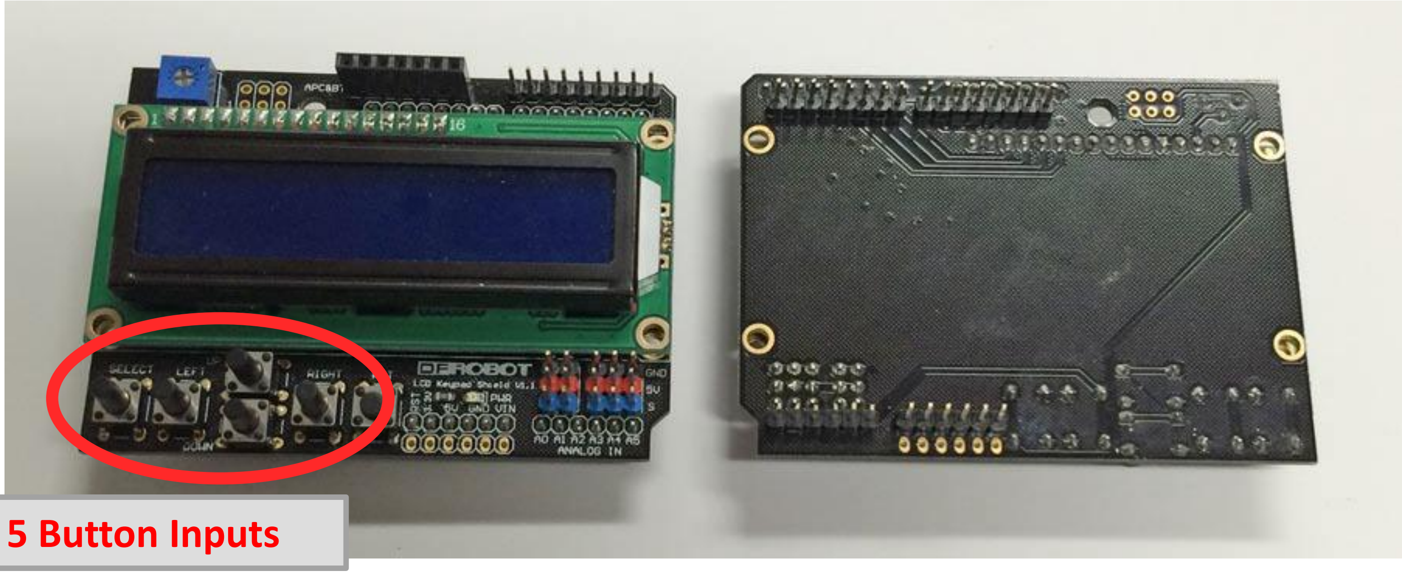
EE 109 Unit 6

LCD Interfacing

LCD BOARD

The EE 109 LCD Shield

- The LCD shield is a 16 character by 2 row LCD that mounts on top of the Arduino Uno.
- The shield also contains five buttons that can be used as input sources.



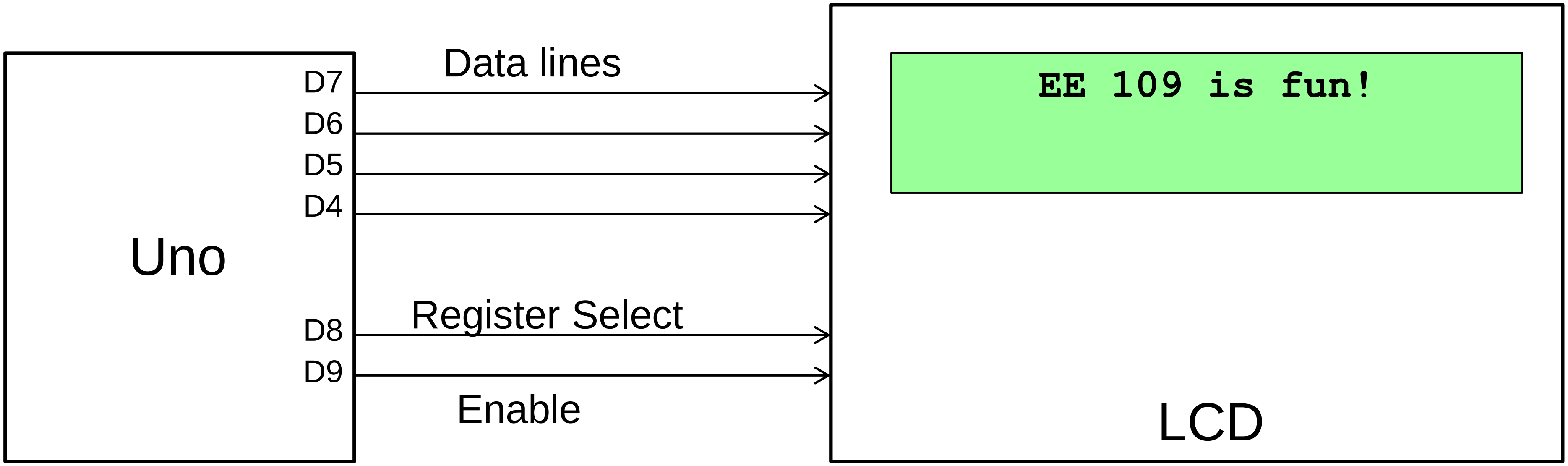
5 Button Inputs

How Do We Use It?

