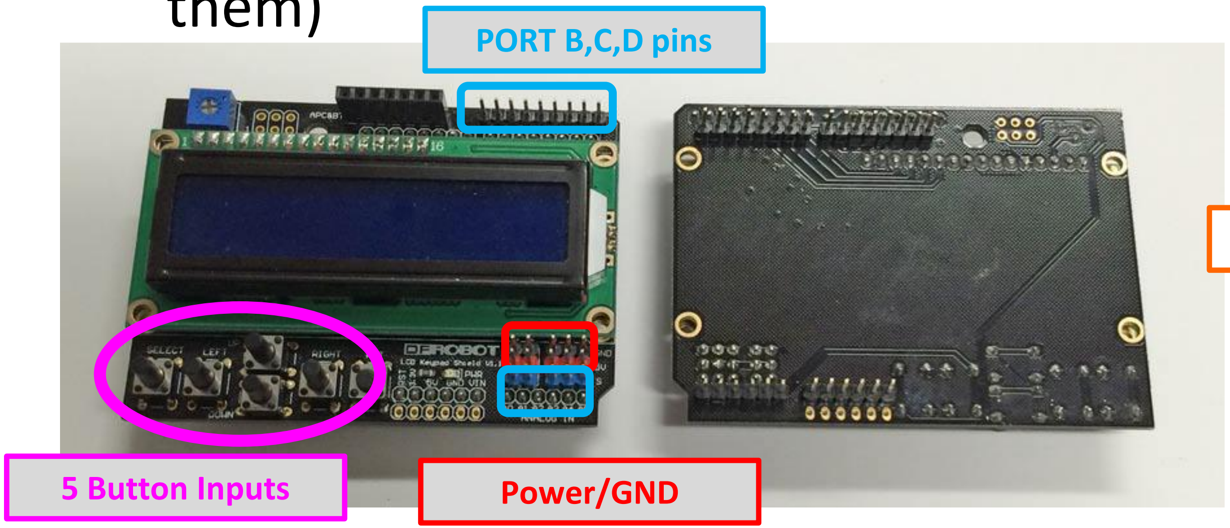
EE 109 Unit 5

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 5 buttons that can be used as input sources (but they use an analog interface, so we'll wait to use them)



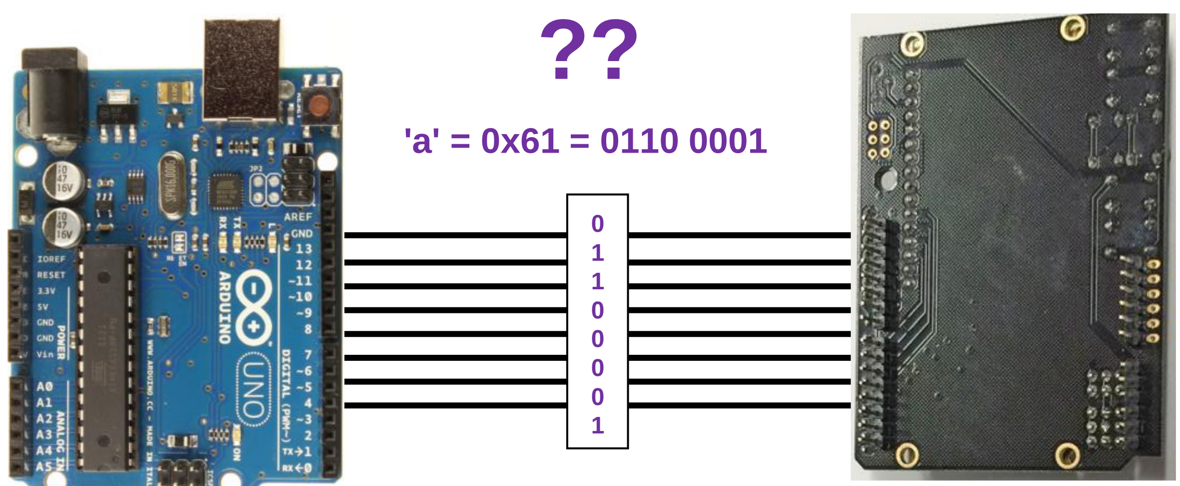
How Do We Use It?

By sending it:

- **Data**: ____________________ codes
  - Send `'a' = 0x61` and that will appear on the screen
    - Note: the cursor will automatically move over _____ position
  - You can then send it the next character code
- ________________: Numeric codes for non-printing tasks
  - Clear/erase the screen
  - Return the cursor to the upper left
  - Move the cursor to a specific location
  - Upload new fonts.

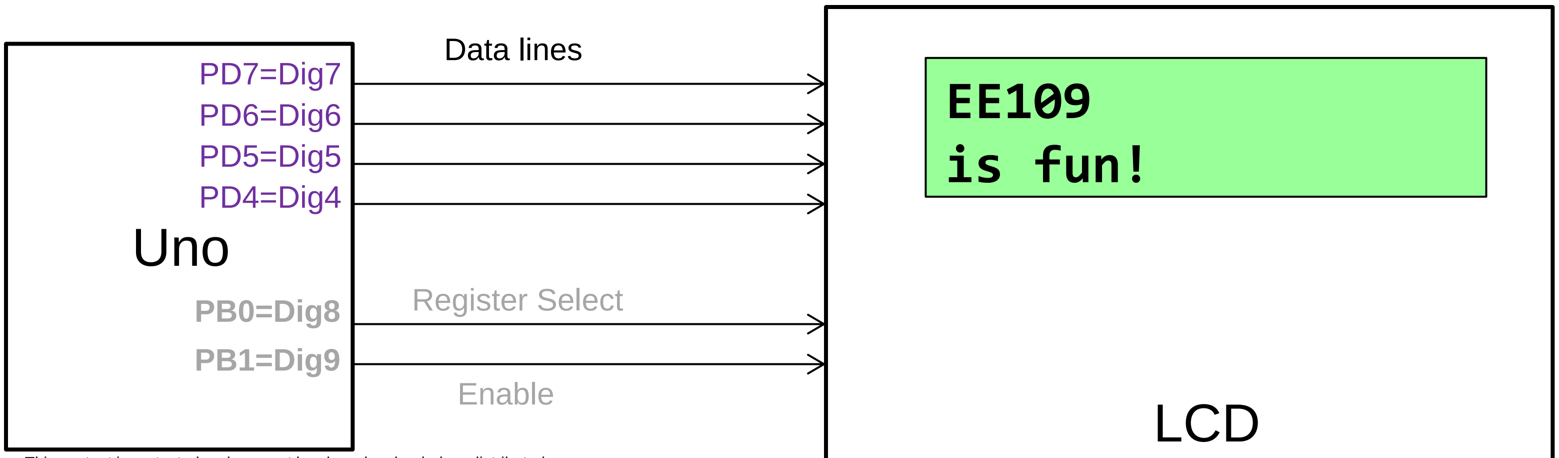
How Do We Communicate? (1)

- ASCII characters are usually 8-bit values. Should we use 8-wires?
  - We could. And some LCD's do.
  - But that would be 8 of our ____________________ pins on the Arduino
- To save pins, though, we use a **___-bit interface** (see next slide).
  - Other LCD panels may use a **1-bit serial** interface, **an 8-bit parallel** interface, etc.



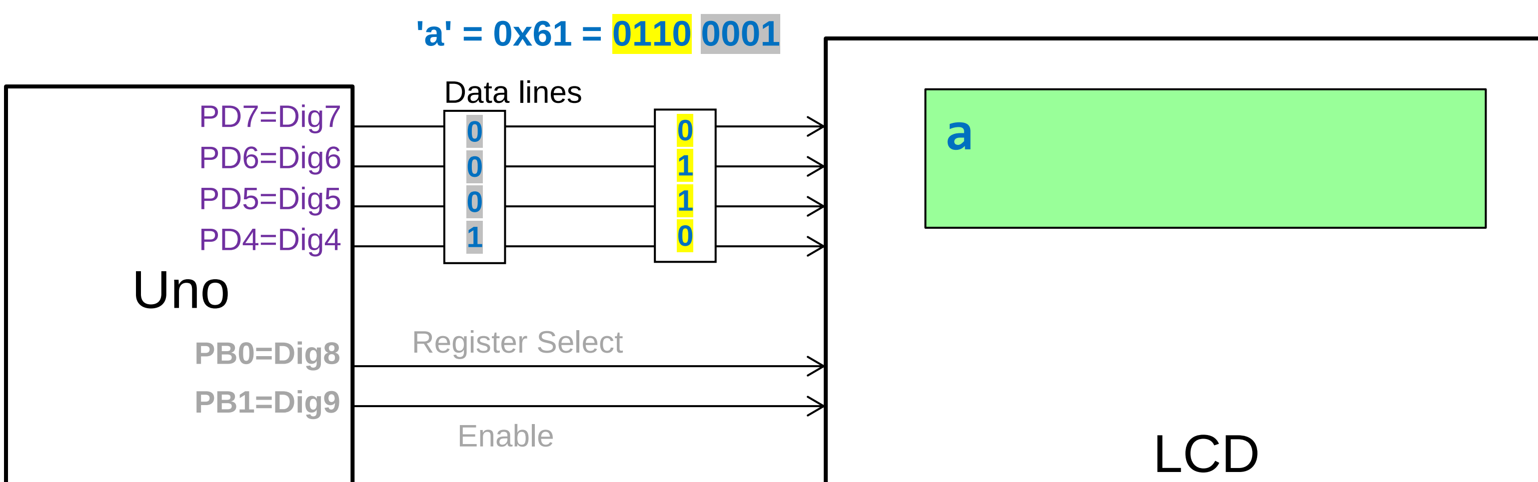
How Do We Communicate? (2)

- The LCD uses a "parallel" interface (4-bits sent per transfer) to receive information from the µC (Note: µC => microcontroller)
- **Data** and **commands** are transferred **4 bits** at a time.



