Speedometer

Embedded Hardware Design Project

Project By:

Pavan Daxini	**200801205**	Documentation
Parth Rao	**200801206**	Incremental Rotatory Encoder
Yash Soni	**200801207**	Documentation & LCD Interfacing
Sonali Dubey	**200801208**	Documentation & LCD Interfacing
Parth Mehta	**200801209**	SPI interfacing and Speed Calculation

INTRODUCTION

The aim this project is to implement the functionality of an electronic speedometer on an ATmega32/16 kit. The wheel turn indicator (Absolute encoder) is implemented on a separate microcontroller (slave) and its values are being passed on to the main microcontroller (master) through Serial Peripheral Interface communication (SPI). The master microcontroller computes the time elapsed and calculates the distance traversed. Finally the calculated speed is displayed on a 16x2 LCD.

Index

1. General Description

1.1 Speedometer Overview

Greek words *"hodós"* meaning "path" and "*métron*" meaning "measure"
An **speedometer** (**milometer**) indicates distance travelled by a vehicle. The device may be electronic, mechanical, or a combination of the two.

1.2 Mechanical Speedometer

Mechanical speedometers are turned by a flexible cable made from a tightly wound spring. The cable usually spins inside a protective metal tube with rubber housing. On a motorcycle, a little wheel rolling against the bike wheel turns the cable, and the gear ratio on the speedometer has to be calibrated to the size of this small wheel. On a car, a gear engages the output shaft of the transmission, turning the cable.
The cable snakes its way up to the instrument panel, where it is connected to the input shaft of the speedometer.

Generally Speedometers are marked as turns: mile ratio. i.e. if an speedometer is marked as 1690:1 that means the input shaft of this speedometer has to spin 1,690 times before the speedometer will register 1 mile.

1.3 Electronic Speedometer

These have a magnet attached to one of the wheels and a pickup attached to the frame. Once per revolution of the wheel, the magnet passes by the pickup, generating a voltage in the pickup. The computer counts these voltage spikes, or pulses, and uses them to calculate the distance travelled.
Many modern cars use a system like this, too. Instead of a magnetic pickup on a wheel, they use a toothed wheel mounted to the output of the transmission and a magnetic sensor that counts the pulses as each tooth of the wheel goes by. Some cars use a slotted wheel and an optical pickup, like a computer mouse does. Just like on the bicycle, the computer in the car knows how much distance the car travels with each pulse, and uses this to update the speedometer reading.

2. System Requirements

2.1 General Requirements

The speedometer system has the following general requirements:

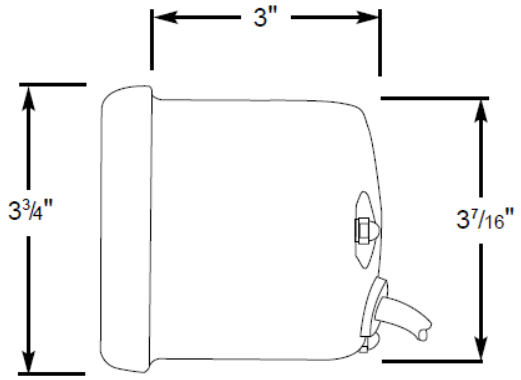
- The energy source must be provided by the vehicle
- The cost of production should be low
- The software architecture must be modular

2.2 Functional Requirements

The speedometer system should have following Functional Requirements:

- As we are measuring the Instantaneous speed, the wheel turn indicator must be able to calculate the exact degree rotation at a faster speed.
- The clock timing should be accurate i.e. no glitches should be present.

2.3 Dimensional Requirements



General Speedometer Dimensions

2.4 Memory Requirements

No permanent storage is required. All the data is stored on the programming stack

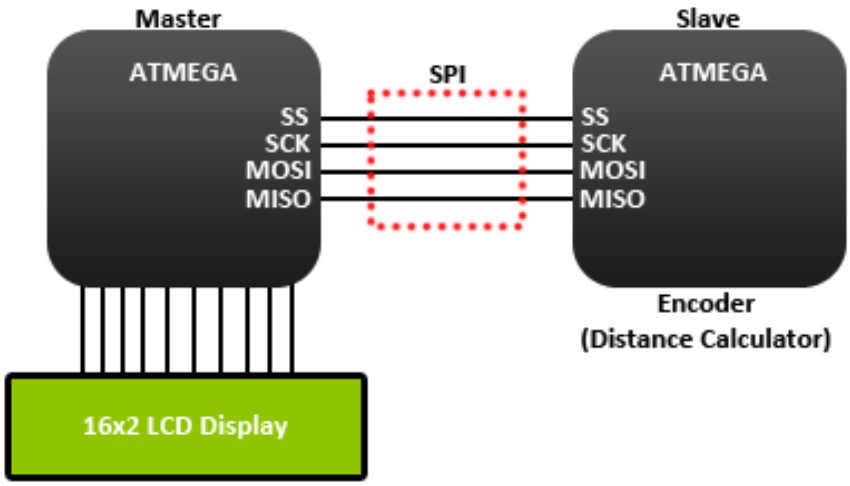
2.5 Display Requirements

The speed output is displayed on the LCD, so following things need to be checked:

- LCD refresh rate should be fast enough to display the instantaneous speed
- It should be able to interface with the ATMEL microcontroller which we are using

3. Implementation Basics

3.1 Block Diagram



3.2 Block Description

Master Microcontroller	The speedometer hardware is based on the Atmel AVR32 The slave sends in the distance encoded data to the master through SPI communication. Master calculates the speed by dividing the distance traversed by the time elapsed.
Slave Microcontroller (Distance calculator)	An Absolute encoder is a sensor used in vehicles to determine the distance travelled by measuring the degree of rotation of the wheel. The output of such an encoder is a digital signal. The same functionality is being implemented on the slave. The signal is transferred using SPI (Synchronous Peripheral Interface).
LCD	Speed calculated by the master microcontroller is displayed on a 16x2 LCD display.

4. Overview of the AVR Micro-Controller

4.1 Introduction

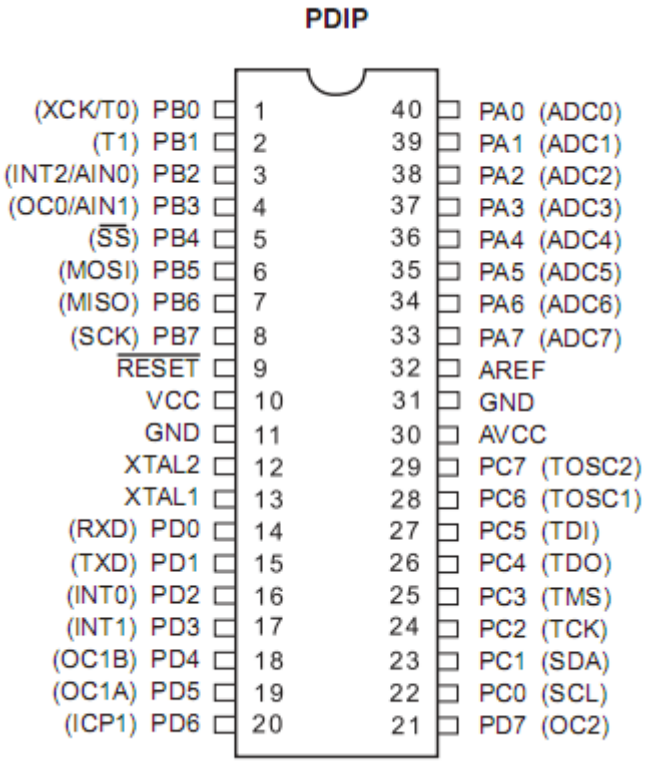
The ATmega32 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega32 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

