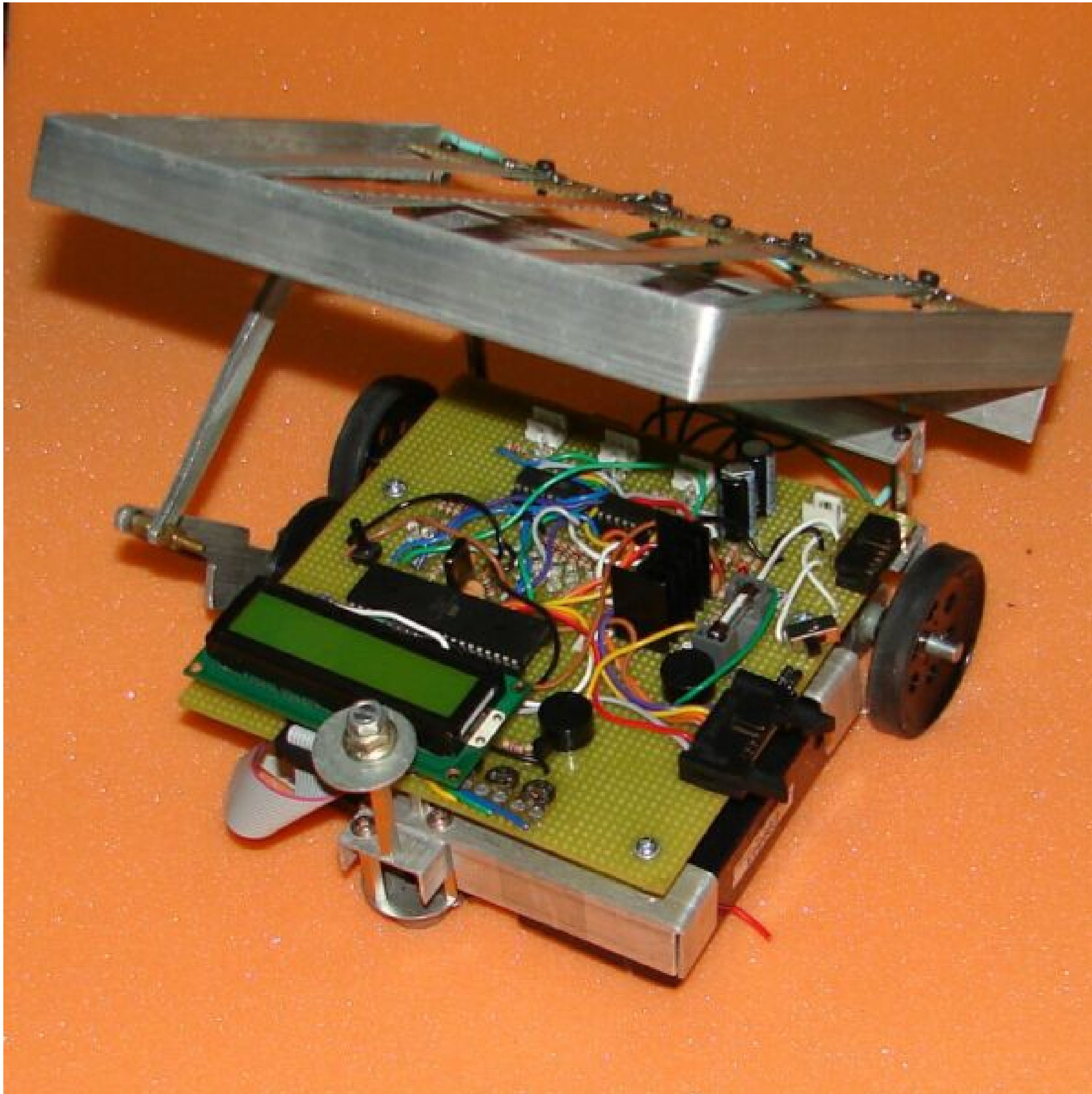
Building Low Cost Autonomous Robots

Chapter 4: Final Coding



Sachitanand Malewar
Director, NEX Robotics
http://www.nex-robotics.com/sachitanand.php

CHAPTER 4

FINAL CODING

In this chapter we are going to write important functions for the robot like LCD display, velocity and direction control, acquiring analog data etc.

