

University of Florida
Department of Electrical and Computer Engineering
EEL 5666
Intelligent Machines Design Laboratory
Final Report

NaviGator

A solution to traffic jams frustration

Date: 8/01/05
Name: Jonathan Mau
TAs: William Dubel
Steven Pickles
Instructors: A.A. Arroyo
E.M. Schwartz

Table of Contents

I.	Abstract	3
II.	Executive Summary	4
III.	Introduction	5
IV.	Integrated System	5
V.	Mobile Platform	6
VI.	Actuation	8
VII.	Sensors	
	a.	9
	b.	10
	c.	12
	d.	15
VIII.	Behaviors	16
IX.	Experimental Layout and Results	17
X.	Conclusion	18
XI.	Documentation	
	a.	19
	b.	20
XII.	Appendix – Source Code	21

Abstract

The purpose of the robot, NaviGator, is to simulate a concept applicable to the real world. The user will remotely control a guide car where the follow car, the robot, will actively follow the guide car's path, while maintaining a safe distance. Measurements and relevant information will be displayed on the LCD. In the case of contact or an obstructing obstacle, the robot will stop its motion and shut down the system.

Executive Summary

The purpose of NaviGator is to follow a remote control car to simulate autopilot control during a traffic jam. It uses as its brains the Atmel chip based Mekatronix ATmega128 board. The robot's platform is based on a hacked remote control car and specially cut balsa wood. It uses the existing rear-wheel drive and single-axle steering setup. Sensors will be mounted in the front and in the back of the robot. Ultrasonic sensors are placed in the front and on the left and right side where an infrared sensor is mounted in the rear, and bump sensors are placed in the front and back of the robot.

With the guide car in front of it, NaviGator uses ultrasonic sensors to perform its main task. Upon taking measurements from each ultrasonic sensor, it then decides the appropriate action to allow it to follow the guide car at a safe distance. Should the sensors fail to do their jobs, there are bump switches in the front and back of the NaviGator to signal it upon detection of a contact; furthermore, to make it more robust, NaviGator also has an infrared sensor mounted on its rear to detect obstacles before they make contact. NaviGator's special sensor is a digital compass to determine the direction of its travel.

Introduction

Imagine sitting in your car during rush hour and you are stuck in traffic. You think to yourself how you have been driving bumper to bumper for the last half hour and your frustration grows as you listen to your preferred radio station traffic report. The brake pad is like a pump to your irritation. Every time you depress the brakes, your patience grows thinner.

Now imagine the same scenario, but this time with a Guide-Follow System (GFS) installed in your car. You no longer need to push the gas and brake every other minute because your car is equipped with an automated system that will control the car to follow the car in front of you that is doing the stop-n-go. You are free to relax while everyone else down the stretch is pounding his or her fists into the steering wheel.

Integrated System

NaviGator's integrated system mainly comprises of the Mekatronix ATmega128 microcontroller. It does all the processing for the sensors: two bump switches used to detect contact, one infrared sensor used to detect obstacles, two ultrasonic sensors used to measure the distance and alignment from the robot to the guide car, and the digital compass used to specify the orientation the robot is in. It also controls two DC motors for the robot's movements. The microcontroller itself uses a 9.6 V standard remote control car rechargeable battery. The motors require a separate supply as long as it has a common ground with the main battery of 7.2 V.

Each sensor is polled round-robin style in exception of the bump switches, which require immediate action upon involvement. The bump switches are implemented via interrupts that are constantly monitored in the background of the program execution. The other sensors are called upon, as they are needed. Figure 1 shows an illustrated high-level block diagram of the system.

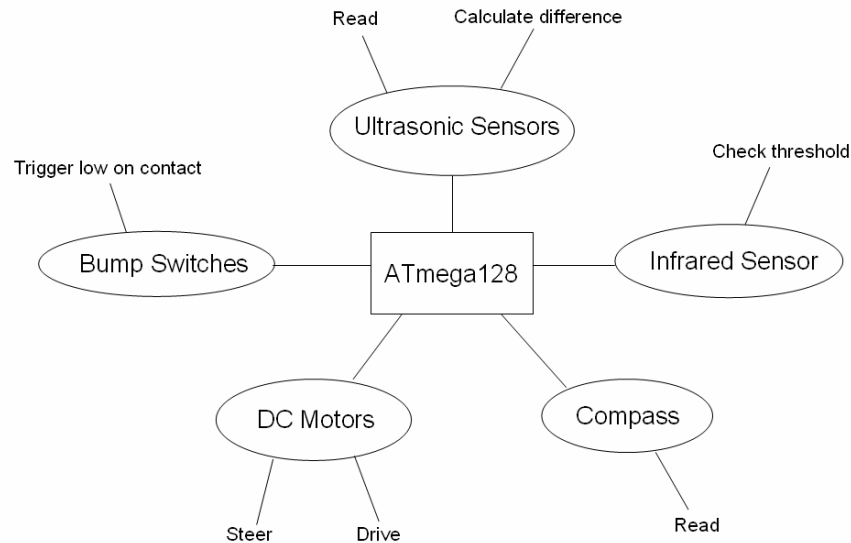


Figure 1. Block Diagram

NaviGator is designed to propose a low-cost solution to a common problem. In using four of the most basic sensors in the field of robotics, it is able to achieve its task in an effective manner.

Mobile Platform

NaviGator's body and locomotion come from the base of a hacked remote control car. In an attempt to reduce costs, an old remote control car was salvaged to become what is now known as the NaviGator. The idea is intuitive because it is the easiest way to design a

robot similar to a car. Figure 2 shows the beginning stages of hacking the remote control car.

However, the base of the remote control car is limited in space. I built on top of it a layer to mount parts on from the class-supplied balsa wood. The base layer holds the microcontroller, the motor controller, and the digital compass. Perpendicular to the layer and mounted in the rear is a piece of wood that holds the infrared sensor, the LCD, and the switches for power, reset, and download/run. The base layer is mounted on top of the remote control car base with four screws, and under it, lay a six-pack of batteries and a standard remote control car rechargeable NiCad battery.

The ultrasonic sensors are mounted on another wooden piece at the head of the robot. The bump switches are situated at bottom of the robot in both the front and rear. NaviGator's mobile platform is hidden under a shell of a different remote control car that was modified to fit.

I learned that using an existing platform from a remote control car should not be something to take lightly. The size of the car and the way it is shaped plays a huge role in the physical aspect of a robot. One needs to look at the amount of space available and the layout of the remote control before choosing it.