How Do We Communicate? (3)

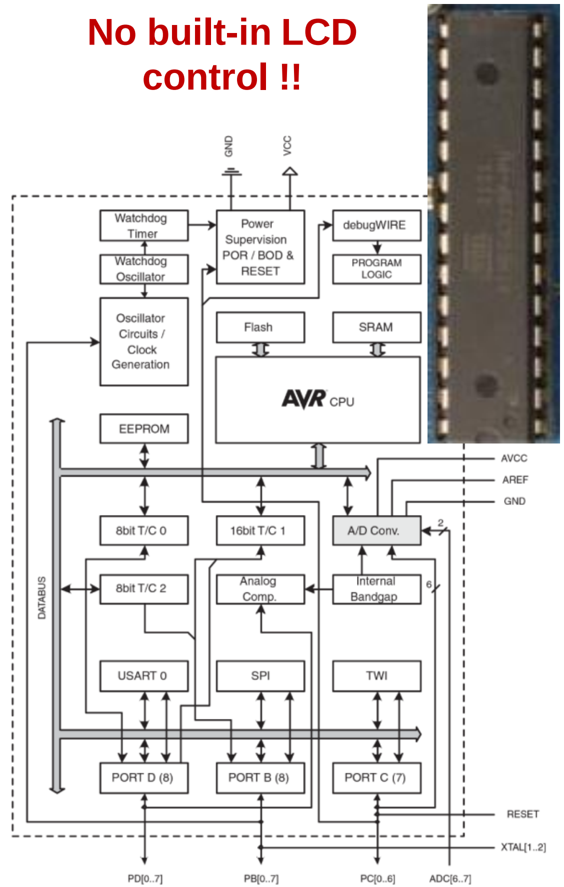
- **Data** or **commands** are transferred **4 bits** at a time and are connected to Group ___, bits _____ (__________).
- To send an 8-bit value, the LCD expects…
  - First, the _____-Significant (___-) 4-bits [i.e. the ______ 4 bits]
  - Then, the _______-Significant (___-) 4-bits [i.e. the _______ 4 bits]



Apply What You've Learned

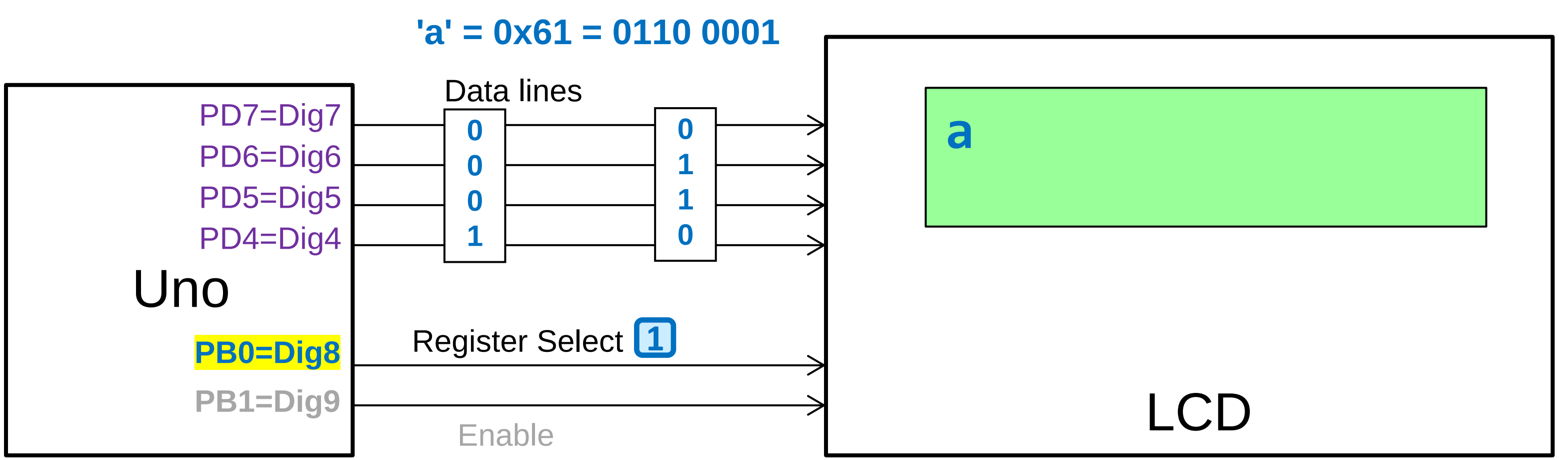
- The Arduino has several built-in hardware (HW) modules to control certain devices, BUT not the LCD
- So, we must use our ______________ skills by applying certain bitwise operations (**&, |, ^**) to specific bits of **PORTB** and **PORTD**
- As we go through this unit, anytime you see a 0 or 1 being sent to the LCD, realize **you'll need to perform some kind of bitwise operation on a specific PORT bit**

No built-in LCD control !!



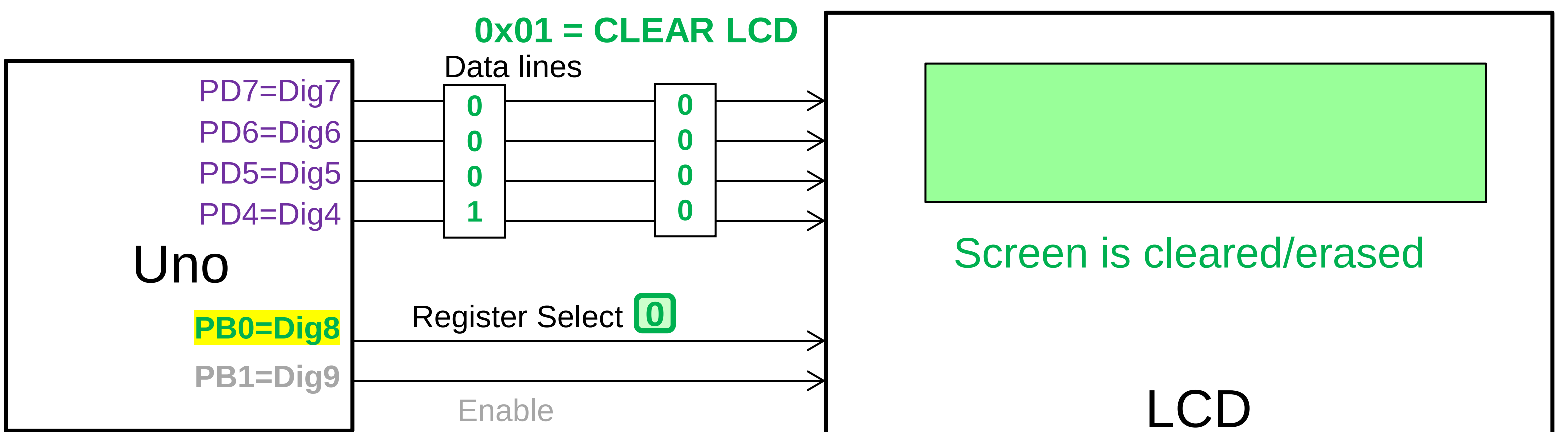
Data vs. Command (1)

- How does the LCD know if the 8-bit value is **data** or a **command**?
- Via the _________________________ (RS) bit on Group B, bit 0 (i.e. PB0)
  - RS (PB0) = ____ means **Data** (ASCII)



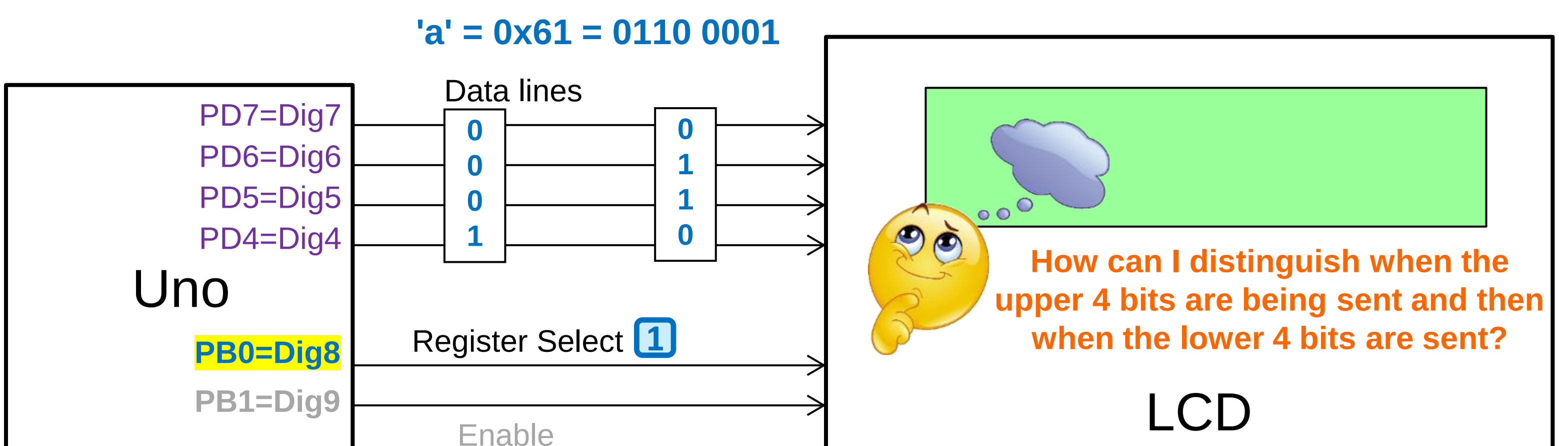
Data vs. Command (2)

- The Register Select (RS) bit (PB0) determines if the 8-bit value being sent is interpreted as **data** or a **command**
  - RS (PB0) = **1** means **Data** (ASCII)
  - RS (PB0) = **0** means **Command**
- The command codes are defined by the LCD in its documentation
  - More on a future slide.