In the first step we are going to do LCD interfacing. If you don't want to use LCD then you can skip this step and go to I/O interfacing section.

LCD interfacing:

At first it looks not so important but it will be very useful when you are going to calibrate your line sensors. It will be very useful when you deploy your robot in the contest arena and your sensors can display amount of ambient light interfering with your sensors. It can also show on what node your robot is and where it is heading. If you display node number (column and row number of the white line) you can very easily identify problem if robot is missing the nodes.

At first LCD interfacing seems bit complicated but it is very easy if you use standardized code. You don't have to reinvent the wheel. You can use LCD display code from our projects as open source content.

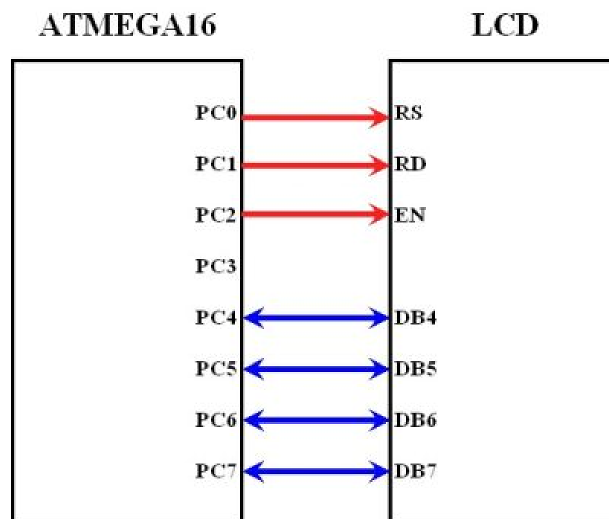


Figure 4.1 LCD interfacing

Interfacing the microcontroller with a LCD requires three control signals and 8 data lines in normal mode. In this mode we will require 11 bits. Since we have very limited I/O pins we are going to use three control lines and 4 data bits. Above figure shows basic LCD interfacing. The three control lines are referred to as EN, RS, and RW.

1. The EN line is called "Enable" and it is connected to PORTC's 2nd pin. This control line is used to tell the LCD that you have sent data to it or are ready to receive data from it. This is indicated by a high-to-low transition on this line. To send data to the LCD, your program should make sure that this line is low (0) and then set the other two control lines as required and put data on the data bus. When this is done, make EN high (1) and wait for the minimum amount of time as specified by the LCD datasheet, and end by bringing it to low (0) again.

2. The RS line is the "Register Select" line and it is connected to PORTC's 0th pin. When RS is low (0), the data is treated as a command or special instruction by the LCD (such as clear screen, position cursor, etc.). When RS is high (1), the data being sent is treated as text data which should be displayed on the screen.

3. The RW or RD line is the "Read/Write" control line and it is connected to PORTC's 1st pin. 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 querying (or reading from) the LCD.

The data bus is bidirectional, 8 bit wide. We are going to use its upper 4 bits only. So we have to load byte nibble by nibble.

