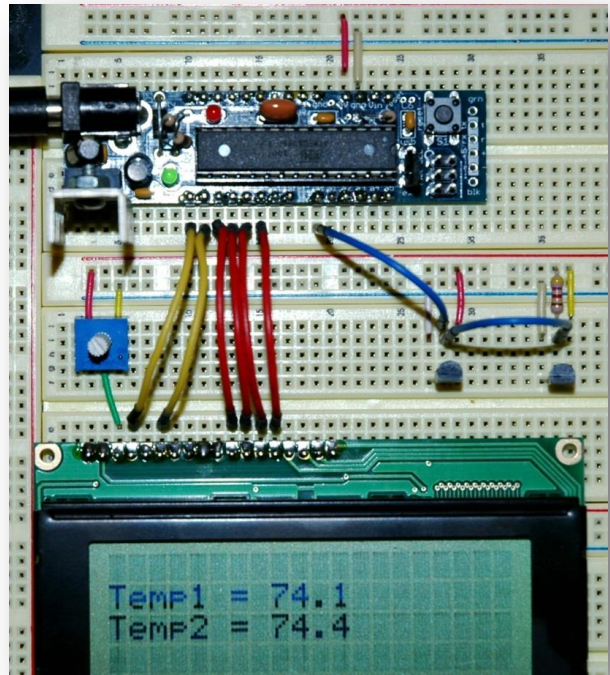


Build a Dual-Temperature Maxim DS18B20 Thermometer

Bruce E. Hall, W8BH



1) INTRODUCTION

The Maxim DS18B20 is a low-cost temperature measurement sensor. This device digitally reports temperature in degrees C with 9 to 12-bit precision, -55C to 125C (+/-0.5C). Each sensor has a unique 64-Bit Serial number etched into it, allowing for a large number of sensors to be used on one data bus. This sensitive and versatile part is the cornerstone of many data-logging and temperature control projects.

There are lots of tutorials available on the web. Read this one if:

- You are interested in learning about the Dallas “one-wire” protocol.
- You program in C language on Atmel AVR microprocessors.
- You like compact, memory-efficient code.
- Your spouse said, “Make something useful this time!”

In this tutorial I will show you a very basic set of DS18B20 routines, and use those routines to implement a dual-temperature digital thermometer.

2) THE DALLAS ONE-WIRE INTERFACE

[1-Wire](#) is a device communications bus system designed by Dallas Semiconductor Corp. that provides low-speed data, signaling, and power over a single wire. 1-Wire is similar in concept to [I²C](#), but with lower data rates over longer wires. It is typically used to communicate with small inexpensive devices such as digital thermometers. A typical circuit consists of a master device which initiates data transfer, such as a microcontroller, and one or more slave devices like the DS18B20.

Physically, like I²C, the 1-wire bus is implemented as open-collector circuit which is passively held at logic-high levels by an external resistor. Communication is enabled by actively pulling the data line low according to a predefined protocol.

For my data line I will be using the Atmel microcontroller Port C, bit 0 (PC0). Rather than hard-coding this pin at multiple code locations, it is better to define it once:

```
#define THERM_PORT    PORTC
#define THERM_DDR     DDRC
#define THERM_PIN     PINC
#define THERM_IO      0
```

We define the port (PortC), the data direction register (DDRC), the port input register (PINC), and the bit number (0). If you change the pin in the future, you'll only need to change it here.

Next are a few useful macros to set, clear, and read individual IO port bits:

```
#define ClearBit(x,y) x &= ~_BV(y)    // equivalent to cbi(x,y)
#define SetBit(x,y)  x |= _BV(y)     // equivalent to sbi(x,y)
#define ReadBit(x,y) x & _BV(y)      // call with PINx and bit#
```

Now that we can control any single port bit, it is easy to restrict control to the data line using our first set of definitions:

```
#define THERM_INPUT()  ClearBit(THERM_DDR,THERM_IO)    // make pin an input
#define THERM_OUTPUT() SetBit(THERM_DDR,THERM_IO)     // make pin an output
#define THERM_LOW()    ClearBit(THERM_PORT,THERM_IO)  // take (output) pin low
#define THERM_HIGH()   SetBit(THERM_PORT,THERM_IO)    // take (output) pin high
#define THERM_READ()   ReadBit(THERM_PIN,THERM_IO)    // get (input) pin value
```