4.2 Pin Descriptions

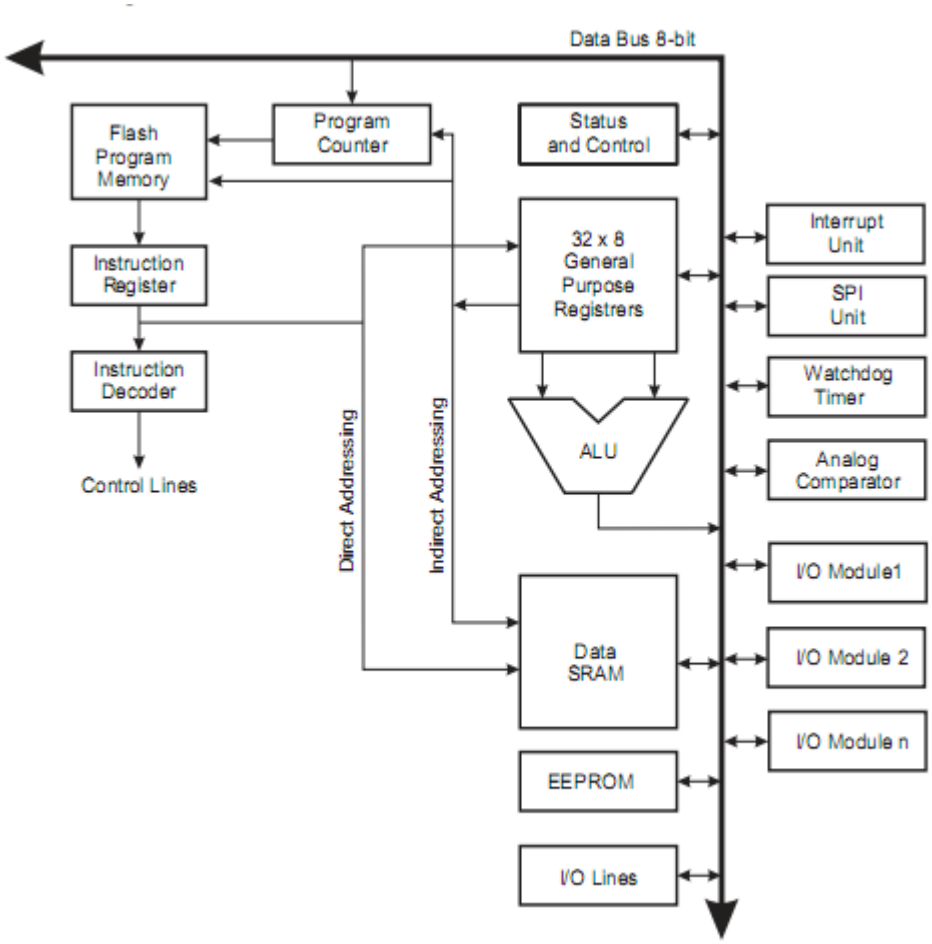
VCC	Voltage Supply
GND	Ground
PORTA (PA0...PA7)	Port A serves as the analog inputs to the A/D Converter. Port A also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. Port pins can provide internal pull-up resistors (selected for each bit). The Port A output buffers have symmetrical drive characteristics with both high sink and source capability. When pins PA0 to PA7 are used as inputs and are externally pulled low, they will source current if the internal pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active, even if the clock is not running.
PORTB (PB0...PB7)	Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.
PORTC (PC0...PC7)	Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running. If the JTAG interface is enabled, the pull-up resistors on pins PC5 (TDI), PC3 (TMS) and PC2 (TCK) will be activated even if a reset occurs. The TD0 pin is tri-stated unless TAP states that shift out data are entered.
PORTD (PD0...PD7)	Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink And source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

$\overline{\text{RESET}}$	Reset Input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running. Shorter pulses are not guaranteed to generate a reset.
AREF	AREF is the analog reference pin for the A/D Converter.

4.3 Pin Configuration



4.4 Internal Block Diagram



5. Peripherals and Equipment used

5.1 Serial Peripheral Interface – SPI

The Serial Peripheral Interface (SPI) allows high-speed synchronous data transfer between the ATmega32 and peripheral devices or between several AVR devices.
For this project, we are using Master and Slave Operation.

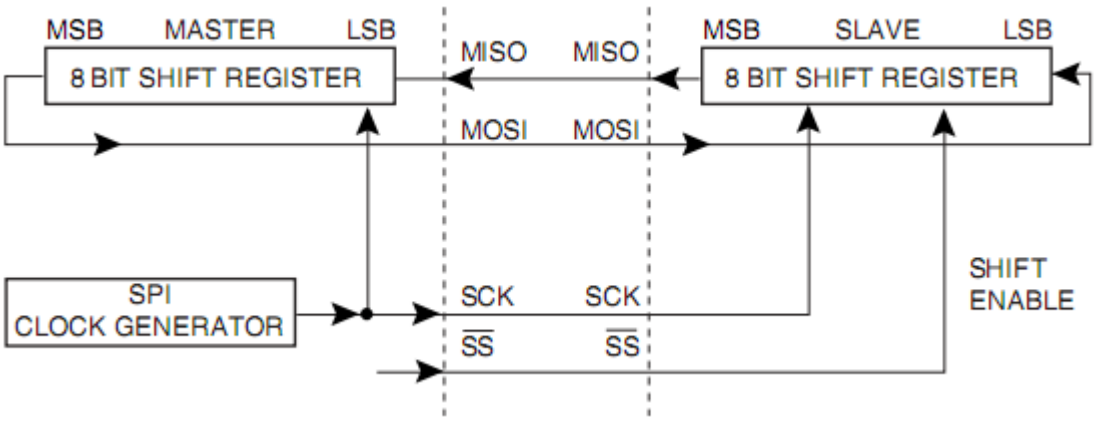
- The system consists of two Shift Registers, and a Master clock generator. The SPI Master initiates the communication cycle when pulling low the Slave Select SS pin of the desired Slave. Master and Slave prepare the data to be sent in their respective Shift Registers, and the Master generates the required clock pulses on the SCK line to interchange data. Data is always shifted from Master to Slave on the Master Out – Slave In, MOSI, line, and from Slave to Master on the Master In – Slave Out, MISO, line. After each data packet, the Master will synchronize the Slave by pulling high the Slave Select, SS, line.

5.1.1 Master Mode

When **configured as a Master**, the SPI interface has no automatic control of the SS line. This must be handled by user software before communication can start. When this is done, writing a byte to the SPI Data Register starts the SPI clock generator, and the hardware shifts the eight bits into the Slave. After shifting one byte, the SPI clock generator stops, setting the end of Transmission Flag (SPIF). If the SPI Interrupt Enable bit (SPIE) in the SPCR Register is set, an interrupt is requested. The Master may continue to shift the next byte by writing it into SPDR, or signal the end of packet by pulling high the Slave Select, SS line. The last incoming byte will be kept in the Buffer Register for later use.

5.1.2 Slave Mode

When **configured as a Slave**, the SPI interface will remain sleeping with MISO tri-stated as long as the SS pin is driven high. In this state, software may update the contents of the SPI Data Register, SPDR, but the data will not be shifted out by incoming clock pulses on the SCK pin until the SS pin is driven low. As one byte has been completely shifted, the end of Transmission Flag, SPIF is set. If the SPI Interrupt Enable bit, SPIE, in the SPCR Register is set, an interrupt is requested. The Slave may continue to place new data to be sent into SPDR before reading the incoming data. The last incoming byte will be kept in the Buffer Register for later use.



SPI Master-slave Interconnection

- When the SPI is enabled, the data direction of the MOSI, MISO, SCK, and SS pins is overridden according to table shown below.

Pin	Direction, Master SPI	Direction, Slave SPI
MOSI	User Defined	Input
MISO	Input	User Defined
SCK	User Defined	Input
$\overline{SS}$	User Defined	Input

Pin Description Table

5.1.3 SPI Control Register – SPCR

Bit	7	6	5	4	3	2	1	0	
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – SPIE: SPI Interrupt Enable

  This bit causes the SPI interrupt to be executed if SPIF bit in the SPSR Register is set and the if the global interrupt enable bit in SREG is set.

- Bit 6 – SPE: SPI Enable

  When the SPE bit is written to one, the SPI is enabled. This bit must be set to enable any SPI operations.

- Bit 5 – DORD: Data Order

  When the DORD bit is written to one, the LSB of the data word is transmitted first.

  When the DORD bit is written to zero, the MSB of the data word is transmitted first.

- Bit 4 – MSTR: Master/Slave Select

  This bit selects Master SPI mode when written to one, and Slave SPI mode when written logic zero. If SS is configured as an input and is driven low while MSTR is set, MSTR will be cleared, and SPIF in SPSR will become set. The user will then have to set MSTR to re-enable SPI Master mode.

- Bit 3 – CPOL: Clock Polarity

  When this bit is written to one, SCK is high when idle. When CPOL is written to zero, SCK is low when idle. Refer to Figure 67 and Figure 68 for an example.

- Bit 2 – CPHA: Clock Phase

  The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK.

- Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0

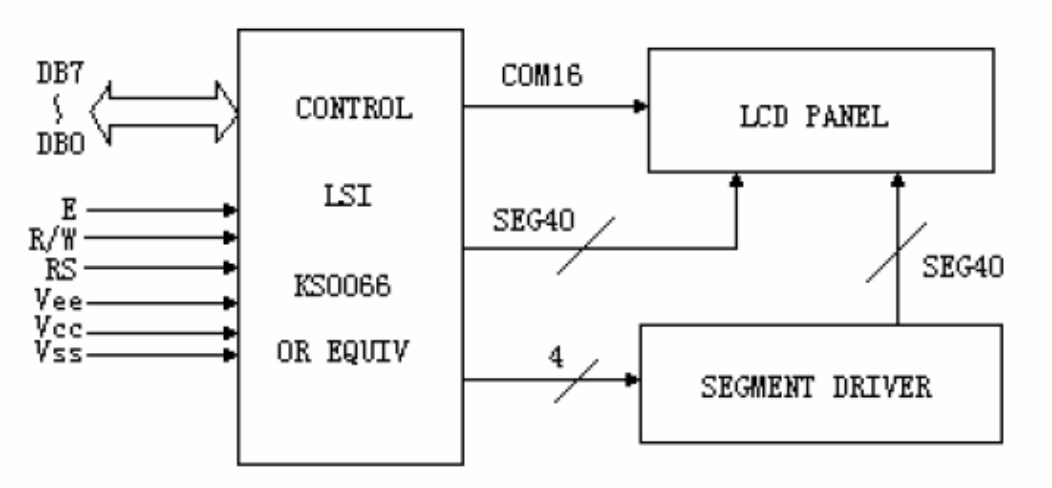
  These two bits control the SCK rate of the device configured as a Master. SPR1 and SPR0 have no effect on the Slave.

