ECE3411 – Fall 2015

Lecture 2b.

General Purpose Digital Input LCD Interfacing

Marten van Dijk, Syed Kamran Haider
Department of Electrical & Computer Engineering
University of Connecticut
Email: {vandijk, syed.haider}@engr.uconn.edu

UCONN

Based on the Atmega328P datasheet and material from Bruce Land's video lectures at Cornel

Ports and their control registers

- I/O ports are labelled B, C, D: special functions are set up for each
- Can set any bit of any port to be input or output within 1 cycle
- Let x be in {B,C,D}
  - DDRx takes an 8 bit value:
    - If a bit is 1, then the corresponding pin is an output
    - If a bit is 0, then the corresponding pin is an input
  - PORTx is an I/O register:
    - Write to a bit in PORTx sets the corresponding port/pin if the corresponding DDRx bit is set to 1
  - PINx contains inputs

- E.g.,
  1. DDRx says output
  2. Set PORT
  3. Read PIN is the value just set in the PORT

- The above registers control each I/O pin independently at a logical level

ATmega328P Header file snippet

```
#define  PINB    _SFR_IO8(0x03)
#define  PINB0   0
#define  PINB1   1
#define  PINB2   2
#define  PINB3   3
#define  PINB4   4
#define  PINB5   5
#define  PINB6   6
#define  PINB7   7
```

```
#define  DDRB    _SFR_IO8(0x04)
#define  DDB0    0
#define  DDB1    1
#define  DDB2    2
#define  DDB3    3
#define  DDB4    4
#define  DDB5    5
#define  DDB6    6
#define  DDB7    7
```

```
#define  PORTB   _SFR_IO8(0x05)
#define  PORTB0  0
#define  PORTB1  1
#define  PORTB2  2
#define  PORTB3  3
#define  PORTB4  4
#define  PORTB5  5
#define  PORTB6  6
#define  PORTB7  7
```

Reading a logic value from a Port

Suppose we want to read the logic value of 7th pin of Port B:

1. Read the register PINB in a character variable, i.e.
   `char reg = PINB`
2. Let PINB register has a value 0b10101010 then
   `reg = 0b10101010`
3. Create a mask to mask out all the bits in 'reg' except for 7th bit position, i.e.
   `0b10000000 = (1<<7) = (1<<PINB7)`
4. Use the mask to mask out all the bits except for the 7th bit, and decide based on the resultant value, i.e.

```
if( reg & (1<<PINB7) ) { /* 7th pin is logic 1 */ }
else                   { /* 7th pin is logic 0 */ }
```

Tristate Buffer

- In a naïve button circuit, a closed button connects a pin to the MCU to Gnd:
  - When it opens, the MCU end of the button/switch (i.e. pin) dangles in the air
  - It acts as an antenna picking up high/low voltages depending on what frequency the local radio stations / "noisy" electrical appliances broadcast
  - Unreliable!
- Need a pull-up resistor (10kOhm) at the pin, so that if the switch is open, the voltage at the pin is pulled to high
  - If the switch is closed, the resistance to Gnd is much lower so that the voltage at the pin is close to zero
- The pull-up resistor is implicitly implemented by setting the output of the pin to high as a result of programming PORTx

Tristate Buffer

A (PORT)	B (DDR)	C (PIN)
0	1	Low impedance High out 0
1	1	Low impedance High out 1
0	0	High impedance
1	0	High impedance

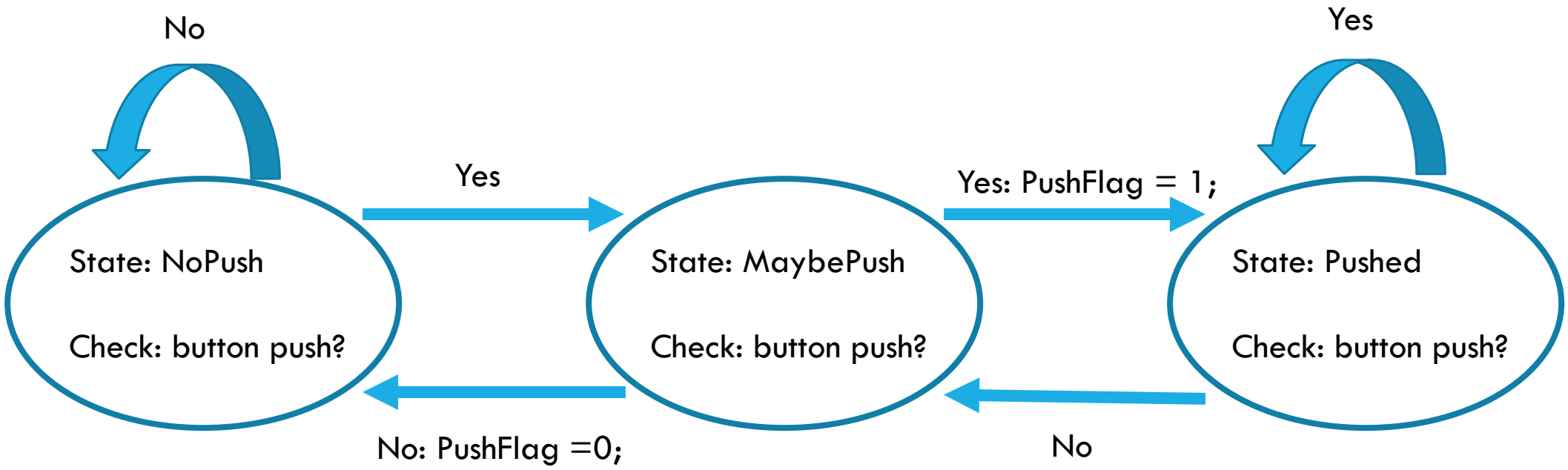
- DDR (B) = 0 and PORT (A) = 1: Eliminates static effects/noise and allows to read port/pin in a coherent fashion → PORT (A) = 1 activates the pull-up resistor and makes reading PIN (C) reliable
- DDR (B) = 0 and PORT (A) = 0: Is good for creating high impedance if you do not want the PIN to have any current at all