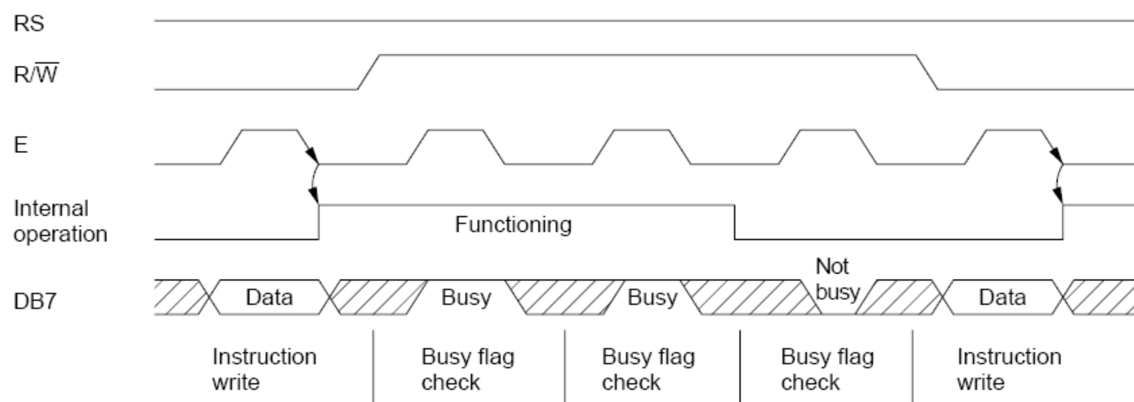


Figure 4.2: LCD Interface timing diagram

Before we can display any data on the LCD we need to initialize the LCD for proper operation. The first instruction we send must tell the LCD whether we'll be communicating with it using an 8-bit or 4-bit data bus. We also opt to use a 5x8 dot character font (as opposed to a 5x10 dot font). These two options are selected by sending the 38H to the LCD as a command. Remember that the RS line must be low if we are sending a command to the LCD. In the second and third instruction we reset and clear the display of the LCD. The fourth instruction sets the cursor to move in incremental direction. In fifth and sixth instruction we turn ON the display and place the cursor at the start. Check the Init_LCD( ) function to see how all this is put in code.

Step 1 project gives you the first LCD interfaced code. Using this code you can print ASCII character in LCD. I would like to give special thanks to Rohit Chauhan and Harsh Shrivastav for their LCD code

In this code following variables were defined in the main function

```
unsigned char
lcd_data1[16]={0x20,'N','E','X',0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20
,0x20,0x20,0x20,0x20};
unsigned char
lcd_data2[16]={0x20,0x20,0x20,'R','O','B','O','T','I','C','S',0x20,0x20
,0x20,0x20,0X20};
```

Will result following result on the text.

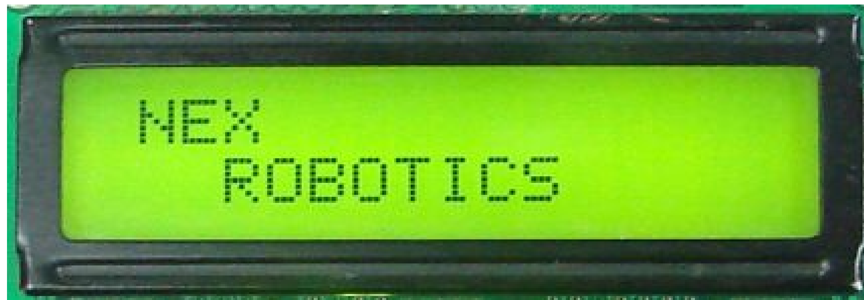


Figure: 4.3 LCD Display

I/O interfacing:

Controlling the buzzer

In this section first we are going to interface buzzer. Buzzer is primarily used for debugging of indicating some event like block is place on top of battery is low etc.

Buzzer is shearing PORTC of the microcontroller with the LCD. So we have to activate it in such a way that it will not effect LCD functions.

To make code simple and modular we are going to write simple functions to turn buzzer on and off.

This code is available in Step_2_turn_buzzer_on_off project

Buzzer_toggel_delay is a simple delay function. You can change its delay value by changing initial values of the a,b and c inside the each for loops.
If you reduce value of a,b or c it will give you larger delay.