5.2 LCD Usage (Model: JHD162A)

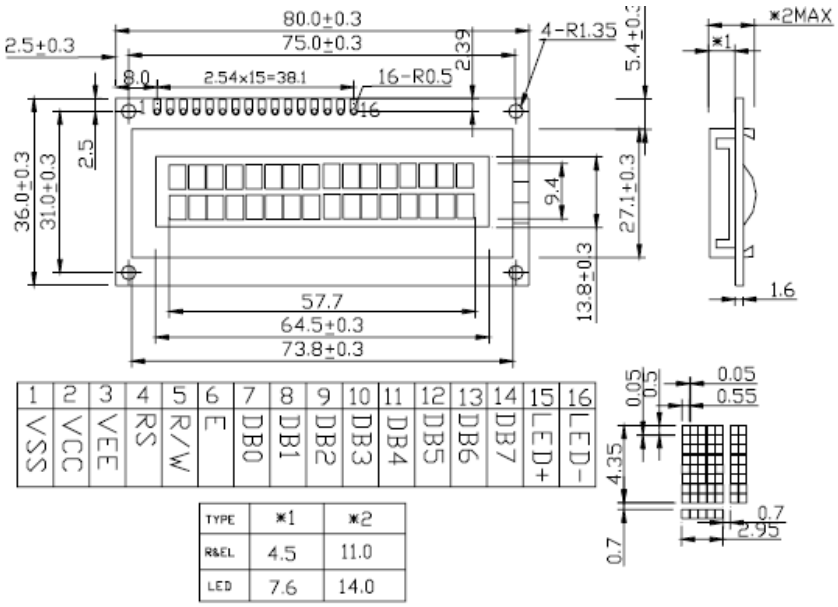
5.2.1 Characteristics

- Character Dots： 5 x 8
- Driving Mode： 1/16D
- Display Content： 16 CHAR x 2ROW
- Parameters: (VDD=5.0V±10%,VSS=0V,Ta=25℃)

5.2.2 Application Circuit



5.2.3 Dimension/Display Circuit



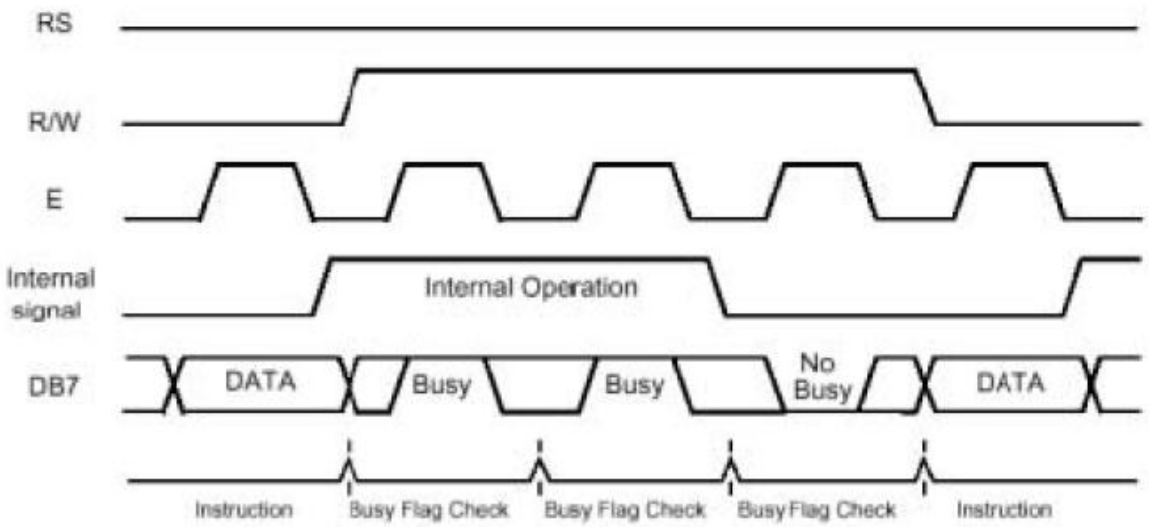
5.2.4 Pin Configuration

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
VSS	VCC	VEE	RS	R/W	E	DB0	DB1	DB2	DB3	DB4	DB5	DB6	DB7	LED+	LED-

5.2.5 Description/Working

- The LCD can be used in 2 modes – 4 bit mode and 8 bit mode
- In 8 bit mode 8 data pins are there which sends a character (1 byte) directly to the LCD. In 4 bit mode the upper nibble is sent first and then lower nibble.
- The three control lines are referred to as EN, RS, and RW.
- The EN line is called "Enable." This control line is used to tell the LCD that you are sending it data. To send data to the LCD, your program should make sure this line is low (0) and then set the other two control lines and/or put data on the data bus. When the other lines are completely ready, bring EN high (1) and wait for the minimum amount of time required by the LCD datasheet (this varies from LCD to LCD), and end by bringing it low (0) again.
- The RS line is the "Register Select" line. When RS is low (0), the data is to be treated as a command or special instruction (such as clear screen, position cursor, etc.). When RS is high (1), the data being sent is text data which should be displayed on the screen. For example, to display the letter "T" on the screen you would set RS high.
- The RW line is the "Read/Write" control line. When RW is low (0), the information on the data bus is being written to the LCD. When RW is high (1), the program is effectively reading the LCD. Only one instruction ("Get LCD status") is a read command. All others are write commands--so RW will almost always be low.
- Finally, the data bus consists of 4 or 8 lines (depending on the mode of operation selected by the user). In the case of an 8-bit data bus, the lines are referred to as DB0, DB1, DB2, DB3, DB4, DB5, DB6, and DB7. IN 4 bit mode, we use DB4, DB5, DB6, and DB7. Rest 4 pins are not shorted or ground.

5.2.6 Timing Diagram



5.3 Memory usage

- Slave program memory: (text + data + boot loader): 5664 bytes
- Master program memory: (text + data + boot loader): 708 bytes
- SRAM Data memory (data + bss + noinit): 277 bytes

6. Working (Distance & Time)

6.1 Overview

There are two AVR Microcontrollers being used. One is used to give input to the other. The first microcontroller (slave) serves as an absolute encoder. An Absolute encoder is a sensor used in vehicles to determine the distance travelled by measuring the degree of rotation of the wheel. The output of such an encoder is a digital signal. The same functionality is being implemented on the slave. The signal is transferred from one microcontroller to the other (master) using SPI (Synchronous Peripheral Interface). The second controller calculates the speed according to the degree of rotation and displays it on a 16X2 LCD.

6.2 Distance calculation (Slave Microcontroller)

- Arranging an actual Absolute Encoder was difficult so we implemented it's functionality on a Microcontroller.
- We have assumed the Maximum RPM for any vehicle is 1000
- After every 12^{o} of rotation of the wheel, the value is updated in the ECDR (Encoder Data Register). Hence the value in the ECDR is sampled 30 times in one complete rotation of the wheel
- A time delay is introduced in between 2 values which the slave transmits. This delay time depends on the Value of RPM.
- Data goes out from the Slave into the Master. Therefore the MISO pin in DDRB is set to 1 and MOSI, SCK & SS pins are set to 0.
- We use DDRB because PORT B's alternate functions include the SPI pins.

6.3 Distance Calculation (Master Microcontroller)

- A get_slave_status() function has been defined which fetches the data from the SPDR whenever called
- This function is called every 10ms and the degree difference of the 2 consecutive samples gives the distance travelled and hence the speed via the following formula:

  12*deg_diff*CIRCUMFERENCE/360
- This value is displayed on the LCD.

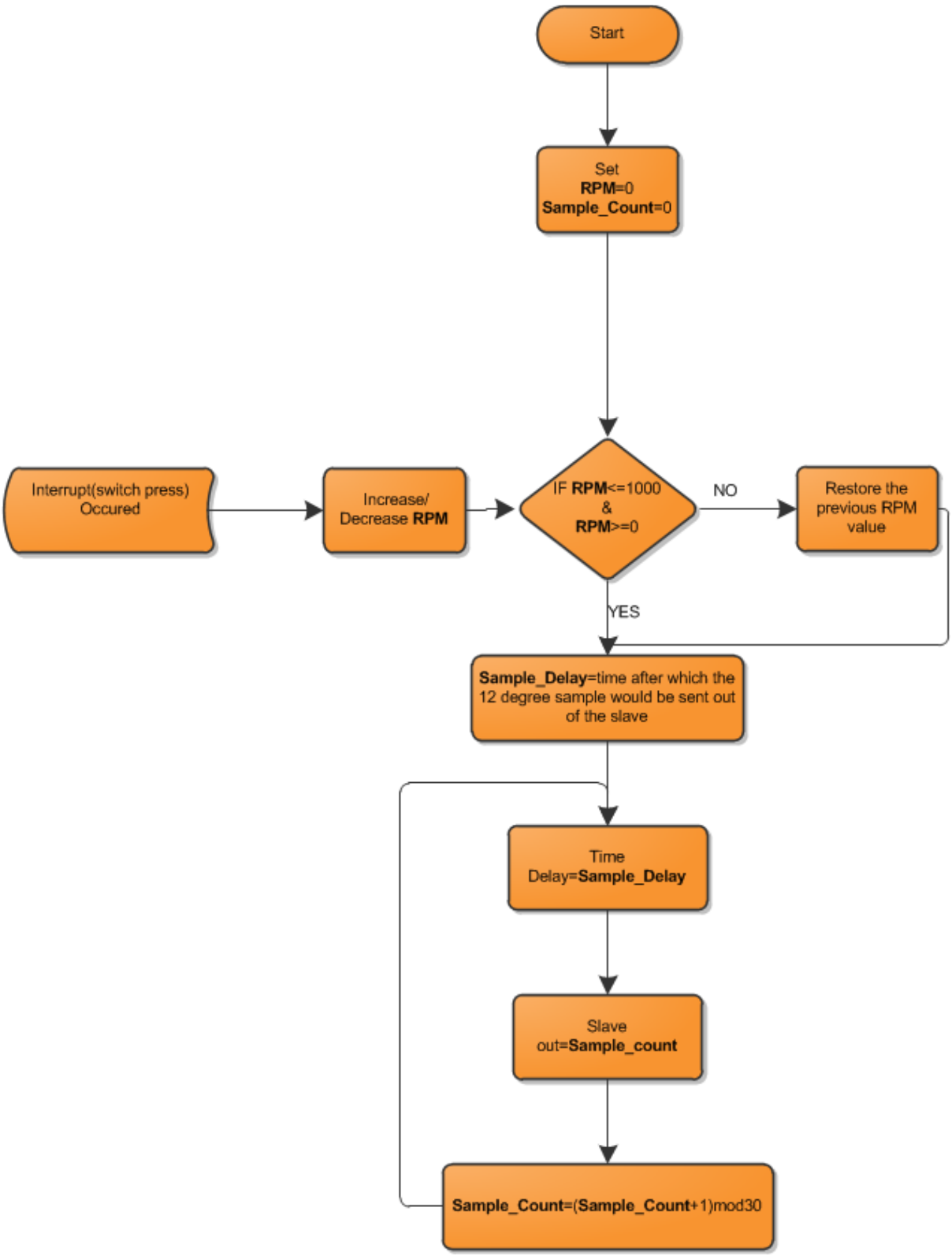
6.4 Time Calculation (Master Microcontroller)