Debounce State Machine

- Capture a button push is a very fast process (compared to e.g setting a LED which is quite slow)
- When you press a switch closed, two surface are brought into contact with each other → no perfect match and electrical contact will be made and unmade a few times till the surfaces are firm enough together
  - The same is true when you release a button, but in reverse
  - Bouncing between high and low voltage is often at a timescale of a few um to a few ms → very often you do not see it
- No debouncing SW:

```
unsigned char PushFlag_NoDebounce; //message indicating a button push

void Task_PollingButton_NoDebounce(void)
{
  //button push of the switch connected to B.7
  if (~PINB & 0x80) PushFlag_NoDebounce = 1;
  else PushFlag_NoDebounce = 0;
}
```

Debounce State Machine



Checks happen every 30ms

- What happens if this time is increased?
- What happens if this time is decreased?

Debounce State Machine

```
unsigned char PushFlag_Debounce;

unsigned char PushState; //state machine
#define NoPush 1
#define Maybe 2
#define Pushed 3

void Task_PollingButton_Debounce(void)
{
  switch (PushState)
  {
    case NoPush:
      if (~PINB & 0x08) PushState=Maybe;
      else PushState=NoPush;
      break;
```

```
    case Maybe:
        if (~PINB & 0x08)
        {
           PushState=Pushed;
           PushFlag_Debounce=1;
         }
         else
        {
           PushState=NoPush;
           PushFlag_Debounce=0;
         }
         break;
      case Pushed:
        if (~PINB & 0x08) PushState=Pushed;
        else PushState=Maybe;
        break;
   }
}
```

Debounce State Machine

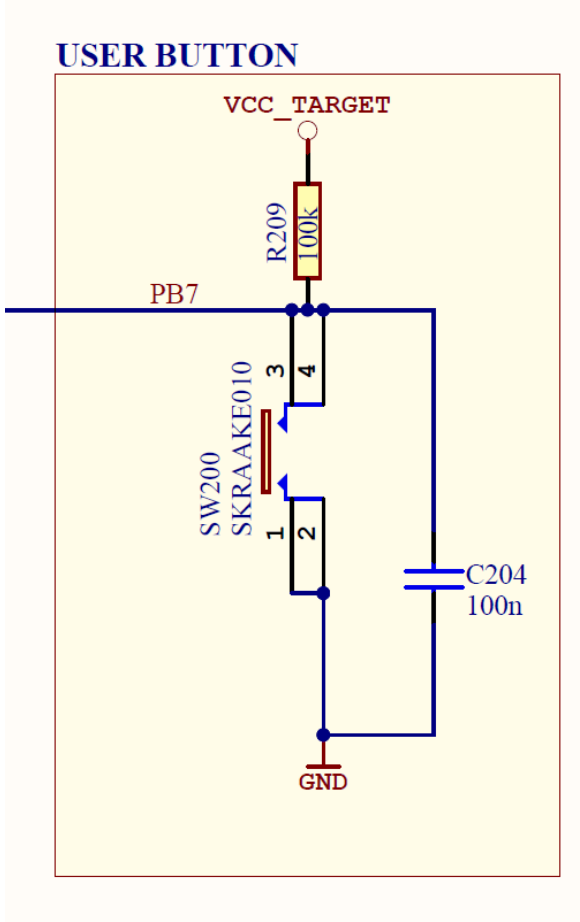
- A SW debounce state machine can also be made for a keypad
- Depending on the application, you may want to add more actions to the Finite State Machine (FSM). In other words, you may want to synchronize your FSM with other tasks.
  - E.g., as soon as PushFlag =1 is set, I may want to increment a counter
  - E.g., during the state transition from NoPush to Maybe, the actual time (possibly as a translation from the HW timers hardcoded in the MCU) is recorded. As soon as NoPush changes into Pushed, this recorded time is considered to correspond to the moment of the most recent button push.

Hardware Debouncer

- HW debouncers are also possible:
  - Just by using a low pass filter (a capacitor across the two contacts of the switch)
  - However everyone debounces in SW, saving a few cents per capacitor

- Figure shows the schematic of the push button onboard ATmega328p Xplained Mini kit
  - This is Hardware Debounced switch (Notice the capacitor C204)
  - The switch is connected to PB7

- We will do software debouncing for this switch as well anyway.



LCD

- LCD has a command state machine:
  - Erase, Draw character, etc.
- Notice that (see http://www.atmel.com/Images/Atmel-42287-ATmega328P-Xplained-Mini-User-Guide_UserGuide.pdf) the MCU is programmed through port B and C:
  - Cannot use PB3, PB4, PB5, PC6 to connect to LCD
  - If these would be connected to the databus for the LCD, then if a LCD read operation is interrupted, then the LCD is driving the bus → programmer cannot program the chip → program failure

2.2.4. Target Programming

The J204 header enable direct connection to the SPI bus with an external programmer for programming of the ATmega328P.