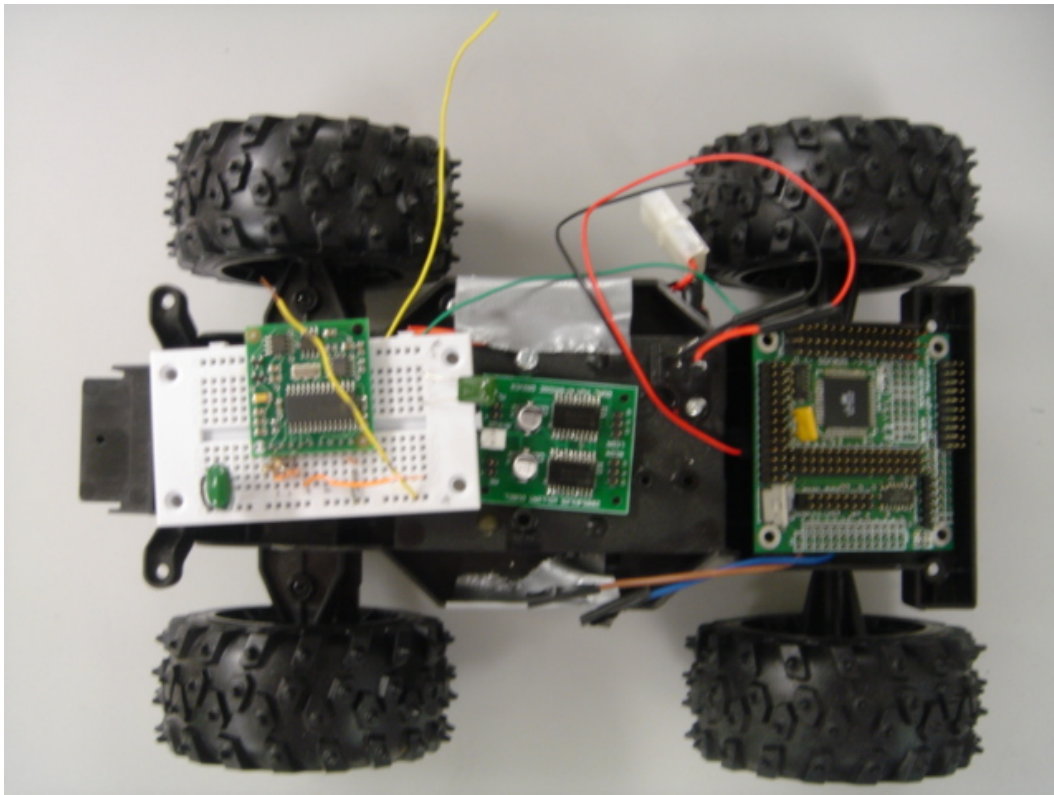


Figure 2. Layout of the remote control car with parts before platform was built

Actuation

As was mentioned, NaviGator uses the two DC motors of the remote control car. It uses the same single-axle steering analogous to a real automobile to motivate the possibility of a similar autopilot system being implemented in the real world.

The DC motors required a minimum of approximately ± 2.4 V for movement without load which can be handled by the 5 V regulator on the microcontroller; however, when under load and the initial spinning current spikes, the two motors can draw up to 2 A. To solve this problem, a separate battery pack was introduced, and I added an H-bridge motor controller.

The controller uses two PWM Intelligent Motor chips, TPIC0107B, was purchased pre-built from the TA, William Dubel, and is able to drive two DC motors. The controller simply needed a power source, and a separate direction and PWM input for each motor. The H-bridge design simply acts as a switch to provide positive and negative voltage given the direction and PWM inputs. This allowed the DC motors to be driven in both directions.

Given another opportunity, I would use servomotors in lieu of the DC motors, mainly because they can provide rotations that are more precise. In the programming portion, I had a lot of trouble turning the amount that I wanted from the motors. A big problem to the DC motor driven single-axle steering setup is that even when there is no power to the motor, it would not reposition itself to the beginning position, which is straight. I had to modify my code to compensate. For example, if the last action taken by the robot was driving to the left and I want the next action to be driving straight, the robot would not drive straight as I was unable to reset the position on the motor.

Sensors

Bump Sensors

The bump sensors are simply active low switches, which upon contact; the robot stops its motion and shuts down. Since in the real world, when a car makes contact with something, the first action for the driver is to stop. I chose to implement the bump switches to act as a backup safety feature should the other sensors fail to detect impending obstacles. I am using the Zippy Subminiature Microswitches SM-05, which can be seen in Figure 3. They can handle up to a current of 5 A and a voltage of 250 VAC. Two switches are used: one

in the front and one in the back. Upon contact, the bump switches short the input port to ground.



Figure 3. Zippy Microswitch

The code for implementing the bump switches can be seen in the Appendix. Initially, I programmed NaviGator to poll the switches at certain points in the program; however, the method proved to be inefficient. I modified the program to use interrupts instead and is a much better method for bump switches.

Sensors: Infrared

One Sharp GP2D12 infrared sensor is used at the rear of the robot to detect obstacles. An image of the sensor taken from the Acroname website is shown in Figure 4. The infrared sensor was chosen because it is easy to implement and is half the price of an ultrasonic sensor. It is very effective for close range detection under favorable environments. The idea behind its addition is to avoid utilizing the bump switches. When NaviGator becomes too close to the guide car, it will attempt to back up for a little bit, but it is not cleared to do so unless the infrared sensor allows it.



Figure 4. Infrared Sensor

The sensor emits infrared light and detects the reflection; thereby, making it an ideal low-cost sensor. Another advantage is that the color of the reflective object does not play a major factor in the performance. The sensor takes a continuous distance reading and reports the distance as an analog voltage with a distance range of 10cm (~4") to 80cm (~30"), which is then converted to a digital reading, as can be seen in the adc class of the program in the Appendix. The interface is 3-wire with power, ground and the output voltage. It operates on 4.5 to 5.5 V and draws a maximum current of 50 mA. The detecting distance is between 10 cm to 80 cm.

One disadvantage of the GP2D12 is that its output is not linear with the distance of the reflective object. This fact can be seen from Sharp's datasheet, which is shown in Figure 5. If I were using the sensors for a more variable and further distance detection, I would create a lookup table of output voltages to the actual distances instead of relying on a linear math equation. However, for my purposes, I only need to act upon a reading that is below a certain threshold (an obstacle is detected to be too close), so I did not need to create one.