Functions:

```
void buzzer_on (void)
{
 unsigned char portc_restore = 0; //used to buffer port C values
 portc_restore = PINC; //reading values of the PORTC register
 portc_restore = portc_restore | 0x08; //setting PC3 bit logic 1 to
turn on the buzzer
 PORTC = portc_restore; //pushing value back to the port
}


void buzzer_off (void)
{
 unsigned char portc_restore = 0; //used to buffer port C values
 portc_restore = PINC; //reading values of the PORTC register
 portc_restore = portc_restore & 0xF7; //resetting PC3 bit logic 0 to
turn off the buzzer
 PORTC = portc_restore; //pushing value back to the port
}



void buzzer_toggel_delay (void)
{
 unsigned char a,b,c = 0;
 for(a=1; a; a++)
  for(b=1; b; b++)
   for(c=245; c; c++);
}
```

Now we are going to toggle the buzzer. Write the following code in the main loop.

```
void main(void)
{
 //insert your functional code here...
 while(1)
 {
  buzzer_on();
  buzzer_toggel_delay();
  buzzer_off();
  buzzer_toggel_delay();
 }
}
```

You can turn buzzer on and off by calling buzzer_on and buzzer_off functions alternately sandwiching them between delay function and running them in while loop.

Moving your robot forward, backward, left and right.

We have studied functionalities of the L293D motor control IC in the previous chapters. Now using this IC we are going to control our motors.

This code is available in Step_3_robot motion project

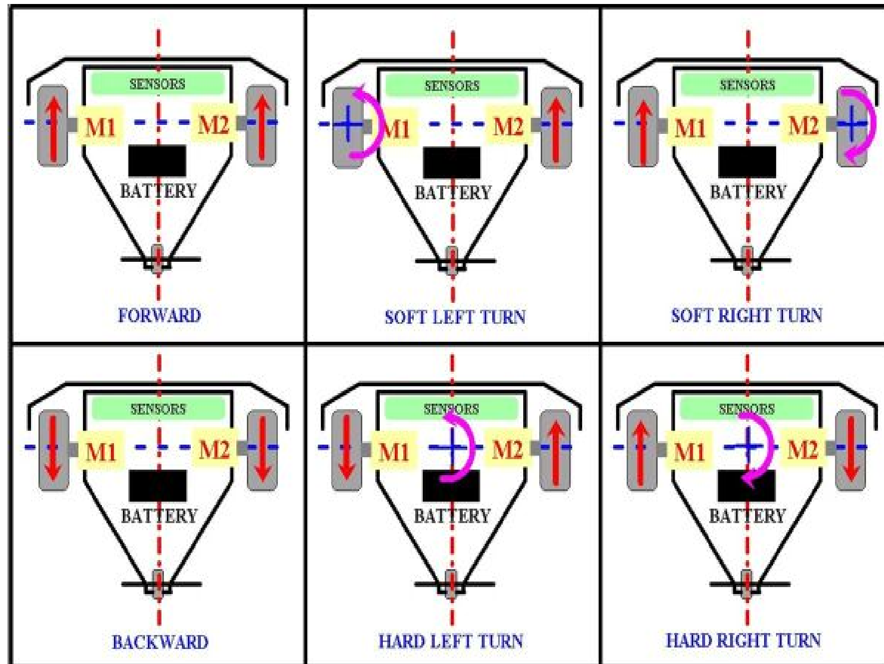


Figure 4.4 Possible robot motion

Above figure shows all the possible motions for the robot. First we will design the truth table for all these possibilities. At the end we will connect motors in such a way that this sequence will be obeyed.

Direction	PB0	PB1	Left motor status	PB2	PB3	Right motor status	HEX Value
Forward	1	0	Forward	1	0	Forward	0x05
Back	0	1	Back	0	1	Back	0x0A
Left	0	1	Back	1	0	Forward	0x06
Soft Left	0	0	Stopped	1	0	Forward	0x04
Right	1	0	Forward	0	1	Back	0x09
Soft Right	1	0	Forward	0	0	Stopped	0x01
Stop	0	0	Stopped	0	0	Stopped	0x00

Table 4.1 Motor direction

Gift Loader tray motor's direction is controlled by PB4 and PB5 pins of the microcontroller.