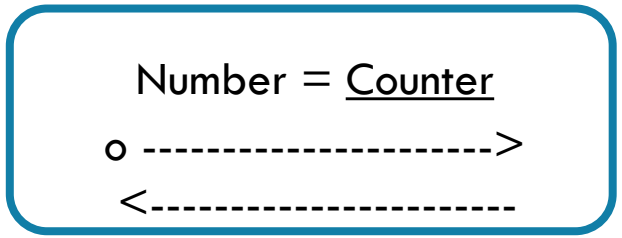
Table 2-5 SPI Header

J204 pin	ATmega328P pin	Function
1	PB4	MISO
2		VCC target
3	PB5	SCK
4	PB3	MOSI
5	PC6	RESET
6		GND

LCD

- LCD must be properly connected, otherwise the LCD does not acknowledge and the program hangs forever
- LCD library (lcd_lib.c and lcd_lib.h) uses #include <util/delay.h>
  - Allows using delay_ms() and delay_us()
  - We will use interrupts to program delay_ms() in next lectures (so that other computations can take place in the meantime)
  - Principle: Never use ms delays, but sometimes you may use a us delay because this is hard to get by using an interrupt
- Need to tell the LCD the clock rate of the MCU by setting
  #define F_CPU 16000000UL

LCD Example Display



How do we store the constant string "Number=\0" ?

Many AVRs have limited amount of RAM in which to store data, but may have more Flash space available. The AVR is a Harvard architecture processor, where Flash is used for the program, RAM is used for data, and they each have separate address spaces.

- Let's use flash for storing data!

```
//For accessing program space:
#include <avr/pgmspace.h>

const   int8_t   LCD_number[]   PROGMEM="Number=\0";
```

Is the same as char

Name

[] tells C to look at the actual number of characters in the string and reserve and appropriate a chunk to hold it

Keyword tells C to store the string in program memory (flash)

- All strings in C are terminated by a \0 (i.e., the all-zero byte)
- The string "Number=\0" is converted into ASCII integers, each integer is stored in 1 byte

LCD Example

```
/**
 * Written by Ruibing Wang (rw98@cornell.edu)
 * Mods for 644 by brl4@cornell.edu
 * Feb 2010
 *
 * Slightly modified for ECE-3411
 * by Marten van Dijk, Jan 2014
 */

#define F_CPU 16000000UL

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <util/delay.h>
#include "lcd_lib.h"
```

```
const uint8_t LCD_initialize[] PROGMEM = "LCD Initialized\0";
const uint8_t LCD_number[] PROGMEM = "Number=\0";

// LCD display buffer: general purpose print buffer in RAM
// LCD can print 16 characters, 17th character holds \0
uint8_t lcd_buffer[17];

uint16_t count;          // a number to display on the LCD
uint8_t anipos, dir;     // move a character around
```

LCD Example (cont.)

```
// task  writes to LCD every 200 mSec
void task (void)
{
        // increment time counter and format string
        sprintf(lcd_buffer,"%-i",count++);
        LCDGotoXY(7, 0);
        // display the count
        LCDstring(lcd_buffer, strlen(lcd_buffer));

        // now move a char left and right
        LCDGotoXY(anipos,1);            //second line
        LcdDataWrite(' ');

        if (anipos>=7) dir=-1;   // check boundaries
        if (anipos<=0 ) dir=1;
        anipos =anipos+dir;
        LCDGotoXY(anipos,1);            //second line
        LcdDataWrite('o');
}
```

Prints to a string destination (not a file unit); C does internal transformation from integer to string format.

LCD Example (cont.)

```
int main(void)
{
        // Initializations:
        initialize_LCD();                       //initialize the display
        LCDcursorOFF();                         // Turn off the cursor
        CopyStringtoLCD(LCD_initialize, 0, 0);
        _delay_ms(2000);                        // Display message for 2 seconds

        LCDclr();       //clear the display
        // put some stuff on LCD starting at char=0 line=0
        CopyStringtoLCD(LCD_number, 0, 0);

        // Initialize animation state variables
        count=0;
        anipos = 0;
        LCDGotoXY(anipos,1);    //second line
        LcdDataWrite('o');

        while(1)        //main task scheduler loop
        {
                task();
                _delay_ms(200);
        }
}
```

This stalls any other computation … In next lectures we will use HW timer interrupts that can be used to wake up task() every 200ms. During task() idle time of 200ms other tasks can be completed.

