Temperature Sensor

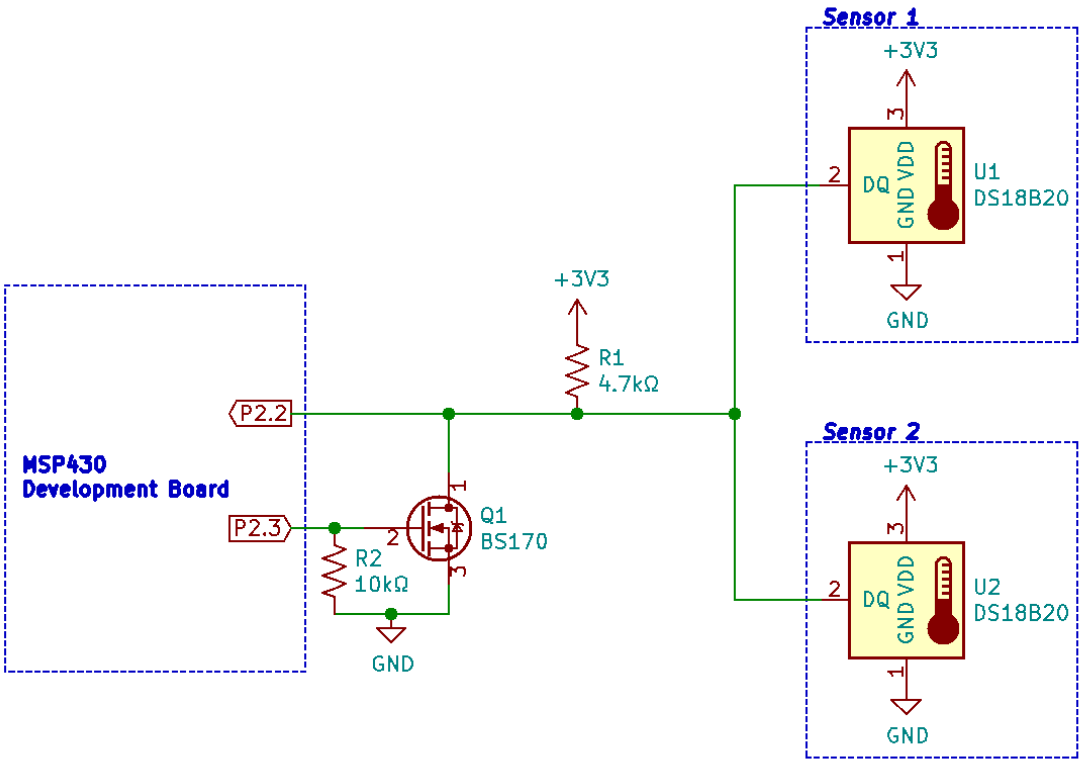

Figure 1 – Schematic diagram of the test circuit for the temperature sensor

The DS18B20 is a temperature sensor that uses Maxim Integrated one-wire bus. The sensor can be configured to be 9-bit to 12-bit resolution. The temperature data is given in degree Celsius as a 2 bytes number, where bits 15 to 12 are the sign of the number, bits 10 to 4 represent the integer value, and bits 3 to 0 are the fractional values as shown below:

BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
S	S	S	S	S	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

The resolution works as follow:

- 12-bit resolution: All bits are assigned
- 11-bit resolution: Bits 15 to 1 are assigned, and bit 0 always read a logic ‘0’
- 10-bit resolution: Bits 15 to 2 are assigned, and bits 1 and 0 always read a logic ‘0’
- 9-bit resolution: Bits 15 to 3 are assigned, and bits 2, 1 and 0 always read a logic ‘0’

In order to configure the sensor and read the temperature, we need to pass in a command to do so. The sensor can read the command data serially according to 1-wire bus protocol. All commands are one byte long. In order to communicate with the sensor, it is necessary to send a match address command proceeded by its unique address (identified as Lasered ROM code in the datasheet).

The bus protocol needs to follow a specific algorithm and must meet all the time constrains. To initiate communication with the sensor, first we need to send a reset pulse and wait for a feedback. The reset pulse is achieved by pulling the bus low for $480\mu s$ and releasing it. If the sensor acknowledges the reset pulse, it will pull the bus low for a period between $60\mu s$ to $120\mu s$. After the reset procedure is done, we can start transmitting commands to the sensor, and every time a new command must be preceded by a reset pulse.

To read temperature, the master must send a command to the slave to convert the current temperature and then command the slave to transmit the contents of its memory.

Each bit is transmitted between the master and the slave according to the following time diagram:

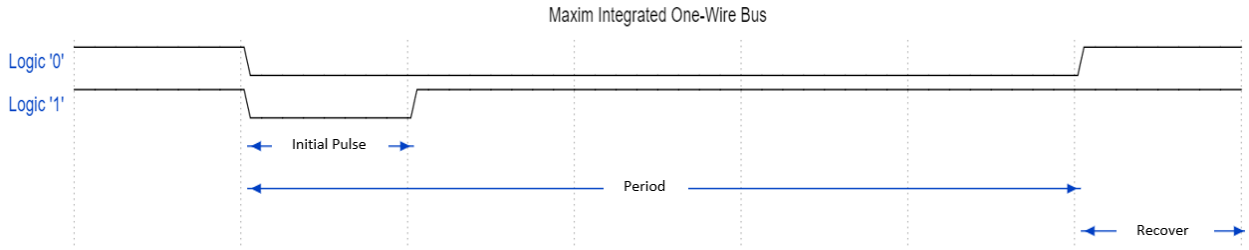

Figure 2 – One-wire bus time diagram

When transmitting a bit, the master sends an initial pulse by pulling the bus low for at least $1\mu s$. If the master is sending a logic '1', it must release the bus within $15\mu s$ after the falling edge of the initial pulse. If the master is transmitting a logic '0', it must release hold the bus low for a minimum period of $60\mu s$.