- The clock frequency of the microcontroller is set to 1 MHz
- Thus there will be 10^6 cycles per second.
- So, first we prescale the frequency of the clock by 1024 by setting the 0 and 2 pin of TCCR0 to 1.
- Now there will be (10^6 /1024) = 973 cycles per second.
- The clock register TCCNT0 (8 bits) goes from 0 to 255 and then resets to 0.
- For 973 counts, we divide it by 4. Thus, we require 4 cycles 243 clock cycles each. So for every cycle, we set the initial value of TCCNT0 to 12.
- The clock gives an Overflow Interrupt whenever it overflows. So we need to enable the catching of Overflow Interrupt.
- For enabling any interrupt, the 'I bit' of SREG needs to be set to 1. This is done by the command 'sei()' .
- The Overflow register of Timer/Counter0 is set on by setting TIMSK0 = 0x01.
- So, after every Overflow interrupt, the value of 'flag' is incremented by one. After the value the value of 'flag' becomes three, at next overflow interrupt, one second would have elapsed.
- After this cycle, the value of 'sec' gets incremented by one and value of 'count' is reset to '0'.
- Thus, after every one second, 'sec' is incremented by 1.

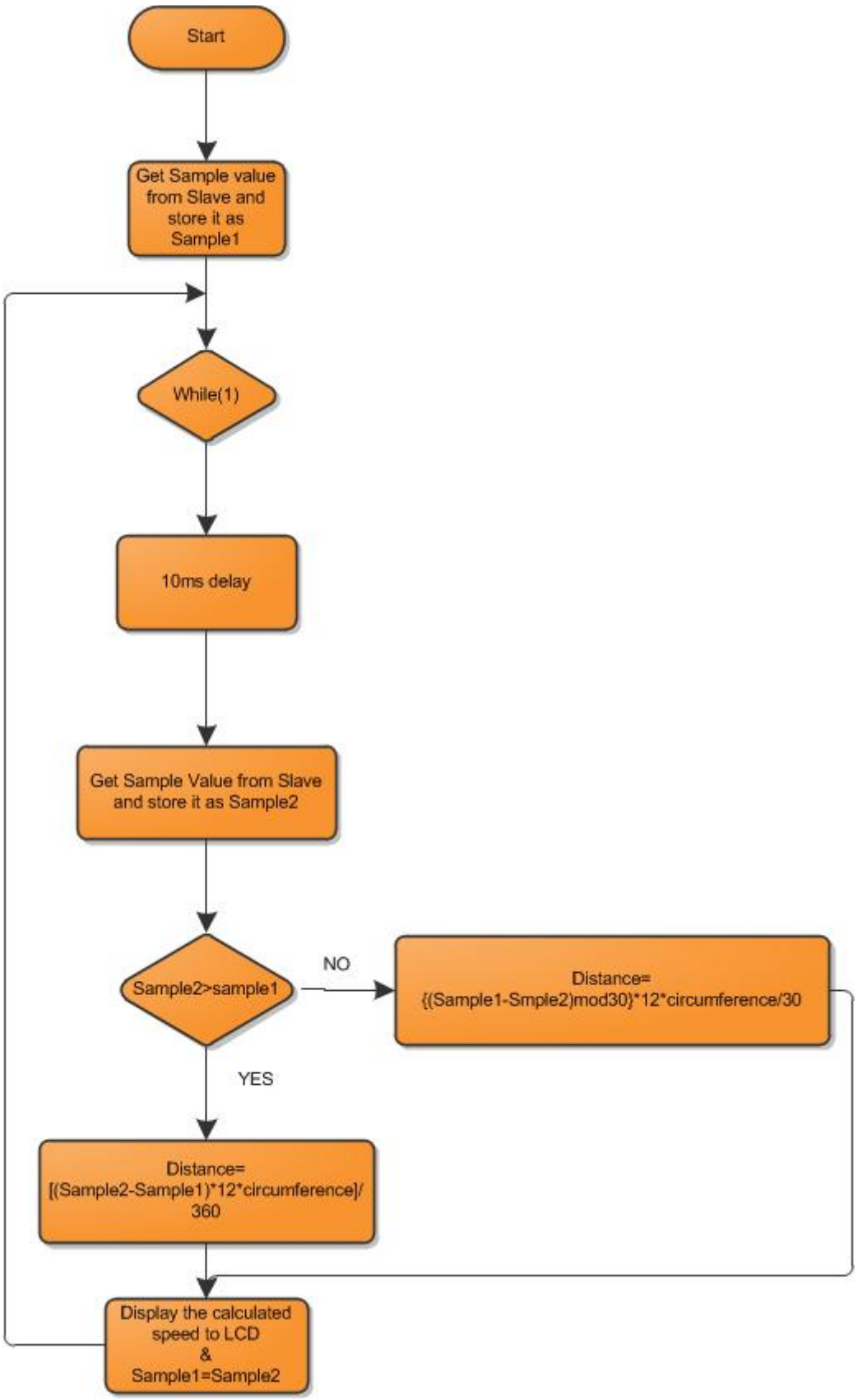
Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Slave Flowchart



Master Flowchart



7. Appendix

This Appendix Contains the following Source Codes:

Spi.c

```
// for SPI MASTER
// master takes the input of the slave..

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "lcd.h"
#include <stdlib.h>
#include <string.h>
#define CIRCUMFERENCE 116

void set_print_lcd(int x,int y, char* c)
{
        lcd_gotoxy(x,y); //function in the lcd library to move the cursor
        lcd_puts(c);   //function in the lcd library to display a string of characters on the lcd.
}



char get_slave_status()
{
        SPDR = 'a';
        while(!(SPSR & (1<<SPIF)));
        return SPDR;
}



int main()
{
        DDRA = 0xFF;                    // FOR the LCD
        PORTA = 0x00;
        lcd_init(LCD_DISP_ON);
        lcd_clrscr();
// SS is high so spi of slave is disabled initially
        PORTB = PORTB | 0b00010000;
// MISO=ip, MOSI=op, SS=op, SCK=op;
        DDRB  = (1<<PB4)|(1<<PB5)|(1<<PB7);
// for enabling pull up on the pin configured as input;
        PORTB |= (1<<PB6);
 // spi cntrol reg SPIE,SPI,MASTR,PRESCALER setting
        SPCR = 0b11010011;
//for clearing the SPIE flag
        char a = SPSR;
```

```
        a = SPDR;

// global interrupt enable
//      sei();
// ss is low so slave is enabled and spi is running.
        PORTB = PORTB & 0b11101111; // ss is low

        int sample_1;
        int sample_2,deg;
        double speed;
        char *buff;
        char *temp = "parth  ";
        sample_1 =     get_slave_status();

        while(1)
        {
                sample_2 = get_slave_status();

                if(sample_1>= 15 && sample_2<15) deg = (30-sample_1) + sample_2;
                else deg = sample_2 - sample_1;
                speed = (12*deg*CIRCUMFERENCE)/360;//speed in m/s
                _delay_ms(10);
                strcpy(temp,itoa(speed,buff,10));
                set_print_lcd(0,0,temp);
        }

}
```