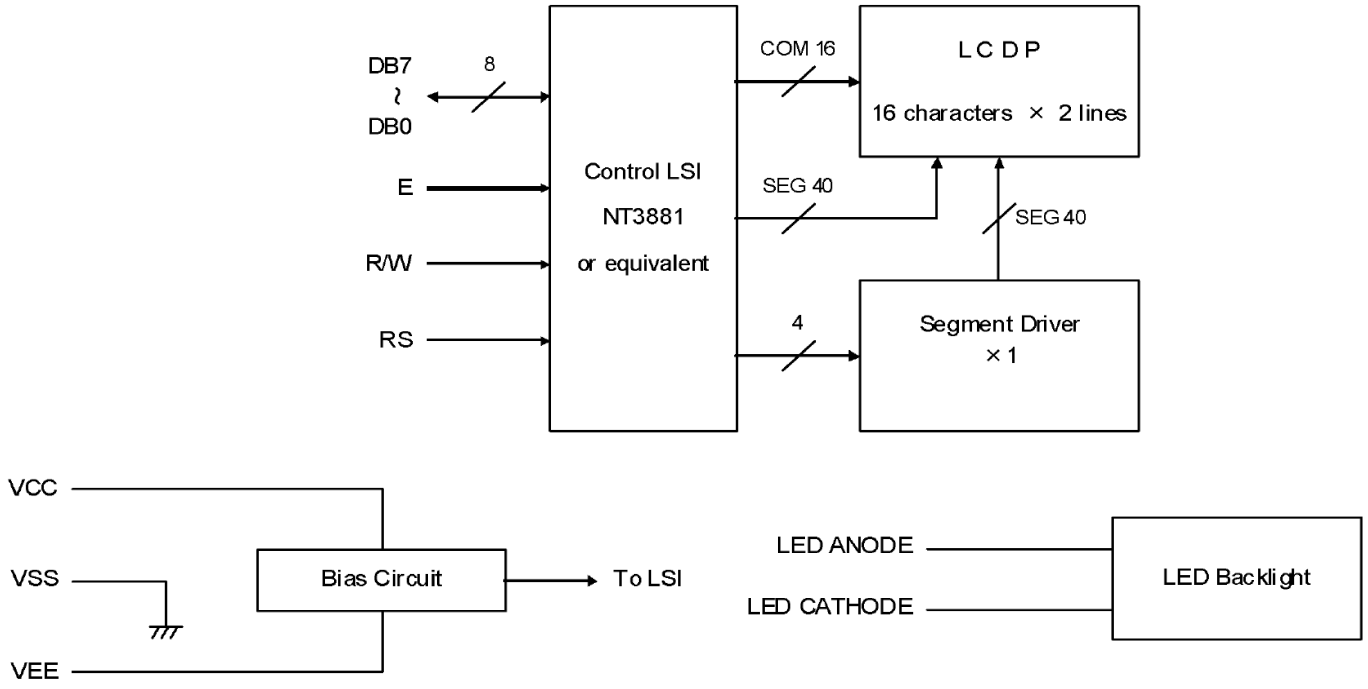
LCD Pin Assignment

Taken from LCD Datasheet available here

No.	Symbol	Level	Function
1	V_{SS}	—	Power Supply (0V, GND)
2	V_{CC}	—	Power Supply for Logic
3	V_{EE}	—	Power Supply for LCD Drive
4	RS	H / L	Register Select Signal
5	R/W	H / L	Read/Write Select Signal H : Read L : Write
6	E	H / L	Enable Signal (No pull-up Resister)
7	DB0	H / L	Data Bus Line / Non-connection at 4-bit operation
8	DB1	H / L	Data Bus Line / Non-connection at 4-bit operation
9	DB2	H / L	Data Bus Line / Non-connection at 4-bit operation
10	DB3	H / L	Data Bus Line / Non-connection at 4-bit operation
11	DB4	H / L	Data Bus Line
12	DB5	H / L	Data Bus Line
13	DB6	H / L	Data Bus Line
14	DB7	H / L	Data Bus Line
15	LED CATHODE	—	LED Cathode Terminal
16	LED ANODE	—	LED Anode Terminal

Block Diagram

- Because of limited number of I/O pins on Xplained Mini kit, we use LCD in 4-bit mode



Pin1: V_{SS} → GND
Pin2: V_{CC} → 5V
Pin3: V_{EE} → GND
Pin4: RS → PC4
Pin5: R/W → GND
Pin6: E → PC5
Pin7: DB0 → N/C
Pin8: DB1 → N/C
Pin9: DB2 → N/C
Pin10: DB3 → N/C
Pin11: DB4 → PC0
Pin12: DB5 → PC1
Pin13: DB6 → PC2
Pin14: DB7 → PC3

Pin16: ANODE → 5V

Pin15: CATHODE → GND

Taken from LCD Datasheet available here

Write Operation Timing

```
void LcdCommandWrite_UpperNibble(uint8_t cm)
{
        // Give the higher half of 'cm' to DATA_PORT
        DATA_PORT = (DATA_PORT & 0xf0) | (cm >> 4);

        // Setting RS=0 to choose the instruction register
        // as we are writing a command
        CTRL_PORT &= ~(1<<RS);

        // Setting ENABLE=1
        CTRL_PORT |= (1<<ENABLE);

        // Allow the LCD controller to successfully read command in,
        // minimum 40 µs
        _delay_ms(1);

        // Setting ENABLE=0
        CTRL_PORT &= ~(1<<ENABLE);

        // Allow long enough delay for instruction writing
        _delay_ms(1);
}
```