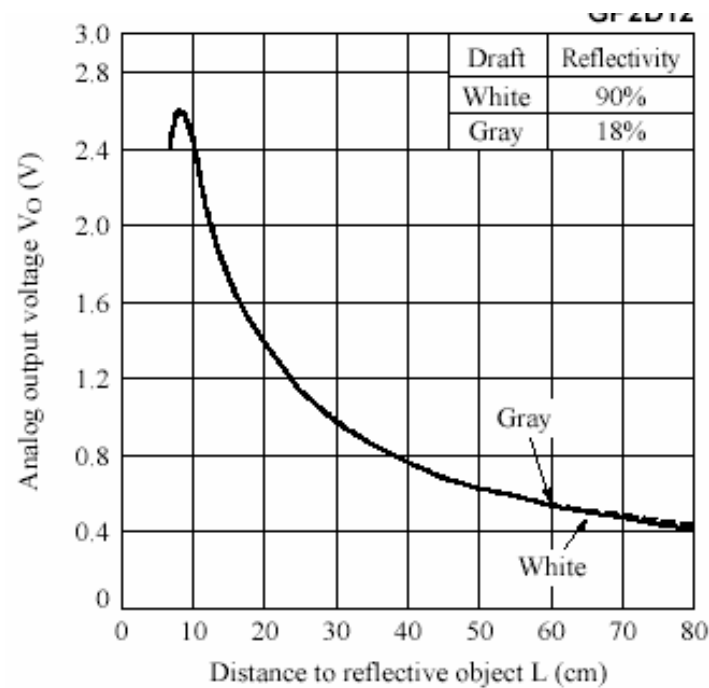


Figure 5. Analog Output Voltage vs. Distance to Reflective Object

Ultrasonic Sensors

NaviGator uses two Devantech SRF04 ultrasonic sensors. As mentioned above, the sensors are placed on the front sides of the robot. This arrangement allows the robot to measure its alignment with respect to the guide car. If the guide car turns left slightly, the right sonar will measure a distance greater than the distance measured by the left sonar. This information will activate the robot to turn left slightly until the distance measured by both sensors is similar. At the same time, the ultrasonic sensors maintain a certain distance between the robot and the guide car. To help reduce erroneous measurements, an average of three readings is calculated. An image of the sensor taken from the Acroname website can be seen in Figure 6.

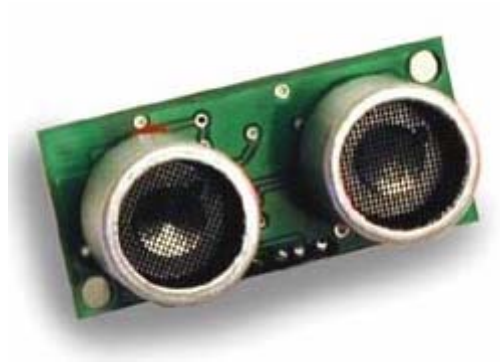


Figure 6. Ultrasonic Sensor

The ultrasonic sensors are able to measure distances between 3 cm to 3 m. It requires an operating voltage of 5 V and draws an average current of 30 mA. The operating frequency is 40 kHz. The pin connections are shown below in Figure 8. As you can see, the sensor has four connections: 5 V supply, echo output, trigger input, and ground.

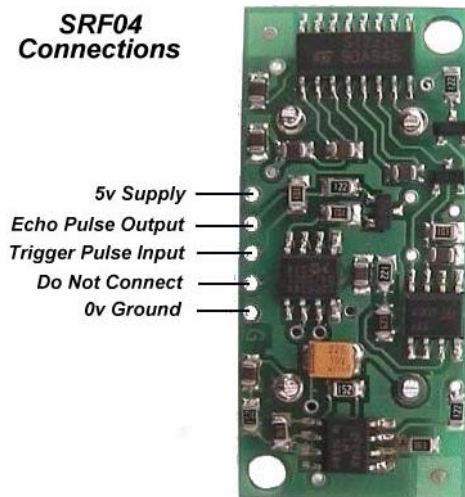


Figure 7. Ultrasonic Sensor Connections

The ultrasonic sensor was designed to require a trigger pulse input and provides an output echo pulse. The length of the echo is timed to find the distance between the sensor and an object. The timing diagram is shown in Figure 8.

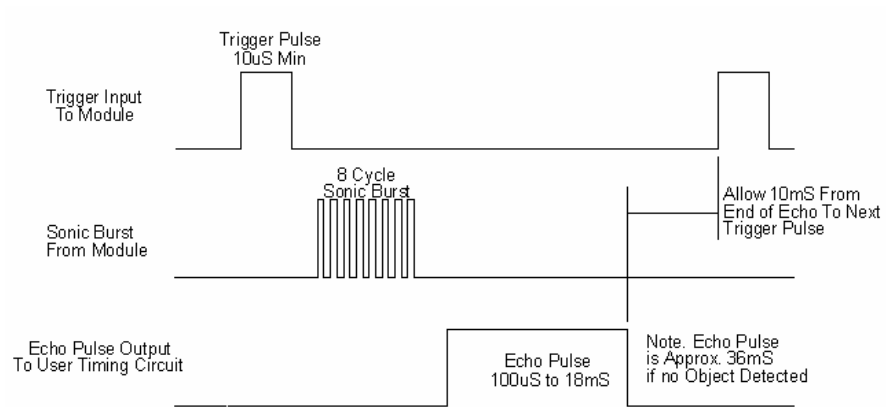


Figure 8. Ultrasonic Sensor Timing Diagram

The sensor requires a 10 μ S pulse to the trigger input to start the ranging. The SRF04 will in turn send an 8 cycle sonic burst at 40 kHz and raise its echo line high. It then waits for a responding echo, and as soon as it detects one, it lowers the echo line. If nothing is detected after 36 mS, the echo line is lowered automatically by the sensor. The echo line becomes a pulse with a width that is proportional to the distance to the object and the robot is programmed to time the pulse and translate the range into inches.

The sensor uses a transducer to emit the ultrasonic sound and listens for its reflection. The beam pattern of the transducer is shown in Figure 9, taken from the manufacturer's datasheet. As you can see, the beam spreads in all directions rather than traveling a straight line. It is because of this fact that two ultrasonic sensors cannot be fired at the same time since they can pick up each other's echo and report a false reading. Since I am using two SRF04's and they are relatively close to each other, I am firing them sequentially at 65 mS apart at a very slight angle pointed towards the center of the robot. The code for the ultrasonic sensors was taken from the BDMicro website and modified, and can be seen in the Appendix.

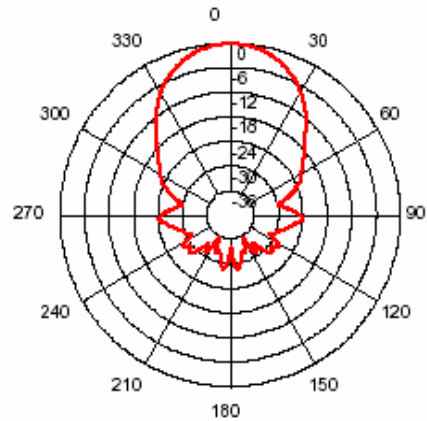


Figure 9. Ultrasonic Sensor Beam Pattern

Digital Compass

NaviGator's special sensor is the Devantech CMPS03 digital compass. It is able to detect Earth's magnetic field and is a great aid to navigation in robotics; however, NaviGator is only making readings and displaying them on the LCD. The idea is to simulate the automobiles that are equipped with digital compasses on the dashboards.

The compass requires 5 V at 15 mA. There are two methods in taking readings from its outputs. If you connect pull-up resistors to pins 2 and 3, which can be seen in Figure 10, you can measure the PWM signal from pin 4. The outputting pulse length varies from 1mS (0 degrees) to 36.99 mS (359.9 degrees).

