Laboratory 09: LM75 I2C Interface and Basic I2C Transactions

CHRISTOPHER NIELSEN + CHRISTOPHER SHAMAH
KENNETH SHORT
Bench No: 17
ID#114211318
ID#112229076
Lab Section L01

Questions 1. What is the maximum SCL clock frequency for the LM75? Explain how you determined it. At what SCL frequency are you operating your LM75, explain how you know that.

The minimum SCL period for I2C operation was stated in the LM75 datasheet as 2.5 microseconds, which when converted to a frequency is 400Khz

2. What is the value to be loaded into the MBAUD register to get the desired maximum SCL clock frequency. Show your calculations.

Fscl = Fclk_per / (10 + 2*BUAD +Fclk_per*Tr)
10 + 2*BUAD +Fclk_per*Tr = Fclk_per / Fscl
BAUD = (Fclk_per / Fscl - Fclk_per*Tr - 10)/2
BAUD = (4Mhz/400Khz-4Mhz*.282us-10)/2
BAUD = (10-1.128-10)/2
BUAD = -1.128/2
BAUD = -0.564

We load it in as a 1 because that is the closest we can get, and as BAUD increases the frequency decreases, so we are still within the maximum frequency

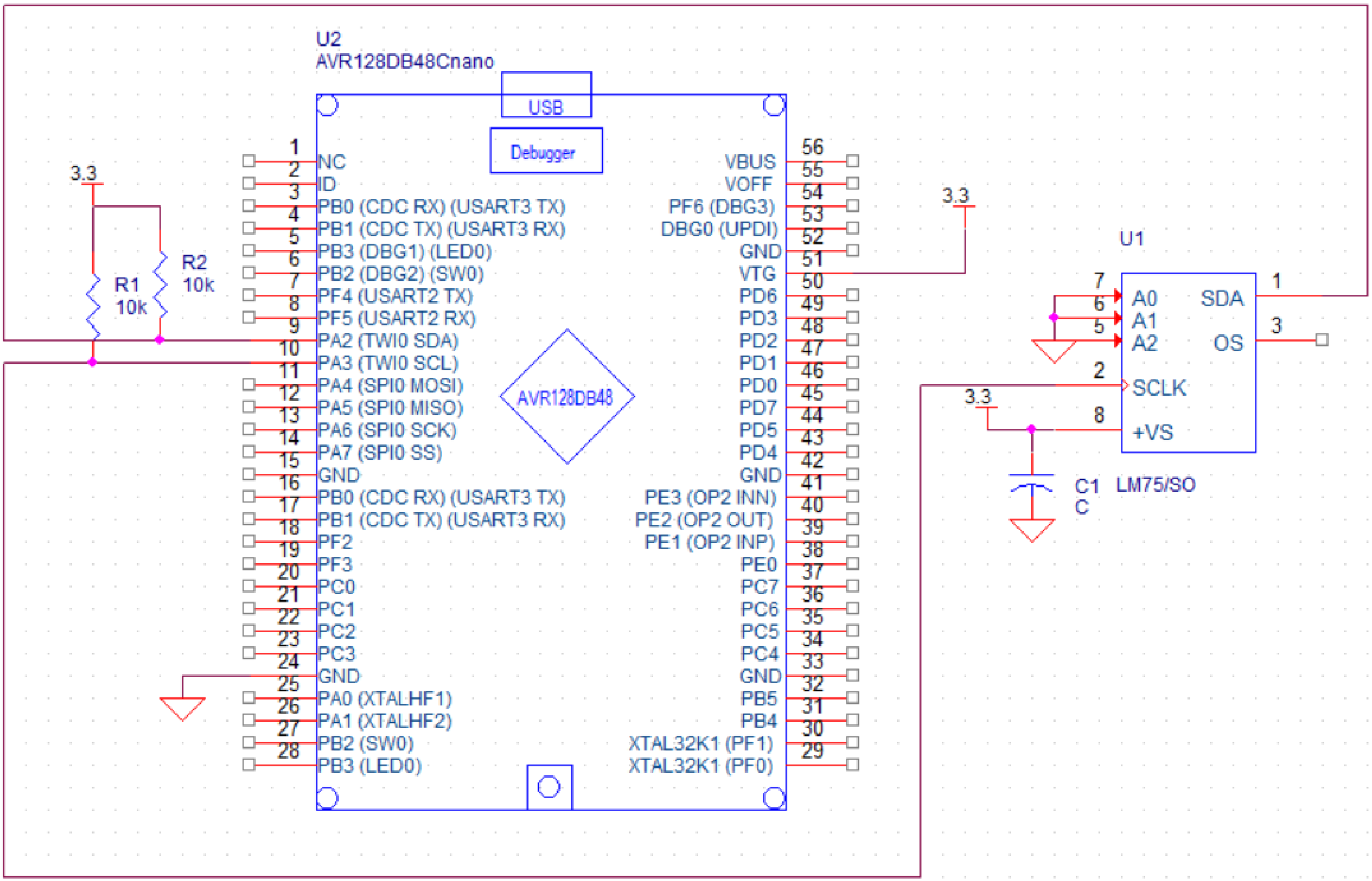
3. What value should the DBGRUN bit have in the DBGCTRL register when you are debugging your software and why?
We want to write a 1 to this bit so that the twi will continue to operate as normal, otherwise there are possible bad behaviors that could occur, such as if we pause while in the middle of transmitting an acknowledgement bit.