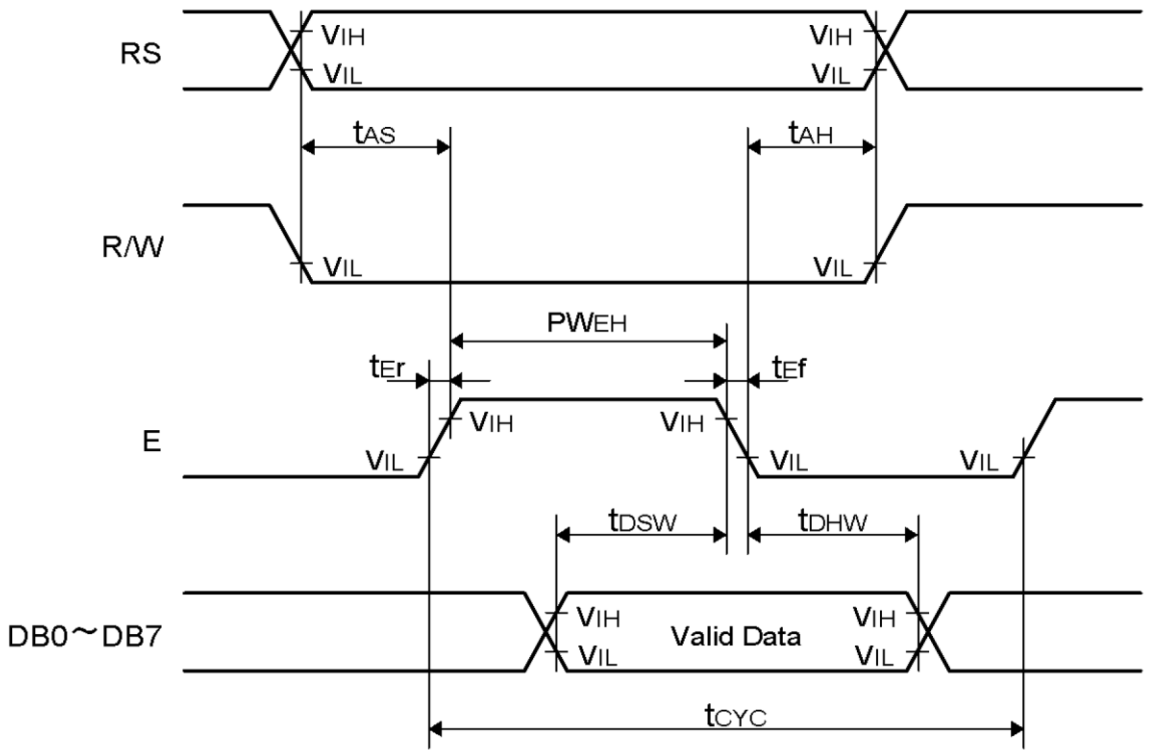


Fig.1 Write Operation Timing

Taken from LCD Datasheet available here

Read Operation Timing

- Read operation also follows similar timing as Write operation
  - Typically only a 'Busy Flag' is to be read
- We don't read 'Busy Flag', instead we provide the LCD controller long enough time to process the command
- Hence we only perform LCD writes
  - R/W signal is connected to GND, i.e. to always perform writes
  - This saves another I/O pin

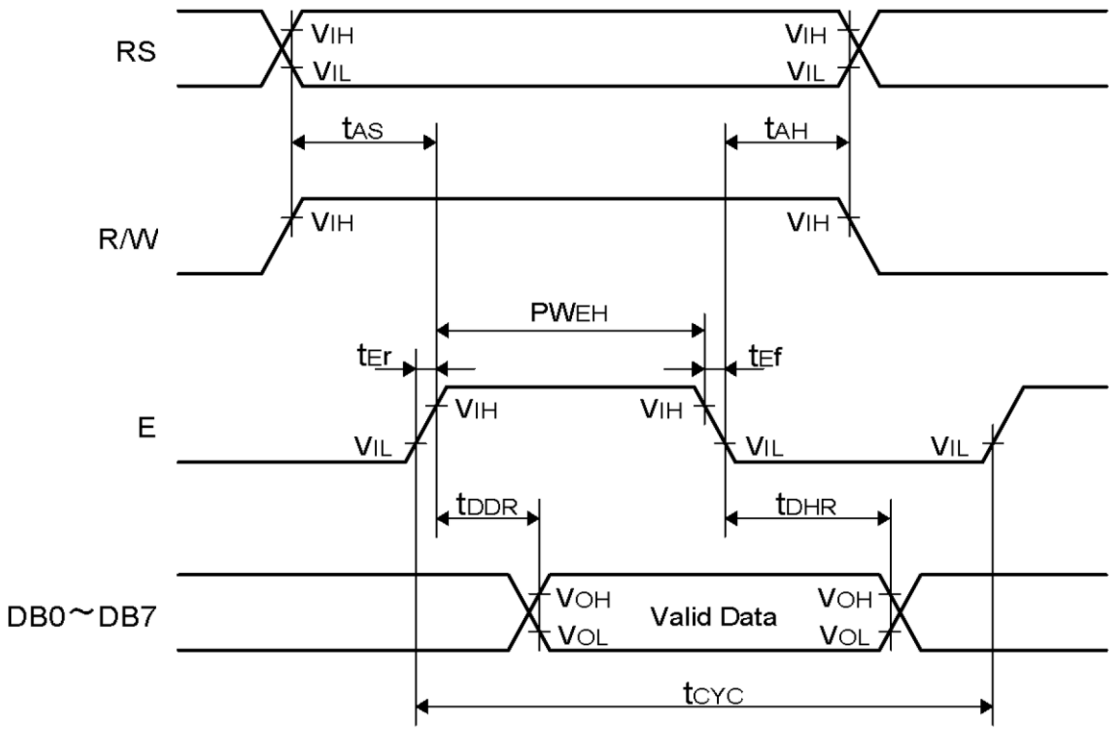


Fig.2 Read Operation Timing

Taken from LCD Datasheet available here

Timing Characteristics

Taken from LCD Datasheet available here

VCC=5.0V±10%

Parameter	Symbol	Conditions	Min.	Max.	Units
Enable Cycle Time	t_{CYC}	Fig.1, 2	500	—	ns
Enable Pulse Width	PW_{EH}	Fig.1, 2	300	—	ns
Enable Rise/Fall Time	t_{Er}, t_{Ef}	Fig.1, 2	—	25	ns
Address Setup Time	t_{AS}	Fig.1, 2	60	—	ns
Address Hold Time	t_{AH}	Fig.1, 2	10	—	ns
Write Data Setup Time	t_{DSW}	Fig.1	100	—	ns
Write Data Hold Time	t_{DHW}	Fig.1	10	—	ns
Read Data Delay Time	t_{DDR}	Fig.2	—	190	ns
Read Data Hold Time	t_{DHR}	Fig.2	20	—	ns

LCD Instruction Set

Taken from LCD Controller Datasheet available here

Instruction Set

Instruction	Code										Function	Execution time (max) (f_{osc} = 250KHz)
	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Display Clear	0	0	0	0	0	0	0	0	0	1	Clear entire display area, restore display from shift, and load address counter with DD RAM address 00H.	1.64ms
Display/ Cursor Home	0	0	0	0	0	0	0	0	1	*	Restore display from shift and load address counter with DD RAM address 00H.	1.64ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Specify direction of cursor movement and display shift mode. This operation takes place after each data transfer (read/write).	40μs
Display ON/OFF	0	0	0	0	0	0	1	D	C	B	Specify activation of display (D) cursor (C) and blinking of character at cursor position (B).	40μs
Display/ Cursor Shift	0	0	0	0	0	1	S/C	R/L	*	*	Shift display or move cursor.	40μs

LCD Instruction Set (cont.)