- By sending it data (i.e. ASCII characters one at a time) that it will display for us
- By sending it special commands to do things like:
  - Move the cursor to a specific location
  - Clear the screen contents
  - Upload new fonts/special characters

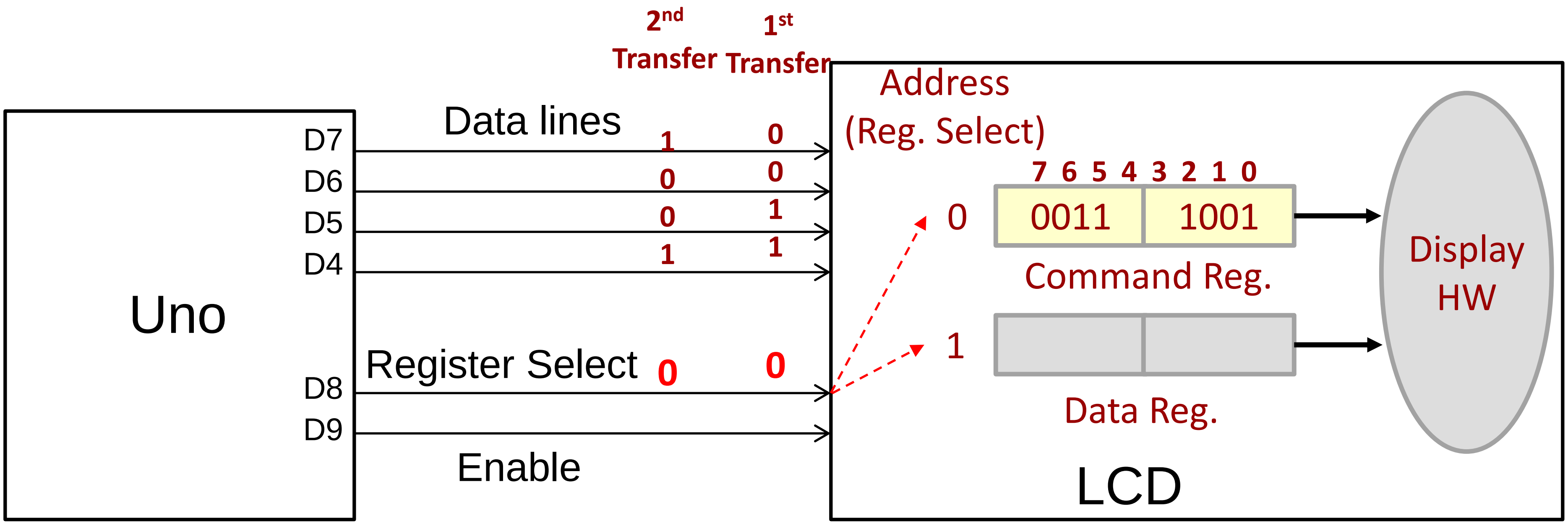
How Do We Communicate?

- The LCD uses a "parallel" interface (4-bits sent per transfer) to communicate with the µC (Note: µC => microcontroller)
- Data is transferred 4 bits at a time and uses 2 other signals (**Register Select** and **Enable**) to control ***where*** the 4-bits go and ***when*** the LCD should capture them



How Do We Communicate?

- To send an 8-bit byte we must send it in two groups of 4 bits
  - First the upper 4-bits followed by the lower 4-bits
- RS=0 sets the destination as the command reg.
- RS=1 sets the destination as the data reg.



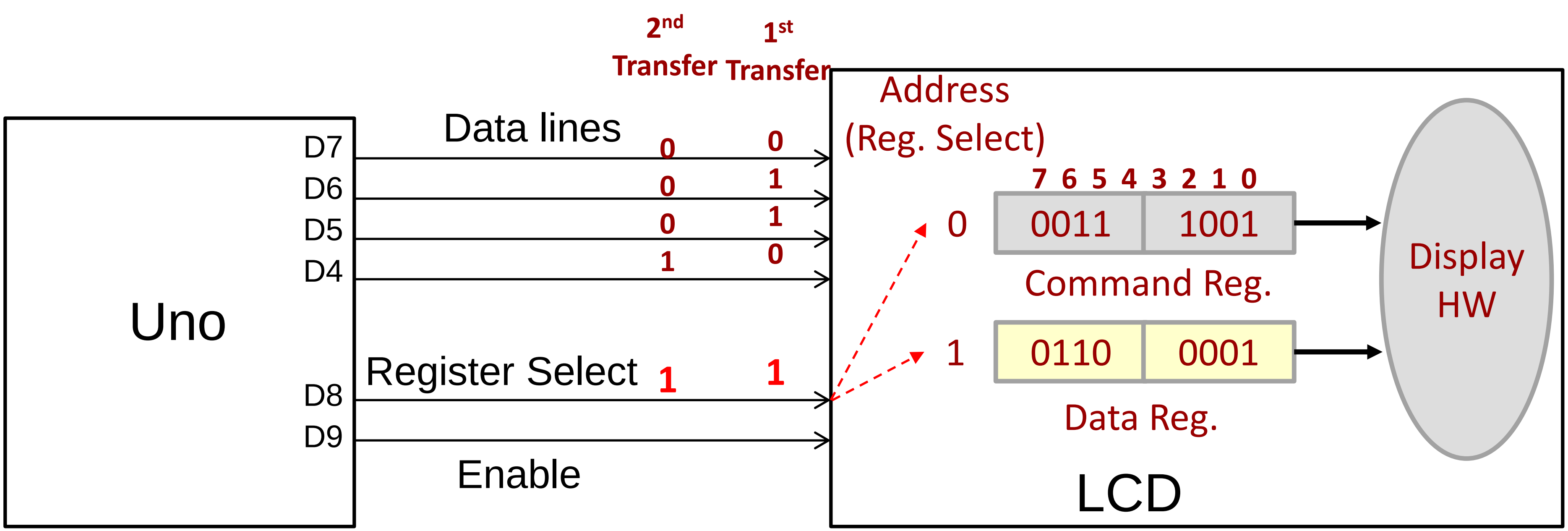
Commands and Data

- LCD contains two 8-bit registers which it uses to control its actions: Command and Data
- A Register Select (RS) signal determines which register is the destination of the data we send it (RS acts like an address selector)
  - RS = 0, info goes into the command register
  - RS = 1, info goes into the data register
- To perform operations like clear display, move cursor, turn display on or off, write the command code to the command register.
- To display characters on the screen, write the ASCII code for the character to the data register.