To begin the data transfer, the microcontroller pulls the data line low for 480 microseconds. Then, after waiting 60 additional μ S, the DS18B20 will pull the line low to indicate its presence. In fact, this sequence will “reset” every slave device on the bus, and each will signal its presence by pulling the data line low. Pseudocode for the reset operation:

- Set data line as output & pull data line low
- Wait 480 μ S – the entire “send” timeslot
- Release data line
- Wait 60 μ S
- Read data line, looking for slave response
- Wait for remainder of the “receive” slot

The only missing pieces are the microsecond timing delays. Fortunately, there is a function for microsecond delay in <util/delay.h> named “_delay_us”. To use it you must specify the microcontroller clock speed in Hz. My clock speed is 16 MHz.

```
#define F_CPU          16000000L    // run CPU at 16 MHz
#include <util/delay.h>              // used for _delay_us function
```

Here is the code for the reset operation:

```

byte therm_Reset()
{
    byte i;
    THERM_OUTPUT();           // set pin as output
    THERM_LOW();              // pull pin low for 480uS
    _delay_us(480);
    THERM_INPUT();           // set pin as input
    _delay_us(60);           // wait for 60uS
    i = THERM_READ();         // get pin value
    _delay_us(420);          // wait for rest of 480uS period
    return i;
}

```

Writing a bit to a slave device is also done by a strict timing protocol. Each timing slot is 60 μ S wide, and begins with master pulling down the data line for 1 μ S . To write a one, the master releases the line, allowing it to float back to a logic high state. To write a zero, the master holds the line low for the entire slot:

```

void therm_WriteBit(byte bit)
{
    THERM_OUTPUT();           // set pin as output
    THERM_LOW();              // pull pin low for 1uS
    _delay_us(1);
    if (bit) THERM_INPUT();   // to write 1, float pin
    _delay_us(60);
    THERM_INPUT();           // wait 60uS & release pin
}

```

Reading bits also occurs in 60 μ S time slots. Again, the master holds down the data line for 1 μ S. Then the master waits 14 additional μ S before reading the line:

```

byte therm_ReadBit()
{
    byte bit=0;
    THERM_OUTPUT();           // set pin as output
    THERM_LOW();              // pull pin low for 1uS
    _delay_us(1);
    THERM_INPUT();           // release pin & wait 14 uS
    _delay_us(14);
    if (THERM_READ()) bit=1;  // read pin value
    _delay_us(45);           // wait rest of 60uS period
    return bit;
}

```

Reading and writing byte values is just 8 consecutive bit operations. Our 1-wire interface is complete. But how can we the temperature?

3) DS18B20 COMMANDS

The DS18B20 has responds to 10 different commands. See the [datasheet](#) for complete details. We will focus on those needed for this application:

ReadROM	Reads the 8 byte ID of a slave device.
MatchROM	Address a device by its 8-byte ROM ID. All other devices will be disabled.
SkipROM	Address all slave devices on the bus, without specifying a particular device.

ConvertT	Initiate temperature conversion.
ReadScratchpad	Read the 8 byte scratchpad. The first two bytes contain the temperature data.

The usual sequence for a DS18B20 command is:

- 1-wire reset command, which resets all slave devices.
- ROM command (MatchROM is specify a single device, or SkipROM for all devices)
- Non-ROM command (ConvertT or ReadScratchpad).

For example, suppose we have only one slave device on the bus, and we want to measure the temperature. Two command sets like above are required: the first to start the temperature conversion, and the second to read the data. The command sequence would be as follows:

- 1-wire reset
 - SkipROM
 - ConvertT
-
- 1-wire reset
 - SkipROM
 - ReadScratchpad

