sends a data byte to slave
{
    TWDR = data;                           // load data to be sent
    TWCR = TW_SEND;                        // and send it
    while (!TW_READY);                     // wait
    return (TW_STATUS!=0x28);
}

byte I2C_ReadACK ()                        // reads a data byte from slave
{
    TWCR = TW_ACK;                         // ack = will read more data
    while (!TW_READY);                     // wait
    return TWDR;
    //return (TW_STATUS!=0x28);
}

byte I2C_ReadNACK ()                       // reads a data byte from slave
{
    TWCR = TW_NACK;                        // nack = not reading more data
    while (!TW_READY);                     // wait
    return TWDR;
    //return (TW_STATUS!=0x28);
}

void I2C_WriteByte(byte busAddr, byte data)
{
    I2C_Start(busAddr);                    // send bus address
    I2C_Write(data);                       // then send the data byte
    I2C_Stop();
}

void I2C_WriteRegister(byte busAddr, byte deviceRegister, byte data)
```

```
{
    I2C_Start(busAddr);                 // send bus address
    I2C_Write(deviceRegister);          // first byte = device register address
    I2C_Write(data);                    // second byte = data for device register
    I2C_Stop();
}

byte I2C_ReadRegister(byte busAddr, byte deviceRegister)
{
    byte data = 0;
    I2C_Start(busAddr);                 // send device address
    I2C_Write(deviceRegister);          // set register pointer
    I2C_Start(busAddr+READ);            // restart as a read operation
    data = I2C_ReadNACK();              // read the register data
    I2C_Stop();                         // stop
    return data;
}


//      ---------------------------------------------------------------------------
//      DS1307 RTC ROUTINES

#define DS1307             0xD0         // I2C bus address of DS1307 RTC
#define SECONDS_REGISTER   0x00
#define MINUTES_REGISTER   0x01
#define HOURS_REGISTER     0x02
#define DAYOFWK_REGISTER   0x03
#define DAYS_REGISTER      0x04
#define MONTHS_REGISTER    0x05
#define YEARS_REGISTER     0x06
#define CONTROL_REGISTER   0x07
#define RAM_BEGIN          0x08
#define RAM_END            0x3F

void DS1307_GetTime(byte *hours, byte *minutes, byte *seconds)
// returns hours, minutes, and seconds in BCD format
{
    *hours = I2C_ReadRegister(DS1307,HOURS_REGISTER);
    *minutes = I2C_ReadRegister(DS1307,MINUTES_REGISTER);
    *seconds = I2C_ReadRegister(DS1307,SECONDS_REGISTER);
    if (*hours & 0x40)                  // 12hr mode:
    *hours &= 0x1F;                     // use bottom 5 bits (pm bit = temp & 0x20)
    else *hours &= 0x3F;                // 24hr mode: use bottom 6 bits
}

void DS1307_GetDate(byte *months, byte *days, byte *years)
// returns months, days, and years in BCD format
{
    *months = I2C_ReadRegister(DS1307,MONTHS_REGISTER);
    *days = I2C_ReadRegister(DS1307,DAYS_REGISTER);
    *years = I2C_ReadRegister(DS1307,YEARS_REGISTER);
}

void SetTimeDate()
// simple, hard-coded way to set the date.
{
    I2C_WriteRegister(DS1307,MONTHS_REGISTER,  0x08);
    I2C_WriteRegister(DS1307,DAYS_REGISTER,    0x31);
    I2C_WriteRegister(DS1307,YEARS_REGISTER,   0x13);
    I2C_WriteRegister(DS1307,HOURS_REGISTER,   0x08+0x40);  // add 0x40 for PM
    I2C_WriteRegister(DS1307,MINUTES_REGISTER, 0x51);
    I2C_WriteRegister(DS1307,SECONDS_REGISTER, 0x00);
}



//      ---------------------------------------------------------------------------
//      APPLICATION ROUTINES
```

```
void ShowDevices()
// Scan I2C addresses and display addresses of all devices found
{
    LCD_Line(1); LCD_String("Found:");
    byte addr = 1;
    while (addr>0)
    {
        LCD_Char(' ');
        addr = I2C_FindDevice(addr);
        if (addr>0) LCD_Hex(addr++);
    }
}

void LCD_TwoDigits(byte data)
// helper function for WriteDate()
// input is two digits in BCD format
// output is to LCD display at current cursor position
{
    byte temp = data>>4;
    LCD_Char(temp+'0');
    data &= 0x0F;
    LCD_Char(data+'0');
}

void WriteDate()
{
    byte months, days, years;
    DS1307_GetDate(&months,&days,&years);
    LCD_TwoDigits(months);
    LCD_Char('/');
    LCD_TwoDigits(days);
    LCD_Char('/');
    LCD_TwoDigits(years);
}


void WriteTime()
{
    byte hours, minutes, seconds;
    DS1307_GetTime(&hours,&minutes,&seconds);
    LCD_TwoDigits(hours);
    LCD_Char(':');
    LCD_TwoDigits(minutes);
    LCD_Char(':');
    LCD_TwoDigits(seconds);
}

void LCD_TimeDate()
{
    LCD_Line(0);  WriteTime();
    LCD_Line(1);  WriteDate();
}


//      -------------------------------------------------------------------------
//      PROGRAM LOOP

void MainLoop()
{
    while(1)
    {
        LCD_TimeDate();                 // put time & date on LCD
        msDelay(1000);                  // one second between updates
    }
}


//      -------------------------------------------------------------------------
```

```c
//      MAIN PROGRAM

int main(void)
{
    InitAVR();                      // set port direction
    LCD_Init();                     // initialize HD44780 LCD controller
    I2C_Init();                     // set I2C clock frequency
    LCD_String("Ready.");
    ShowDevices();                  // show that I2C is working OK
    msDelay(4000);
    LCD_Clear();
    MainLoop();                     // display time
}
```