Command	Code
Clear LCD	0x01
Curser Home (Upper-Left)	0x02
Display On	0x0f
Display Off	0x08
Move cursor to top row, column i	0x80+i
Move cursor to bottom row, column i	0xc0+i

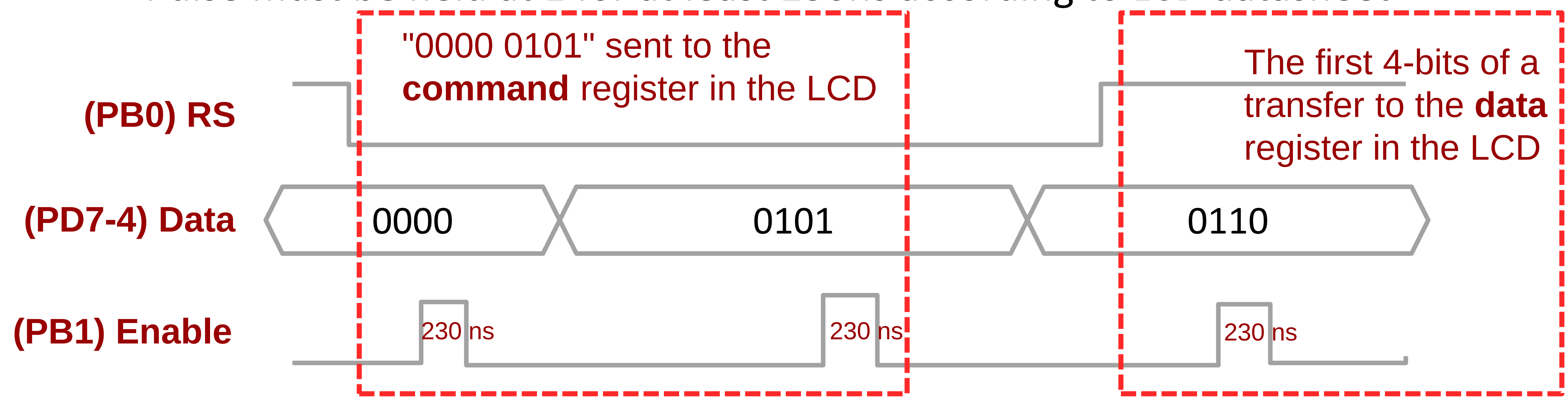
How Do We Communicate?

- To send an 8-bit byte we must send it in two groups of 4 bits
  - First the upper 4-bits followed by the lower 4-bits
- RS=0 sets the destination as the command reg.
- RS=1 sets the destination as the data reg.



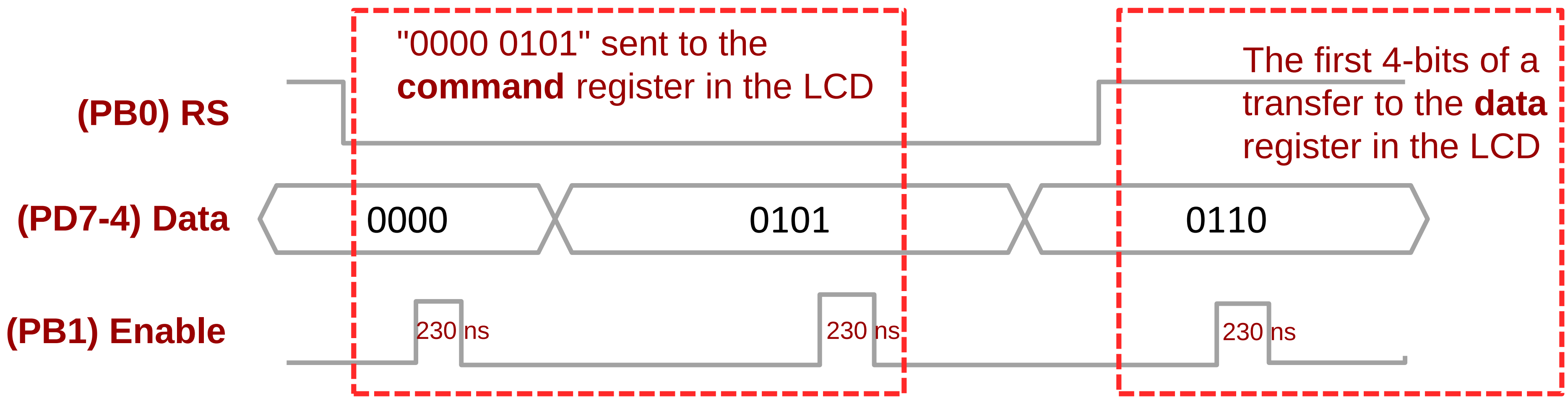
Another View

- Data from the Uno is transferred by placing four bits on the data lines (Port D bits 7-4).
- The Register Select (RS) line determines whether the data goes to the LCD's "Command Register" or "Data Register"
  - **RS=0** => Command Register **RS=1** => Data Register
- The Enable (E) line acts as a "clock" signal telling the LCD to capture the data and examine the RS bit on the 0-1-0 transition
  - Pulse must be held at 1 for at least 230ns according to LCD datasheet



Another View

- Data from the Uno is transferred by placing four bits on the data lines (Port D bits 7-4).
- Whether sending info to the "command" or "data" register, the LCD still wants a full byte (8-bits) of data so we must do 2 transfers
  - We always send the upper 4-bits of the desired data first
  - Then we transfer the lower 4-bits



Who's Job Is It?