Instruction Set

Instruction	Code										Function	Execution time (max) (f_{osc} = 250KHz)
	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Function Set	0	0	0	0	1	DL	N	F	*	*	Set interface data length (DL), number of display line (N), and character font (F).	40μs
RAM Address Set	0	0	0	1	ACG						Load the address counter with a CG RAM address. Subsequent data access is for CG RAM data.	40μs
DD RAM Address Set	0	0	1	ADD							Load the address counter with a DD RAM address. Subsequent data access is for DD RAM data.	40μs
Busy Flag/ Address Counter Read	0	1	BF	AC							Read Busy Flag (BF) and contents of Address Counter (AC).	0μs
CG RAM/ DD RAM Data Write	1	0	Write data								Write data to CG RAM or DD RAM.	40μs
CG RAM/ DD RAM Data Read	1	1	Read data								Read data from CG RAM or DD RAM.	40μs

Note 1: Symbol "*" signifies an insignificant bit (disregard).

Note 2: Correct input value for "N" is predetermined for each model.

LCD Instruction Set (cont.)

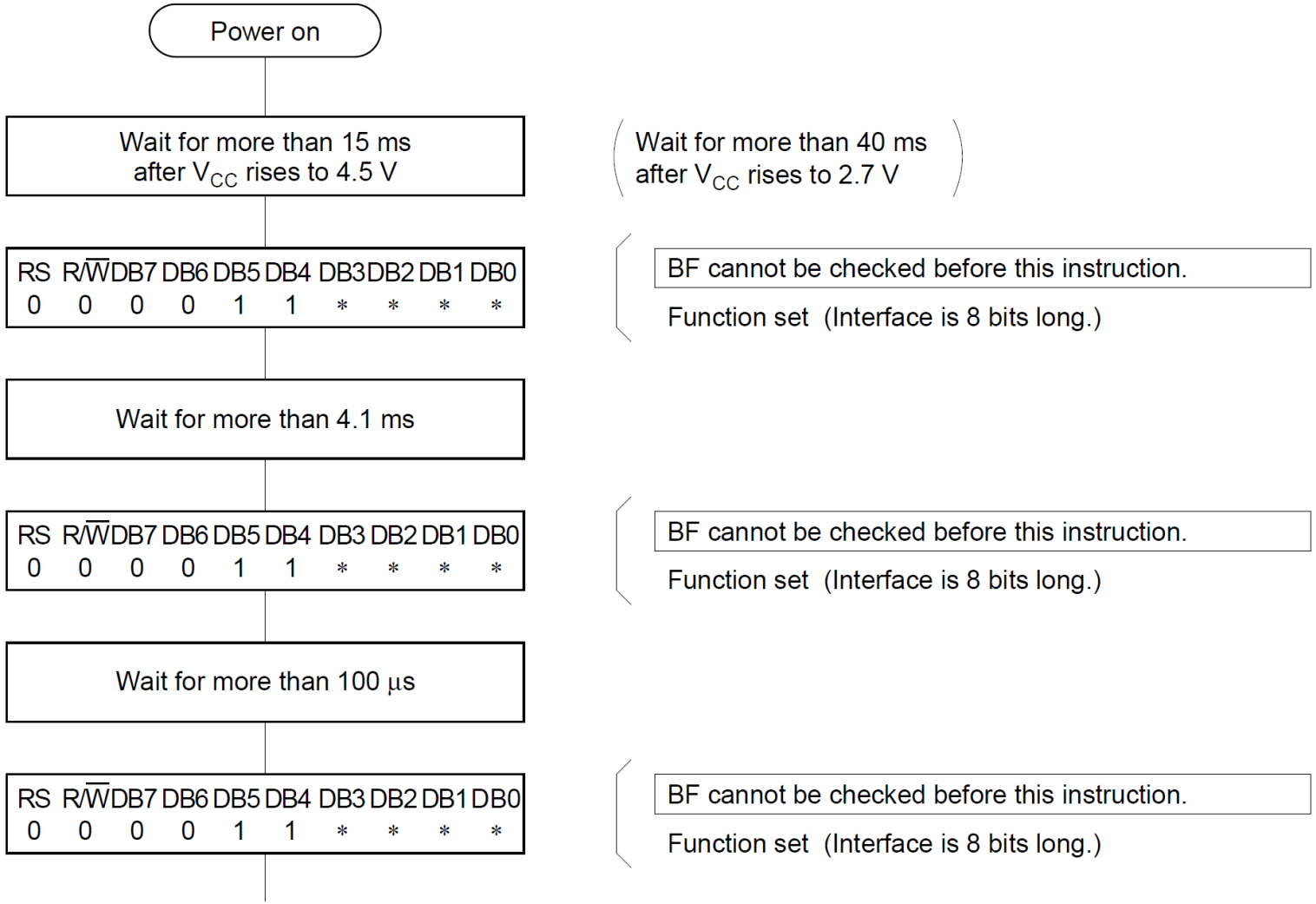
Instruction Set (continued)