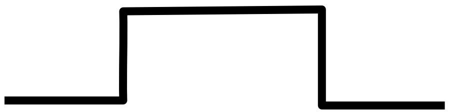
Capturing Data (1)

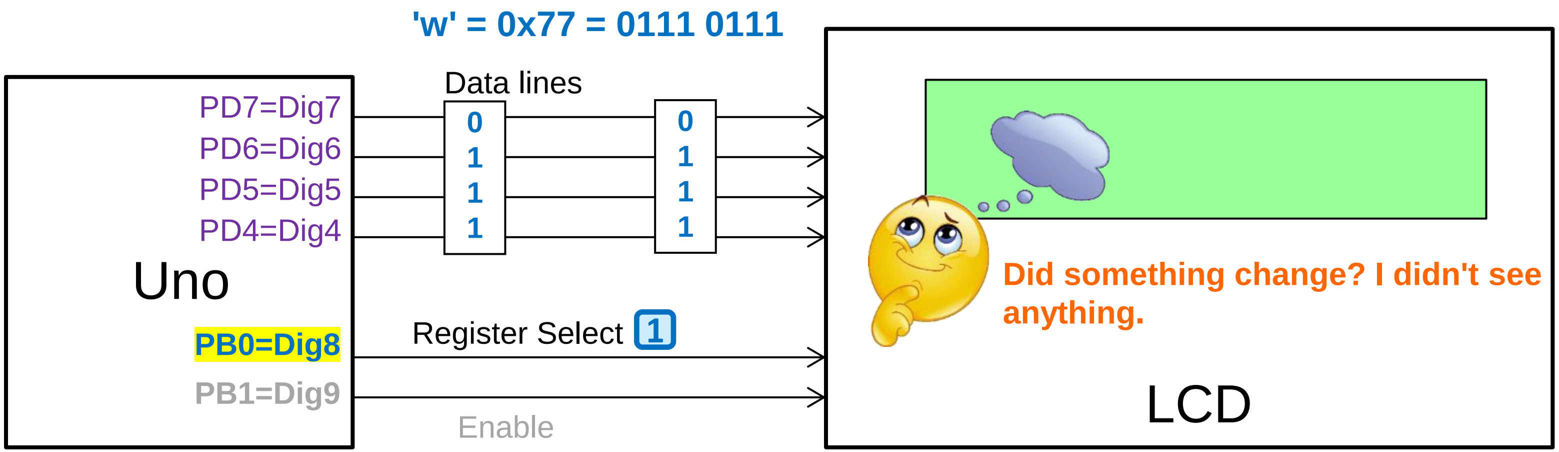
- How does the LCD know ________ we are sending the upper 4 vs. the lower 4 bits of our data?
  - **Option 1**: Look for ________________ in the data lines (e.g. 0x6 => 0x1)



Capturing Data (2)

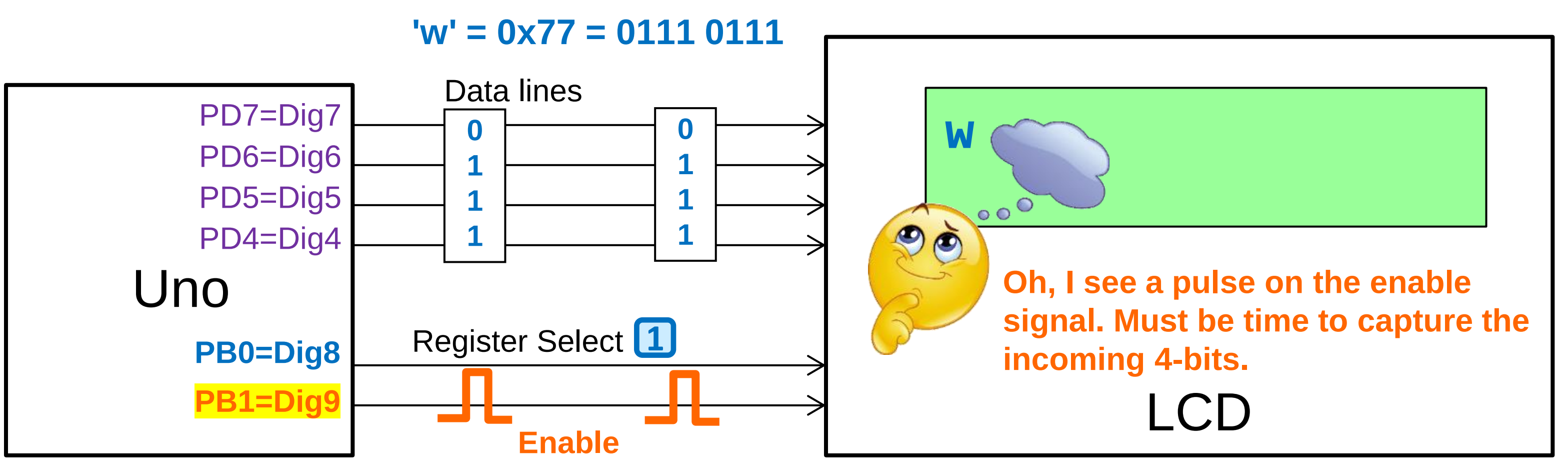
- **Option 1**: Looking for transitions in the data
  - __________ work (ASCII `'w' = 0x77`)
  - As an analogy: What data have I sent you?





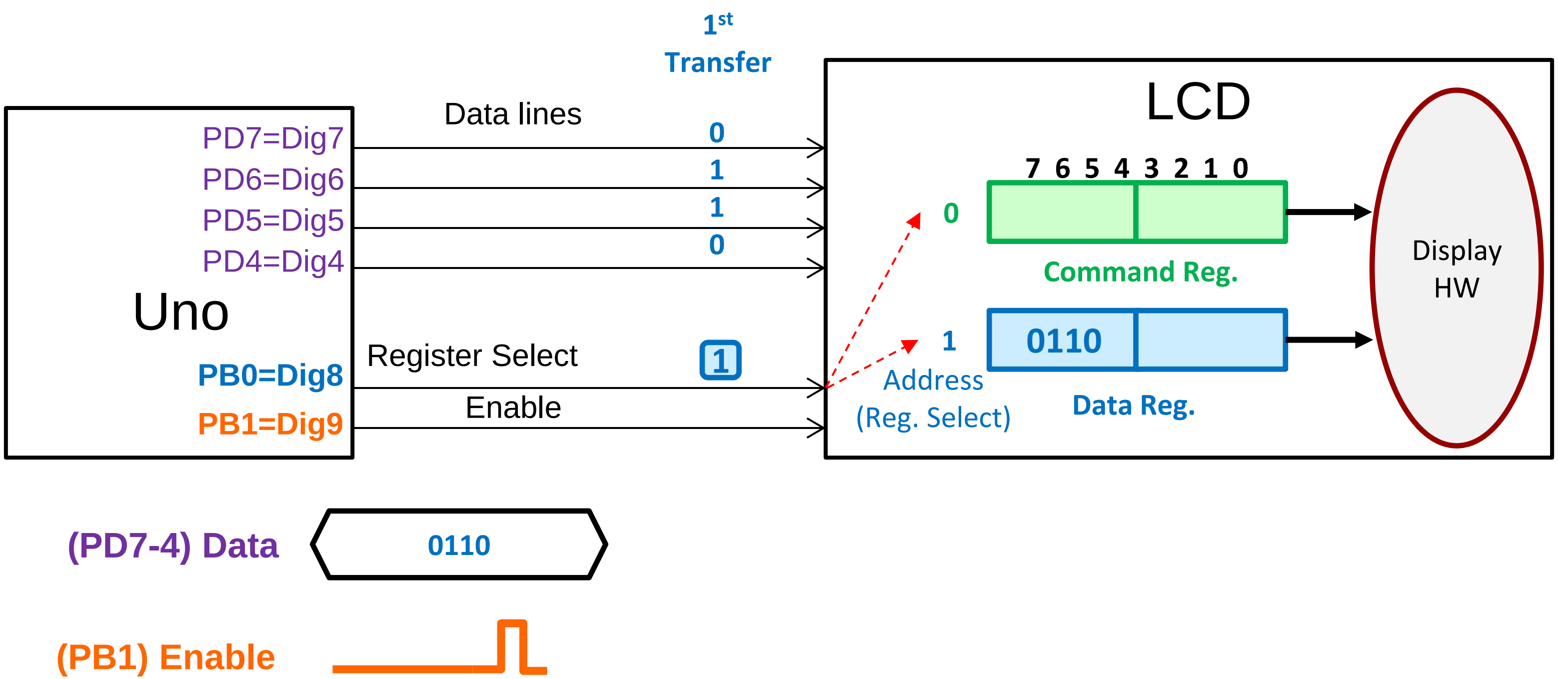
Capturing Data (3)

- **Option 2**: Use a ______________ signal to indicate when the 4-bits are ready
  - Enable bit on group B, bit 1 (PB1)
  - To signal the LCD that 4-bits of data are ready to be collected, the enable must make a __________ (**low-high-low**) transition
  - Pulse must be held at 1 for at least 230ns according to LCD datasheet



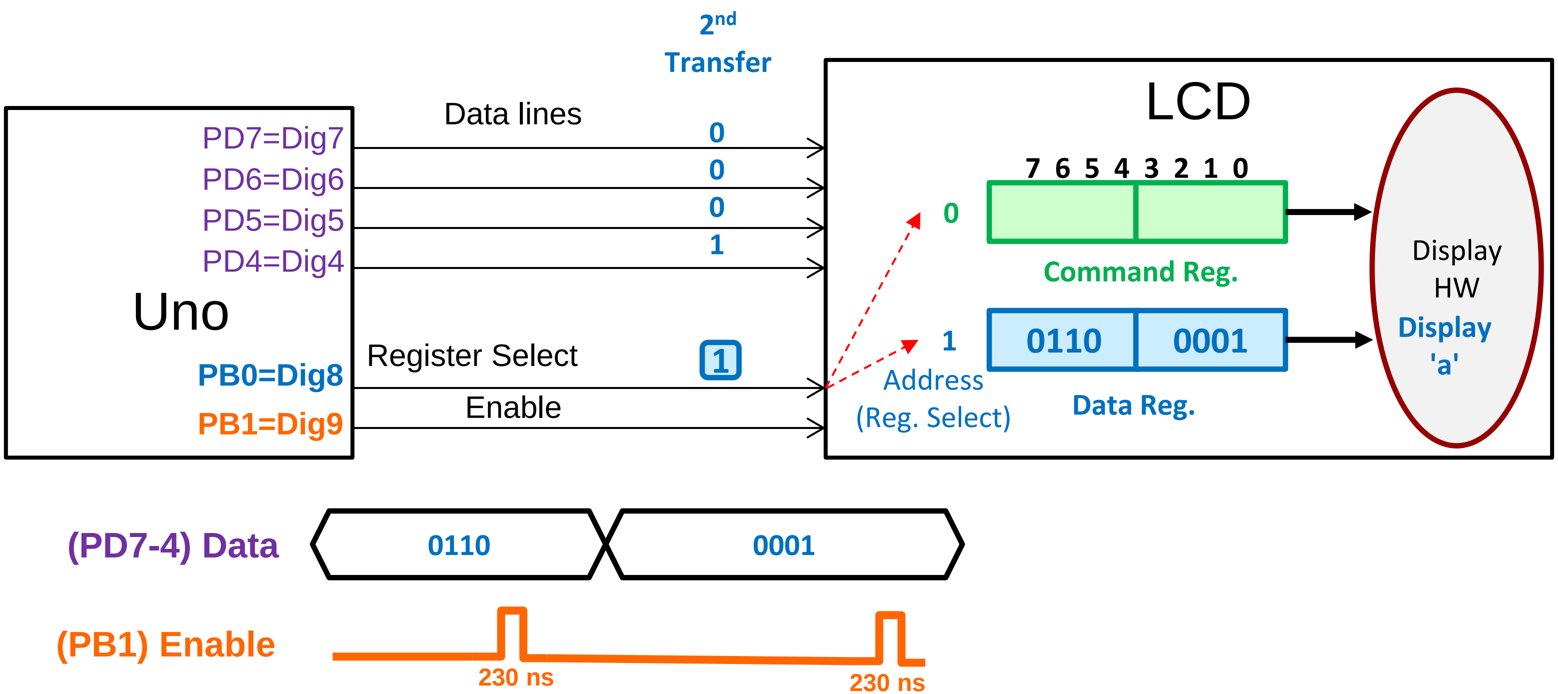
Example (1)

- To send the 8-bit ASCII code for an `'a' (0x61)`, use digital I/O to
  - Set RS=1 (destination as the **data** register)
  - First, send the **upper** four bits of the ASCII code: $6 = 0110_2$.