- So who is producing the values on the RS and Data lines and the 0-1-0 transition on the E line?
- You!! With your digital I/O (setting and clearing PORT bits)

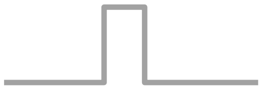
```
// Turn on bit 0 of PORTD
PORTD |= ___

// Delay 1 us > 230ns needed
// A better way in a few slides
_delay_us(1);

// Turn off bit 0 of PORTD
PORTD &= _____
```

This code would produce some voltage pattern like this on PD0

(PD0)



Note: The LCD connection doesn't use PD0, you'll need to modify this appropriately to generate the E signal

Other LCD Interface

- Other LCD devices may use
  - Only one signal (a.k.a. serial link) to communicate between the μC and LCD
    - This makes wiring easier but requires more complex software control to "serialize" the 8- or 16-bit numbers used inside the μC
  - 8-data wires plus some other control signals so they can transfer an entire byte
    - This makes writing the software somewhat easier

LCD LAB PREPARATION

Step 1

- Mount the LCD shield on the Uno without destroying the pins
- Download the “test.hex” file and Makefile from the web site, and modify the Makefile to suite your computer.
- Run “make test” to download test program to the Uno+LCD.
- Should see a couple of lines of text on the screen.

Step 2

- Develop a set of functions that will abstract the process of displaying text on the LCD
  - A set of functions to perform specific tasks for a certain module is often known as an **API** (application programming interface)
  - Once the API is written it gives other application coders a nice simple interface to do high-level tasks
- Download the skeleton file and examine the functions outlines on the next slides

LCD API Development Overview

- Write the routines to control the LCD in layers
  - Top level routines that your code or others can use: write a string to LCD, move the cursor, initialize LCD, etc.
  - Mid level routines: write a byte to the command register, write a byte to the data register
  - Low level routines: controls the 4 data lines and E to transfer a nibble to a register
- Goal: Hide the ugly details about how the interface actually works from the user who only wants to put a string on the display.

Low Level Functions

- **`lcd_writenibble(unsigned char x)`**
  - Assumes RS is already set appropriately
  - Send four bits from 'x' to the LCD
    - Takes 4-bits of x and copies them to PD[7:4] (where we've connected the data lines of the LCD)
    - **SEE NEXT SLIDES ON COPYING BITS**
    - Produces a 0-1-0 transition on the Enable signal
  - Must be consistent with mid-level routines as to which 4 bits to send, MSB or LSB
  - Uses: logical operations (AND/OR) on the PORT bits

This will be your challenge to write in lab!

Mid-Level Functions

- `lcd_writecommand(unsigned char x)`
  - Send the 8-bit byte ‘x’ to the LCD as a command
  - Set RS to 0, send data in two nibbles, delay
  - Uses: `lcd_writenibble()`
- `lcd_writedata(unsigned char x)`
  - Send the 8-bit byte ‘x’ to the LCD as data
  - Set RS to 1, send data in two nibbles, delay
  - Uses: `lcd_writenibble()`
- Could do as one function
  - `lcd_writebyte(unsigned char x, unsigned char rs)`

This will be your challenge to write these two functions in lab!

High Level API Routines

- `lcd_init()`
  - Mostly complete code to perform initialization sequence
  - **See lab writeup for what code you MUST add.**
  - Uses: `lcd_writenibble()`, `lcd_writecommand()`, delays
- `lcd_moveto(unsigned char row, unsigned char col)`
  - Moves the LCD cursor to “row” (0 or 1) and “col” (0-15)
  - Translates from row/column notation to the format the LCD uses for positioning the cursor (see lab writeup)
  - Uses: `lcd_writecommand()`
- `lcd_stringout(char *s)`
  - Writes a string of character starting at the current cursor position
  - Uses: `lcd_writedata()`

Activity: Code-Along

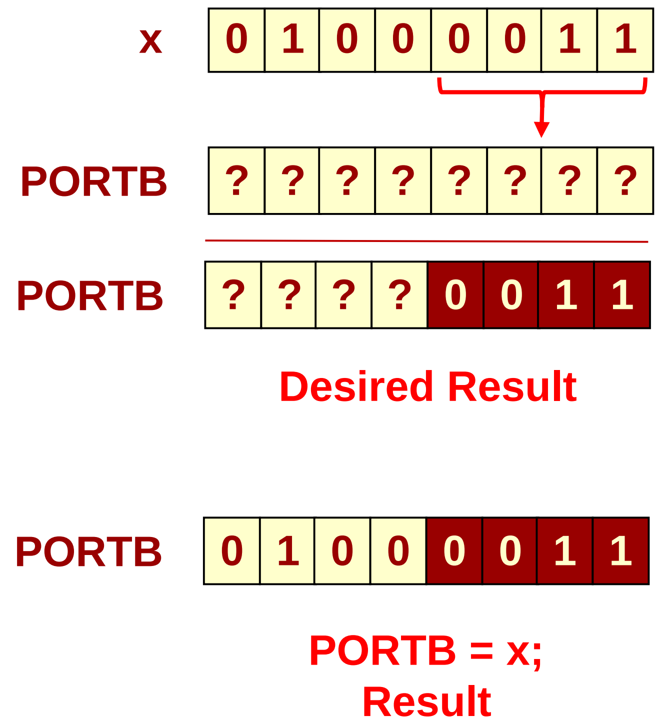
- Assuming the `lcd_writecommand()` and `lcd_writedata()` functions are correctly written, code the high-level functions:
  - `void lcd_stringout(char* str);`
  - `void lcd_moveto(int row, int col);`