Spi_slave.c

```
// for SPI slave
//

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <math.h>
#include <inttypes.h>
#include <string.h>
#include "lcd.h"
#include <stdlib.h>

char read;
char write = '0';
int count=0;
int rpm=500;
int delay;
long double delay1;


void set_print_lcd(int x,int y, char *str)
{
	lcd_gotoxy(x,y); //function in the lcd library to move the cursor
	lcd_puts(str);   //function in the lcd library to display a string of characters on the lcd.
}


void count_delay(int delay)
{
	int d_ct= delay/255;
	int r_ct= delay%255;
	int i;
	for(i=1;i<=d_ct;i++)
	{
		_delay_us(255);
	}
	_delay_us(r_ct);
	return;
}
```

```
SIGNAL(SIG_INTERRUPT0)
{
        if(rpm<1000)
        {   rpm+=100;  }

        delay1=(2000/rpm) *1000;     // [1000/((rpm/60)*30)]*1000
        delay=round(delay1); // rounding delay by half way from 0
        count_delay(delay);
}

SIGNAL(SIG_INTERRUPT1)
{
        if(rpm>0)
        {   rpm-=100;   }

        delay1=(2000/rpm) *1000;     // [1000/((rpm/60)*30)]*1000
        delay=round(delay1); // rounding delay by half way from 0
        count_delay(delay);
}




SIGNAL(SIG_SPI)
{


                read = SPDR;

                SPDR = count;

//              set_print_lcd(1,1,read);
}



int main()
{
        char *buff = "parth ";
        char *temp = "parth  ";

        DDRA = 0xff;
        PORTA = 0x00;

        DDRD = 0xFF;
```

```
        DDRB  = (1<<PB6); // MISO=op, MOSI=ip, SS=ip, SCK=ip;
        PORTB |= 0b10111111; // for pull up enable pins configured as input
        SPCR = 0b11000011; // spi cntrol reg
//      SPSR = 0x00;//spi s_reg

        lcd_init(LCD_DISP_ON);
        char a = SPSR;
        a = SPDR;
        GICR|= (1<<INT1) | (1<<INT0);
        MCUCR = (1<<ISC11)|(1<<ISC01);
        lcd_clrscr();
        sei();

        delay=(2000/rpm) *1000;      // [1000/((rpm/60)*30)]*1000
        delay=round(delay);   // rounding delay by half way from 0


        count_delay(delay);

        while(1)
        {
                count=(count+1)%30;


                /*
                lcd_clrscr();
                strcpy(temp,itoa(count,buff,10));
                set_print_lcd(0,0,temp);
                */
                count_delay(delay);




        }


}
```