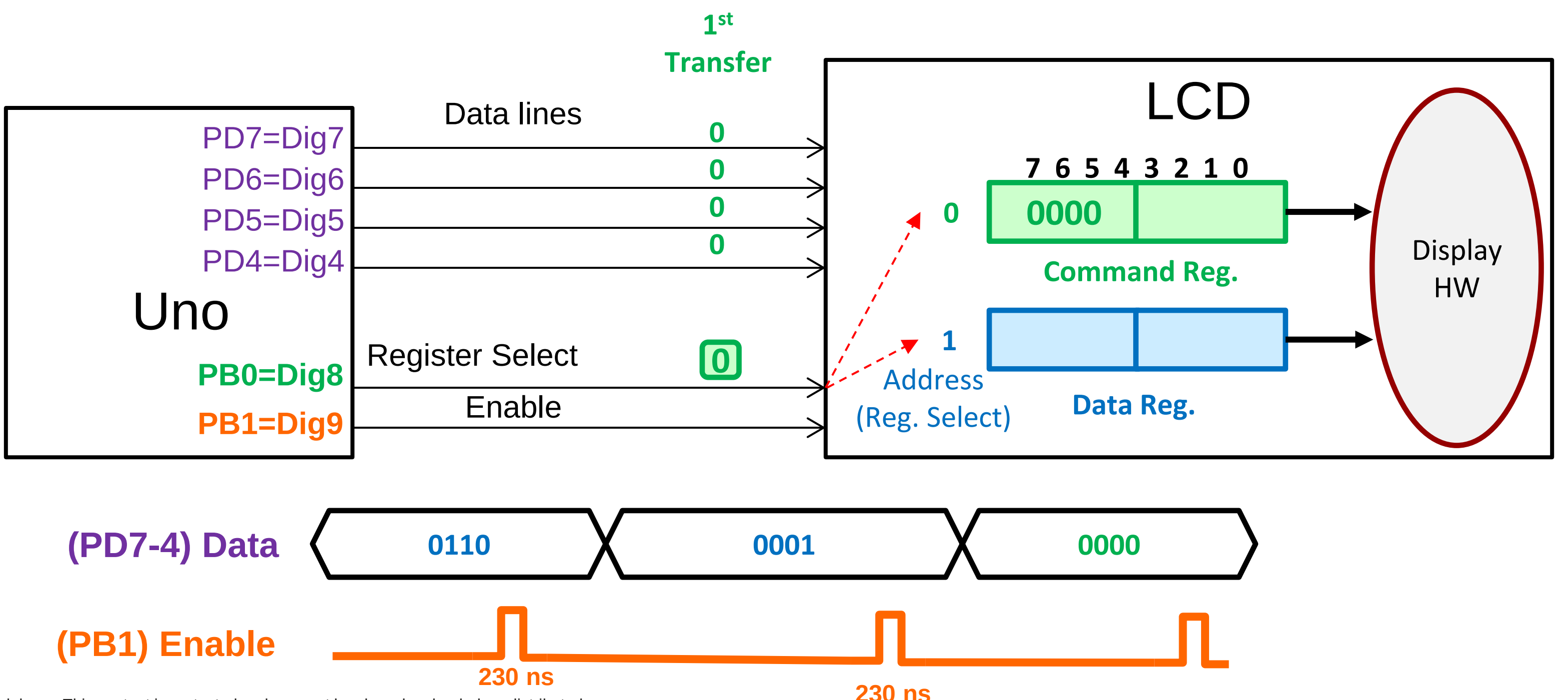
Example (2)

- To send the 8-bit ASCII code for an `'a' (0x61)`, use digital I/O to
  - Set RS=1 (destination as the **data** register)
  - Then, send the **lower** four bits of the ASCII code: 1 = 0001_2.



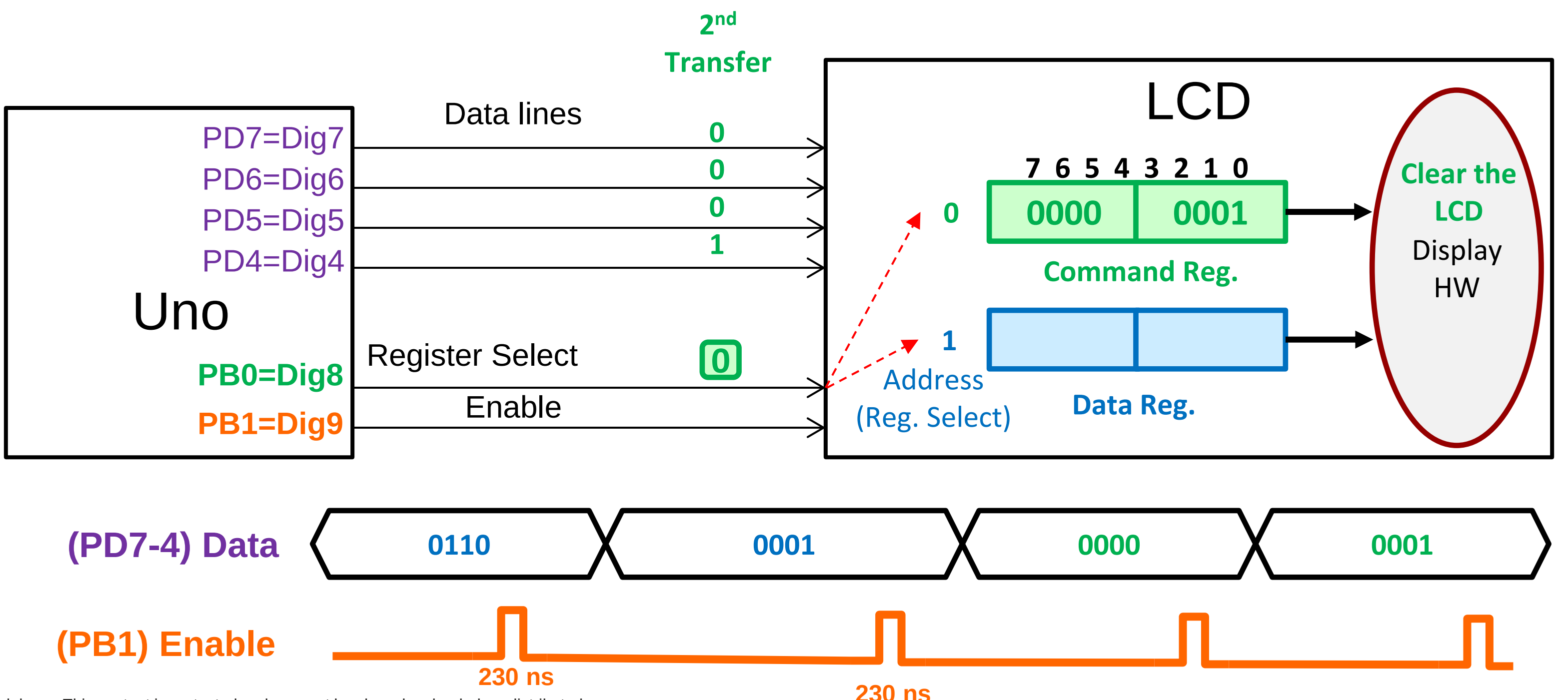
Example (3)

- To send the 8-bit command of (`0x01`) to clear the LCD, use digital I/O to:
  - Clearing RS=0 indicates the destination as the **command** register
  - First, send the **upper** four bits of the ASCII code: **0 = 0000_2**.



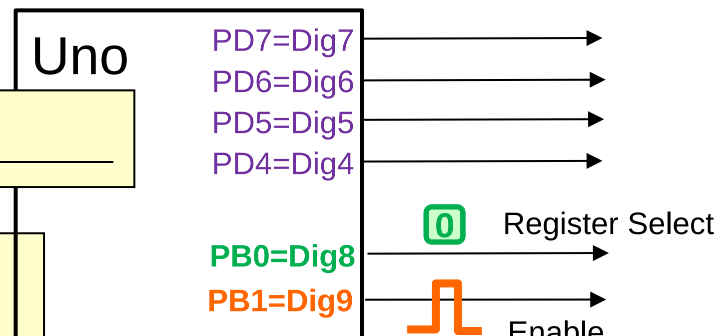
Example (4)

- To send the 8-bit command of (`0x01`) to clear the LCD, use digital I/O to:
  - Clearing RS=0 indicates the destination as the **command** register
  - Then, send the **lower** four bits of the ASCII code: **1 = 0001_2**.



Whose Job Is It? Yours!

- Recall, how are we producing the values on the RS and Data lines and the 0-1-0 transition on the E line?
- With basic digital I/O (setting and clearing PORT bits)!




```
// RS = 0
PORTB ________________
```

```
// E => 1
PORTB |= _________

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

// E => 0
PORTB &= _________
```

Note: This is not the final way we want to implement the Enable pulse (see later slides)

Command Codes and Cursor Location

- To perform the operations shown at the right, send the given code to the **command** register.
- See below for an illustration of the commands to move the cursor.