4. When viewing SDA and SCL on the Saleae Logic Analyzer, what do the red square and green circle represent?

The red square and green circle represent the START and STOP , or when the bus becomes active versus when the bus is released and becomes “idle” again

5. In one short sentence, explain whether the master or the slave should be providing the acknowledge after a byte transfer

The master should be providing the acknowledgement to tell the slave to continue reporting the data as necessary.


U2
AVR128DB48Cnano
USB
Debugger
AVR128DB48
1 NC
2 ID
3 PB0 (CDC RX) (USART3 TX)
4 PB1 (CDC TX) (USART3 RX)
5 PB3 (DBG1) (LED0)
6 PB2 (DBG2) (SW0)
7 PF4 (USART2 TX)
8 PF5 (USART2 RX)
9 PA2 (TWI0 SDA)
10 PA3 (TWI0 SCL)
11 PA4 (SPI0 MOSI)
12 PA5 (SPI0 MISO)
13 PA6 (SPI0 SCK)
14 PA7 (SPI0 SS)
15 GND
16 PB0 (CDC RX) (USART3 TX)
17 PB1 (CDC TX) (USART3 RX)
18 PF2
19 PF3
20 PC0
21 PC1
22 PC2
23 PC3
24 GND
25 PA0 (XTALHF1)
26 PA1 (XTALHF2)
27 PB2 (SW0)
28 PB3 (LED0)
56 VBUS
55 VOFF
54 PF6 (DBG3)
53 DBG0 (UPDI)
52 GND
51 VTG
50 PD6
49 PD3
48 PD2
47 PD1
46 PD0
45 PD7
44 PD5
43 PD4
42 GND
41 PE3 (OP2 INN)
40 PE2 (OP2 OUT)
39 PE1 (OP2 INP)
38 PE0
37 PC7
36 PC6
35 PC5
34 PC4
33 GND
32 PB5
31 PB4
30 XTAL32K1 (PF1)
29 XTAL32K1 (PF0)
3.3
R1
10k
R2
10k
3.3
U1
7 A0
6 A1
5 A2
2 SCLK
8 +VS
1 SDA
3 OS
3.3
C1
C
LM75/SO

```
/*
 * display_LM75_temp.c
 *
 * Created: 4/7/2024 2:15:40 PM
 * Author : MysticOwl
 */




/*
 * DOG_LCD_BasicTest.c
 *
 * Created: 3/25/2024 11:12:25 PM
 * Author : MysticOwl
 */
#include <avr/io.h>
#define F_CPU 4000000 //freq
#include <util/delay.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>


#define LM75_ADDR 0x48     // LM75 address
#define TEMPERATURE_ADDR 0x00   //slave address of temperature register

uint8_t temp_reg_high;  //high byte of LM75 Temperature register
uint8_t temp_reg_low;   //low byte of LM75 Temperature register
uint16_t LM75_temp_reg = 0; //LM75 Temperature register contents
int16_t LM75_temp = 0;  //right adjusted 2's complement, resolution 0.1C
char dsp_buff1[17];     //buffer for line 1 of LCD image

//Function prototypes
void TWI0_LM75_init(void);  //Initialize TWI0 module to talk to LM75

//Write a byte to the specified I2C slave. Parameters are slave address,
//address of register in slave to be written, and data to be written.
int TWI0_LM75_write(unsigned char saddr, unsigned char raddr, unsigned char data);
//void LM75_TWI0_init(void) ; //Initialize LM75
uint16_t TWI0_LM75_read(unsigned char saddr);



// Display buffer for DOG LCD using sprintf()
char dsp_buff1[17];
char dsp_buff2[17];
char dsp_buff3[17];

void lcd_spi_transmit_CMD (char cmd);  //macro for multiple write spi functions
```

```
  for a setup command
void lcd_spi_transmit_DATA (char cmd); //macro for multiple write spi functions
  for any data to be sent
void init_spi_lcd (void); //init spi0 settings for avr
void init_lcd_dog (void); //finish init commands for dog
void update_lcd_dog(void); //send buffer for line data

void init_spi_lcd (){

    PORTA.DIR |= PIN4_bm; /* Set MOSI pin direction to output */
    PORTA.DIR &= ~PIN5_bm; /* Set MISO pin direction to input */
    PORTA.DIR |= PIN6_bm; /* Set SCK pin direction to output */
    PORTA.DIR |= PIN7_bm; /* Set SS pin direction to output */
    PORTA.OUT |= PIN7_bm; /* Set SS pin direction to output */

    PORTC.DIR |= PIN0_bm; //Reg select output to the display memory
    PORTC.OUT &= ~PIN0_bm;

    SPI0.CTRLB |= (SPI_SSD_bm | 0x03 ); // mode 3 as per the dog waveforms

    SPI0.CTRLA = SPI_ENABLE_bm | SPI_MASTER_bm;

    init_lcd_dog();
}


void lcd_spi_transmit_CMD (char data){

    PORTA_OUT &= ~PIN7_bm; //Slave select ON
    PORTC.OUT &= ~PIN0_bm; // register select 0, command setting
    SPI0.DATA = data;

    while (!(SPI0.INTFLAGS & SPI_IF_bm))  /* waits until data is exchanged*/