Compare this command sequence with the code example below. When doing a temperature conversion, the slave device holds the data line low until the conversion is complete.

```
void therm_ReadTempExample()
// Returns the two temperature bytes from the scratchpad
{
    therm_Reset();                // skip ROM & start temp conversion
    therm_WriteByte(THERM_SKIPROM);
    therm_WriteByte(THERM_CONVERTTEMP);
    while (!therm_ReadBit());      // wait until conversion completed

    therm_Reset();                // read first two bytes from scratchpad
    therm_WriteByte(THERM_SKIPROM);
    therm_WriteByte(THERM_READSCRATCH);
    byte t0 = therm_ReadByte();    // first byte
    byte t1 = therm_ReadByte();    // second byte
}
```

The two bytes in this code example, t0 and t1, hold the 12-bit Celsius temperature data:

T0:	2 ³	2 ²	2 ¹	2 ⁰	2 ⁻¹	2 ⁻²	2 ⁻³	2 ⁻⁴
T1:	Sign	Sign	Sign	Sign	Sign	2 ⁶	2 ⁵	2 ⁴

Complicated, isn't it? First, combine the whole number components from both bytes into a single byte. We need the upper 4 bits of T0, which need to be shifted 4 places to the left (T0 >>4). Ignoring the sign bits for now, we need the lower 3 bits of T1 shifted 4 places to the right (T1 & 0x0F) << 4. Putting it all together, whole = (t0>>4) + ((T1 & 0x07)<<4).

Now we need to consider the fractional binary component of T0. The value of bit 0 is 2^{-4} , which is 1/16 or 0.0625 = 625/10000. Therefore, if we take the value of these 4 bits and multiply by 625, we get the number of 10,000^{ths}. Interesting! For example, consider the fractional binary value 0.0101. Binary 0101 equals 5, so the decimal equivalent is $5 * 0.0625 = 0.3125$.

The following lines convert the raw t0 and t1 temperature data into its base-10 equivalent:

```
whole = (t1 & 0x07) << 4;           // grab lower 3 bits of t1
whole |= t0 >> 4;                   // and upper 4 bits of t0
decimal = t0 & 0x0F;                // decimals in lower 4 bits of t0
decimal *= 625;                      // conversion factor for 12-bit resolution
```

Every DS18B20 tutorial I read gives temperature readings in Celsius, or uses floating point arithmetic for Fahrenheit conversion. Using floating point division requires a lot of code space, so I devised with another way. Consider the Celcius to Farhenheit conversion formula:

$$F = 1.8C + 32$$

Multiplying both sides by 10 allows us to stick with integer math:

$$10F = 18C + 320.$$

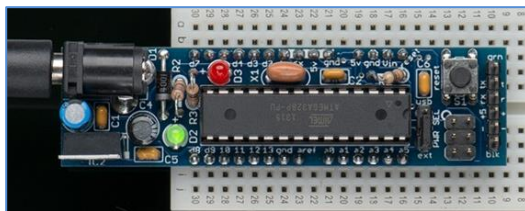
Now, do the same thing with our raw Celcius data, which is in the form $C = \text{bbbbbb.bbbb}$. Multiplying both sides by 16 will shift the decimal 4 places to the right, thereby removing it:

$$16C = (\text{bbbbbb.bbbb}) * 16 = \text{bbbbbbbbb}$$

In the Fahrenheit formula we need 18C, so we are 2C short. An integer divide-by-8 on 16C will give us 2C, with loss of the lowest two bits. However, this loss is negligible if we display our temperature with a precision of 0.1 F. The following code will accomplish our Fahrenheit conversion using integer math only:

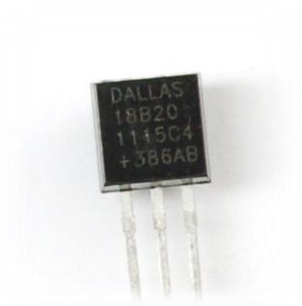
```
int c16 = (t1<<8) + t0;           // result is temp*16, in celcius
int c2 = c16/8;                   // get t*2, with fractional part lost
int f10 = c16 + c2 + 320;          // F=1.8C+32, so 10F = 18C+320 = 16C + 2C + 320
byte whole = f10/10;              // get whole part
byte decimal = f10 % 10;          // get fractional part
```

4) 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.