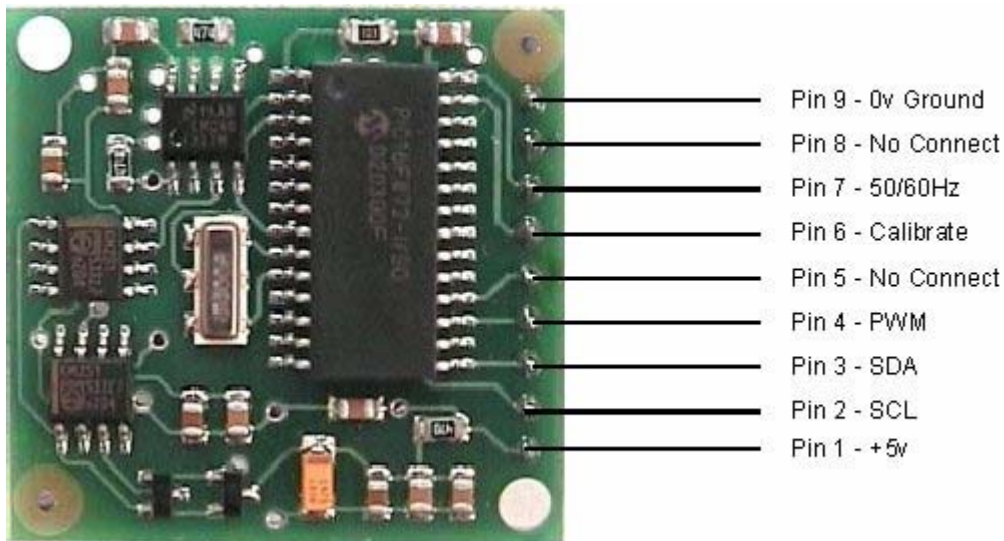


Figure 10. Digital Compass

I had opted for the second method, using the I2C bus (pins 2 and 3). This is a more direct way in using the compass, in that; you are able to obtain a direct readout of the angle in degrees from it. The section of code for the digital compass was taken from Robot-Electronics and can be seen in the Appendix.

Behaviors

Successful operation of NaviGator assumes that there is indeed a guide car directly in front of it on start. It is not designed for complete autopilot, i.e., it was not meant to follow guide cars completely at high speeds. The proposed project was to simulate a solution for traffic jams, so a successful run of NaviGator also requires that the operator of the guide car does not drive too fast.

The main programming portion of NaviGator lies in the procedure to follow the guide car as it is turning. NaviGator is able to follow accordingly given that the operator does not

speed off while turning. Below shows the pseudocode for maintaining alignment to the guide car while driving straight and/or turning.

```
while (LSonar and RSonar > 3)
    drive forward;
while (no contact){
    while (LSonar or RSonar ≤ 3)
        drive backward a little bit;
    while (|LSonar-RSonar| > 0)
        if (LSonar > RSonar)
            drive right;
        else
            drive left;
}
```

NaviGator only takes readings when it is stationary. I did not see it necessary for it to take measurements constantly, as the following is to be done at low speeds. Also, I did not want to take the readings of two ultrasonic sensors and the compass while the motors are on. I did not want the distance of travel to be dependent on other parts in that way.

Experimental Layout and Results

NaviGator is able to achieve its tasks while the above requirements are met. As long as the guide car does not speed off, NaviGator will be able to follow the remote control car. The project shows that a similar system can be implemented in the real world.

A big problem that arose was the DC motors not rotating consistently. The RPM of the motor is dependent on the amount of charge left in the battery so it was difficult to find appropriate PWM pulse lengths and the amount of time to keep the motors on. This will cause NaviGator to misjudge how much the motors should turn to maintain a distance.

Conclusion

Overall, the project was a rewarding experience. I was forced to cram everything towards the end of the semester because I had numerous problems with my microcontroller in the beginning. In an attempt to save a few dollars, I ended up paying more in both time and money. It took me several weeks to try to get the ISP dongle to download my programs only to forego the idea.

Nonetheless, I would very much like to see my idea implemented in the real world. The method of sensing to maintain distance and alignment may not need to be ultrasonic sensors or the like, as they may need cleaning constantly from dirt and gravel. A video camera could be used and be calibrated to follow the specific color of the car that is in front.

Documentation

Sources for Parts

Acroname: <http://www.acroname.com>

Sharp GP2D12 IR Sensor

Devantech SRF04 Ultrasonic Sensors

Devantech CMPS03 Digital Compass

eBay:

Mekatronix ATmega128 Microcontroller

Mr. Robot: <http://www.mrrobot.com>

MB2325 Board

Old Parts:

Hacked Remote Control Car

LCD

Radio Shack:

Various Parts

Sparkfun: <http://www.sparkfun.com>

ISP Dongle

Wal-Mart:

Standard Remote Control Car

9.6 V NiCad Battery

William Dubel:

H-Bridge Motor Driver

References

TA's: Steven Pickles, William Dubel

Avrfreaks:

<http://www.avrfreaks.net>

BDMicro:

<http://bdmicro.com/code/>

Peter Fleury:

<http://homepage.sunrise.ch/mysunrise/peterfleury/index.html>

Robot Electronics:

CMPS03 Robot Compass Module

<http://www.robot-electronics.co.uk/htm/cmeps3doc.shtml>

SRF04 Ultrasonic Ranger Technical Specification

<http://www.robot-electronics.co.uk/htm/srf04tech.htm>

Appendix – Source Code

```

/*****
* Title: main2.c
*
* The main program.
*****/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <avr/signal.h>
#include <compat/twi.h>
#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>

#include "lcd.h"           // LCD
#include "timer.h"        // delay
#include "sonar.h"        // sonars
#include "motor.h"        // motor
#include "cmeps03.h"      // compass
#include "i2c.h"          // compass
#include "adc.h"          // IR

// IR declaration
#define IRval ADC_sample(2);
uint8_t distIR;

void readIR(char * buffer){
    uint8_t temp1;
    uint8_t avgValue;

    lcd_clrscr();
    temp1 += IRval;
    temp1 += IRval;
    temp1 += IRval;
    avgValue = temp1 / 3;
    itoa(avgValue, buffer, 7);
}

/* compass declarations */
#define BAUD_RR ((CPU_FREQ/(16L*9600L) - 1))

#define CPU_FREQ 16000000L /* set to clock frequency in Hz */

#if CPU_FREQ == 16000000L
#define OCR_1MS 125
#elif CPU_FREQ == 14745600L

```

```

#define OCR_1MS 115
#endif

int def_putc(char ch)
{
    /* output character to UART0 */
    while ((UCSR0A & _BV(UDRE)) == 0)
        ;
    UDR0 = ch;
    return ch;
}

// bumper flag
static volatile uint8_t bumpFlag; // use volatile when variable is accessed from interrupts

SIGNAL(SIG_INTERRUPT5) // front bumper (PE5)
/* signal handler for external interrupt int5 */
{
    lcd_clrscr();
    lcd_puts("Contact in the front!");
    bumpFlag = 1;
    drive_stop();
}

SIGNAL(SIG_INTERRUPT6) // rear bumper (PE6)
/* signal handler for external interrupt int6 */
{
    lcd_clrscr();
    lcd_puts("Contact in the back!");
    bumpFlag = 1;
    drive_stop();
}

int main(void)
{
    // sonar variables
    char bufferLeftSonar[10];
    char bufferRightSonar[10];

    // compass variables
    int16_t bearing;
    double degree;
    char dir[10];

    // IR variables
    char bufferIR[7];

    // initializations
    lcd_init(LCD_DISP_ON);