Instruction	Code										Function	Execution time (max) (f_{osc} = 250KHz)
	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
	I/D = 1 : Increment S = 1 : Display Shift On D = 1 : Display On C = 1 : Cursor Display On B = 1 : Cursor Blink On S/C = 1 : Shift Display R/L = 1 : Shift Right DL = 1 : 8-Bit N = 1 : Dual Line F = 1 : 5x10 dots BF = 1 : Internal Operation BF = 0 : Ready for Instruction					I/D = 0 : Decrement S/C = 0 : Move Cursor R/L = 0 : Shift Left DL = 0 : 4-Bit N = 0 : Signal Line F = 0 : 5x8 dots					DD RAM : Display Data RAM CG RAM : Character Generator RAM ACG : Character Generator RAM Address ADD : Display Data RAM Address AC : Address Counter	

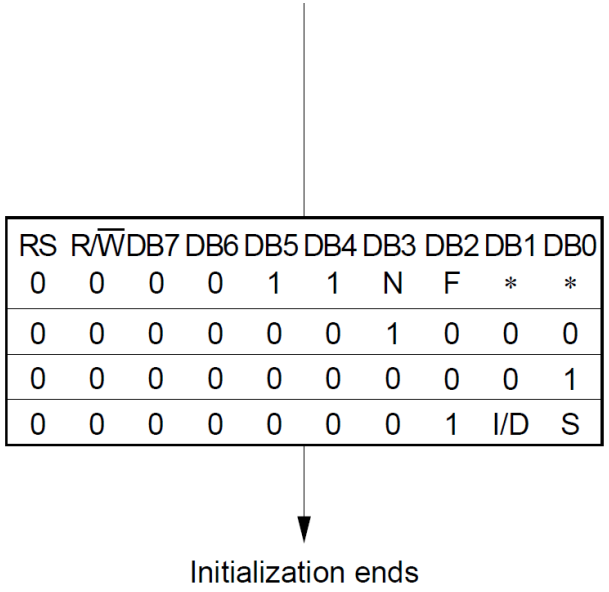
Note 1: Symbol "*" signifies an insignificant bit (disregard).

Note 2: Correct input value for "N" is predetermined for each model.

LCD Initialization: 8-bit Mode



LCD Initialization: 8-bit Mode (cont.)



RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	1	N	F	*	*
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	I/D	S

BF can be checked after the following instructions. When BF is not checked, the waiting time between instructions is longer than the execution instuction time. (See Table 6.)

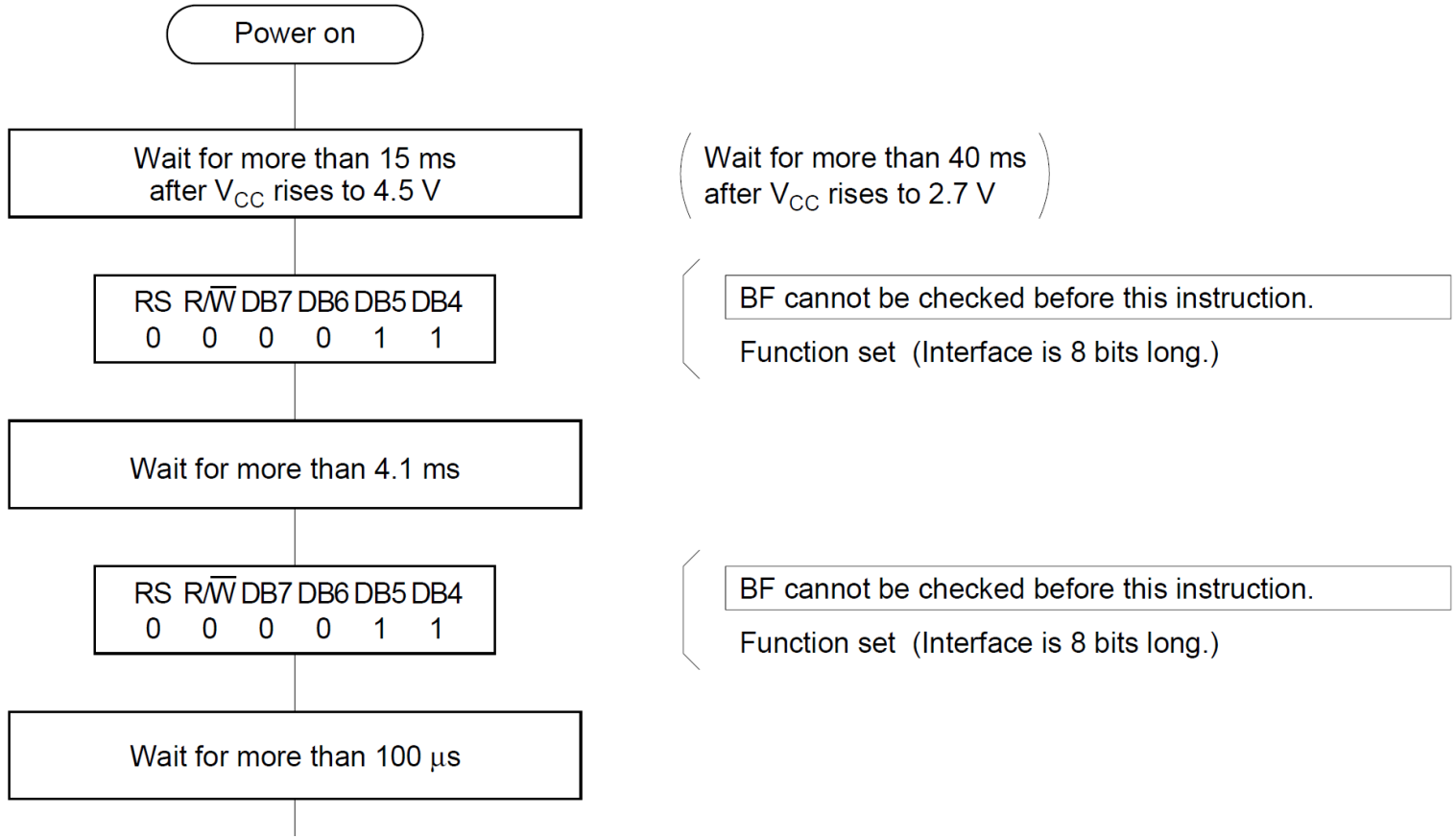
Function set (Interface is 8 bits long. Specify the number of display lines and character font.)
The number of display lines and character font cannot be changed after this point.

