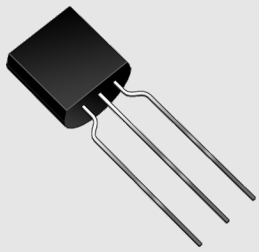


Using DS18B20 digital temperature sensor on AVR microcontrollers

Description and application

Version 1.0 (Preliminary)

Research Project Documents
Gerard Marull Paretas
September 2007

Index

1. Introduction

A few months ago I received two samples of DS18B20 from Maxim. However, I asked for another chip, not DS18B20, so I decided to notify to Maxim. They agreed about the mistake and a few days later I received the correct ones, but those two chips remained on my desktop until one day I decided to examine them. I had thought they would not be very interesting, as they only had 3 pads. Maybe a voltage regulator, a kind of transistor... But I got a surprise when I read that it was a digital temperature sensor! At first, I could not understand how it could be a digital temperature sensor: it only had 3 pads and 2 of them had to be Ground and Vcc. How could one transfer digital data through only one wire? I had been always using two wires or more, one of them working as a clock source. However, it is possible to transfer data through only one wire making it controlled by the microcontroller at very precise intervals. One common example is the serial line of your computer. The baud rate of both sides (the computer and a device) must be known in order to send and receive data without errors. I used serial line before getting DS18B20, but when you set it up on an AVR microcontroller, you just give some parameters and the device does the rest. But in this case the work will be done by us. To continue, you just need the materials listed below:

- 1- AVR Microcontroller (ex. Atmega8)
- 1- DS18B20 Digital temperature sensor
- 1- 4K7 Resistor
- A working AVR Toolchain

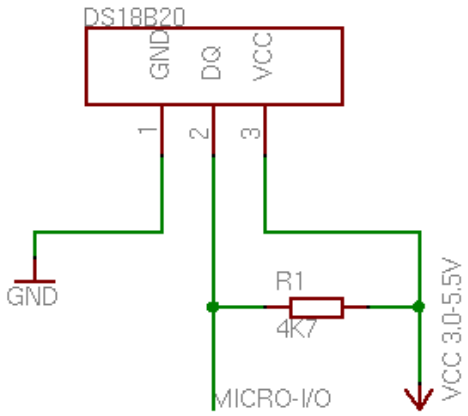
If you find any mistake in this document, please, report it at gerardmarull@gmail.com.

Notes: In the preliminary version of this document, a copy of the GNU Free Documentation License is not included, but you can access it by visiting http://www.gnu.org/licenses/fdl.txt.

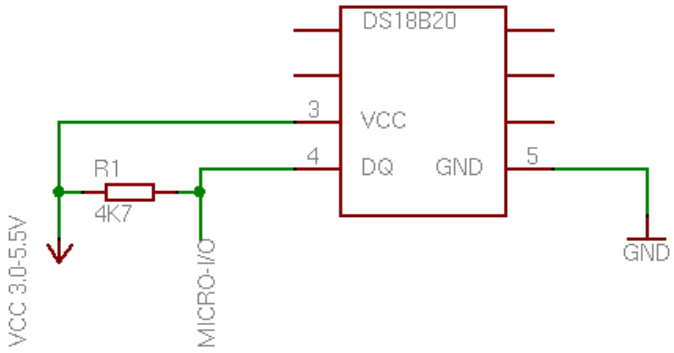
2. Making the circuit

The sensor can come in three different packages: TO-92, SO-8 and µSOP-8. You can choose the most convenient as all operate in the same way. The schematic is shown below for each package type. For *DQ* connection you can choose any of your microcontroller's I/O pin.