To implement writenibble() these slides will help you…

COPYING BITS

Copying Multiple Bits

- Suppose we want to copy a portion of a variable or register into another BUT WITHOUT affecting the other bits
- Example: Copy the lower 4 bits of X into the lower 4-bits of PORTB…but leave the upper 4-bits of PORTB UNAFFECTED
- Assignment doesn't work since it will overwrite ALL bits of PORTB
  - PORTB = x; // changes all bits of PORTB



Copying Into a Register

- Solution…use these steps:
- Step 1: Define a mask that has 1's where the bits are to be copied

```
#define   MASKBITS   0x0f
```

- Step 2: Clear those bits in the destination register using the MASK

```
PORTB &= ~MASKBITS
```

- Step 3: Mask the appropriate field of x and then OR it with the destination, PORTB

```
PORTB |= (x & MASKBITS);
```

x: 0 1 0 0 0 0 1 1

Step 1 MASKBITS = 0 0 0 0 1 1 1 1

Step 2 PORTB: ? ? ? ? ? ? ? ?

& 1 1 1 1 0 0 0 0

PORTB: ? ? ? ? 0 0 0 0

Step 3 x: 0 1 0 0 0 0 1 1

& MASKBITS 0 0 0 0 1 1 1 1

0 0 0 0 0 0 1 1

| PORTB: ? ? ? ? 0 0 0 0

Result PORTB: ? ? ? ? 0 0 1 1

Do We Need Step 2...Yes!!!

- Can't we just do step 1 and 3 and OR the bits of x into PORTB

```
#define   MASKBITS   0x0f
PORTB |= (x & MASKBITS);
```

- No, because what if the destination (PORTB) already had some 1's where we wanted 0's to go...
- ...Just OR'ing wouldn't change the bits to 0
- That's why we need step 2
  - Step 2: Clear those bits in the destination register using the MASK

```
PORTB &= ~MASKBITS;
```

x	0	1	0	0	0	0	1	1
PORTB	?	?	?	?	1	1	1	0

What if PORTB just happened to have these bits initially

Step 1 & 3

x	0	1	0	0	0	0	1	1
& MASKBITS	0	0	0	0	1	1	1	1
	0	0	0	0	0	0	1	1
| PORTB	?	?	?	?	1	1	1	0
Result PORTB	?	?	?	?	1	1	1	1

Copying To Different Bit Locations

- What if the source bits are in a different location than the destination
  - Ex. Copy lower 4 bits of x to upper 4 bits of PORTB
- Step 1: Define a mask that has 1's **where the bits are to be copied**