Lch.c

```
/************************************************************************
**
 Title    :  HD44780U LCD library
 Author:    Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
 File:       $Id: lcd.c,v 1.14.2.1 2006/01/29 12:16:41 peter Exp $
 Software:  AVR-GCC 3.3
 Target:    any AVR device, memory mapped mode only for AT90S4414/8515/Mega

 DESCRIPTION
     Basic routines for interfacing a HD44780U-based text lcd display

     Originally based on Volker Oth's lcd library,
     changed lcd_init(), added additional constants for lcd_command(),
     added 4-bit I/O mode, improved and optimized code.

     Library can be operated in memory mapped mode (LCD_IO_MODE=0) or in
     4-bit IO port mode (LCD_IO_MODE=1). 8-bit IO port mode not supported.

     Memory mapped mode compatible with Kanda STK200, but supports also
     generation of R/W signal through A8 address line.

 USAGE
     See the C include lcd.h file for a description of each function

************************************************************************
**/
#include <inttypes.h>
#include <avr/io.h>
#include <avr/pgmspace.h>
#include "lcd.h"



/*
** constants/macros
*/
#define DDR(x) (*(&x - 1))      /* address of data direction register of port x */
#if defined(__AVR_ATmega64__) || defined(__AVR_ATmega128__)
   /* on ATmega64/128 PINF is on port 0x00 and not 0x60 */
   #define PIN(x) ( &PORTF==&(x) ? _SFR_IO8(0x00) : (*(&x - 2)) )
#else
          #define PIN(x) (*(&x - 2))    /* address of input register of port x          */
#endif


#if LCD_IO_MODE
```

```
#define lcd_e_delay()   __asm__ __volatile__( "rjmp 1f\n 1:" );
#define lcd_e_high()    LCD_E_PORT  |=  _BV(LCD_E_PIN);
#define lcd_e_low()     LCD_E_PORT  &= ~_BV(LCD_E_PIN);
#define lcd_e_toggle()  toggle_e()
#define lcd_rw_high()   LCD_RW_PORT |=  _BV(LCD_RW_PIN)
#define lcd_rw_low()    LCD_RW_PORT &= ~_BV(LCD_RW_PIN)
#define lcd_rs_high()   LCD_RS_PORT |=  _BV(LCD_RS_PIN)
#define lcd_rs_low()    LCD_RS_PORT &= ~_BV(LCD_RS_PIN)
#endif

#if LCD_IO_MODE
#if LCD_LINES==1
#define LCD_FUNCTION_DEFAULT    LCD_FUNCTION_4BIT_1LINE
#else
#define LCD_FUNCTION_DEFAULT    LCD_FUNCTION_4BIT_2LINES
#endif
#else
#if LCD_LINES==1
#define LCD_FUNCTION_DEFAULT    LCD_FUNCTION_8BIT_1LINE
#else
#define LCD_FUNCTION_DEFAULT    LCD_FUNCTION_8BIT_2LINES
#endif
#endif

#if LCD_CONTROLLER_KS0073
#if LCD_LINES==4

#define KS0073_EXTENDED_FUNCTION_REGISTER_ON  0x24   /* |0|010|0100 4-bit mode
extension-bit RE = 1 */
#define KS0073_EXTENDED_FUNCTION_REGISTER_OFF 0x20   /* |0|000|1001 4 lines mode
*/
#define KS0073_4LINES_MODE                    0x09   /* |0|001|0000 4-bit mode, extension-bit
RE = 0 */

#endif
#endif

/*
** function prototypes
*/
#if LCD_IO_MODE
static void toggle_e(void);
#endif

/*
** local functions
*/
```

```
/*****************************************************************************
 delay loop for small accurate delays: 16-bit counter, 4 cycles/loop
*****************************************************************************/
static inline void _delayFourCycles(unsigned int __count)
{
   if ( __count == 0 )
      __asm__ __volatile__( "rjmp 1f\n 1:" );    // 2 cycles
   else
      __asm__ __volatile__ (
            "1: sbiw %0,1" "\n\t"
            "brne 1b"                        // 4 cycles/loop
            : "=w" (__count)
            : "0" (__count)
           );
}


/*****************************************************************************
delay for a minimum of <us> microseconds
the number of loops is calculated at compile-time from MCU clock frequency
*****************************************************************************/
#define delay(us)  _delayFourCycles( ( ( 1*(XTAL/4000) )*us)/1000 )


#if LCD_IO_MODE
/* toggle Enable Pin to initiate write */
static void toggle_e(void)
{
   lcd_e_high();
   lcd_e_delay();
   lcd_e_low();
}
#endif


/*****************************************************************************
Low-level function to write byte to LCD controller
Input:    data   byte to write to LCD
        rs     1: write data
               0: write instruction
Returns:  none
*****************************************************************************/
#if LCD_IO_MODE
static void lcd_write(uint8_t data,uint8_t rs)
```

```
{
   unsigned char dataBits ;


   if (rs) {   /* write data       (RS=1, RW=0) */
     lcd_rs_high();
   } else {    /* write instruction (RS=0, RW=0) */
     lcd_rs_low();
   }
   lcd_rw_low();

   if ( ( &LCD_DATA0_PORT == &LCD_DATA1_PORT) && ( &LCD_DATA1_PORT ==
&LCD_DATA2_PORT ) && ( &LCD_DATA2_PORT == &LCD_DATA3_PORT )
     && (LCD_DATA0_PIN == 0) && (LCD_DATA1_PIN == 1) && (LCD_DATA2_PIN == 2) &&
(LCD_DATA3_PIN == 3) )
   {
      /* configure data pins as output */
      DDR(LCD_DATA0_PORT) |= 0x0F;

      /* output high nibble first */
      dataBits = LCD_DATA0_PORT & 0xF0;
      LCD_DATA0_PORT = dataBits |((data>>4)&0x0F);
      lcd_e_toggle();

      /* output low nibble */
      LCD_DATA0_PORT = dataBits | (data&0x0F);
      lcd_e_toggle();

      /* all data pins high (inactive) */
      LCD_DATA0_PORT = dataBits | 0x0F;
   }
   else
   {
      /* configure data pins as output */
      DDR(LCD_DATA0_PORT) |= _BV(LCD_DATA0_PIN);
      DDR(LCD_DATA1_PORT) |= _BV(LCD_DATA1_PIN);
      DDR(LCD_DATA2_PORT) |= _BV(LCD_DATA2_PIN);
      DDR(LCD_DATA3_PORT) |= _BV(LCD_DATA3_PIN);

      /* output high nibble first */
      LCD_DATA3_PORT &= ~_BV(LCD_DATA3_PIN);
      LCD_DATA2_PORT &= ~_BV(LCD_DATA2_PIN);
      LCD_DATA1_PORT &= ~_BV(LCD_DATA1_PIN);
      LCD_DATA0_PORT &= ~_BV(LCD_DATA0_PIN);
          if(data & 0x80) LCD_DATA3_PORT |= _BV(LCD_DATA3_PIN);
          if(data & 0x40) LCD_DATA2_PORT |= _BV(LCD_DATA2_PIN);
          if(data & 0x20) LCD_DATA1_PORT |= _BV(LCD_DATA1_PIN);
```

```
        if(data & 0x10) LCD_DATA0_PORT |= _BV(LCD_DATA0_PIN);
    lcd_e_toggle();

    /* output low nibble */
    LCD_DATA3_PORT &= ~_BV(LCD_DATA3_PIN);
    LCD_DATA2_PORT &= ~_BV(LCD_DATA2_PIN);
    LCD_DATA1_PORT &= ~_BV(LCD_DATA1_PIN);
    LCD_DATA0_PORT &= ~_BV(LCD_DATA0_PIN);
        if(data & 0x08) LCD_DATA3_PORT |= _BV(LCD_DATA3_PIN);
        if(data & 0x04) LCD_DATA2_PORT |= _BV(LCD_DATA2_PIN);
        if(data & 0x02) LCD_DATA1_PORT |= _BV(LCD_DATA1_PIN);
        if(data & 0x01) LCD_DATA0_PORT |= _BV(LCD_DATA0_PIN);
    lcd_e_toggle();

    /* all data pins high (inactive) */
    LCD_DATA0_PORT |= _BV(LCD_DATA0_PIN);
    LCD_DATA1_PORT |= _BV(LCD_DATA1_PIN);
    LCD_DATA2_PORT |= _BV(LCD_DATA2_PIN);
    LCD_DATA3_PORT |= _BV(LCD_DATA3_PIN);
  }
}
#else
#define lcd_write(d,rs) if (rs) *(volatile uint8_t*)(LCD_IO_DATA) = d; else *(volatile
uint8_t*)(LCD_IO_FUNCTION) = d;
/* rs==0 -> write instruction to LCD_IO_FUNCTION */
/* rs==1 -> write data to LCD_IO_DATA */
#endif


/*************************************************************************
Low-level function to read byte from LCD controller
Input:    rs     1: read data
              0: read busy flag / address counter
Returns:  byte read from LCD controller
*************************************************************************/
#if LCD_IO_MODE
static uint8_t lcd_read(uint8_t rs)
{
  uint8_t data;


  if (rs)
    lcd_rs_high();               /* RS=1: read data      */
  else
    lcd_rs_low();                /* RS=0: read busy flag */
  lcd_rw_high();                  /* RW=1  read mode      */
```

```
  if ( ( &LCD_DATA0_PORT == &LCD_DATA1_PORT) && ( &LCD_DATA1_PORT ==
&LCD_DATA2_PORT ) && ( &LCD_DATA2_PORT == &LCD_DATA3_PORT )
    && ( LCD_DATA0_PIN == 0 )&& (LCD_DATA1_PIN == 1) && (LCD_DATA2_PIN == 2) &&
(LCD_DATA3_PIN == 3) )
  {
    DDR(LCD_DATA0_PORT) &= 0xF0;        /* configure data pins as input */

    lcd_e_high();
    lcd_e_delay();
    data = PIN(LCD_DATA0_PORT) << 4;    /* read high nibble first */
    lcd_e_low();

    lcd_e_delay();                      /* Enable 500ns low      */

    lcd_e_high();
    lcd_e_delay();
    data |= PIN(LCD_DATA0_PORT)&0x0F;   /* read low nibble       */
    lcd_e_low();
  }
  else
  {
    /* configure data pins as input */
    DDR(LCD_DATA0_PORT) &= ~_BV(LCD_DATA0_PIN);
    DDR(LCD_DATA1_PORT) &= ~_BV(LCD_DATA1_PIN);
    DDR(LCD_DATA2_PORT) &= ~_BV(LCD_DATA2_PIN);
    DDR(LCD_DATA3_PORT) &= ~_BV(LCD_DATA3_PIN);

    /* read high nibble first */
    lcd_e_high();
    lcd_e_delay();
    data = 0;
    if ( PIN(LCD_DATA0_PORT) & _BV(LCD_DATA0_PIN) ) data |= 0x10;
    if ( PIN(LCD_DATA1_PORT) & _BV(LCD_DATA1_PIN) ) data |= 0x20;
    if ( PIN(LCD_DATA2_PORT) & _BV(LCD_DATA2_PIN) ) data |= 0x40;
    if ( PIN(LCD_DATA3_PORT) & _BV(LCD_DATA3_PIN) ) data |= 0x80;
    lcd_e_low();

    lcd_e_delay();                      /* Enable 500ns low      */

    /* read low nibble */
    lcd_e_high();
    lcd_e_delay();
    if ( PIN(LCD_DATA0_PORT) & _BV(LCD_DATA0_PIN) ) data |= 0x01;
    if ( PIN(LCD_DATA1_PORT) & _BV(LCD_DATA1_PIN) ) data |= 0x02;
    if ( PIN(LCD_DATA2_PORT) & _BV(LCD_DATA2_PIN) ) data |= 0x04;
    if ( PIN(LCD_DATA3_PORT) & _BV(LCD_DATA3_PIN) ) data |= 0x08;
    lcd_e_low();
```

```
   }
   return data;
}
#else
#define lcd_read(rs) (rs) ? *(volatile uint8_t*)(LCD_IO_DATA+LCD_IO_READ) : *(volatile
uint8_t*)(LCD_IO_FUNCTION+LCD_IO_READ)
/* rs==0 -> read instruction from LCD_IO_FUNCTION */
/* rs==1 -> read data from LCD_IO_DATA */
#endif


/*************************************************************************
loops while lcd is busy, returns address counter
*************************************************************************/
static uint8_t lcd_waitbusy(void)

{
   register uint8_t c;

   /* wait until busy flag is cleared */
   while ( (c=lcd_read(0)) & (1<<LCD_BUSY)) {}

   /* the address counter is updated 4us after the busy flag is cleared */
   delay(2);

   /* now read the address counter */
   return (lcd_read(0));  // return address counter

}/* lcd_waitbusy */


/*************************************************************************
Move cursor to the start of next line or to the first line if the cursor
is already on the last line.