When receiving a bit, the master also sends an initial pulse by pulling the bus low. However, it must release the bus before $15\mu s$ after the falling edge of the initial pulse because the data from the slave is only guaranteed by the manufacturers to be stable within the first $15\mu s$ after the falling edge of the initial pulse. After the initial pulse, the slave will hold the bus low if it desires to transmit a logic '0', and it will allow the bus to return to its idle state if it desired to transmit a logic '1'.

Every transmit and receive has a minimum period of $60\mu s$, and each transaction must allow a minimum of $1\mu s$ of recover time.

Since this protocol has very specific time constraints, the functions that control the bus are written in assembly language, so the timing can be fully optimized. This protocol, although efficient in terms of hardware, is very inefficient timewise.

To control the bus, two pins of the MSP430 microcontroller are being used even though this is a 1-wire bus. As seen in the schematic diagram, the gate of a N-MOSFET transistor is attached to port P2.3, so the microcontroller can pull the bus low. Using an external transistor to control the bus allowed the software to be less complex and to run faster. The bus is pulled high in its idle state by a $4.7k\Omega$ resistor as recommended by Maxim Integrated. This way, pin P2.2 acts as an input port only.

Making changes

```
// define input and output constants for the one-wire bus
#define TS_BUS          P2IN                    // P2.2 is the input port for the 1-wire bus
#define TS_OUT          P2OUT                   // P2.3 is driving an external pull-down transistor that will pull the bus low when needed

#define TS_INIT_INPORT  P2DIR
#define TS_INIT_OUTPORT P2DIR

#define TS_OUTBIT       BIT2                    // Transistor is being driven by P2.2
#define TS_INBIT        BIT3                    // 1-wire bus is directly connected to P2.3

#define TS_CYCLE_DELAY_W  18                    // amount of cycles needed to delay for the write bit
#define TS_CYCLE_DELAY_R  11                    // amount of cycles needed to delay for the read bit

// The MSP430F5529 SCLK is 1.048 MHz, so these macros define the amount of clk cycles needed to achieve a certain amount of time
#define TS_15us         16
#define TS_30us         31
#define TS_45us         47
#define TS_60us         63
#define TS_480us        503
```

Figure 3 – Screenshot of the section where main changes should be made

Changing the frequency of the main clock

In order to optimize the timing of the sensor, the read/write functions that control the pins on the MSP430 were written in assembly code. That means that the frequency of each read/write were designed considering how many cycles each instruction takes to execute, so the frequency of the MSP430 plays a crucial role in these functions.

If you desire to change the clock speed of the MSP430, ensure that you also change the TS_CYCLE_DELAY macro in DS18B20.h.

TS_CYCLE_DELAY_W macro determines how many clock cycles the write should delay to satisfy the minimum frequency of the DS18B20. The equation for that number is:

$$TS\_CYCLE\_DELAY\_W = \frac{60\mu s \times f_{clk} - 30 \text{ cycles}}{3 \text{ cycles}}$$

Where:

- 30 cycles is the amount of clock cycles needed to compute the desired logic level and write it the pin
- 3 cycles is how long the delay loop takes
- $60\mu s \times f_{clk}$ accounts for how many clock cycles there are in $60\mu s$

TS_CYCLE_DELAY_R macro determines how many clock cycles the read should delay to satisfy the minimum frequency of the DS18B20. The equation for that number is:

$$TS\_CYCLE\_DELAY\_R = \frac{60\mu s \times f_{clk} - 9 \text{ cycles}}{3 \text{ cycles}}$$

The only difference here is that the read only need 9 cycles to read and store the bit in memory.

Use the following formula to redefine the time constants:

$$n = t \times f_{clk}$$

Where:

- n is the number of cycles
- t is time
- f_{clk} is the new frequency of the clock

Example:

Changing the constant for TS_45us with a clock at 4MHz:

$$n = 45\mu\text{s} \times 4\text{MHz} = 180$$

Changes in the code:

```
#define TS_45us          180
```

IMPORTANT:

- The default frequency is 2^{20}MHz
- There are 4 cycles for the initial pulse, so if you change the frequency to a value greater than 4MHz, you will need to add delays there too, which will make those two equations above invalid
- I don't guarantee that the code will work if any changes are made

Changing the ports

Changing the port should be straight forward. Just change TS_BUS, TS_OUT, TS_INIT_INPORT, and TS_OUTPORT to the port you want. Just don't change the suffix.

Change TS_OUTBIT to the BIT that corresponds to the transmitter pin, and TS_INBIT to the receiver pin.

Example how to change the transmitter to P4.0 and the receiver to P1.5.

```
#define TS_BUS              P1IN
#define TS_OUT              P4OUT
#define TS_INIT_INPORT      P1DIR
#define TS_INIT_OUTPORT     P4DIR
#define TS_INBIT            BIT5
#define TS_OUTBUT           BIT0
```