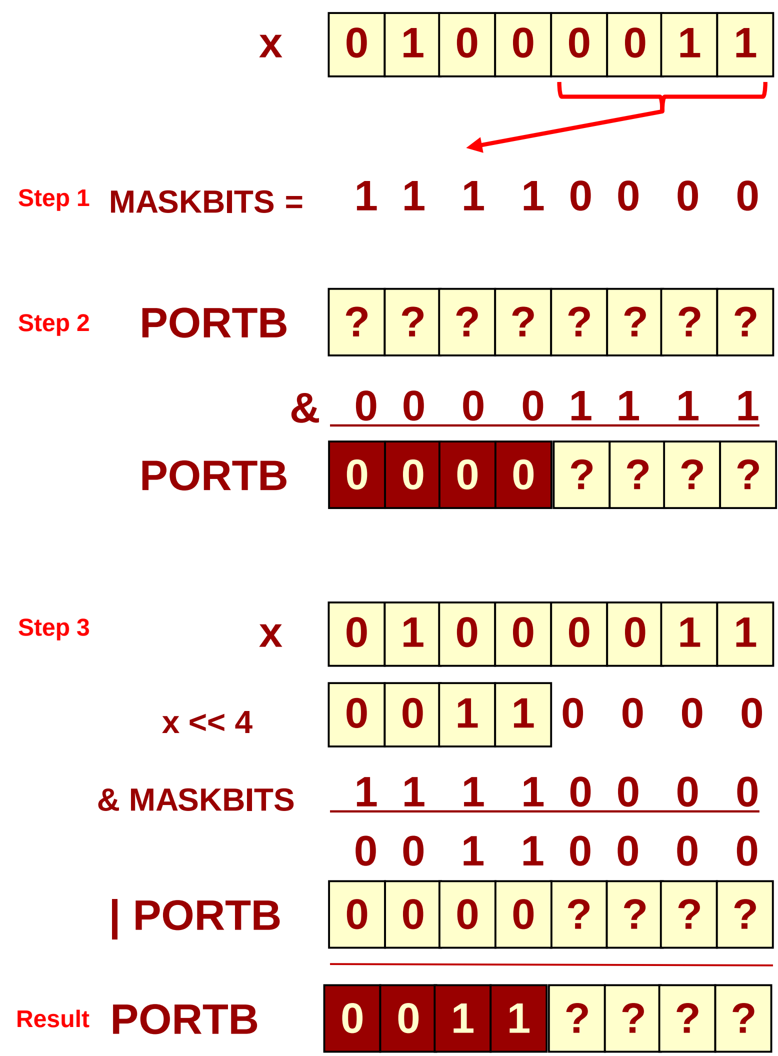
```
#define  MASKBITS   0xf0
```

- Step 2: Clear those bits in the destination register using the MASK

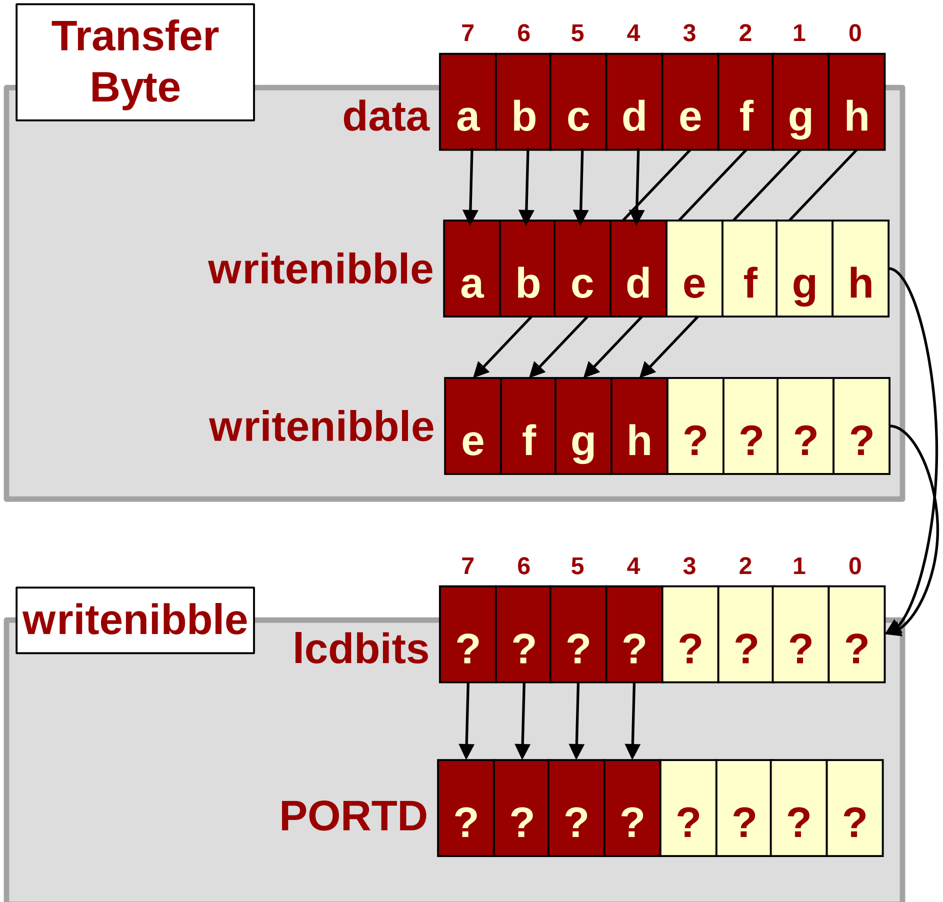
```
PORTB &= ~MASKBITS
```

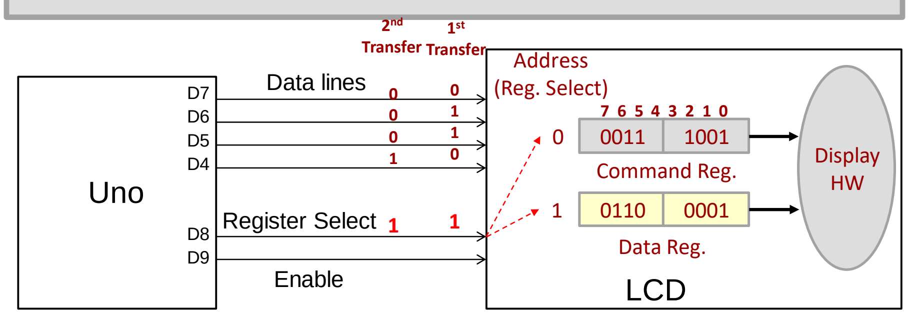
- Step 3: Shift the bits of x to align them appropriately, then perform the regular step 3

```
PORTB |= ((x<<4) & MASKBITS);
```



Coding a Byte Transfer to the LCD





Ensuring the Enable pulse is long enough

THE DEVIL IN THE DETAILS…

Does your code do the right thing?

- LCD lab required the program to generate an Enable (E) pulse.
- Example: The writenibble() routine controls the PB1 bit that is connected to the LCD Enable line.

```
PORTB |= (1 << PB1);        // Set E to 1
PORTB &= ~(1 << PB1);       // Clear E to 0
```

- Creates a 0→1→0 pulse to clock data/commands into LCD.
- But is it a pulse that will work with the LCD?
- Rumors circulated that the E pulse had to be made longer by putting a delay in the code that generated it.
- Don’t Guess. Time to read the manual, at least a little bit.