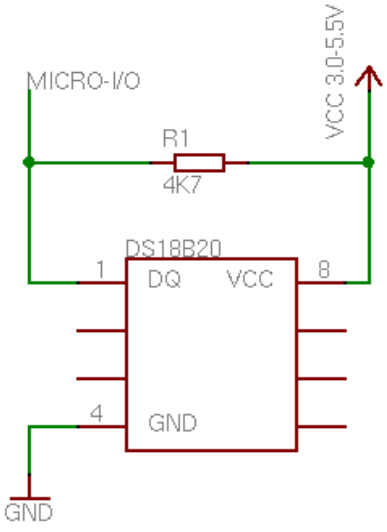
TO-92 Package Schematic



SO-8 Package Schematic



uSOP-8 Package Schematic



3. Working with DS18B20

In this chapter I will explain how the sensor works and even the necessary C code to get it running under an AVR microcontroller. Of course, external topics related to microcontrollers such as the compilation of the code will not be treated. Another requirement is to have a way to report the temperature that you obtain from the sensor: a serial connection with the computer or a LCD would be fine. Some of the features that the sensor offers are not treated in this version of the document (ex. Having multiple sensors in the same bus, CRC computation or setting Alarm triggers). They might be included in future versions of this document.

3.1. A critical operation: timing

Since we only have one wire to communicate with the sensor, it is necessary to read/write data at very precise intervals. All these intervals are defined in DS18B20's datasheet. If you take a brief look, you will see that most of them are just a matter of a few micro-seconds (I will use the µs abbreviation from now on). This might seem too small for humans, but not for microcontrollers. A common AVR microcontroller can run up to 16Mhz (some at 20Mhz) so each clock cycle is very short. In this document I will use the internal 8Mhz clock of an Atmega1281 (also included in many other models) which means that each clock cycle will take 0.125µs (1/8000000Hz). AVR Instructions (also called AVR assembler instructions) can take from 1 to 4 clock cycles to execute, so it is easy to make exact µs delays.
You will probably program in C language, where you do not know which assembler code the compiler will produce, but you can get it easily.
Let us start with a simple delay function that will be able to execute from 0 to 65535 loops:

```
inline __attribute__((gnu_inline)) void therm_delay(uint16_t delay){
        while(delay--) asm volatile("nop");
}
```

There is a thing that you might not know: the *inline* and *gnu_inline* attributes. This means that the code inside *therm_delay* function will be inserted where *therm_delay* is called instead of making the call, which would require *calling* and *returning* from the function and even *pushing* and *popping* some registers to the *stack* (a waste of time not desirable in this case).
Now, make another function (I will name it *therm_reset*) and call *therm_delay* inside it. This way we will be able to see which assembler does this code produce. To do that, you have to invoke the compiler with *-S* flag. In my case, I have used the following command (*avr-gcc* version is 4.2.1):

```
# avr-gcc -S -mcall-prologues -std=gnu99 -funsigned-char
-funsigned-bitfields -fpack-struct -fshort-enums -mmcu=atmega1281
-Wall -Wstrict-prototypes -Os yourfile.c -o yourfile.s
```

After that, you should be able to see the file *yourfile.s* with some assembler code inside. If you look for *therm_delay* function code, you should find something like this:

```
.global   therm_delay
          .type     therm_delay, @function
therm_delay:
/* prologue: frame size=0 */
```

```
/* prologue end (size=0) */
        rjmp .L2
.L3:
/* #APP */
        nop
/* #NOAPP */
.L2:
        sbiw r24,1
        ldi r18,hi8(-1)
        cpi r24,lo8(-1)
        cpc r25,r18
        brne .L3
/* epilogue: frame size=0 */
        ret
/* epilogue end (size=1) */
/* function therm_delay size 9 (8) */
        .size    therm_delay, .-therm_delay
```

This is only the function code. If you go to *therm_reset*, you should see something similar:

```
.global  therm_reset
        .type    therm_reset, @function
therm_reset:
/* prologue: frame size=0 */
/* prologue end (size=0) */
        ldi r24,lo8(480)
        ldi r25,hi8(480)
        rjmp .L14
.L15:
/* #APP */
        nop
/* #NOAPP */
.L14:
        sbiw r24,1
        ldi r18,hi8(-1)
        cpi r24,lo8(-1)
        cpc r25,r18
        brne .L15
```