*************************************************************************/
static inline void lcd_newline(uint8_t pos)
{
   register uint8_t addressCounter;


#if LCD_LINES==1
   addressCounter = 0;
#endif
#if LCD_LINES==2
   if ( pos < (LCD_START_LINE2) )
      addressCounter = LCD_START_LINE2;
   else
```

```
      addressCounter = LCD_START_LINE1;
#endif
#if LCD_LINES==4
#if KS0073_4LINES_MODE
   if ( pos < LCD_START_LINE2 )
      addressCounter = LCD_START_LINE2;
   else if ( (pos >= LCD_START_LINE2) && (pos < LCD_START_LINE3) )
      addressCounter = LCD_START_LINE3;
   else if ( (pos >= LCD_START_LINE3) && (pos < LCD_START_LINE4) )
      addressCounter = LCD_START_LINE4;
   else
      addressCounter = LCD_START_LINE1;
#else
   if ( pos < LCD_START_LINE3 )
      addressCounter = LCD_START_LINE2;
   else if ( (pos >= LCD_START_LINE2) && (pos < LCD_START_LINE4) )
      addressCounter = LCD_START_LINE3;
   else if ( (pos >= LCD_START_LINE3) && (pos < LCD_START_LINE2) )
      addressCounter = LCD_START_LINE4;
   else
      addressCounter = LCD_START_LINE1;
#endif
#endif
   lcd_command((1<<LCD_DDRAM)+addressCounter);

}/* lcd_newline */


/*
** PUBLIC FUNCTIONS
*/

/************************************************************************
Send LCD controller instruction command
Input:   instruction to send to LCD controller, see HD44780 data sheet
Returns: none
************************************************************************/
void lcd_command(uint8_t cmd)
{
   lcd_waitbusy();
   lcd_write(cmd,0);
}


/************************************************************************
Send data byte to LCD controller
Input:   data to send to LCD controller, see HD44780 data sheet
```

```
Returns: none
*************************************************************************/
void lcd_data(uint8_t data)
{
   lcd_waitbusy();
   lcd_write(data,1);
}



/*************************************************************************
Set cursor to specified position
Input:    x  horizontal position  (0: left most position)
          y  vertical position    (0: first line)
Returns:  none
*************************************************************************/
void lcd_gotoxy(uint8_t x, uint8_t y)
{
#if LCD_LINES==1
   lcd_command((1<<LCD_DDRAM)+LCD_START_LINE1+x);
#endif
#if LCD_LINES==2
   if ( y==0 )
      lcd_command((1<<LCD_DDRAM)+LCD_START_LINE1+x);
   else
      lcd_command((1<<LCD_DDRAM)+LCD_START_LINE2+x);
#endif
#if LCD_LINES==4
   if ( y==0 )
      lcd_command((1<<LCD_DDRAM)+LCD_START_LINE1+x);
   else if ( y==1)
      lcd_command((1<<LCD_DDRAM)+LCD_START_LINE2+x);
   else if ( y==2)
      lcd_command((1<<LCD_DDRAM)+LCD_START_LINE3+x);
   else /* y==3 */
      lcd_command((1<<LCD_DDRAM)+LCD_START_LINE4+x);
#endif

}/* lcd_gotoxy */


/*************************************************************************
*************************************************************************/
int lcd_getxy(void)
{
   return lcd_waitbusy();
}
```

```
/*************************************************************************
Clear display and set cursor to home position
*************************************************************************/
void lcd_clrscr(void)
{
   lcd_command(1<<LCD_CLR);
}


/*************************************************************************
Set cursor to home position
*************************************************************************/
void lcd_home(void)
{
   lcd_command(1<<LCD_HOME);
}


/*************************************************************************
Display character at current cursor position
Input:    character to be displayed
Returns:  none
*************************************************************************/
void lcd_putc(char c)
{
   uint8_t pos;


   pos = lcd_waitbusy();   // read busy-flag and address counter
   if (c=='\n')
   {
      lcd_newline(pos);
   }
   else
   {
#if LCD_WRAP_LINES==1
#if LCD_LINES==1
      if ( pos == LCD_START_LINE1+LCD_DISP_LENGTH ) {
         lcd_write((1<<LCD_DDRAM)+LCD_START_LINE1,0);
      }
#elif LCD_LINES==2
      if ( pos == LCD_START_LINE1+LCD_DISP_LENGTH ) {
         lcd_write((1<<LCD_DDRAM)+LCD_START_LINE2,0);
      }else if ( pos == LCD_START_LINE2+LCD_DISP_LENGTH ){
         lcd_write((1<<LCD_DDRAM)+LCD_START_LINE1,0);
```

```
      }
#elif LCD_LINES==4
      if ( pos == LCD_START_LINE1+LCD_DISP_LENGTH ) {
         lcd_write((1<<LCD_DDRAM)+LCD_START_LINE2,0);
      }else if ( pos == LCD_START_LINE2+LCD_DISP_LENGTH ) {
         lcd_write((1<<LCD_DDRAM)+LCD_START_LINE3,0);
      }else if ( pos == LCD_START_LINE3+LCD_DISP_LENGTH ) {
         lcd_write((1<<LCD_DDRAM)+LCD_START_LINE4,0);
      }else if ( pos == LCD_START_LINE4+LCD_DISP_LENGTH ) {
         lcd_write((1<<LCD_DDRAM)+LCD_START_LINE1,0);
      }
#endif
      lcd_waitbusy();
#endif
      lcd_write(c, 1);
   }

}/* lcd_putc */


/*************************************************************************
Display string without auto linefeed
Input:    string to be displayed
Returns:  none
*************************************************************************/
void lcd_puts(const char *s)
/* print string on lcd (no auto linefeed) */
{
   register char c;

   while ( (c = *s++) ) {
      lcd_putc(c);
   }

}/* lcd_puts */


/*************************************************************************
Display string from program memory without auto linefeed
Input:    string from program memory be be displayed
Returns:  none
*************************************************************************/
void lcd_puts_p(const char *progmem_s)
/* print string from program memory on lcd (no auto linefeed) */
{
   register char c;
```

```
   while ( (c = pgm_read_byte(progmem_s++)) ) {
      lcd_putc(c);
   }

}/* lcd_puts_p */


/*************************************************************************
Initialize display and select type of cursor
Input:    dispAttr LCD_DISP_OFF          display off
               LCD_DISP_ON           display on, cursor off
               LCD_DISP_ON_CURSOR      display on, cursor on
               LCD_DISP_CURSOR_BLINK   display on, cursor on flashing
Returns:  none
*************************************************************************/
void lcd_init(uint8_t dispAttr)
{
#if LCD_IO_MODE
   /*
    *  Initialize LCD to 4 bit I/O mode
    */

   if ( ( &LCD_DATA0_PORT == &LCD_DATA1_PORT) && ( &LCD_DATA1_PORT ==
&LCD_DATA2_PORT ) && ( &LCD_DATA2_PORT == &LCD_DATA3_PORT )
     && ( &LCD_RS_PORT == &LCD_DATA0_PORT) && ( &LCD_RW_PORT ==
&LCD_DATA0_PORT) && (&LCD_E_PORT == &LCD_DATA0_PORT)
     && (LCD_DATA0_PIN == 0 ) && (LCD_DATA1_PIN == 1) && (LCD_DATA2_PIN == 2) &&
(LCD_DATA3_PIN == 3)
     && (LCD_RS_PIN == 4 ) && (LCD_RW_PIN == 5) && (LCD_E_PIN == 6 ) )
   {
      /* configure all port bits as output (all LCD lines on same port) */
      DDR(LCD_DATA0_PORT) |= 0x7F;
   }
   else if ( ( &LCD_DATA0_PORT == &LCD_DATA1_PORT) && ( &LCD_DATA1_PORT ==
&LCD_DATA2_PORT ) && ( &LCD_DATA2_PORT == &LCD_DATA3_PORT )
        && (LCD_DATA0_PIN == 0 ) && (LCD_DATA1_PIN == 1) && (LCD_DATA2_PIN == 2) &&
(LCD_DATA3_PIN == 3) )
   {
      /* configure all port bits as output (all LCD data lines on same port, but control lines on
different ports) */
      DDR(LCD_DATA0_PORT) |= 0x0F;
      DDR(LCD_RS_PORT)    |= _BV(LCD_RS_PIN);
      DDR(LCD_RW_PORT)    |= _BV(LCD_RW_PIN);
      DDR(LCD_E_PORT)     |= _BV(LCD_E_PIN);
   }
   else
   {
```

```
    /* configure all port bits as output (LCD data and control lines on different ports */
    DDR(LCD_RS_PORT)    |= _BV(LCD_RS_PIN);
    DDR(LCD_RW_PORT)    |= _BV(LCD_RW_PIN);
    DDR(LCD_E_PORT)     |= _BV(LCD_E_PIN);
    DDR(LCD_DATA0_PORT) |= _BV(LCD_DATA0_PIN);
    DDR(LCD_DATA1_PORT) |= _BV(LCD_DATA1_PIN);
    DDR(LCD_DATA2_PORT) |= _BV(LCD_DATA2_PIN);
    DDR(LCD_DATA3_PORT) |= _BV(LCD_DATA3_PIN);
  }
  delay(16000);       /* wait 16ms or more after power-on      */

  /* initial write to lcd is 8bit */
  LCD_DATA1_PORT |= _BV(LCD_DATA1_PIN);  // _BV(LCD_FUNCTION)>>4;
  LCD_DATA0_PORT |= _BV(LCD_DATA0_PIN);  // _BV(LCD_FUNCTION_8BIT)>>4;
  lcd_e_toggle();
  delay(4992);        /* delay, busy flag can't be checked here */

  /* repeat last command */
  lcd_e_toggle();
  delay(64);          /* delay, busy flag can't be checked here */

  /* repeat last command a third time */
  lcd_e_toggle();
  delay(64);          /* delay, busy flag can't be checked here */

  /* now configure for 4bit mode */
  LCD_DATA0_PORT &= ~_BV(LCD_DATA0_PIN);   // LCD_FUNCTION_4BIT_1LINE>>4
  lcd_e_toggle();
  delay(64);          /* some displays need this additional delay */

  /* from now the LCD only accepts 4 bit I/O, we can use lcd_command() */
#else
  /*
   * Initialize LCD to 8 bit memory mapped mode
   */

  /* enable external SRAM (memory mapped lcd) and one wait state */
  MCUCR = _BV(SRE) | _BV(SRW);

  /* reset LCD */
  delay(16000);                       /* wait 16ms after power-on     */
  lcd_write(LCD_FUNCTION_8BIT_1LINE,0);   /* function set: 8bit interface */
  delay(4992);                        /* wait 5ms                     */
  lcd_write(LCD_FUNCTION_8BIT_1LINE,0);   /* function set: 8bit interface */
  delay(64);                          /* wait 64us                    */
  lcd_write(LCD_FUNCTION_8BIT_1LINE,0);   /* function set: 8bit interface */
  delay(64);                          /* wait 64us                    */
```

```
#endif

#if KS0073_4LINES_MODE
   /* Display with KS0073 controller requires special commands for enabling 4 line mode */
          lcd_command(KS0073_EXTENDED_FUNCTION_REGISTER_ON);
          lcd_command(KS0073_4LINES_MODE);
          lcd_command(KS0073_EXTENDED_FUNCTION_REGISTER_OFF);
#else
   lcd_command(LCD_FUNCTION_DEFAULT);     /* function set: display lines  */
#endif
   lcd_command(LCD_DISP_OFF);            /* display off               */
   lcd_clrscr();                         /* display clear             */
   lcd_command(LCD_MODE_DEFAULT);        /* set entry mode            */
   lcd_command(dispAttr);                /* display/cursor control      */

}/* lcd_init */
```