Display off

Display clear

Entry mode set

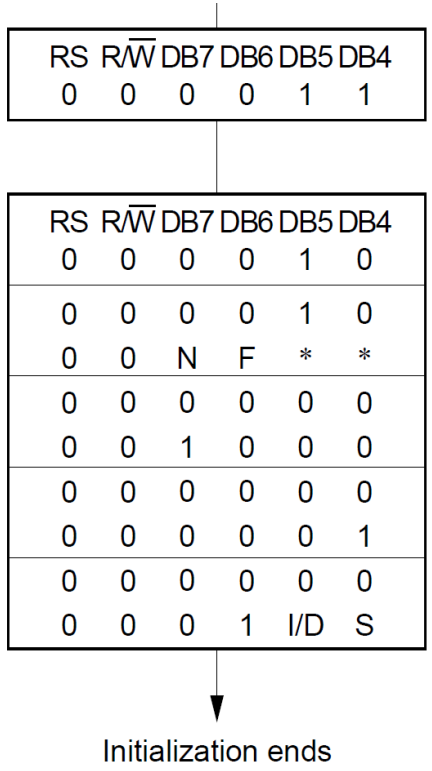
LCD Initialization: 4-bit Mode




```
LcdCommandWrite_UpperNibble(0x30);

_delay_ms(4.1);

LcdCommandWrite_UpperNibble(0x30);

_delay_us(100);
```

LCD Initialization: 4-bit Mode (cont.)

```
LcdCommandWrite_UpperNibble(0x30);

// function set: 4-bit interface
LcdCommandWrite_UpperNibble(0x20);

// 4-bit interface, 2 lines, 5x8 font
LcdCommandWrite(0x28);

// turn display off, cursor off, no blinking
LcdCommandWrite(0x08);

// clear display, set address counter  to zero
LcdCommandWrite(0x01);

// entry mode set
LcdCommandWrite(0x06);
```



BF cannot be checked before this instruction.

Function set (Interface is 8 bits long.)

BF can be checked after the following instructions. When BF is not checked, the waiting time between instructions is longer than the execution instuction time. (See Table 6.)

Function set (Set interface to be 4 bits long.)
Interface is 8 bits in length.

Function set (Interface is 4 bits long. Specify the number of display lines and character font.)
The number of display lines and character font cannot be changed after this point.

Display off

Display clear

Entry mode set

LCD Command Write (4-bit Mode)

```
void LcdCommandWrite(uint8_t cm)
{
        // First send higher 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (cm >> 4);      //give the higher half of cm to DATA_PORT
        CTRL_PORT &= ~(1<<RS);                           //setting RS=0 to choose the instruction register
        CTRL_PORT |= (1<<ENABLE);                        //setting ENABLE=1
        _delay_ms(1);                                    // allow the LCD controller to successfully read command in
        CTRL_PORT &= ~(1<<ENABLE);                       // Setting ENABLE=0
        _delay_ms(1);                                    // allow long enough delay for instruction writing

        // Send lower 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (cm & 0x0f);    //give the lower half of cm to DATA_PORT
        CTRL_PORT &= ~(1<<RS);                           //setting RS=0 to choose the instruction register
        CTRL_PORT |= (1<<ENABLE);                        //setting ENABLE=1
        _delay_ms(1);                                    // allow the LCD controller to successfully read command in
        CTRL_PORT &= ~(1<<ENABLE);                       // Setting ENABLE=0
        _delay_ms(1);                                    // allow long enough delay for instruction writing
}
```

LCD Data Write (4-bit Mode)

```
void LcdDataWrite(uint8_t da)
{
        // First send higher 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (da >> 4);      //give the higher half of cm to DATA_PORT
        CTRL_PORT |= (1<<RS);                            //setting RS=1 to choose the data register
        CTRL_PORT |= (1<<ENABLE);                        //setting ENABLE=1
        _delay_ms(1);                                    // allow the LCD controller to successfully read command in
        CTRL_PORT &= ~(1<<ENABLE);                       // Setting ENABLE=0
        _delay_ms(1);                                    // allow long enough delay

        // Send lower 4-bits
        DATA_PORT = (DATA_PORT & 0xf0) | (da & 0x0f);    //give the lower half of cm to DATA_PORT
        CTRL_PORT |= (1<<RS);                            //setting RS=1 to choose the data register
        CTRL_PORT |= (1<<ENABLE);                        //setting ENABLE=1
        _delay_ms(1);                                    // allow the LCD controller to successfully read command in
        CTRL_PORT &= ~(1<<ENABLE);                       // Setting ENABLE=0
        _delay_ms(1);                                    // allow long enough delay
}
```