There are just two further instructions: *ldi*. *ldi* (*LoaD Immediate*) is used to load a constant value into a register. In this case I am loading the number 480, and as the function uses an *uint16_t* (2-bytes) to store the number of delays, it needs to use two registers (*r24* and *r25*). I will just explain the meaning of two more instructions that will be useful to understand this part. The first is *nop*, which is an instruction that does nothing, and the second is *brne* (*BRanch if Not Equal*) which is a conditional instruction that will jump to *.L8* (in this case) if *r25* and *r18* are not equal (compared in the instruction before with *cpc*). You can refer *AVR Instruction Set* document (available at www.atmel.com/avr) to know the meaning of the other instructions, as teaching AVR assembler is not the purpose of this document.
Now, is the moment to start counting the time that this delay loop would take if it was executed. First, we must separate the instructions *ldi*, *ldi* and *rjmp* that will not enter the continuous loop. They are only used to load a value and jump to the start of the loop. Looking at *AVR Instruction Set* document we can see and calculate the values shown in the following table:

Instruction	Clock Cycles
ldi	1

ldi	1
rjmp	2
TOTAL CYCLES	4

Table 3.1.A

For the second part (those instructions involved in the loop) you have to look at *L14* (where the loop starts) and *L15* (the code executed on each loop):

Instruction	Clock Cycles
nop	1
sbiw	2
ldi	1
cpi	1
cpc	1
brne	2 if true, 1 if false
TOTAL CYCLES	*Read below*

Table 3.1.B

We can calculate the total delay of a loop cycle easily: adding all these instruction clock cycles but taking care of just one thing: *brne* is normally true (it uses 2 cycles and jumps to *L15*) except in the last loop, when *brne* condition is false, so it takes only 1 cycle and *L15* is not executed as the code continues with the next instruction. This means we have to subtract 2 cycles from the total (1 from *nop* and 1 from *brne*).
Therefore, the total delay of this loop is: *(8 cycles * number of loops) – 2 cycles*

Finally we have to join everything:
4 cycles (initialization) + ((8 cycles * number of loops) – 2 cycles) =
(8 cycles * number of loops) + 2 cycles
Considering that at 8Mhz each cycle takes 0.125us, the final delay inµs will be:
(8 * 0.125µs * number of loops) + 2 * 0.125µs = (1µs*number of loops) + 0.25µs =
(number of loops)µs + 0.25µs

Since the number of loops is multiplied by 1, putting the number of µs we want, we will get the same delay in µs directly (plus 0.25µs, which is depreciable in this case), so there is no need for any conversion before. However, if your clock speed is not as lucky as mine, you can create a macro that calculates how many loops would be necessary by just giving the µs you need:

```
#define F_CPU    3000000UL       //Your clock speed in Hz (3Mhz here)
#define LOOP_CYCLES      8       //Number of cycles that the loop takes
#define us(num) (num/(LOOP_CYCLES*(1/(F_CPU/1000000.0))))
```

Using that macro, your code would look like follows:

```
therm_delay(us(480)); //We want to make a 480µs delay
```

The value will be automatically transformed by the compiler preprocessor into the corresponding loops needed by your clock. In this code, where we simulate having a 3Mhz clock, it would calculate 180 loops.