Check the LCD controller datasheet

Timing Characteristics

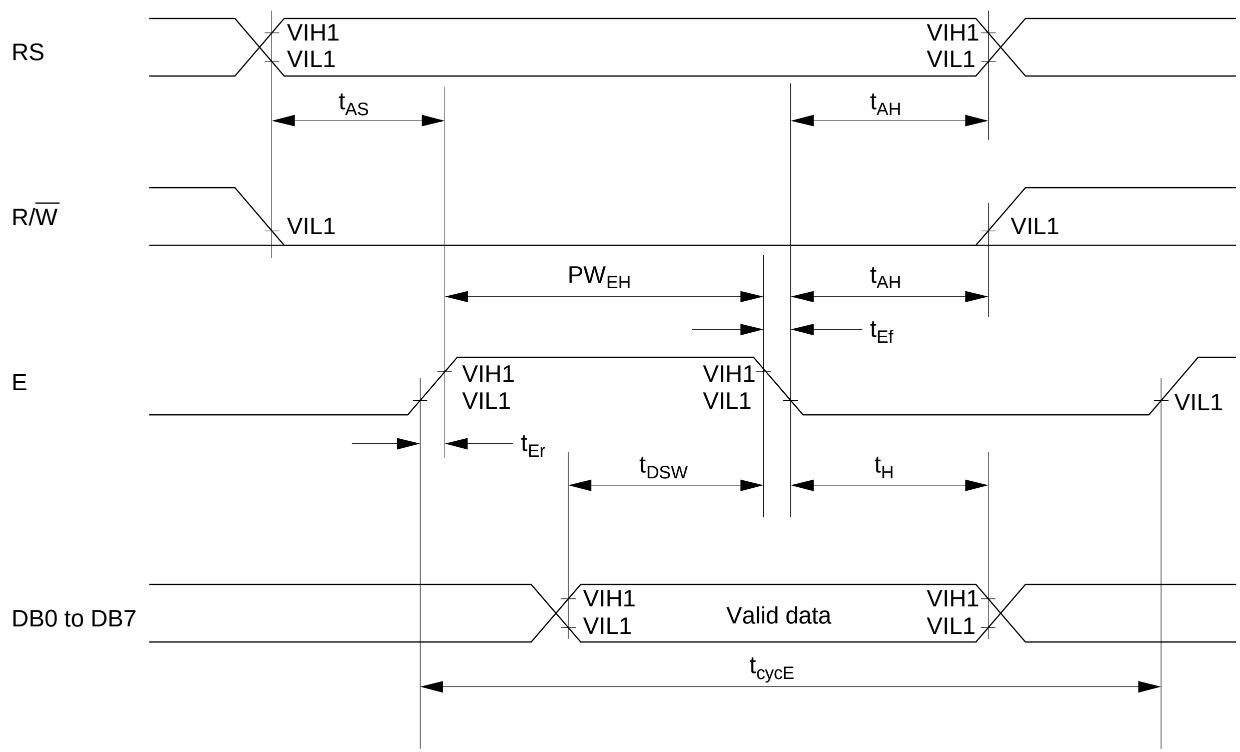


Figure 27 Write Operation

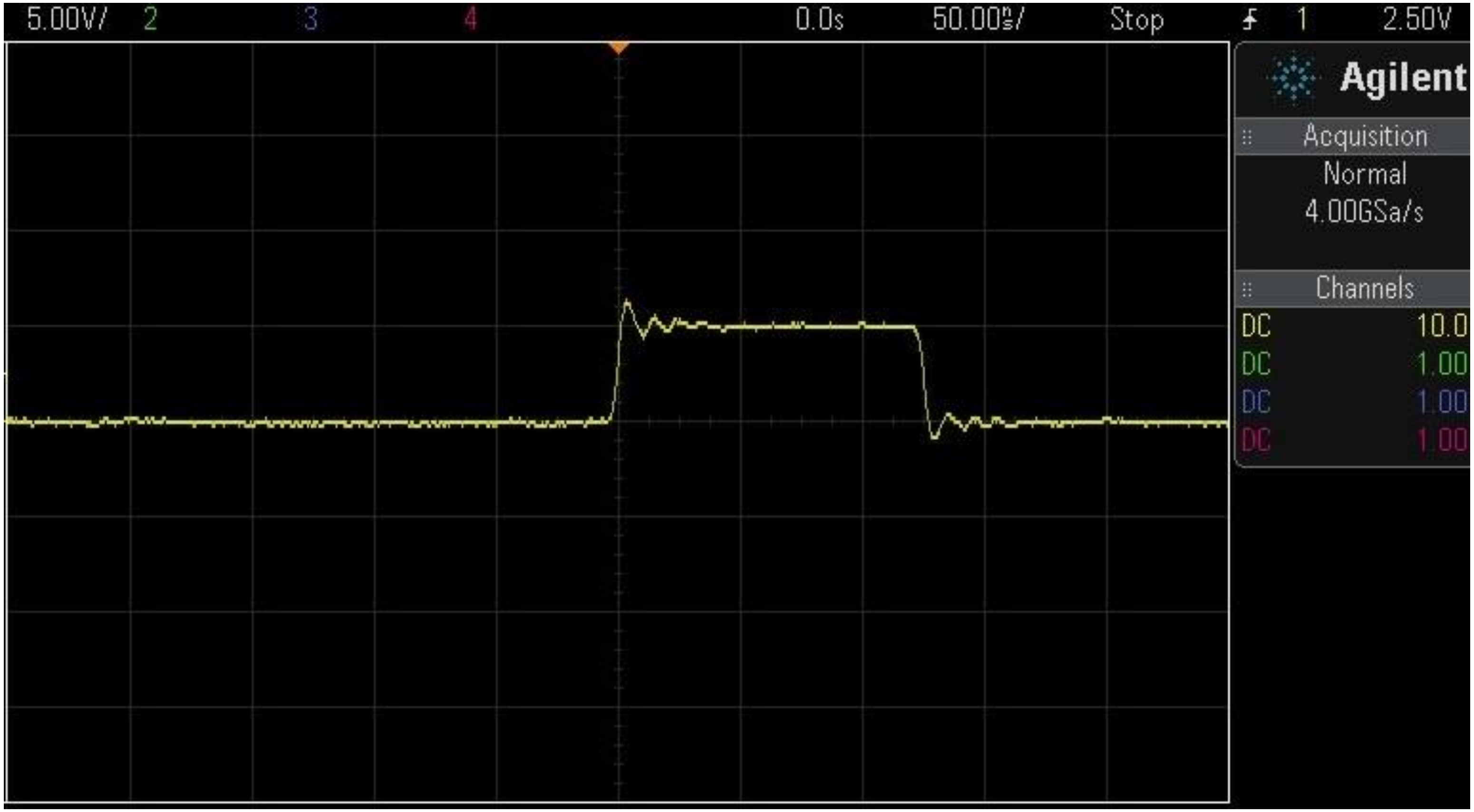
Check the generated code

- Can check the code generated by the compiler to see what is happening.
- For the creation of the E pulse the compiler generated this code:

```
SBI   PORTB, 1      ; Set Bit Immediate, PORTB, bit 1
CBI   PORTB, 1      ; Clear Bit Immediate, PORTB, bit 1
```