    {
        asm volatile ("nop");
    }
    volatile uint8_t dummy;
    dummy = SPI0_DATA;


    PORTF_OUT = PIN7_bm; //Slave select OFF

}

void lcd_spi_transmit_DATA (char data){

    PORTA_OUT &= PIN7_bm; //Slave select ON
    PORTC.OUT = PIN0_bm;  // register select 1, data setting
    SPI0.DATA = data;
    while (!(SPI0.INTFLAGS & SPI_IF_bm))  /* waits until data is exchanged*/
    {
    asm volatile ("nop");
    }


    PORTF_OUT = PIN7_bm; //Slave select OFF

}



void init_lcd_dog(){

        //start_dly_40ms:
        _delay_ms(90);    //startup delay.

        //func_set1:
        lcd_spi_transmit_CMD(0x39);   // send function set #1 //tell for 3 lines
          and data interface at 8 bits
        _delay_us(30);  //delay for command to be processed


        //func_set2:
        lcd_spi_transmit_CMD(0x39); //send function set #2 // again??
        _delay_us(30);  //delay for command to be processed


        //bias_set:
        lcd_spi_transmit_CMD(0x1E); //set bias value.
        _delay_us(30);  //delay for command to be processed


        //power_ctrl:
        lcd_spi_transmit_CMD(0x55); //~ 0x50 nominal for 5V
        //~ 0x55 for 3.3V (delicate adjustment).
        _delay_us(30);  //delay for command to be processed


        //follower_ctrl:
        lcd_spi_transmit_CMD(0x6C); //follower mode on...
        _delay_ms(220); //delay for command to be processed SPECIAL CASE


        //contrast_set:
        lcd_spi_transmit_CMD(0x7F); //~ 77 for 5V, ~ 7F for 3.3V
        _delay_us(30);  //delay for command to be processed
```

```c
        //display_on:
        lcd_spi_transmit_CMD(0x0c); //display on, cursor off, blink off
        _delay_us(30);  //delay for command to be processed


        //clr_display:
        lcd_spi_transmit_CMD(0x01); //clear display, cursor home
        _delay_us(420); //delay for command to be processed


        //entry_mode:
        lcd_spi_transmit_CMD(0x06); //clear display, cursor home
        _delay_us(30);  //delay for command to be processed


}



// Updates the LCD display lines 1, 2, and 3, using the
// contents of dsp_buff_1, dsp_buff_2, and dsp_buff_3, respectively.
void update_lcd_dog(void) {

    init_spi_lcd();     //init SPI port for LCD.



    // send line 1 to the LCD module.
    lcd_spi_transmit_CMD(0x80); //init DDRAM addr-ctr
    _delay_us(30);  //delay for command to be processed
    for (int i = 0; i < 16; i++) {
        lcd_spi_transmit_DATA(dsp_buff1[i]);
        _delay_us(30);  //delay for command to be processed
    }

    // send line 2 to the LCD module.
    lcd_spi_transmit_CMD(0x90); //init DDRAM addr-ctr
    _delay_us(30);  //delay for command to be processed
    for (int i = 0; i < 16; i++) {
        lcd_spi_transmit_DATA(dsp_buff2[i]);
        _delay_us(30);  //delay for command to be processed
    }

    // send line 3 to the LCD module.
    lcd_spi_transmit_CMD(0xA0); //init DDRAM addr-ctr
    _delay_us(30);  //delay for command to be processed
    for (int i = 0; i < 16; i++) {
        lcd_spi_transmit_DATA(dsp_buff3[i]);
        _delay_us(30);  //delay for command to be processed
    }
}




int main(void)
{

    init_spi_lcd();
    TWI0_LM75_init();   //Initialize TWI0 to talk LM75 slave.


    while (1)
    {
        while((TWI0.MSTATUS & 0x03) != 0x01) ; /* wait until I2C bus idle */
        LM75_temp_reg = TWI0_LM75_read(LM75_ADDR);
        LM75_temp = ((int16_t)LM75_temp_reg) >> 7;
//      sprintf(dsp_buff1, "Temp = %4d", (LM75_temp >> 1)); //integer result
//      sprintf(dsp_buff1, "Temp = %4d.%d", (LM75_temp >> 1),((LM75_temp%2) ? 5 : 0) ); //only for pos temps
        sprintf(dsp_buff1, "Temp = %.1f", ((float)(LM75_temp)/2.0)); //requires vprintf library
//      _delay_ms(1000);
        update_lcd_dog();