3.2. Useful definitions

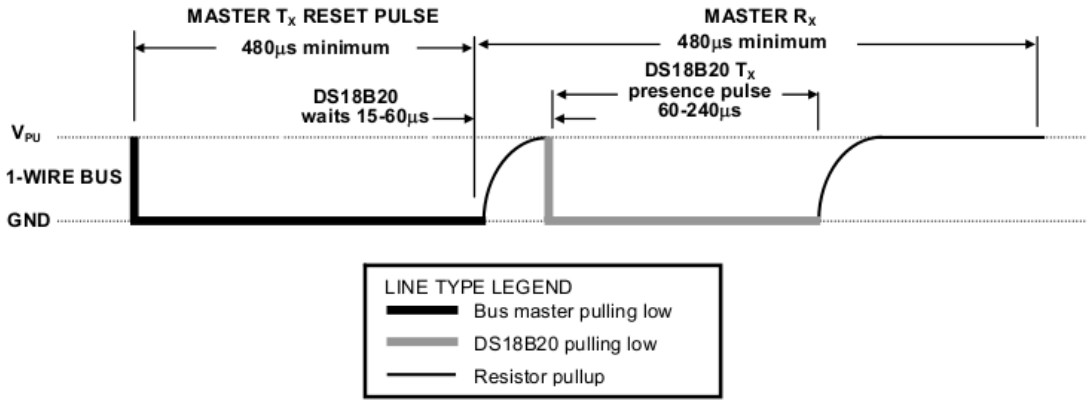
Since you will need to change the level or the direction of the pin where your thermometer is connected, I recommend you to put the following definitions in your code to make it more readable and easy to write:

```
/* Thermometer Connections (At your choice) */
#define THERM_PORT           PORTC
#define THERM_DDR            DDRC
#define THERM_PIN            PINC
#define THERM_DQ             PC0
/* Utils */
#define THERM_INPUT_MODE()              THERM_DDR&=~(1<<THERM_DQ)
#define THERM_OUTPUT_MODE()             THERM_DDR|=(1<<THERM_DQ)
#define THERM_LOW()                     THERM_PORT&=~(1<<THERM_DQ)
#define THERM_HIGH()                    THERM_PORT|=(1<<THERM_DQ)
```

Note that the definitions described in the above chapter (3.1) should also be included in order to get the code (described in the comingchapters) working.

3.3. Thermometer initialization

The first thing you will need to do before any transaction with the thermometer is to send an initialization sequence. This sequence consists of a reset pulse transmitted by the master (microcontroller) followed by a presence pulse sent by the thermometer. The following graphic shows a timing diagram of the sequence:



As you can see, in the reset pulse the master pulls the line low for a minimum of 480 µs and finally it releases the line. After that there is a period of 15-60µs to let the thermometer receive the rising edge (caused by the 4K7 resistor connected to VCC+) and emit the presence pulse, which is 60-240µs long. Just note that the second part (including the 15-60µs delay) must be 480µs long, too.
A clear implementation in C is shown below:

```
uint8_t therm_reset(){

        uint8_t i;
```