```

```

lcd_clrscr();
lcd_puts("Initializing");
init_timer();
init_motor();
ADC_init();

// bumper declarations
DDRE = 0x00; // use all pins on port E for input
PORTE = 0xff; // activate internal pull-up
EIMSK = _BV(INT5) | _BV(INT6); // enable external INT 5,6
MCUCR = _BV(ISC51) | _BV(ISC61); // falling egde: INT 5,6
sei(); // enable interrupts

/* set the I2C bit rate generator to 100 kb/s */
TWSR &= ~0x03;
TWBR = 28;
TWCR |= _BV(TWEN);

ms_sleep(500);

//straightenWheel();

while(!bumpFlag)
{
    // display compass reading
    bearing = bearing16();
    if (bearing >= 0)
    {
        degree = ((double)bearing) / 10.0;
        itoa(degree, dir, 10);
    }
    else
        dir[0] = "NA";

    getAlignVal();
    while (!bumpFlag && ((leftSonar > 6) && (rightSonar > 6)) )
    {
        if (alignVal > 0)
            if (leftSonar > rightSonar)
            {
                turn_right();
                ms_sleep(80);
            }
            else
            {
                turn_left();
                ms_sleep(80);
            }
        }
    }
}

```

```

        drive_fwd();
        ms_sleep(130);
        drive_stop();
        getAlignVal();
        //getLeftSonar();
        //getRightSonar();
    }

    while (!bumpFlag && ((leftSonar > 3) && (rightSonar > 3)) )
    {
        if (alignVal > 0)
            if (leftSonar > rightSonar)
            {
                turn_right();
                ms_sleep(80);
            }
            else
            {
                turn_left();
                ms_sleep(80);
            }

        drive_fwd();
        ms_sleep(40);
        drive_stop();
        getAlignVal();
        //getLeftSonar();
        //getRightSonar();
    }

    readSonars(bufferLeftSonar, bufferRightSonar);
    lcd_clrscr();
    lcd_puts("Lt Sonar: ");
    lcd_puts(bufferLeftSonar);
    lcd_puts(" ");
    lcd_puts(dir);
    lcd_puts(" deg");
    lcd_command(0xC0);
    lcd_puts("Rt Sonar: ");
    lcd_puts(bufferRightSonar);

    // if too close
    while (((leftSonar < 3) || (rightSonar < 3)) && (bumpFlag != 1))
    {
        lcd_clrscr();
        lcd_puts("Too close!");
        ms_sleep(800);
        getAlignVal();
    }

```

```

//getLeftSonar();
//getRightSonar();

distIR = IRval;
/* convert integer into string */
//readIR(bufferIR);

// if still too close, back up, if no obstacles in back
if (((leftSonar < 3) || (rightSonar < 3)) && (!bumpFlag))
{
    if (distIR < 60)                // then no obstacle
    {
        if (alignVal > 0)
            if (leftSonar > rightSonar)
            {
                turn_left();
                ms_sleep(40);
            }
            else
            {
                turn_right();
                ms_sleep(40);
            }

        bearing = bearing16();
        if (bearing >= 0)
        {
            degree = ((double)bearing) / 10.0;
            itoa(degree, dir, 10);
        }
        else
            dir[0] = "NA";

        readSonars(bufferLeftSonar, bufferRightSonar);
        lcd_clrscr();
        lcd_puts("Lt Sonar: ");
        lcd_puts(bufferLeftSonar);
        lcd_puts(" ");
        lcd_puts(dir);
        lcd_puts(" deg");
        lcd_command(0xC0);
        lcd_puts("Rt Sonar: ");
        lcd_puts(bufferRightSonar);
        drive_bwd();
        ms_sleep(30);
        drive_stop();
        getLeftSonar();
        getRightSonar();
    }
}

```

```

        else
        {
            lcd_clrscr();
            lcd_puts("Obstacle in the back!");
            lcd_command(0xC0);
            lcd_puts("Cannot back up.");
            ms_sleep(1000);
        }
    }
}

// if at the 3 inch distance
getAlignVal();
if (alignVal > 0)
    if (leftSonar > rightSonar)
    {
        turn_right();
        ms_sleep(40);
    }
    else
    {
        turn_left();
        ms_sleep(40);
    }

bearing = bearing16();
if (bearing >= 0)
{
    degree = ((double)bearing) / 10.0;
    itoa(degree, dir, 10);
}
else
    dir[0] = "NA";

readSonars(bufferLeftSonar, bufferRightSonar);
lcd_clrscr();
lcd_puts("Lt Sonar: ");
lcd_puts(bufferLeftSonar);
lcd_puts(" ");
lcd_puts(dir);
lcd_puts(" deg");
lcd_command(0xC0);
lcd_puts("Rt Sonar: ");
lcd_puts(bufferRightSonar);

if (bumpFlag == 1)
    break;
}
return 0;

```

```

}

/*****
* Title: adc.h
*
* Code to use the Analogue to Digital Converter.
*****/

#include <avr/io.h>
#include <stdio.h>

void ADC_init(void);
void ADC_selectChannel(uint8_t channel);
uint8_t ADC_read(void);
uint8_t ADC_sample(uint8_t channel); //channel: 0-7
uint8_t ADC_sampleN(uint8_t channel, uint8_t n);

/*****
* Title: adc.c
*
* Code to use the Analogue to Digital Converter.
*****/

#include "adc.h"

/***** Initialization: *****/
* set Port F (ADC PORT) as inputs
* set ADCSR = '1000 0000' or '0x80'
* (enable ADC, single conv. mode, no interrupt, no prescalar)
*****/

void ADC_init(void)
{
//configure ADC port (PORTF) as input
DDRF = 0x00;

//outp(0x86, ADCSR);
ADCSR = 0x80;
}

uint8_t ADC_sample(uint8_t channel)
{
ADC_selectChannel(channel);
return ADC_read();
}

uint8_t ADC_read(void)
{
uint8_t value = 0;