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

Col.No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex Addr	80	81	82	83	84	85	86	87	88	89	8a	8b	8c	8d	8e	8f
Row 0	E	E	1	0	9										i	s
Row 1								F	U	N						
Hex Addr	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	ca	cb	cc	cd	ce	cf

Cursor Movement

- The LCD panel uses the same controller chip for several different models with different ________________ (e.g. 2x64, 4x16, etc.) leaving gaps in the cursor addresses/locations (e.g. `0x90-0xbf`)
- As we send data the cursor location increments, but NO ____________ are added for you. You must move the cursor to the next line as needed. Otherwise, the next 48 characters will be lost until the cursor location again falls into an actual display location.

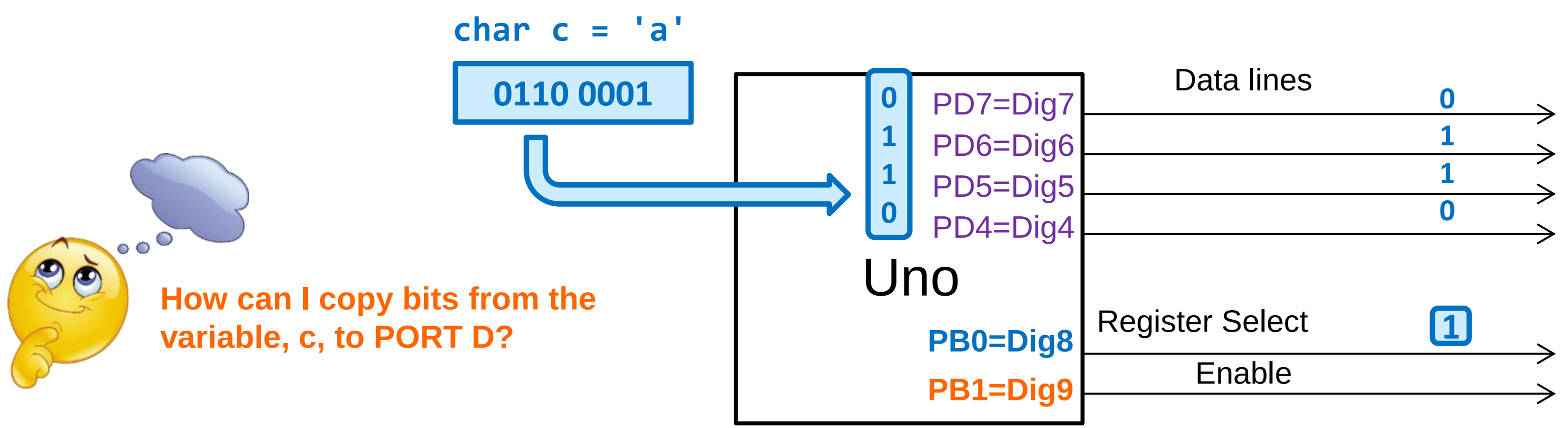
Col.No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	63
Hex Addr	80	81	82	83	84	85	86	87	88	89	8a	8b	8c	8d	8e	8f	90	…	bf
Row 0	F	O	U	R		S	C	O	R	E		A	N	D		S	E	VEN. .	N
Row 1	E	W		N	A	T	I	O	N										
Hex Addr	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	ca	cb	cc	cd	ce	cf	d0	…	ff

48 locations

IMPORTANT RECIPE: HOW TO COPY BITS

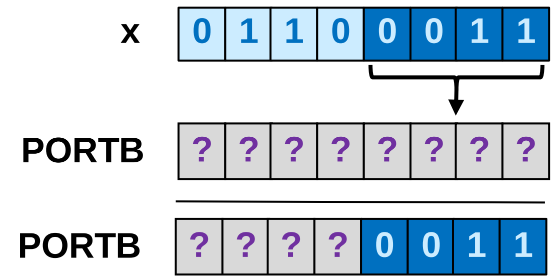
Copying Bits

- We know how to make an individual bit change to 1 or 0
- But how can we take 4-bits from some char variable (e.g. `char c`) and **copy WHATEVER those bits are** to PORTD, bits 7-4?



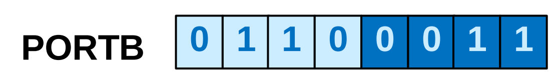
Copying Multiple Bits Recipe (1)

- Let's step aside from the LCD specifics (so that you can apply what we learn here to other generic situations, as well as to write the LCD code).
- **Goal**: Copy a portion of one variable/register into another ________ affecting the other bits
- **Example**: Copy the lower 4 bits of x into the lower 4-bits of PORTB WITHOUT affecting the other bits
- Can we simply use assignment?
  - `PORTB = x;`
- ____! Assignment changes ____ bits of PORTB.



Desired Result

Assignment copies all bits!



Undesirable result of PORTB = x; (upper bits affected)

Recipe: Copying (Aligned) Bits

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

```
#define  MASKBITS   0x0f
```

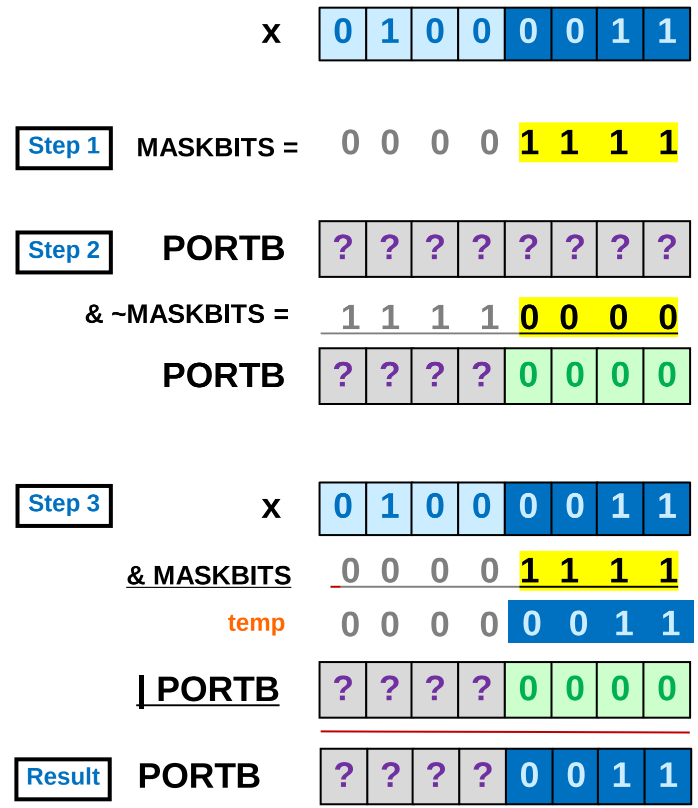
- **Step 2:** _______ those bits in the destination register using the MASK

```
________  &= ~MASKBITS
```

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

```
PORTB |= (________________);
```

temp



Do We Really Need Step 2?

- YES!! Consider if we removed it.
- **Step 1:** Define a mask that has 1's where the bits are to be copied

```
#define  MASKBITS   0x0f
```

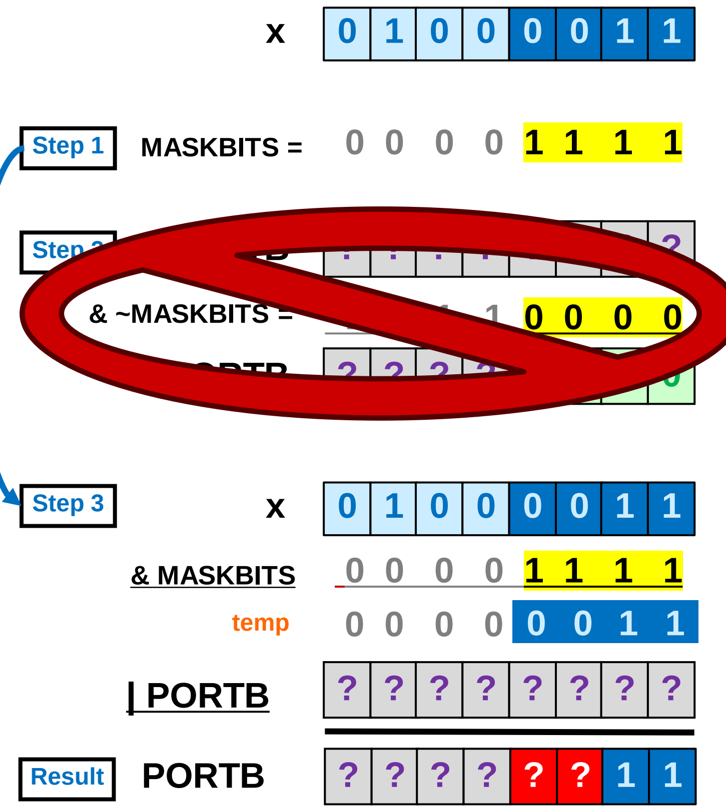
- **Step 2: Clear** those bits in the destination register using the MASK

```
PORTB &= ~MASKBITS
```

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

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

temp

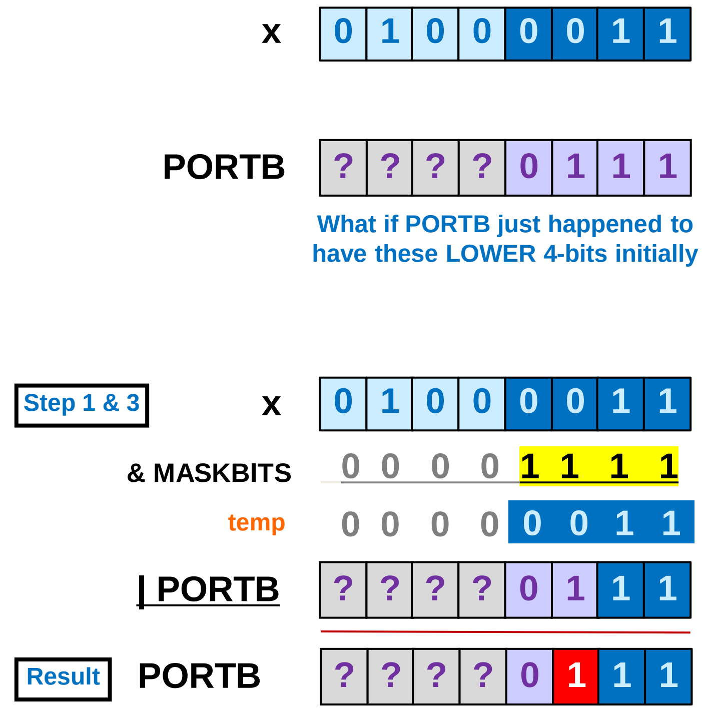


These bits may be ANYTHING and may NOT be the two 0's we want!