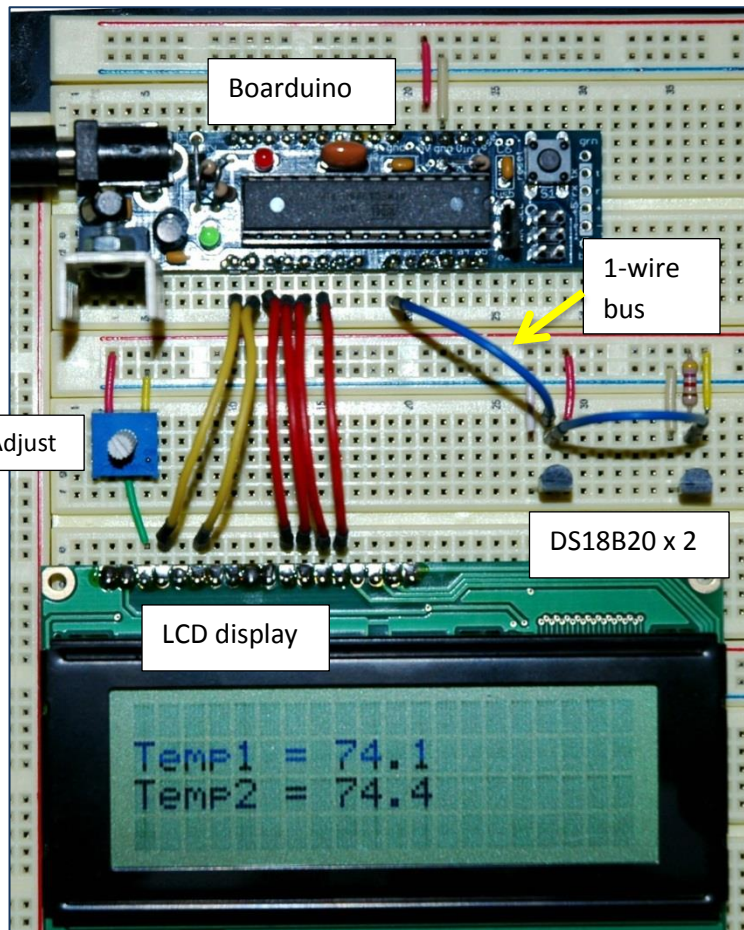
2) Next, you need a DS18B20 chip, which is readily available available from [SparkFun](#) and [Adafruit](#). The Adafruit offering comes with the 4.7K resistor, which is a nice touch.



3) You will need an LCD character display to see the results. Any standard, HD44780-compatible parallel-input display will work, such as this [20x4 character model](#) from Adafruit. See my article on [LCD interfacing](#) for more details.

4) Wire the 1-wire bus. Remember than the data line also needs a pull-up resistor. Install a 4.7K resistor between data line and 5V.

DC Boarduino/328	DS18B20
A0 (PC0)	Pin 2 (middle pin)
Gnd	Pin 1 (left pin)
5V	Pin 3 (right pin)



Here is the breadboard layout. The 1-wire bus is represented by the blue wires. Note DC power connections to the Boarduino and DS18B20 devices.

5) SOURCE CODE:

```
//-----  
// Experiments with the Maxim DS18B20 temperature sensor  
//  
// Author   : Bruce E. Hall <bhall166@gmail.com>  
// Website  : w8bh.net  
// Version  : 1.0  
// Date     : 22 Sep 2013  
// Target   : ATmega328P microcontroller  
// Language : C, using AVR studio 6  
// Size     : 1340 bytes, using -O1 optimization  
//  
// Fuse settings: 8 MHz osc with 65 ms Delay, SPI enable; *NO* clock/8  
  
//-----  
// GLOBAL DEFINES  
  
#define F_CPU      16000000L           // 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)  
#define ReadBit(x,y) x & BV(y)        // call with PINx and bit#  
  
//-----  
// INCLUDES  
  
#include <avr/io.h>                   // deal with port registers  
#include <util/delay.h>               // used for _delay_us function  
#include <stdlib.h>  
#include <stdio.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  
{  
    for (int i=0;i<delay;i++)         // to remove code inlining  
        _delay_ms(1);                // at cost of timing accuracy  
}  
  
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
//      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
}

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_String("0x");       // add prefix "0x" if desired
    LCD_String(st);           // display it on LCD
}

void HexDigit(byte data)
// lower 4 bits of input -> output hex digit
// helper function for other LCD hex routines
{
    if (data<10) data+='0';
    else data+='A'-10;
    LCD_Char(data);
}

void LCD_HexByte(byte data)
// displays a two-character hexadecimal value of data @ current LCD cursor
{
    HexDigit(data>>4);
    HexDigit(data & 0x0F);
}

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
}