Direction	**PB4**	**PB5**	**Hex Value**
Up	0	1	0x2-

Down	1	0	0x1-
Stop	0	0	0x0-
Note: - indicates nibble for robot direction			

Table 4.2 Loader arm configuration

Function:

```
void forward (void)
{
 unsigned char portb_restore = 0; //used to buffer port b values
 portb_restore = PINB; //reading values of the PORTB register
 portb_restore = portb_restore & 0xF0; // setting lower direction
nibbel to 0
 portb_restore = portb_restore | 0x05; // placing forward command in
the lower nibbel
 PORTB = portb_restore;
}
```

You can write functions for other directions by replacing hex value marked red in the above function. Same way functions can be written to control tray loading and unloading.

```
void tray_load (void)
{
 unsigned char portb_restore = 0; //used to buffer port b values
 portb_restore = PINB; //reading values of the PORTB register
 portb_restore = portb_restore & 0x0F; // setting upper direction
nibbel to 0
 portb_restore = portb_restore | 0x20; // placing tray_load command in
the lower nibbel
 PORTB = portb_restore;
}
```

Note the term marked in blue. In this case we are making upper nibble 0 and then set it with proper value.
You can write other functions by replacing value marked in red from the table.

You can also write function like `robo_motion_test` Which will test all the motors of your robot at once.

```
void robo_motion_test (void)
{
  buzzer_on();
  buzzer_toggel_delay();
  buzzer_off();
  buzzer_toggel_delay();
  forward();
  buzzer_toggel_delay();
  back();
  buzzer_toggel_delay();
  left();
  buzzer_toggel_delay();
  right();
  buzzer_toggel_delay();
```

```
  stop();
  buzzer_toggel_delay();
  tray_load();
  buzzer_toggel_delay();
  tray_un_load();
  buzzer_toggel_delay();
  tray_stop();
  buzzer_toggel_delay();
}
```

Controlling velocity of the robot

To control the velocity of the motor, pulse width modulated (PWM) pulses are applied to Enable pins of L293 driver. PWM is the scheme in which the duty cycle of a square wave output from the microcontroller is varied to provide a varying average DC output. What actually happens by applying a PWM pulse is that the motor is switched ON and OFF at a given frequency. In this way, the motor reacts to the time average of the power supply.

Timer1 of the microcontroller is configured to generate PWM with an 8 bit resolution. Timer1 counts from BOTTOM to MAX and then restarts from BOTTOM. The pulse width of the PWM for left and right motors is controlled by OCR1AL and OCR1BL registers of the timer. Timer1 generates a high pulse on PWM pin until the timer reaches the count mentioned in the OCR register and then switches to low until the timer reaches MAX count and then again the timer outputs a high pulse on the PWM pin and restarts counting from the BOTTOM.