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

- We need step 2 to CLEAR the destination bits, because what if the destination (PORTB) already had some 1's where we wanted 0's to go…
- …Just ____________ wouldn't change the bits to ____
  - OR'ing never has the power to make bits 0
  - …and AND'ing never has the power to make 1s
- That's why we need step 2
  - Step 2: **Clear** those bits in the destination register using the MASK

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



Recipe: Copying (Shifted) Bits

- What if the source bits are in a different location than the destination
  - Ex. Copy middle 4 bits of x (bits 5:2) to upper 4 bits of PORTB (bits 7:4)
- Step 1: Define a **mask** that has 1's where the bits are to be copied

```
#define  MASKBITS   0xf0
```

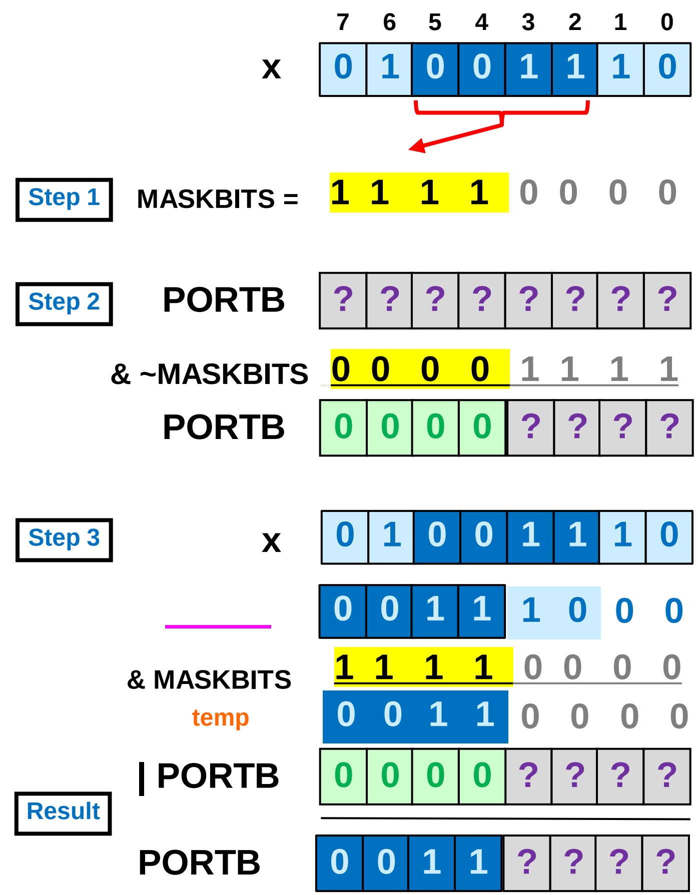
- Step 2: **Clear** those bits in the destination register using the MASK

```
PORTB &= ~MASKBITS
```

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

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

temp



LCD LAB PREPARATION DETAILS

Step 1

- Mount the LCD shield on the Uno without destroying the pins
- Download the `test.hex` file and `Makefile` from the website, and modify the `Makefile` programmer line to suite your computer.
- Run `make test` to download test program to the Uno+LCD.
- You should see a couple of lines of text on the screen similar to what is shown to the right.

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 ______ (application programming interface)
  - Once the API is written it gives other application coders a nice simple interface to do high-level tasks
  - You can then reuse this code in every future lab.
- Download the skeleton file and examine the functions outlines on the next slides

LCD API Development Overview

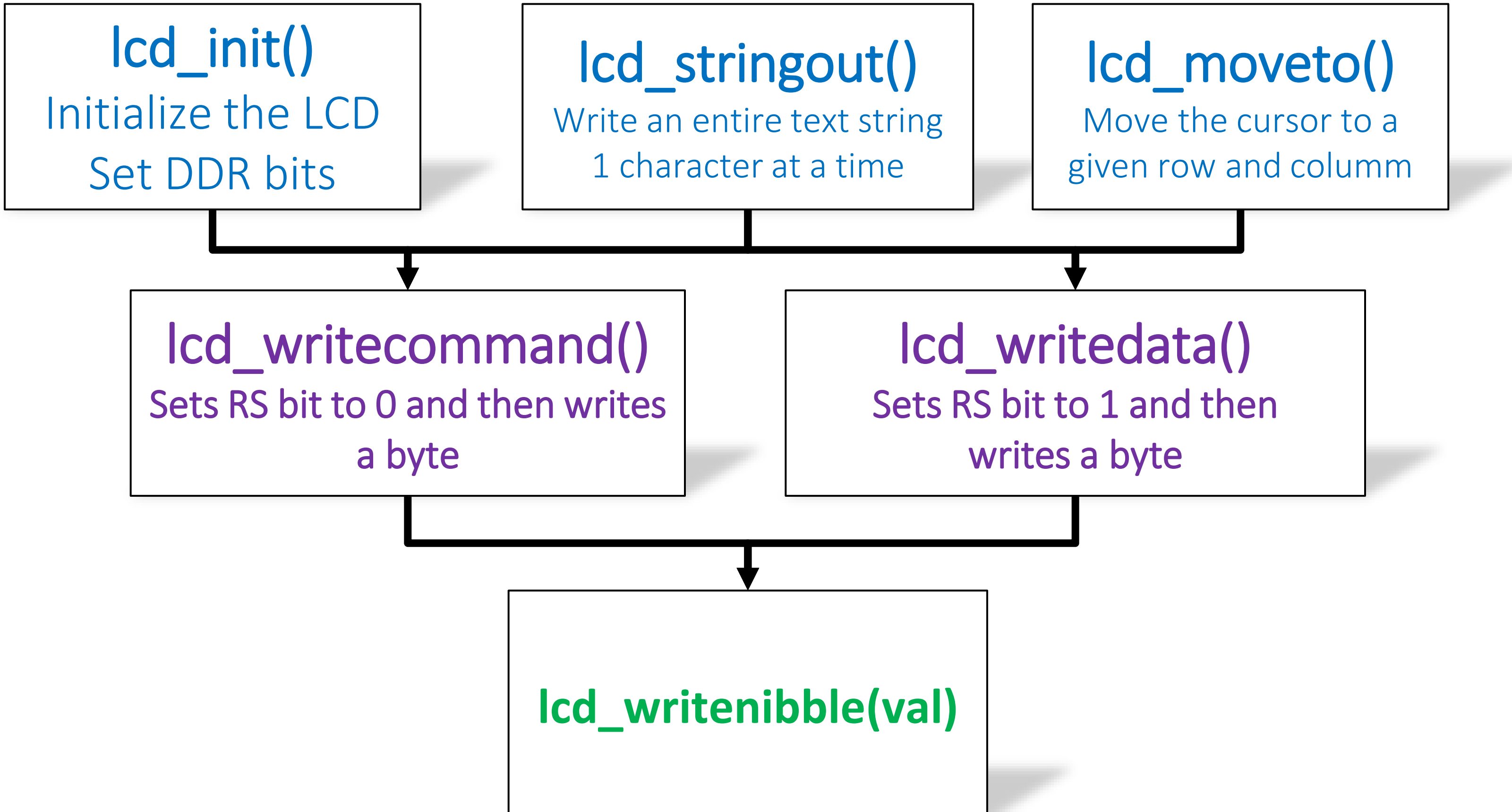
- Write the routines to control the LCD in layers
  - Top-level routines that your code or others can use: write a string to LCD, move the cursor, initialize LCD, etc.
  - Mid-level routines: Set RS appropriate to write a data byte or a command register and calls a low-level function to send groups of 4-bits
  - Low-level routines: Sets the 4 data lines and generates pulse on E to transfer
- **Goal**: Hide the ______________ about how the interface actually works from the user who only wants to put a string on the display.

```
lcd_stringout("hello");
```

```
lcd_writedata('h'); //0x68
lcd_writedata('e'); //0x65
...
lcd_writedata('o'); //0x6f
```

```
// Send 'h'
lcd_writenibble(0x6?);
lcd_writenibble(0x8?);
// Send 'e'
lcd_writenibble(0x6?);
lcd_writenibble(0x5?);
...
```

Code Organization



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)
    - **Use the recipe for copying bits provided earlier**
    - Produces a 0-1-0 transition on the Enable signal
  - Must be consistent with mid-level routines as to which 4 bits of the input 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
  - Calls: `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
  - Calls: `lcd_writenibble()`

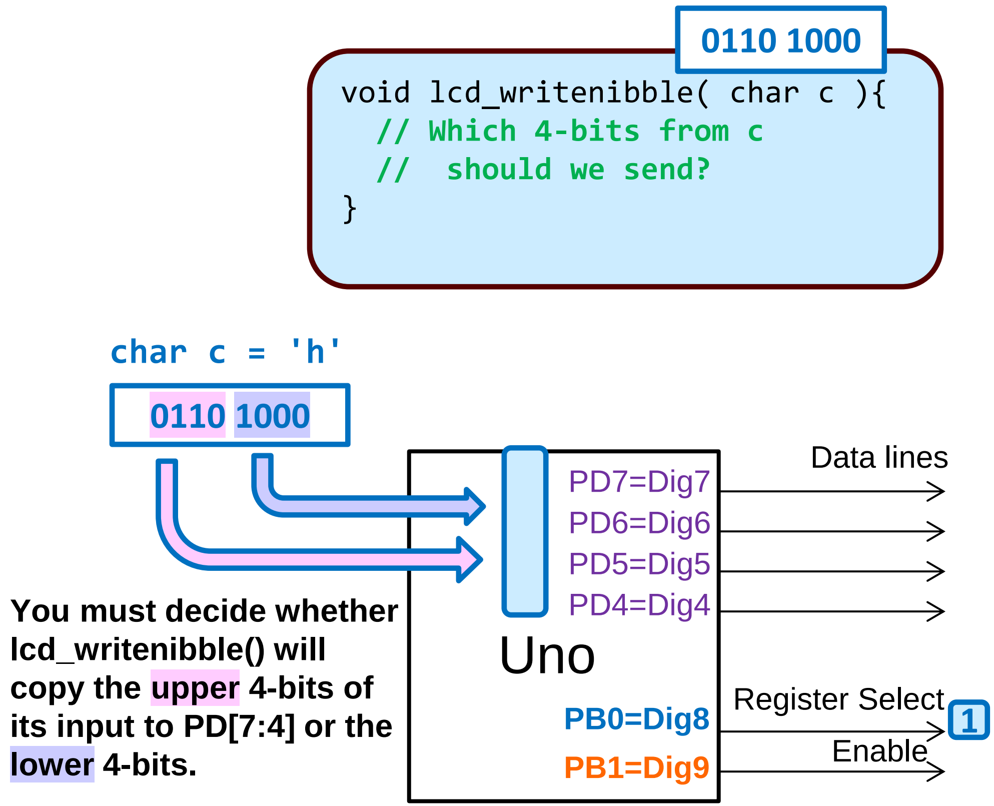
This will be your challenge to write these two functions in lab!