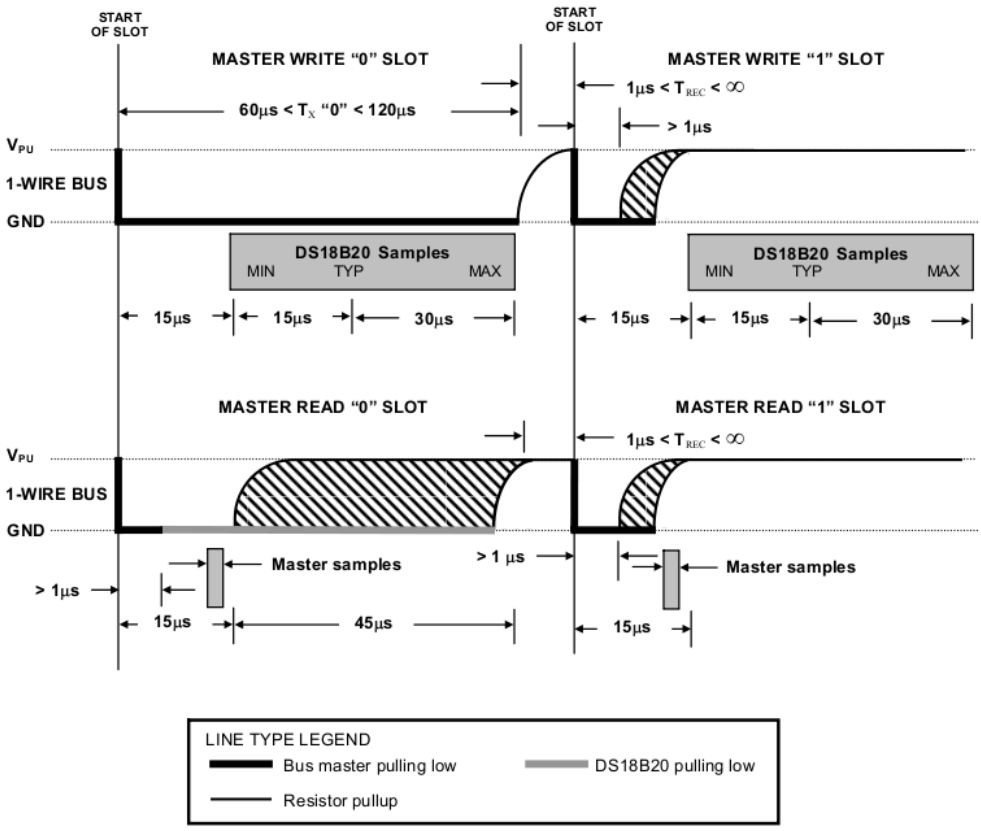
```
        //Pull line low and wait for 480uS
        THERM_LOW();
        THERM_OUTPUT_MODE();
        therm_delay(us(480));

        //Release line and wait for 60uS
        THERM_INPUT_MODE();
        therm_delay(us(60));

        //Store line value and wait until the completion of 480uS period
        i=(THERM_PIN & (1<<THERM_DQ));
        therm_delay(us(420));

        //Return the value read from the presence pulse (0=OK, 1=WRONG)
        return i;

}
```

3.4. Read/Write operations

First of all, I will show how to make functions to read/write individual bits (where you have to take care of timing), and then other functions to read/write an entire byte (which are basically loops executing the individual bits functions).

3.4.1. Reading/Writing individual bits

Let us take a look at the timing diagrams provided in the datasheet



In the write operation, we always have to pull the line low at startup. If we want to write 0 we

will keep the line low until the end of the sequence (60µs) but if we want to write 1, after a delay of 1µs we will release it. An implementation in C is shown below:

```
void therm_write_bit(uint8_t bit){

          //Pull line low for 1uS
          THERM_LOW();
          THERM_OUTPUT_MODE();
          therm_delay(us(1));

          //If we want to write 1, release the line (if not will keep low)
          if(bit) THERM_INPUT_MODE();

          //Wait for 60uS and release the line
          therm_delay(us(60));
          THERM_INPUT_MODE();

}
```

On the other side, we have the read operation. It is quite similar, but has some differences. We also have to start pulling the line low for 1µs. Then we have to release the line and wait for 14µs more (14+1=15µs as shown in the diagram). After that, we can read the value of the line, that will be high if thermometer transmits 1 and low if it transmits 0. Finally, we just have to wait 45µs more to end the 60µs period. An implementation in C is shown below:

```
uint8_t therm_read_bit(void){

          uint8_t bit=0;

          //Pull line low for 1uS
          THERM_LOW();
          THERM_OUTPUT_MODE();
          therm_delay(us(1));

          //Release line and wait for 14uS
          THERM_INPUT_MODE();
          therm_delay(us(14));

          //Read line value
          if(THERM_PIN&(1<<THERM_DQ)) bit=1;

          //Wait for 45uS to end and return read value
          therm_delay(us(45));
          return bit;

}
```

3.4.2. Reading/Writing bytes

Now that we can read/write individual bits, doing this for bytes is quite easy: make loops of 8 cycles and store the result in a variable. The implementation in C is shown below:

```
uint8_t therm_read_byte(void){

          uint8_t i=8, n=0;

          while(i--){
```

```
                    //Shift one position right and store read value
                    n>>=1;
                    n|=(therm_read_bit()<<7);
          }

          return n;

}

void therm_write_byte(uint8_t byte){

          uint8_t i=8;

          while(i--){
                    //Write actual bit and shift one position right to make
the next bit ready
                    therm_write_bit(byte&1);
                    byte>>=1;
          }

}
```

3.5. Available commands

Now that we are able to read/write data from DS18B20, we have to know which commands are available to do the operations we need. I will define them in a table with a short description of each one and its corresponding code. The definitions for your C code will also be given.