- According to the manual, the SBI and CBI instructions each take 2 clock cycles
- 16MHz ⇒62.5nsec/cycle, so pulse will be high for 125nsec

Check with the oscilloscope



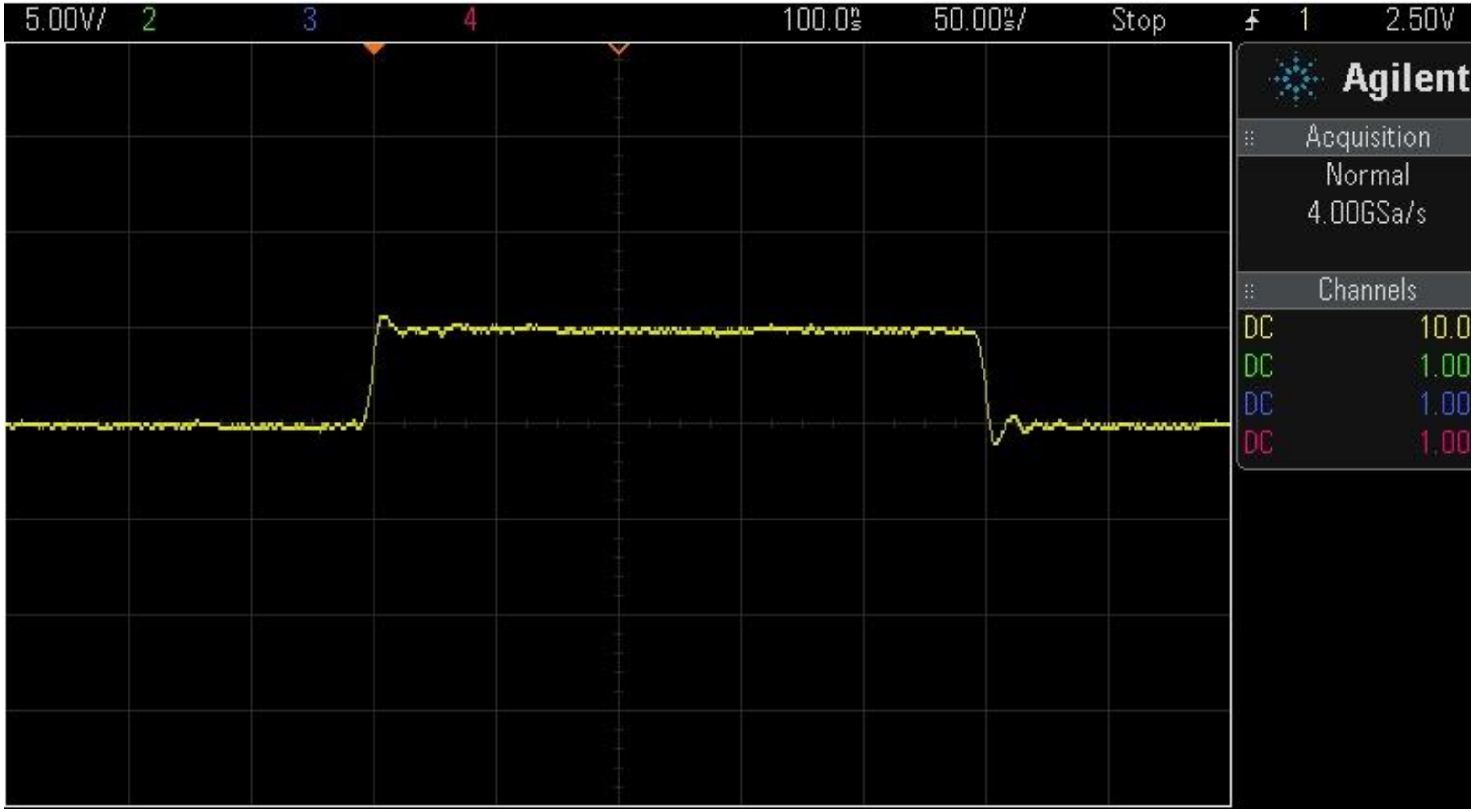
Extend the pulse

- At 125nsec, the E pulse it not long enough although it might work on some boards.
- Can use _delay_us() or _delay_ms() functions but these are longer than needed since the minimum delay is 1 us (=1000 ns) and we only need 230 ns
- Trick for extending the pulse by a little bit:

```
PORTB |= (1 << PB1);       // Set E to 1
PORTB |= (1 << PB1);       // Add another 125nsec to the pulse
PORTB &= ~(1 << PB1);      // Clear E to 0
```

USC Viterbi School of Engineering

Better looking pulse



Extend the pulse (geek way)

- Use the “asm” compiler directive to embed low level assembly code within the C code.
- The AVR assembly instruction “NOP” does nothing, and takes 1 cycle to do it.

```
PORTB |= (1 << PB1);        // Set E to 1
asm(“nop”::);               // NOP delays another 62.5ns
asm(“nop”::);
PORTB &= ~(1 << PB1);       // Clear E to 0
```

Don't guess that things will work

- When working with a device, make sure you know what types of signals it needs to see
  - Voltage
  - Current
  - Polarity (does 1 mean enable/true or does 0)
  - Duration (how long the signal needs to be valid)
  - Sequence (which transitions comes first, etc.)
- Have the manufacturer's datasheet for the device available
  - Most of it can be ignored, but some parts are critical
  - Learn how to read it
- When in doubt ➔ follow the acronym used industry-wide: RTFM (read the *!@^-ing manual)