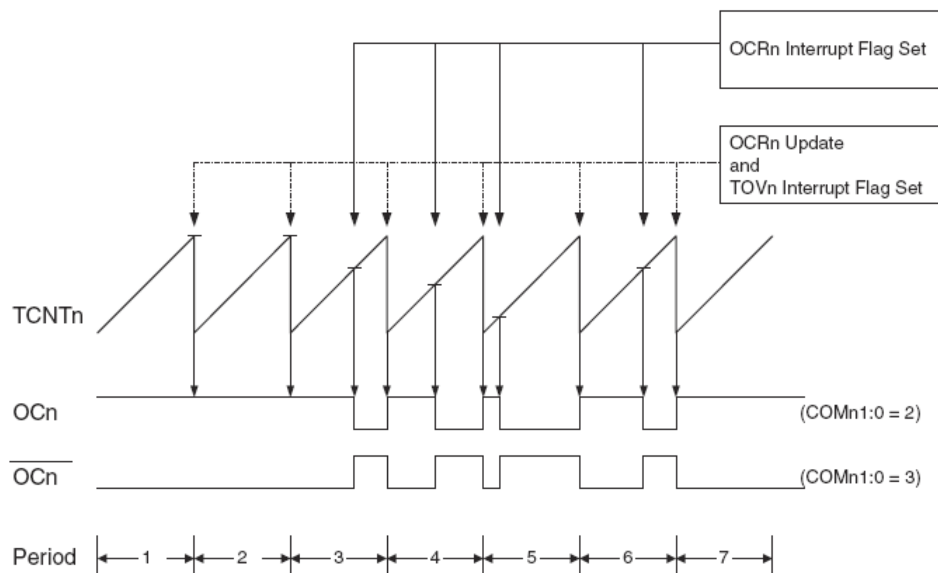


Figure 4.5: Fast PWM mode timing diagram

This code is avail able in the Step_4_velocity_control project

```
void velocity(left_motor,right_motor) //sets the velocity of the left
and the right motor do not give velocity more than 255
{
 if(left_motor > 255) //limiting the max velocity
 left_motor = 255;

 if(right_motor > 255) //limiting the max velocity
 right_motor = 255;

 OCR1AL = (unsigned char) left_motor;
 OCR1BL = (unsigned char) right_motor;
}
```

You can specify any speed between 0 to 255 in above function.

First try `velocity(255,255);` and observe the velocity then call `velocity(180,180);` you will observe that there will be reduction in the velocity.

Lets call above function with `robo_motion_test` function and test change in the velocity because of PWM

```
void main(void)
{
 //insert your functional code here...
 init_devices();
 while(1)
 {
  velocity(180,180);
  robo_motion_test();
 }
}
```

Getting sensor data using Analog to Digital Converter (ADC)

The ATMEGA16 features a 10-bit successive approximation Analog to Digital Converter (ADC). The ADC is connected to an 8-channel Analog Multiplexer which allows 8 single-ended voltage inputs constructed from the pins of Port F. The minimum value represents GND and the maximum value represents the voltage on the AREF pin (5 Volt). The result of ADC is calculated as follows for 8 bit of resolution

$$ADC = \frac{V_{IN} * 2^8}{V_{REF}} = \frac{V_{IN} * 256}{V_{REF}}$$

Two registers control the analog to digital converter. The ADC Control and Status Register (ADCSR) controls the functioning of the ADC and the ADC Multiplexer Select Register (ADMUX) controls which of the eight possible inputs is being measured.

This code is avail able in the Step_5_ADCs project

ADC function

```
unsigned char ADC_conversion (unsigned char ADC_channel_number) //ADC
channel numbers are from 0 to 7

{
 unsigned char i = 0;
 i = ADC_channel_number & 0x0F; // keeping the loweat nibbel
 ADMUX = i | 0x20; //upper nibbel of 0x20 indicates the result is left
adjusted (8 bit ADC conversion)
 ADCSRA |= 0x40; // firing the ADC

 for (i = 1; i < 255; i++); //delay
 i = ADCH;
 return (i);
}
```

Calling the function

```
void main(void)
{
 //insert your functional code here...
 unsigned char sensor_data = 0;
 init_devices();
 sensor_data = ADC_conversion(3);
}
```

In the above example variable sensor_data will receive the analog data.

Using the interrupts

When the gift is place on the top of the tray switch S2 (made up of hexa blades) will get closed and you can initiate your next move. For this you can use interrupts. When switch S2 is closed pin PD2 (INT 0) will become logic 1. We have configured this pin as interrupt which will get triggered by rising edge. When this interrupt occurs you can take necessary action in the Interrupt Service Subroutine (ISR).

This code is avail able in the Step_6_Interrupts project

```
#pragma interrupt_handler int0_isr:2
void int0_isr(void)
{
 //external interupt on INT0
 buzzer_on();
}
```

In the above code when ever interrups occours you can write proper response in the ISR section. Like in the above example when touch sensor switch S2 will closed buzzer will be turned on.

So far we have designed essential functions for the robot. You can build your code based on these codes.

Wish you all the best for the Techfest 2008