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()`

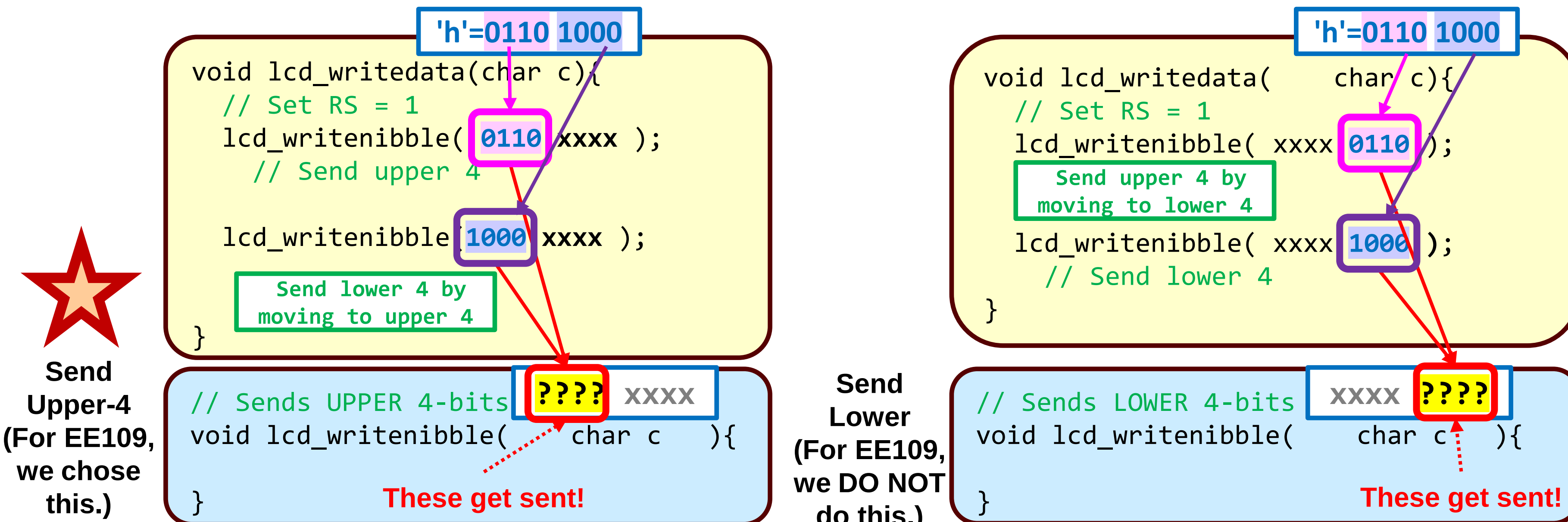
A Choice: Upper-4 or Lower-4

- **Recall**: The smallest variable in C is **1-byte** = **8-bits** (i.e. char)
- When we call lcd_writenibble() we have to pass it at least 1-byte, but it only needs to copy ________ to PORTD.
- So, we must make a choice as to ___________ we assume are the ones we should copy to PORTD



Either Choice Can Work!

- **EITHER** choice is **acceptable** as long as both `lcd_writenibble()` and `lcd_writedata()`/`lcd_writecommand()` agree.
- Consider how to use **shift** operators to help align the desired bits as you pass your arguments to `lcd_writenibble()`

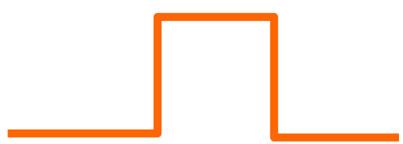


If time permits: Ensuring the Enable pulse is long enough

A STORY: THE DEVIL IN THE DETAILS…

Not That Long Ago...

- At the dawn of EE109, Prof. Weber and Redekopp put together the LCD Lab
- The lab required students to generate the **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.



Not That Long Ago…

- Students were told how to generate the pulse with that code
- But NOBODY's LCD would work?!?
  - Confusion abounded! Professors were perplexed! Students were frustrated!
  - Rumors circulated that the E pulse had to be made longer by putting a delay in the code that generated it.
  - Don't Guess!
- It was time to read the manual (at least a little bit).

Making Things Work Together

- 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.

Making Things Work Together

- Check the LCD controller datasheet

Timing Characteristics

PW_{EH} = Pulse Width Enable High

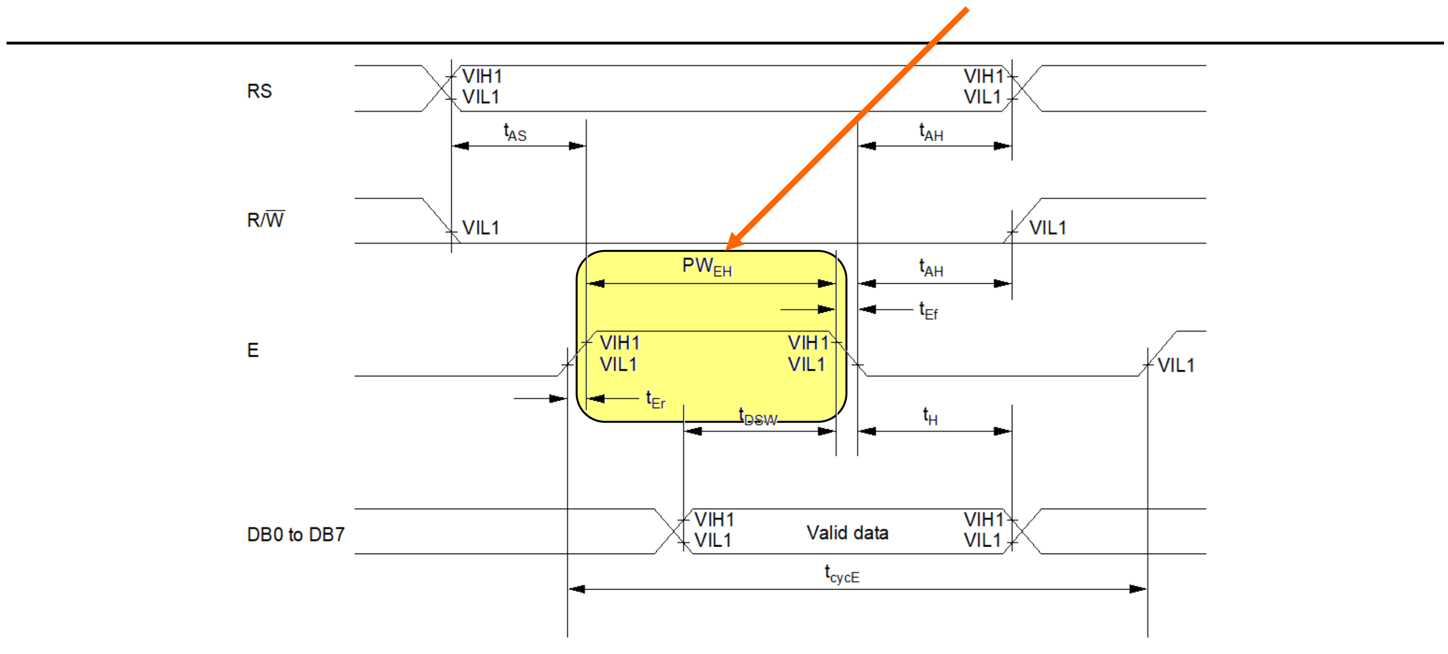
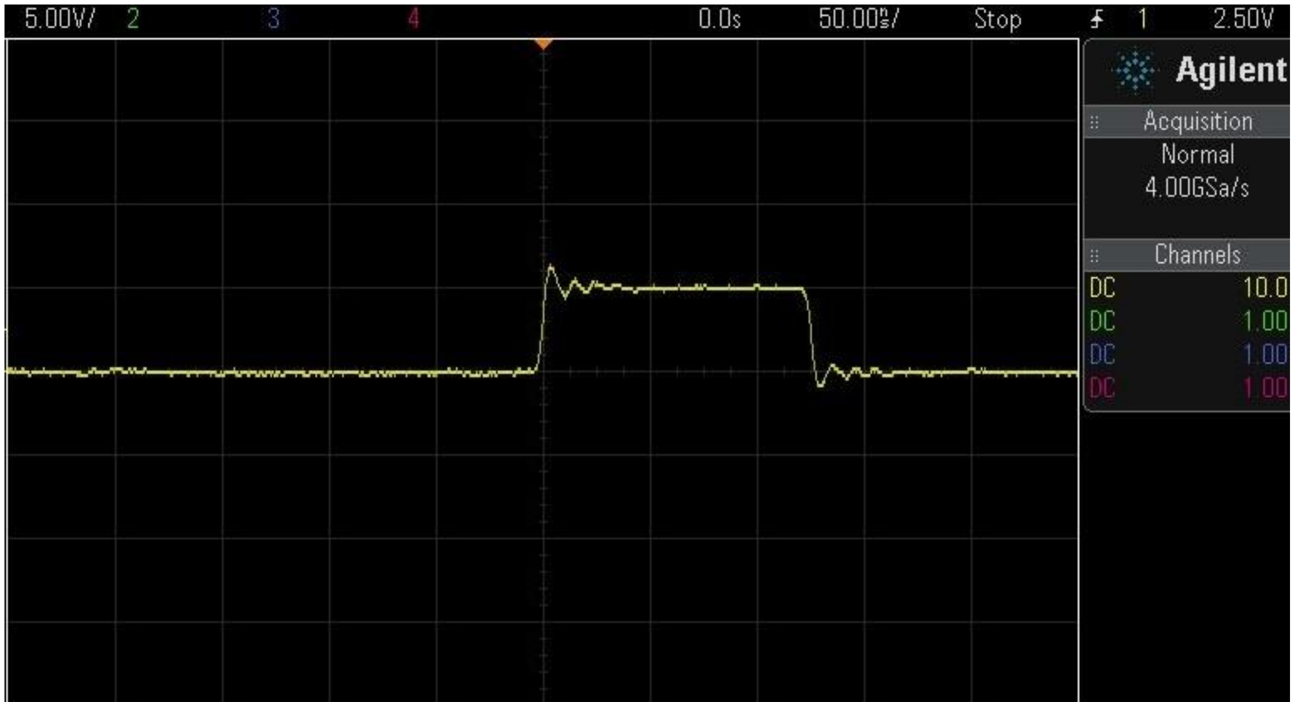