```

```

//sbi(ADCSR,ADSC); //start conversion
ADCSR |= _BV(ADSC);
loop_until_bit_is_set(ADCSR,ADIF); //wait till conversion completes
value = ADCH; //read value
//sbi(ADCSR,ADIF); //reset ADCI flag
return value;
}

void ADC_selectChannel(uint8_t channel)
{
/* select channel */
ADMUX = 0x60; //AVCC with external capacitor at AREF pin, left adjusted
ADMUX |= channel;
}

/*****
* Title: cmps03.h
*
* Code to use the Devantech CMPS03 compass
* taken from Robot-Electronics
*****/

/* $Id: cmps03.h,v 1.2 2003/11/15 22:46:20 bsd Exp $ */

#ifndef __cmpos3_h__
#define __cmpos3_h__

#include <inttypes.h>

int8_t cmpos3_get_byte(uint8_t device, uint8_t addr, uint8_t * value);

int16_t cmpos3_get_word(uint8_t device, uint8_t addr, uint16_t * value);

int16_t bearing16(void);

uint8_t bearing8(void);

void getDir(double pulseLength, char * dir);

#endif

/*****
* Title: cmpos3.c
*
* Code to use the Devantech CMPS03 compass,
* taken from Robot-Electronics
*****/

/* $Id: cmpos3.c,v 1.2 2003/11/15 22:46:20 bsd Exp $ */

```

```

#include <compat/twi.h>

#include <inttypes.h>

#include "cmps03.h"
#include "i2c.h"
#include <string.h>

/*
 * read the data byte at the specified address from the specified device
 */
int8_t cmps03_get_byte(uint8_t device, uint8_t addr, uint8_t * value)
{
    uint8_t v;

    /* start condition */
    if (i2c_start(0x08, 1))
        return -1;

    /* address slave device, write */
    if (i2c_sla_rw(device, 0, TW_MT_SLA_ACK, 1))
        return -2;

    /* write register address */
    if (i2c_data_tx(addr, TW_MT_DATA_ACK, 1))
        return -3;

    /* repeated start condition */
    if (i2c_start(0x10, 1))
        return -4;

    /* address slave device, read */
    if (i2c_sla_rw(device, 1, TW_MR_SLA_ACK, 1))
        return -5;

    /* read data byte */
    if (i2c_data_rx(&v, I2C_NACK, TW_MR_DATA_NACK, 1))
        return -6;

    if (i2c_stop())
        return -7;

    *value = v;

    return 0;
}

```

```

/*
 * read the data byte at the specified address from the specified device
 */
int8_t cmsps03_get_word(uint8_t device, uint8_t addr, uint16_t * value)
{
    uint8_t v1, v2;

    /* start condition */
    if (i2c_start(0x08, 1))
        return -1;

    /* address slave device, write */
    if (i2c_sla_rw(device, 0, TW_MT_SLA_ACK, 1))
        return -2;

    /* write register address */
    if (i2c_data_tx(addr, TW_MT_DATA_ACK, 1))
        return -3;

    /* repeated start condition */
    if (i2c_start(0x10, 1))
        return -4;

    /* address slave device, read */
    if (i2c_sla_rw(device, 1, TW_MR_SLA_ACK, 1))
        return -5;

    /* read data byte 1 */
    if (i2c_data_rx(&v1, I2C_ACK, TW_MR_DATA_ACK, 1))
        return -6;

    /* read data byte 2 */
    if (i2c_data_rx(&v2, I2C_NACK, TW_MR_DATA_NACK, 1))
        return -7;

    if (i2c_stop())
        return -8;

    *value = (v1 << 8) | v2 ;

    return 0;
}

int16_t bearing16(void)
{
    uint16_t d;

```

```

int8_t rc;

rc = cmpls03_get_word(0x60, 2, &d);
if (rc < 0) {
    return -1;
}

return d;
}

uint8_t bearing8(void)
{
    uint8_t d;

    cmpls03_get_byte(0x60, 1, &d);

    return d;
}

void getDir(double pulseLength, char * dir)
{
    if (pulseLength < 23)
        dir = "N";
    else if (pulseLength < 68)
        dir = "NE";
    else if (pulseLength < 113)
        dir = "E";
    else if (pulseLength < 158)
        dir = "SE";
    else if (pulseLength < 203)
        dir = "S";
    else if (pulseLength < 248)
        dir = "SW";
    else if (pulseLength < 293)
        dir = "W";
    else if (pulseLength < 338)
        dir = "NW";
    else
        dir = "N";
}
/*
int getDir(double pulseLength)
{
    if (pulseLength < 23)
        return 1;
    else if (pulseLength < 68)
        return 2;
    else if (pulseLength < 113)
        return 3;
}

```

```

        else if (pulseLength < 158)
            return 4;
        else if (pulseLength < 203)
            return 5;
        else if (pulseLength < 248)
            return 6;
        else if (pulseLength < 293)
            return 7;
        else if (pulseLength < 338)
            return 8;
        else
            return 1;
    }
}
*/

/*****
* Title: i2c.h
*
* Code to use the I2C.
*****/

#ifndef __i2c_h__
#define __i2c_h__

#include <inttypes.h>

/*
* define some handy I2C constants
*/
#define I2C_START    (_BV(TWINT)|_BV(TWSTA)|_BV(TWEN))
#define I2C_MASTER_TX (_BV(TWINT)|_BV(TWEN))
#define I2C_TIMEOUT  1000
#define I2C_ACK      1
#define I2C_NACK     0

int8_t i2c_stop(void);

void i2c_error(const char * message, uint8_t cr, uint8_t status);

int8_t i2c_start(uint8_t expected_status, uint8_t verbose);

int8_t i2c_sla_rw(uint8_t device, uint8_t op, uint8_t expected_status,
                 uint8_t verbose);

int8_t i2c_data_tx(uint8_t data, uint8_t expected_status, uint8_t verbose);

int8_t i2c_data_rx(uint8_t * data, uint8_t ack, uint8_t expected_status,
                  uint8_t verbose);

```

```

#endif

/*****
* Title: i2c.c
*
* Code to use the I2C.
*****/

#include <compat/twi.h>
#include <avr/pgmspace.h>
#include <stdio.h>
#include <inttypes.h>
#include "lcd.h"
#include "i2c.h"

const char s_i2c_start_error[] PROGMEM = "I2C START CONDITION ERROR";
const char s_i2c_sla_w_error[] PROGMEM = "I2C SLAVE ADDRESS ERROR";
const char s_i2c_data_tx_error[] PROGMEM = "I2C DATA TX ERROR";
const char s_i2c_data_rx_error[] PROGMEM = "I2C DATA RX ERROR";
const char s_i2c_timeout[] PROGMEM = "I2C TIMEOUT";
const char s_i2c_error[] PROGMEM = "I2C ERROR\n";
const char s_fmt_i2c_error[] PROGMEM = " TWCR=%02x STATUS=%02x\n";

extern volatile uint16_t ms_count;

/*
 * signal the end of an I2C bus transfer
 */
int8_t i2c_stop(void)
{
    TWCR = _BV(TWINT)|_BV(TWEN)|_BV(TWSTO);
    while (TWCR & _BV(TWSTO))
        ;
    return 0;
}

/*
 * display the I2C status and error message and release the I2C bus
 */
void i2c_error(const char * message, uint8_t cr, uint8_t status)
{
    i2c_stop();
    //lcd_puts(message);
    //lcd_puts(s_fmt_i2c_error, cr, status);
}