```

```

}

void LCD_PadInteger(int data, byte size, char padChar)
// displays right-justified integer on LCD, padding with specified character
// using this instead of sprintf will save you about 1400 bytes of codespace
{
    char st[8] = ""; // save enough space for result
    itoa(data,st,10); // convert to ascii
    byte len = strlen(st); // string length of converted integer
    sbyte blanks = size-len; // required spaces to pad on left
    if (blanks>=0) // do we need to pad at all?
        while(blanks--) // add padding characters...
            LCD_Char(padChar);
    LCD_String(st); // then the converted integer
}

// -----
// "ONE-WIRE" ROUTINES
//

#define THERM_PORT PORTC
#define THERM_DDR DDRC
#define THERM_PIN PINC
#define THERM_IO 0

#define THERM_INPUT() ClearBit(THERM_DDR,THERM_IO) // make pin an input
#define THERM_OUTPUT() SetBit(THERM_DDR,THERM_IO) // make pin an output
#define THERM_LOW() ClearBit(THERM_PORT,THERM_IO) // take (output) pin low
#define THERM_HIGH() SetBit(THERM_PORT,THERM_IO) // take (output) pin high
#define THERM_READ() ReadBit(THERM_PIN,THERM_IO) // get (input) pin value

#define THERM_CONVERTTEMP 0x44
#define THERM_READSCRATCH 0xBE
#define THERM_WRITESCRATCH 0x4E
#define THERM_COPYSCRATCH 0x48
#define THERM_READPOWER 0xB4
#define THERM_SEARCHROM 0xF0
#define THERM_READROM 0x33
#define THERM_MATCHROM 0x55
#define THERM_SKIPROM 0xCC
#define THERM_ALARMSEARCH 0xEC

// the following arrays specify the addresses of *my* ds18b20 devices
// substitute the address of your devices before using.
byte rom0[] = {0x28,0xE1,0x21,0xA3,0x02,0x00,0x00,0x5B};
byte rom1[] = {0x28,0x1B,0x21,0x30,0x05,0x00,0x00,0xF5};

byte therm_Reset()
{
    byte i;
    THERM_OUTPUT(); // set pin as output
    THERM_LOW(); // pull pin low for 480uS
    _delay_us(480);
    THERM_INPUT(); // set pin as input
    _delay_us(60); // wait for 60uS
    i = THERM_READ(); // get pin value
    _delay_us(420); // wait for rest of 480uS period
    return i;
}

void therm_WriteBit(byte bit)
{
    THERM_OUTPUT(); // set pin as output
    THERM_LOW(); // pull pin low for 1uS
    _delay_us(1);
    if (bit) THERM_INPUT(); // to write 1, float pin
    _delay_us(60);
}