    }




}


void TWI0_LM75_init(void) {

    //29.3.2.1 Initialization
    //If used, the following bits must be configured before enabling the TWI device:
    //• The SDA Setup Time (SDASETUP) bit from the Control A (TWIn.CTRLA) register
    //• The SDA Hold Time SDAHOLD) bit field from the Control A (TWIn.CTRLA) register
    //• The FM Plus Enable (FMPEN) bit from the Control A (TWIn.CTRLA) register

    //Default values work for this Laboratory

    //29.3.2.1.1 Master Initialization
    //The Master Baud Rate (TWIn.MBAUD) register must be written to a value that
```

```
      will result in a valid TWI bus clock
    //frequency. Writing a ‘1’ to the Enable TWI Master (ENABLE) bit in the Master
      Control A (TWIn.MCTRLA) register will
    //enable the TWI master. The Bus State (BUSSTATE) bit field from the Master
      Status (TWIn.MSTATUS) register must
    //be set to 0x1, to force the bus state to Idle.

    TWI0.MBAUD = 0x01;  //Want 400kHz, but to get it BAUD value would be negative.
       4MHz main clock -> ~400KHz I2C clock
    TWI0.MCTRLA = 0x01; //Enable TWI master bit0
    //Smart mode enable SMEN is bit1, it is 0, so smart mode not enabled.
    //Since SMEN = 0, MCMD field in MCTRLB must be written for each byte
    //received by master to create an acknowledge action followed by an operation.
    TWI0.DBGCTRL = 0x01;
    TWI0.MSTATUS = 0x01; //Force bus state to idle


    //29.3.2.2 TWI Master Operation
    //The TWI master is byte-oriented, with an optional interrupt after each byte.
       There are separate interrupt flags for the
    //master write and read operation. Interrupt flags can also be used for polled
       operation. There are dedicated status
    //flags for indicating ACK/NACK received, bus error, arbitration lost, clock
      hold, and bus state.
    //When an interrupt flag is set to ‘1’, the SCL is forced low. This will give
      the master time to respond or handle any
    //data, and will, in most cases, require software interaction. Clearing the
      interrupt flags releases the SCL. The number
    //of interrupts generated is kept to a minimum by an automatic handling of
      most conditions.


}


int TWI0_LM75_write(uint8_t saddr, uint8_t raddr, uint8_t data) {
    while((TWI0.MSTATUS & 0x03) != 0x01) ; /* wait until idle */

    TWI0.MADDR = saddr << 1;         /* send address for write */
    while((TWI0.MSTATUS & 0x40) == 0); /* WIF flag, wait until saddr sent */

    //The next write clears the WIF flag
    TWI0.MDATA = raddr;              /* send memory address */
    while((TWI0.MSTATUS & 0x40) == 0); /* WIF flag, wait until raddr sent */

    //The next write clears the WIF flag
    TWI0.MDATA = data;               /* send data */
    while((TWI0.MSTATUS & 0x40) == 0); /* WIF flag, wait until data sent */
```

```c
    //The next write clears the WIF flag
    TWI0.MCTRLB |= 0x03;          /* issue a stop */

    return 0;
}




uint16_t TWI0_LM75_read(uint8_t saddr)
{
    while((TWI0.MSTATUS & 0x03) != 0x01) ;  // wait until idle

    TWI0.MADDR = ((saddr << 1) | 0x01);     // send slave address and read command

    while((TWI0.MSTATUS & 0x80) == 0);      // RIF flag, wait until byte is received
                                            // WIF flag does not work here
    temp_reg_high = TWI0.MDATA;             //clears the RIF flag

    TWI0.MCTRLB = 0x02;                 //MCMD - issue ack followed by a byte read operation

    while((TWI0.MSTATUS & 0x80) == 0);      // RIF flag, wait until data received

    temp_reg_low = TWI0.MDATA;              //clears the RIF flag

    TWI0.MCTRLB = 0x07;                 //MCMD issue nack followed by a stop

    return (uint16_t)((temp_reg_high << 8) | (temp_reg_low & 0x80));    //read data from received data buffer
}
```