```

```

/*
 * signal an I2C start condition in preparation for an I2C bus
 * transfer sequence (polled)
 */
int8_t i2c_start(uint8_t expected_status, uint8_t verbose)
{
    uint8_t status;

    ms_count = 0;

    /* send start condition to take control of the bus */
    TWCR = I2C_START;
    while (!(TWCR & _BV(TWINT)) && (ms_count < I2C_TIMEOUT))
        ;

    if (ms_count >= I2C_TIMEOUT) {
        if (verbose) {
            i2c_error(s_i2c_start_error, TWCR, TWSR);
            i2c_error(s_i2c_timeout, TWCR, TWSR);
        }
        return -1;
    }

    /* verify start condition */
    status = TWSR;
    if (status != expected_status) {
        if (verbose) {
            i2c_error(s_i2c_start_error, TWCR, status);
        }
        return -1;
    }

    return 0;
}

/*
 * initiate a slave read or write I2C operation (polled)
 */
int8_t i2c_sla_rw(uint8_t device, uint8_t op, uint8_t expected_status,
                 uint8_t verbose)
{
    uint8_t sla_w;
    uint8_t status;

    ms_count = 0;

    /* slave address + read/write operation */
    sla_w = (device << 1) | op;

```

```

TWDR = sla_w;
TWCR = I2C_MASTER_TX;
while (!(TWCR & _BV(TWINT)) && (ms_count < I2C_TIMEOUT))
;

if (ms_count >= I2C_TIMEOUT) {
    if (verbose) {
        i2c_error(s_i2c_sla_w_error, TWCR, TWSR);
        i2c_error(s_i2c_timeout, TWCR, TWSR);
    }
    return -1;
}

status = TWSR;
if ((status & 0xf8) != expected_status) {
    if (verbose) {
        i2c_error(s_i2c_sla_w_error, TWCR, status);
    }
    return -1;
}

return 0;
}

/*
 * transmit a data byte onto the I2C bus (polled)
 */
int8_t i2c_data_tx(uint8_t data, uint8_t expected_status, uint8_t verbose)
{
    uint8_t status;

    ms_count = 0;

    /* send data byte */
    TWDR = data;
    TWCR = I2C_MASTER_TX;
    while (!(TWCR & _BV(TWINT)) && (ms_count < I2C_TIMEOUT))
;

    if (ms_count >= I2C_TIMEOUT) {
        if (verbose) {
            i2c_error(s_i2c_data_tx_error, TWCR, TWSR);
            i2c_error(s_i2c_timeout, TWCR, TWSR);
        }
        return -1;
    }

    status = TWSR;

```

```

if((status & 0xf8) != expected_status) {
    if(verbose) {
        i2c_error(s_i2c_data_tx_error, TWCR, status);
    }
    return -1;
}

return 0;
}

/*
 * receive a data byte from the I2C bus (polled)
 */
int8_t i2c_data_rx(uint8_t * data, uint8_t ack, uint8_t expected_status,
                  uint8_t verbose)
{
    uint8_t status;
    uint8_t b;

    ms_count = 0;

    if(ack) {
        TWCR = _BV(TWINT)|_BV(TWEN)|_BV(TWEA);
    }
    else {
        TWCR = _BV(TWINT)|_BV(TWEN);
    }
    while (!(TWCR & _BV(TWINT)) && (ms_count < I2C_TIMEOUT))
        ;

    if(ms_count >= I2C_TIMEOUT) {
        if(verbose) {
            i2c_error(s_i2c_data_rx_error, TWCR, TWSR);
            i2c_error(s_i2c_timeout, TWCR, TWSR);
        }
        return -1;
    }

    status = TWSR;
    if((status & 0xf8) != expected_status) {
        if(verbose) {
            i2c_error(s_i2c_data_rx_error, TWCR, status);
        }
        return -1;
    }

    b = TWDR;

```

```

*data = b;

return 0;
}

#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
* Modified by: Jonathan Mau
* File    : $Id: lcd.h,v 1.12.2.4 2005/02/28 22:54:41 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 8000000      /**< clock frequency in Hz, used to calculate delay timer
 */

/**
 * @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 24    /**< visibles characters per line of the display */
#define LCD_LINE_LENGTH 0x40  /**< internal line length of the display */
#define LCD_START_LINE1 0x80  /**< DDRAM address of first char of line 1 */
#define LCD_START_LINE2 0xC0  /**< 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 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 0          /**< pin for 4bit data bit 0 */
#define LCD_DATA1_PIN 1          /**< pin for 4bit data bit 1 */
#define LCD_DATA2_PIN 2          /**< pin for 4bit data bit 2 */

```

```

#define LCD_DATA3_PIN 3      /**< pin for 4bit data bit 3 */
#define LCD_RS_PORT LCD_PORT /**< port for RS line */
#define LCD_RS_PIN 4       /**< 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 6        /**< 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 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))

/*@}*/

void LCD_delay(uint16_t delTime);
#endif //LCD_H

/*****
****
Title : HD44780U LCD library
Author: Peter Fleury <pfleury@gmx.ch> http://jump.to/fleury
Modified by: Jonathan Mau
File: $Id: lcd.c,v 1.13.2.5 2005/02/16 19:15:13 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
#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

/*
** 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);
    }
}
#endif

```

```

    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
*****/
/
#ifdef 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 ( 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
    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
    {
#ifdef LCD_WRAP_LINES==1
#ifdef 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)
{
#ifdef 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
    lcd_command(LCD_FUNCTION_DEFAULT); /* function set: display lines */
    lcd_command(LCD_DISP_OFF);        /* display off */
    lcd_clrscr();                      /* display clear */
    lcd_command(LCD_MODE_DEFAULT);    /* set entry mode */
    lcd_command(disAttr);              /* display/cursor control */

} /* lcd_init */

void LCD_delay(uint16_t delTime)
{
    uint16_t temp;
    for(temp = 0; temp < delTime; temp++);
}

/*****
 * Title: motor.h
 *
 * Code to use the DC motors.
 *****/

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

#define frontPWM_PIN PB5           // frontPWM pin (PB5)
#define frontPWM_DDR DDRB         // frontPWM DDR (DDR B)
#define rearPWM_PIN PB7           // rearPWM pin (PB7)
#define rearPWM_DDR DDRB          // rearPWM DDR (DDR B)

#define frontMotor OCR1AL
#define rearMotor OCR1CL