Figure 27 Write Operation

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

Verify With the Oscilloscope

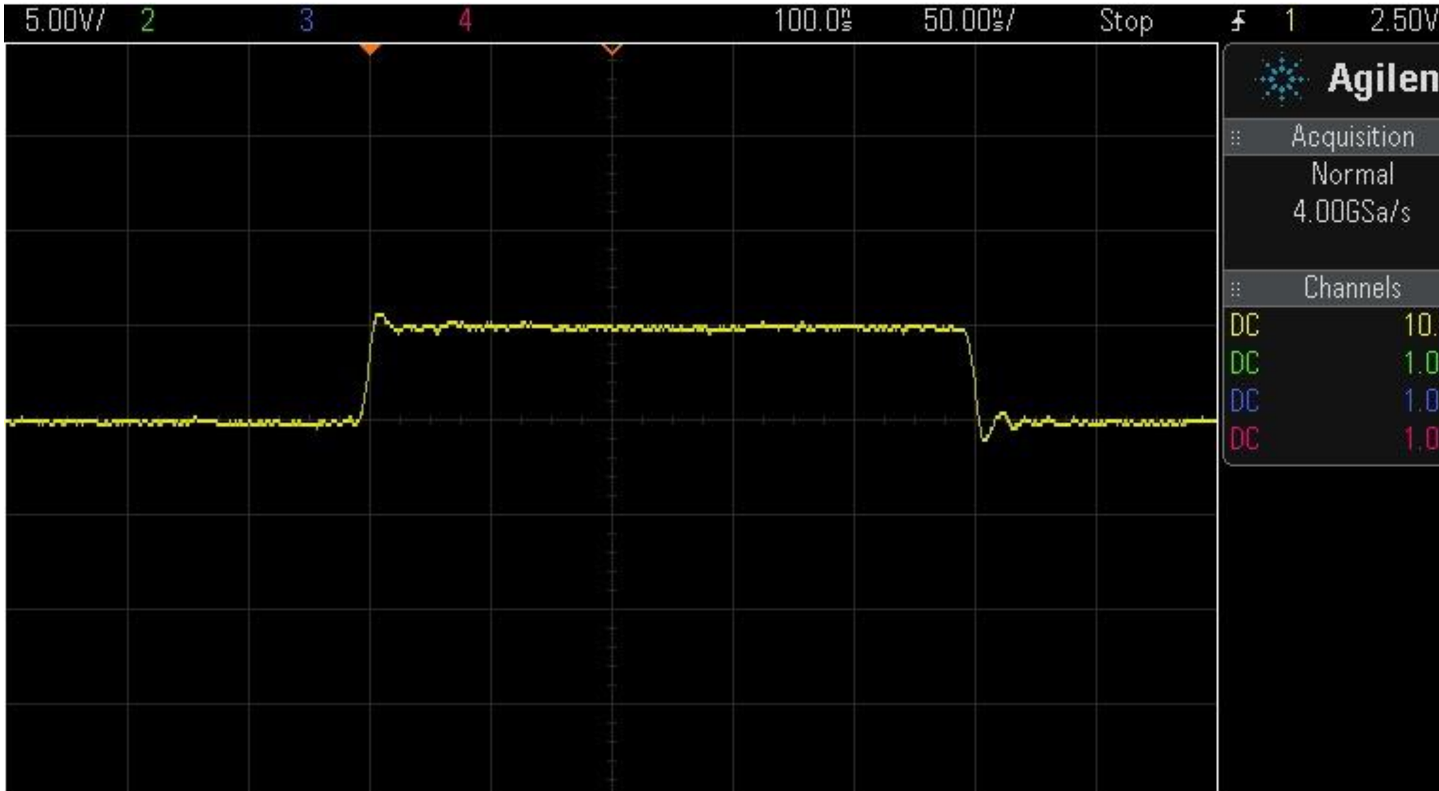


Extend the Pulse

- 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
```

A Working Pulse Length!



Extending the Pulse (The "Geeky" Way)

- Use the "asm" compiler directive to embed low level assembly code within the C code.
- The AVR assembly instruction "NOP" (no-operation) does nothing and takes 1 cycle to do it.

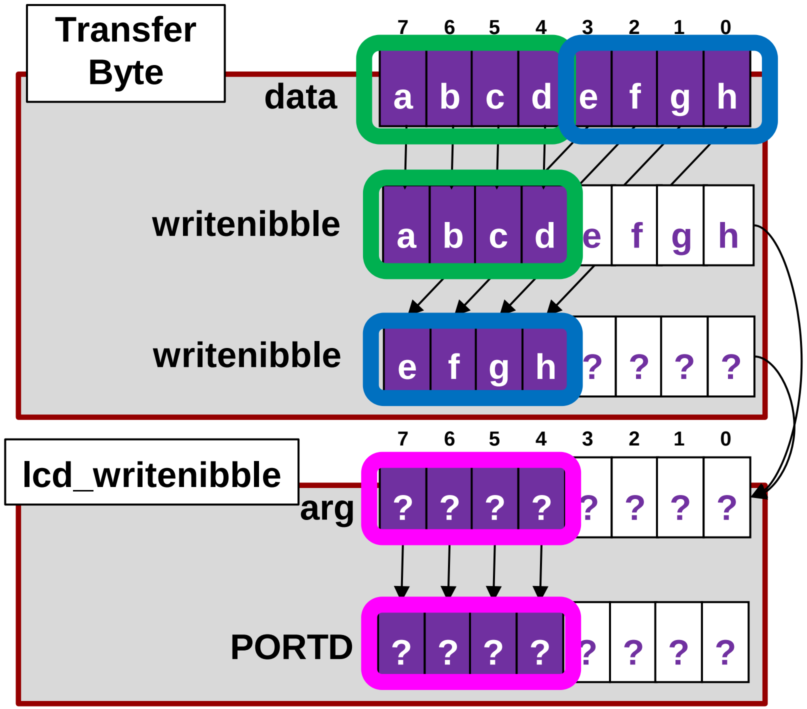
```
PORTB |= (1 << PB1);          // Set E to 1
asm("nop"::);                 // NOP delays another 62.5ns
asm("nop"::);
PORTB &= ~(1 << PB1);         // Clear E to 0
```

Read the Manual

- 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)

BACKUP

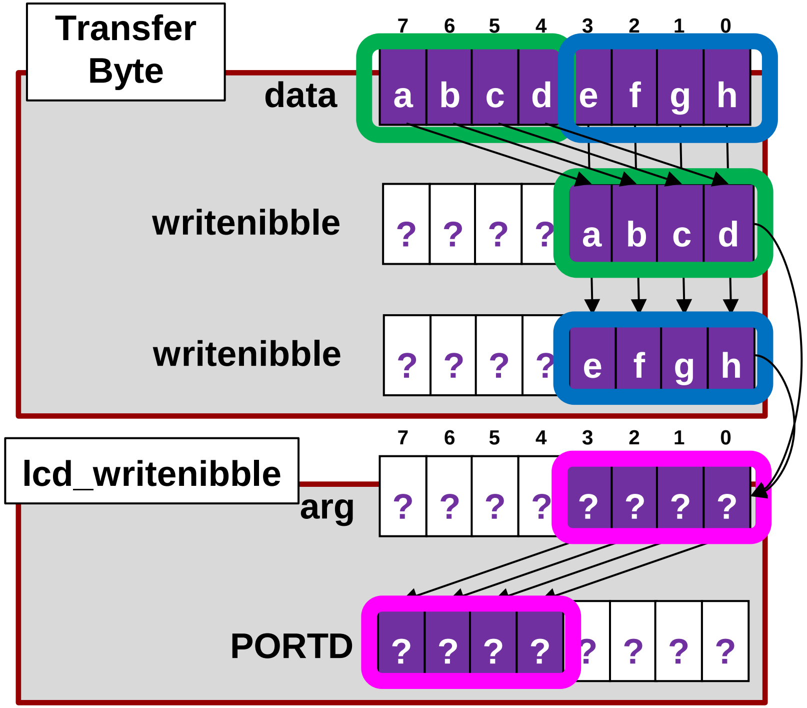
Coding a Byte Transfer to the LCD



Implementation where lcd_writenibble expects data in the upper 4 bits.

- Writing a byte requires two transfers of 4 bits from different physical locations
  - First the **upper 4** and then the **lower 4**
- Since we want only 1 version of `lcd_writenibble` and the smallest argument we can pass is 8-bits, it must assume a **specific group of 4-bits are the desired bits to transfer in the argument**
- `lcd_writedata()` and `lcd_writecommand()` must adjust to place the desired bits in the correct location that `lcd_writenibble()` expects

Coding a Byte Transfer to the LCD



Implementation where lcd_writenibble expects data in the lower 4 bits.

- Writing a byte requires two transfers of 4 bits from different physical locations
  - First the **upper 4** and then the **lower 4**
- Since we want only 1 version of `lcd_writenibble` and the smallest argument we can pass is 8-bits, it must assume a **specific group of 4-bits are the desired bits to transfer in the argument**
- `lcd_writedata()` and `lcd_writecommand()` must adjust to place the desired bits in the correct location that `lcd_writenibble()` expects

Either Choice Can Work!

- **EITHER** choice is **acceptable** as long as both `lcd_writenibble()` and `lcd_writedata()/lcd_writecommand()` agree.
- Consider how to use **shift** operators to help align the desired bits as you pass your arguments to `lcd_writenibble()`