Command	Code	Description
ROM COMMANDS		
Search ROM	0xf0	Used to identify the ROM codes of the available slaves in the bus, which also lets the master determine the total number of slaves.
Read ROM	0x33	This command has the same effect as Search ROM, but it can be used when there is only one device in the bus. If not, data collision will occur.
Match ROM	0x55	This command followed by a 64-bit ROM code is used to address a specific slave in the bus. If a slave matches the code, it will be the only one to respond to the commands. The others will wait for a reset sequence.
Skip ROM	0xcc	This is used to address all the devices in the bus at the same time. This could be useful to send commands such as the start of temperature conversion (Convert T, 0x44) or if there is only one device in the bus.
Alarm Search	0xec	This command is nearly the same as Search ROM, excepting that only devices with a set alarm flag will respond.
FUNCTION COMMANDS		

Convert T	0x44	This command is used to start a temperature conversion, that is stored in the first two bytes of the Scratchpad. The conversion time is resolution dependent, -look at table shown in chapter 3.6 . If you send read slots while conversion is in progress the slave responds with 0 (low) if it is still in progress or 1 (high) if it has finished (only available if it is not parasite powered).
Write Scratchpad	0x4e	This command allows master to write 3 bytes of data to the Scratchpad: TH, TL and Configuration registers. Data must be transferred with least significant bit first.
Read Scratchpad	0xbe	This command allows the master to read the contents of the Scratchpad. It is 9-bytes long and it is transferred starting with the least significant byte. If all bytes are not needed transfer can be stopped by master issuing a reset.
Copy Scratchpad	0x48	This command copies the contents of TH, TL and configuration registers from the Scratchpad to the EEPROM.
Recall E^2	0xb8	This command recalls the alarm trigger values (TH and TL) and configuration data from EEPROM and places them in the Scratchpad.
Read Power Supply	0xb4	This command is used to determine if a slave is externally powered or uses parasite power. Parasite powered slaves will keep the line low after the command, and high the externally powered ones.

Table 3.5.A

A list of these commands translated into C defines:

```
#define THERM_CMD_CONVERTTEMP          0x44
#define THERM_CMD_RSCRATCHPAD          0xbe
#define THERM_CMD_WSCRATCHPAD          0x4e
#define THERM_CMD_CPYSCRATCHPAD        0x48
#define THERM_CMD_RECEEPROM            0xb8
#define THERM_CMD_RPWRSUPPLY           0xb4
#define THERM_CMD_SEARCHROM            0xf0
#define THERM_CMD_READROM              0x33
#define THERM_CMD_MATCHROM             0x55
#define THERM_CMD_SKIPROM              0xcc
#define THERM_CMD_ALARMSEARCH          0xec
```

3.6. The last step: reading the temperature

Now that we are able to send to and read from the thermometer, it is time to learn how to read the temperature, which is in fact the reason why you are reading this document. The temperature is stored in the first two bytes of the Scratchpad, which is 9-bytes long and contains more information than the temperature. Its structure is shown below:

Byte	Content	Startup value

0	Temperature LSB	0x5005 (+85ºC)
1	Temperature MSB	
2	TH Register or User Byte 1	Content stored in EEPROM
3	TL Register or User Byte 2	
4	Configuration register	
5	Reserved (0xff)	-
6	Reserved (0x0c)	-
7	Reserved (0x10)	-
8	CRC	-