Lcd.h

```
#ifndef LCD_H
#define LCD_H
/************************************************************************
 Title    :   C include file for the HD44780U LCD library (lcd.c)
 Author:    Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
 File:       $Id: lcd.h,v 1.13.2.2 2006/01/30 19:51:33 peter Exp $
 Software:  AVR-GCC 3.3
 Hardware:  any AVR device, memory mapped mode only for AT90S4414/8515/Mega
************************************************************************
/

/**
 @defgroup pfleury_lcd LCD library
 @code #include <lcd.h> @endcode

 @brief Basic routines for interfacing a HD44780U-based text LCD display

 Originally based on Volker Oth's LCD library,
 changed lcd_init(), added additional constants for lcd_command(),
 added 4-bit I/O mode, improved and optimized code.

 Library can be operated in memory mapped mode (LCD_IO_MODE=0) or in
 4-bit IO port mode (LCD_IO_MODE=1). 8-bit IO port mode not supported.

 Memory mapped mode compatible with Kanda STK200, but supports also
 generation of R/W signal through A8 address line.

 @author Peter Fleury pfleury@gmx.ch http://jump.to/fleury

 @see The chapter <a href="http://homepage.sunrise.ch/mysunrise/peterfleury/avr-
lcd44780.html" target="_blank">Interfacing a HD44780 Based LCD to an AVR</a>
     on my home page.

*/

/*@{*/

#if (__GNUC__ * 100 + __GNUC_MINOR__) < 303
#error "This library requires AVR-GCC 3.3 or later, update to newer AVR-GCC compiler !"
#endif

#include <inttypes.h>
#include <avr/pgmspace.h>

/**
 *  @name  Definitions for MCU Clock Frequency
```

```
 *  Adapt the MCU clock frequency in Hz to your target.
 */
#define XTAL 4000000            /**< clock frequency in Hz, used to calculate delay timer */


/**
 * @name  Definition for LCD controller type
 * Use 0 for HD44780 controller, change to 1 for displays with KS0073 controller.
 */
#define LCD_CONTROLLER_KS0073 0  /**< Use 0 for HD44780 controller, 1 for KS0073
controller */

/**
 *  @name  Definitions for Display Size
 *  Change these definitions to adapt setting to your display
 */
#define LCD_LINES         2     /**< number of visible lines of the display */
#define LCD_DISP_LENGTH    16     /**< visibles characters per line of the display */
#define LCD_LINE_LENGTH  0x40     /**< internal line length of the display    */
#define LCD_START_LINE1  0x00     /**< DDRAM address of first char of line 1 */
#define LCD_START_LINE2  0x40     /**< DDRAM address of first char of line 2 */
#define LCD_START_LINE3  0x14     /**< DDRAM address of first char of line 3 */
#define LCD_START_LINE4  0x54     /**< DDRAM address of first char of line 4 */
#define LCD_WRAP_LINES     0     /**< 0: no wrap, 1: wrap at end of visibile line */


#define LCD_IO_MODE     1        /**< 0: memory mapped mode, 1: IO port mode */
#if LCD_IO_MODE
/**
 *  @name Definitions for 4-bit IO mode
 *  Change LCD_PORT if you want to use a different port for the LCD pins.
 *
 *  The four LCD data lines and the three control lines RS, RW, E can be on the
 *  same port or on different ports.
 *  Change LCD_RS_PORT, LCD_RW_PORT, LCD_E_PORT if you want the control lines on
 *  different ports.
 *
 *  Normally the four data lines should be mapped to bit 0..3 on one port, but it
 *  is possible to connect these data lines in different order or even on different
 *  ports by adapting the LCD_DATAx_PORT and LCD_DATAx_PIN definitions.
 *
 */
#define LCD_PORT        PORTA       /**< port for the LCD lines   */
#define LCD_DATA0_PORT   LCD_PORT    /**< port for 4bit data bit 0 */
#define LCD_DATA1_PORT   LCD_PORT    /**< port for 4bit data bit 1 */
#define LCD_DATA2_PORT   LCD_PORT    /**< port for 4bit data bit 2 */
#define LCD_DATA3_PORT   LCD_PORT    /**< port for 4bit data bit 3 */
```

```
#define LCD_DATA0_PIN    3          /**< pin for 4bit data bit 0  */
#define LCD_DATA1_PIN    2          /**< pin for 4bit data bit 1  */
#define LCD_DATA2_PIN    1          /**< pin for 4bit data bit 2  */
#define LCD_DATA3_PIN    0          /**< pin for 4bit data bit 3  */
#define LCD_RS_PORT      LCD_PORT     /**< port for RS line        */
#define LCD_RS_PIN       6          /**< pin  for RS line        */
#define LCD_RW_PORT      LCD_PORT     /**< port for RW line        */
#define LCD_RW_PIN       5          /**< pin  for RW line        */
#define LCD_E_PORT       LCD_PORT     /**< port for Enable line     */
#define LCD_E_PIN        4          /**< pin  for Enable line     */

#elif defined(__AVR_AT90S4414__) || defined(__AVR_AT90S8515__) ||
defined(__AVR_ATmega64__) || \
    defined(__AVR_ATmega8515__)|| defined(__AVR_ATmega103__) ||
defined(__AVR_ATmega128__) || \
    defined(__AVR_ATmega161__) || defined(__AVR_ATmega162__)
/*
 *  memory mapped mode is only supported when the device has an external data memory
interface
 */
#define LCD_IO_DATA     0xC000    /* A15=E=1, A14=RS=1               */
#define LCD_IO_FUNCTION  0x8000    /* A15=E=1, A14=RS=0               */
#define LCD_IO_READ     0x0100    /* A8 =R/W=1 (R/W: 1=Read, 0=Write   */
#else
#error "external data memory interface not available for this device, use 4-bit IO port mode"

#endif


/**
 *  @name Definitions for LCD command instructions
 *  The constants define the various LCD controller instructions which can be passed to the
 *  function lcd_command(), see HD44780 data sheet for a complete description.
 */

/* instruction register bit positions, see HD44780U data sheet */
#define LCD_CLR             0     /* DB0: clear display              */
#define LCD_HOME            1     /* DB1: return to home position       */
#define LCD_ENTRY_MODE      2     /* DB2: set entry mode              */
#define LCD_ENTRY_INC       1     /*   DB1: 1=increment, 0=decrement     */
#define LCD_ENTRY_SHIFT     0     /*   DB2: 1=display shift on          */
#define LCD_ON              3     /* DB3: turn lcd/cursor on           */
#define LCD_ON_DISPLAY      2     /*   DB2: turn display on            */
#define LCD_ON_CURSOR       1     /*   DB1: turn cursor on             */
#define LCD_ON_BLINK        0     /*    DB0: blinking cursor ?         */
#define LCD_MOVE            4     /* DB4: move cursor/display          */
#define LCD_MOVE_DISP       3     /*   DB3: move display (0-> cursor) ?  */
```

```
#define LCD_MOVE_RIGHT       2     /*   DB2: move right (0-> left) ?     */
#define LCD_FUNCTION         5     /* DB5: function set                  */
#define LCD_FUNCTION_8BIT    4     /*   DB4: set 8BIT mode (0->4BIT mode) */
#define LCD_FUNCTION_2LINES  3     /*   DB3: two lines (0->one line)     */
#define LCD_FUNCTION_10DOTS  2     /*   DB2: 5x10 font (0->5x7 font)     */
#define LCD_CGRAM            6     /* DB6: set CG RAM address            */
#define LCD_DDRAM            7     /* DB7: set DD RAM address            */
#define LCD_BUSY             7     /* DB7: LCD is busy                   */

/* set entry mode: display shift on/off, dec/inc cursor move direction */
#define LCD_ENTRY_DEC          0x04   /* display shift off, dec cursor move dir */
#define LCD_ENTRY_DEC_SHIFT    0x05   /* display shift on,  dec cursor move dir */
#define LCD_ENTRY_INC_         0x06   /* display shift off, inc cursor move dir */
#define LCD_ENTRY_INC_SHIFT    0x07   /* display shift on,  inc cursor move dir */

/* display on/off, cursor on/off, blinking char at cursor position */
#define LCD_DISP_OFF           0x08   /* display off                        */
#define LCD_DISP_ON            0x0C   /* display on, cursor off             */
#define LCD_DISP_ON_BLINK      0x0D   /* display on, cursor off, blink char */
#define LCD_DISP_ON_CURSOR     0x0E   /* display on, cursor on              */
#define LCD_DISP_ON_CURSOR_BLINK 0x0F   /* display on, cursor on, blink char */

/* move cursor/shift display */
#define LCD_MOVE_CURSOR_LEFT    0x10   /* move cursor left  (decrement)     */
#define LCD_MOVE_CURSOR_RIGHT   0x14   /* move cursor right (increment)     */
#define LCD_MOVE_DISP_LEFT      0x18   /* shift display left                */
#define LCD_MOVE_DISP_RIGHT     0x1C   /* shift display right               */

/* function set: set interface data length and number of display lines */
#define LCD_FUNCTION_4BIT_1LINE  0x20   /* 4-bit interface, single line, 5x7 dots */
#define LCD_FUNCTION_4BIT_2LINES 0x28   /* 4-bit interface, dual line,   5x7 dots */
#define LCD_FUNCTION_8BIT_1LINE  0x30   /* 8-bit interface, single line, 5x7 dots */
#define LCD_FUNCTION_8BIT_2LINES 0x38   /* 8-bit interface, dual line,   5x7 dots */


#define LCD_MODE_DEFAULT     ((1<<LCD_ENTRY_MODE) | (1<<LCD_ENTRY_INC) )



/**
 *  @name Functions
 */


/**
 @brief    Initialize display and select type of cursor
 @param    dispAttr \b LCD_DISP_OFF display off\n
```

```
                \b LCD_DISP_ON display on, cursor off\n
                \b LCD_DISP_ON_CURSOR display on, cursor on\n
                \b LCD_DISP_ON_CURSOR_BLINK display on, cursor on flashing
 @return  none
*/
extern void lcd_init(uint8_t dispAttr);


/**
 @brief    Clear display and set cursor to home position
 @param    void
 @return   none
*/
extern void lcd_clrscr(void);


/**
 @brief    Set cursor to home position
 @param    void
 @return   none
*/
extern void lcd_home(void);


/**
 @brief    Set cursor to specified position

 @param    x horizontal position\n (0: left most position)
 @param    y vertical position\n   (0: first line)
 @return   none
*/
extern void lcd_gotoxy(uint8_t x, uint8_t y);


/**
 @brief    Display character at current cursor position
 @param    c character to be displayed
 @return   none
*/
extern void lcd_putc(char c);


/**
 @brief    Display string without auto linefeed
 @param    s string to be displayed
 @return   none
*/
```

```
extern void lcd_puts(const char *s);


/**
 @brief    Display string from program memory without auto linefeed
 @param    s string from program memory be be displayed
 @return   none
 @see      lcd_puts_P
*/
extern void lcd_puts_p(const char *progmem_s);


/**
 @brief    Send LCD controller instruction command
 @param    cmd instruction to send to LCD controller, see HD44780 data sheet
 @return   none
*/
extern void lcd_command(uint8_t cmd);


/**
 @brief    Send data byte to LCD controller

 Similar to lcd_putc(), but without interpreting LF
 @param    data byte to send to LCD controller, see HD44780 data sheet
 @return   none
*/
extern void lcd_data(uint8_t data);


/**
 @brief macros for automatically storing string constant in program memory
*/
#define lcd_puts_P(__s)         lcd_puts_p(PSTR(__s))

/*@}*/
#endif //LCD_H
```