```

```

    THERM_INPUT();                // wait 60uS & release pin
}

byte therm_ReadBit()
{
    byte bit=0;
    THERM_OUTPUT();                // set pin as output
    THERM_LOW();                  // pull pin low for 1uS
    _delay_us(1);
    THERM_INPUT();                // release pin & wait 14 uS
    _delay_us(14);
    if (THERM_READ()) bit=1;      // read pin value
    _delay_us(45);                // wait rest of 60uS period
    return bit;
}

void therm_WriteByte(byte data)
{
    byte i=8;
    while(i-->0)                  // for 8 bits:
    {
        therm_WriteBit(data&1);  // send least significant bit
        data >>= 1;              // shift all bits right
    }
}

byte therm_ReadByte()
{
    byte i=8, data=0;
    while(i-->0)                  // for 8 bits:
    {
        data >>= 1;              // shift all bits right
        data |= (therm_ReadBit()<<7); // get next bit (LSB first)
    }
    return data;
}

void therm_MatchRom(byte rom[])
{
    therm_WriteByte(THERM_MATCHROM);
    for (byte i=0;i<8;i++)
        therm_WriteByte(rom[i]);
}

void therm_ReadTempRaw(byte id[], byte *t0, byte *t1)
// Returns the two temperature bytes from the scratchpad
{
    therm_Reset();                // skip ROM & start temp conversion
    if (id) therm_MatchRom(id);
    else therm_WriteByte(THERM_SKIPROM);
    therm_WriteByte(THERM_CONVERTTEMP);
    while (!therm_ReadBit());    // wait until conversion completed

    therm_Reset();                // read first two bytes from scratchpad
    if (id) therm_MatchRom(id);
    else therm_WriteByte(THERM_SKIPROM);
    therm_WriteByte(THERM_READSCRATCH);
    *t0 = therm_ReadByte();       // first byte
    *t1 = therm_ReadByte();       // second byte
}

void therm_ReadTempC(byte id[], int *whole, int *decimal)
// returns temperature in Celsius as WW.DDDD, where W=whole & D=decimal
{
    byte t0,t1;
    therm_ReadTempRaw(id,&t0,&t1); // get temperature values
    *whole = (t1 & 0x07) << 4;   // grab lower 3 bits of t1
    *whole |= t0 >> 4;            // and upper 4 bits of t0
    *decimal = t0 & 0x0F;         // decimals in lower 4 bits of t0
}

```

```

    *decimal *= 625;                // conversion factor for 12-bit resolution
}

void therm_ReadTempF(byte id[], int *whole, int *decimal)
// returns temperature in Fahrenheit as WW.D, where W=whole & D=decimal
{
    byte t0,t1;
    therm_ReadTempRaw(id,&t0,&t1);    // get temperature values
    int t16 = (t1<<8) + t0;          // result is temp*16, in celcius
    int t2 = t16/8;                  // get t*2, with fractional part lost
    int f10 = t16 + t2 + 320;         // F=1.8C+32, so 10F = 18C+320 = 16C + 2C + 320
    *whole = f10/10;                  // get whole part
    *decimal = f10 % 10;              // get fractional part
}

inline __attribute__((gnu_inline)) void quickDelay(int delay)
// this routine will pause 0.25uS per delay unit
// for testing only; use _us_Delay() routine for >1uS delays
{
    while(delay--)                  // uses sbiw to subtract 1 from 16bit word
        asm volatile("nop");       // nop, sbiw, brne = 4 cycles = 0.25 uS
}

// -----
// ROM READER PROGRAM

void RomReaderProgram()
// Read the ID of the attached Dallas 18B20 device
// Note: only ONE device should be on the bus.
{
    LCD_String("ID (ROM) Reader:");
    while(1)
    {
        LCD_Line(1);
        // write 64-bit ROM code on first LCD line
        therm_Reset();
        therm_WriteByte(THERM_READROM);
        for (byte i=0; i<8; i++)
        {
            byte data = therm_ReadByte();
            LCD_HexByte(data);
        }
        msDelay(1000);              // do a read every second
    }
}

// -----
// DUAL TEMPERATURE PROGRAM

void DualTempProgram()
{
    while(1)
    {
        int whole=0, decimal=0;

        LCD_Line(1);
        LCD_String("Temp1 = ");
        //therm_ReadTempC(&whole,&decimal);
        //LCD_Integer(whole); LCD_Char('.');
        //LCD_PadInteger(decimal,4,'0');
        therm_ReadTempF(rom1,&whole,&decimal);
        LCD_Integer(whole); LCD_Char('.');
        LCD_Integer(decimal);

        LCD_Line(2);
        LCD_String("Temp2 = ");
        //therm_ReadTempC(&whole,&decimal);
        //LCD_Integer(whole); LCD_Char('.');
        //LCD_PadInteger(decimal,4,'0');
    }
}

```

```
        therm_ReadTempF(rom0,&whole,&decimal);
        LCD_Integer(whole); LCD_Char('.');
        LCD_Integer(decimal);
        msDelay(1000);
        FlashLED();
    }
}
```

```
// -----
//      MAIN PROGRAM
```

```
int main(void)
{
    InitAVR();
    LCD_Init();
    //RomReaderProgram();
    DualTempProgram();
}
```