```

```

void init_motor(void);
void drive_fwd(void);
void drive_bwd(void);
void drive_stop(void);
void drive_left(void);
void drive_right(void);
void straightenWheel(void);
void turn_left(void);
void turn_right(void);

/*****
* Title: motor.c
*
* Code to use the DC motors.
*****/

#include <inttypes.h>
#include <avr/io.h>
#include "motor.h"

// OCR1AL = output compare = front motor -> PB5
// OCR1CL = output compare = rear motor-> PB7

// TCCR1A = timer/counter register 1
// TCCR3A = timer/counter register 3

void init_motor(void)
{
    // set PWM pin as output, required for output toggling
    frontPWM_DDR |= _BV(frontPWM_PIN);           //(PB5)
    rearPWM_DDR |= _BV(rearPWM_PIN);            //(PB7)

    // enable 8 bit PWM, select inverted PWM
    TCCR1A = _BV(WGM10) | _BV(COM1A1) | _BV(COM1A0);

    // timer1 running on 1/8 MCU clock with clear timer/counter1 on Compare Match
    // PWM frequency will be MCU clock / 8 / 512, e.g. with 4Mhz Crystal 977 Hz.
    TCCR1B = _BV(CS11) | _BV(WGM12);
    drive_stop();
}

// OCR1AL = 150 -> no pulse
// OCR3AL = 255 -> no pulse
// OCRnAL = 150 -> drive rear motor
// OCRnAL = 30 -> turn front motor

void drive_fwd(void)
{

```

```
    DDRB = 0xC0;
    PORTB = 0xC0;
        frontMotor = 255;
        rearMotor= 250;
}

void drive_bwd(void)
{
    DDRB = 0x80;
    PORTB = 0x80;
        frontMotor = 255;
        rearMotor = 250;
}

void drive_stop(void)
{
    //uint8_t temp = 0x0F;
    DDRB = 0x0;
    PORTB = 0x0;
}

// PE 3 -> direction pin for front motor
// PB 6 -> direction pin for rear motor

void drive_left(void)
{
    DDRB = 0xF0;
    PORTB = 0xF0;
        rearMotor = 150;
        frontMotor = 30;
}

void drive_right(void)
{
    DDRB = 0xE0;
    PORTB = 0xE0;
        rearMotor = 150;
        frontMotor = 30;
}

void turn_left(void)
{
    DDRB = 0X70;
    PORTB = 0X70;
        frontMotor = 30;
}

void turn_right(void)
{
```

```

        DDRB = 0X60;
        PORTB = 0X60;
        frontMotor = 30;
    }

void straightenWheel(void)
{
    DDRB = 0X70;
    PORTB = 0X70;
    frontMotor = 30;
    ms_sleep(200);

    DDRB = 0X60;
    PORTB = 0X60;
    frontMotor = 30;
    ms_sleep(200);

    DDRB = 0X70;
    PORTB = 0X70;
    frontMotor = 30;
    ms_sleep(100);
}

/*****
* Title: i2c.c
*
* Code to use the sonars.
*****/

#include <avr/io.h>
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>

#include "timer.h"

int leftSonar;
int rightSonar;
int alignVal;

int sonarDist(int sonar, int times);
void getLeftSonar(void);
void getRightSonar(void);
void getAlignVal(void);
void readSonars(char * bufferLeftSonar, char * bufferRightSonar);

/*****
* Title: sonar.c
*

```

* Sonic range finder function obtained from BDMicro

*****/

```
#include "sonar.h"
```

```
// SRF04 Ultrasonic =====
```

```
// Port Pin          0    1    2    3    4    5    6    7
```

```
// LCD Pin                                OUT TRG OUT TRIG
```

```
//                                --#1--- --#2--
```

```
#define SRF_PORT  PORTD
```

```
#define SRF_DDRX  DDRD
```

```
#define SRF_PINX  PIND
```

```
//
```

```
=====
=====
//                                SRF04                                Functions
=====
=====
```

```
//
```

```
=====
=====
```

```
// Function to get distance data from an SRF04, first
```

```
// argument specifies which one. ( 1 = left, 2 = right )
```

```
int sonarDist(int sensor_num, int times)
```

```
{
```

```
    int i,counter = 0;
```

```
    for (i=0; i<times ;i++)
```

```
    {
```

```
        // Trigger a ping on coresponding SRF04 and wait a few usec
```

```
        if(sensor_num == 1){
```

```
            SRF_DDRX = 0x20;           // set output pin  (0x02)
```

```
            SRF_PORT = 0x20;           // trigger
```

```
        (0x02)
```

```
            us_sleep(10);               // length of 10 uS
```

```
            SRF_PORT = 0x00;           // end trigger
```

```
        }else {
```

```
            SRF_DDRX = 0x80;
```

```
            SRF_PORT = 0x80;
```

```
            us_sleep(10);
```

```
            SRF_PORT = 0x00;}
```

```
        // wait a few usec for echo circuit initialization
```

```
        us_sleep(3);
```

```
    while(1)
```

```

        {
            if(sensor_num == 1){
                if(SRF_PINX & 0x10)                // check if echo has
                been received
                    {
                        counter++;                // increment counter
                        us_sleep(1);            // wait a few usec
                    }
                else break;                // break if echo was received
            }
            else
            {
                if(SRF_PINX & 0x40)
                {
                    counter++;
                    us_sleep(1);
                }
                else break;
            }
        }
        ms_sleep(50); // wait 50msec between pulses for echo to settle
    }
    return (counter/times);                // return average (calibrated for inches)
}

void getLeftSonar(void)
{
    leftSonar = sonarDist(1,3);
}

void getRightSonar(void)
{
    rightSonar = sonarDist(2,3);
}

void readSonars(char * bufferLeftSonar, char * bufferRightSonar)
{
    getLeftSonar();
    getRightSonar();

    /* convert integer into string and display on LCD*/
    itoa(leftSonar, bufferLeftSonar, 10);
    itoa(rightSonar, bufferRightSonar, 10);
}

void getAlignVal(void)
{
    getLeftSonar();
    getRightSonar();
}

```

```

        alignVal = abs(leftSonar - rightSonar);
    }

/*****
* Title: timer.h
*
* timer functions obtained from BDMicro
*****/

#include <avr/interrupt.h>
#include <avr/signal.h>

// Global vars
volatile uint16_t ms_count;
volatile uint16_t us_count;

void ms_sleep(uint16_t ms);
void us_sleep(uint16_t us);
void init_timer(void);

/*****
* Title: timer.c
*
* timer functions obtained from BDMicro
*****/

#include "timer.h"

// Delay for a specified number of milliseconds
// Note: will not work for more than about 3100ms because of integer overflow!
// 65536/21 = 3120ms max
void ms_sleep(uint16_t ms)
{
    TCNT0 = 0;
    ms_count = 0;
    while (ms_count != ms*21);
}

// Delay for specified number of microseconds * 48
void us_sleep(uint16_t us)
{
    TCNT0 = 0;
    us_count = 0;
    while (us_count != us);
}

// millisecond counter interrupt vector
SIGNAL(SIG_OUTPUT_COMPARE0)
{

```

```
ms_count++;
us_count++;
}

// initialize timer 0 to generate an interrupt every 48usec
void init_timer(void)
{
    TIFR |= _BV(OCIE0);
    TCCR0 = _BV(WGM01)|_BV(CS02)|_BV(CS00);    // CTC, prescale = 128
    TCNT0 = 0;
    TIMSK |= _BV(OCIE0);    // enable output compare interrupt
    OCR0 = 6;                // match in 48usec
}
```