Table 3.6.A

A useful thing inside the Scratchpad is the Configuration register. As you should know, the thermometer can work in four different resolutions (9,10,11 and 12 bits). It comes with a 12-bit mode selected by default, but it can be changed at any moment using the *Write Scratchpad* command (0x4e) and writing there the desired resolution. In the following tables the structure of Configuration register and information of each resolution is shown (including necessary values to change it):

CONFIGURATION REGISTER							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	R1	R0	1	1	1	1	1

Table 3.6.B

R1	R0	Resolution bits	Decimal steps	Conversion time
0	0	9	0.5	93.75ms
0	1	10	0.25	187.5ms
1	0	11	0.125	375ms
1	1	12 (default)	0.0625	750ms

Table 3.6.C

As I said, the temperature is stored in the first 2 bytes of the Scratchpad, but we need to know how to interpret them. We can obtain three important values from the temperature bytes: if it is positive or negative, the temperature integer digit and the number of decimal steps. The structure is detailed below:

Least Significant Byte							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I^3	I^2	I^1	I^0	D^{-1}	D^{-2}	D^{-3}	D^{-4}
Most Significant Byte							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
S	S	S	S	S	I^6	I^5	I^4

Table 3.6.D

I = Integer digits of Celsius degrees (signed) D = Number of decimal steps* S = Sign (0 = +, 1 = -)	+85.0000 ºC
* *For 9-bit resolution only bit 3 is valid, bits 3 and 2 for 10-bit, bits 3, 2 and 1 for 11-bit and all bits for 12-bit. It even needs to be multiplied by decimal steps constant (shown in 4th column of table 3.6.C).*	

Table 3.6.E

Now is the time to make the C implementation. It has been programmed to work with only one thermometer in the same bus and in 12-bit resolution. It should be easy to change to another resolution (only take care of decimal steps value). At the end, the temperature is joined into a char array forming a string like "+85.0000 C" which you can use to send via serial to your computer, draw into a LCD, or whatever you want. Note that making use of *sprintf()* function, saves us from having to take care of the temperature integer digit sugn and S bits of 2 temperature bytes. However, if you do not use *sprintf()* function, you can convert a signed number to unsigned using *two's complement: (~(variable)+1)*.

```
#define THERM_DECIMAL_STEPS_12BIT 625 //.0625

void therm_read_temperature(char *buffer){

        // Buffer length must be at least 12bytes long! ["+XXX.XXXX C"]
        uint8_t temperature[2];
        int8_t digit;
        uint16_t decimal;

        //Reset, skip ROM and start temperature conversion
        therm_reset();
        therm_write_byte(THERM_CMD_SKIPROM);
        therm_write_byte(THERM_CMD_CONVERTTEMP);
        //Wait until conversion is complete
        while(!therm_read_bit());
        //Reset, skip ROM and send command to read Scratchpad
        therm_reset();
        therm_write_byte(THERM_CMD_SKIPROM);
        therm_write_byte(THERM_CMD_RSCRATCHPAD);

        //Read Scratchpad (only 2 first bytes)
        temperature[0]=therm_read_byte();
        temperature[1]=therm_read_byte();
        therm_reset();

        //Store temperature integer digits and decimal digits
        digit=temperature[0]>>4;
        digit|=(temperature[1]&0x7)<<4;
        //Store decimal digits
        decimal=temperature[0]&0xf;
        decimal*=THERM_DECIMAL_STEPS_12BIT;

        //Format temperature into a string [+XXX.XXXX C]
        sprintf(buffer, "%+d.%04u C", digit, decimal);

}
```

4. Bibliography

MAXIM, “DS18B20 Datasheet”, http://www.maxim-ic.com/

MAXIM, “APPLICATION NOTE 162: Interfacing the DS18X20/DS1822 1-Wire Temperature Sensor in a Microcontroller Environment ”, http://www.maxim-ic.com/

ATMEL, “AVR Instruction Set ”, http://www.atmel.com/avr

http://www.atmel.com/avr

http://www.maxim-ic.com