Unit 4

Microcontrollers (Arduino) Overview

Digital I/O
Introduction

• The primary way that software controls hardware is by manipulating individual bits

• We need to learn how to:
  – Set a bit to a 1
  – Clear a bit to a 0
  – Check the value of a given bit (is it 0 or 1)

• Because computers do not access anything smaller than a byte (8-bits) we must use logic operations to manipulate individual bits within a byte
Using software to perform logic on individual (or groups) of bits

BIT FIDDLING
Numbers in Other Bases in C/C++

• Suppose we want to place the binary value 00111010 into a char variable, v [i.e. `char v;`]
  – We could convert to decimal on our own \( (58_{10}) \)
    \[ v = 58; \]
  – All compilers support hexadecimal using the '0x' prefix
    \[ v = 0x3a; \]
  – Our Arduino compiler supports binary using the '0b' prefix
    \[ v = 0b00111010; \]

• Important note: Compilers convert EVERYTHING to equivalent binary. The 3 alternatives above are equivalent because the compiler will take all 3 and place 00111010 in memory.
  – Use whichever base makes the most sense in any given situation
  – *It is your (the programmer's) choice*...the compiler will end up converting to binary once it is compiled
Modifying Individual Bits

• Suppose we want to change only a single bit (or a few bits) in a variable [i.e. `char v;`] without changing the other bits
  – Set the LSB of v to 1 w/o affecting other bits
    • Would this work? `v = 1;`
  – Set the upper 4 bits of v to 1111 w/o affecting other bits
    • Would this work? `v = 0xf0;`
  – Clear the lower 2 bits of v to 00 w/o affecting other bits
    • Would this work? `v = 0;`
  – No!!! Assignment changes ALL bits in a variable

• Because the smallest unit of data in computers is usually a byte, manipulating individual bits requires us to use BITWISE OPERATIONS.
  – Use AND operations to clear individual bits to 0
  – Use OR operations to set individual bits to 1
  – Use XOR operations to invert bits
  – Use AND to check a bit(s) value in a register
Using Logic to Change Bits

- ANDs can be used to control whether a bit passes unchanged or results in a '0'
- ORs can be used to control whether a bit passes unchanged or results in a '1'
- XORs can be used to control whether a bit passes unchanged or is inverted

You already knew the above ideas. It is just T1-T3.
Bitwise Operations

• The C AND, OR, XOR, NOT bitwise operations perform the operation on each pair of bits of 2 numbers

<table>
<thead>
<tr>
<th>0xF0</th>
<th>AND</th>
<th>0x3C</th>
<th>OR</th>
<th>0xFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0000</td>
<td>AND</td>
<td>0011 1100</td>
<td>OR</td>
<td>1111 1100</td>
</tr>
<tr>
<td>~0xF0</td>
<td>NOT</td>
<td>0x0F</td>
<td>NOT</td>
<td>1111 0000</td>
</tr>
</tbody>
</table>

```c
#include <stdio.h> // C-Library
    // for printf()

int main()
{
    char a = 0xf0;
    char b = 0x3c;
    printf("a & b = %x\n", a & b);
    printf("a | b = %x\n", a | b);
    printf("a ^ b = %x\n", a ^ b);
    printf("~a = %x\n", ~a);
    return 0;
}
```

C bitwise operators:

& = AND
| = OR
^ = XOR
~ = NOT
Bitwise Operations & Masking

• Bitwise operations are often used for "bit fiddling"
  – Change the value of a bit in a register w/o affecting other bits
• Determine appropriate constant bit patterns (aka masks) that will change some bits while leaving others unchanged
  • Masks can be written in binary, hex, or even decimal (it is the programmer's choice...whatever is easiest)
• Examples (Assume an 8-bit variable, v)
  – Clear the LSB to '0' w/o affecting other bits
    • \( v = v \& 0xfe; \) or equivalently
    • \( v = v \& \sim(0x01); \)
  – Set the MSB to '1' w/o affecting other bits
    • \( v = v \mid 0x80; \)
  – Flip the LS 4-bits w/o affecting other bits
    • \( v = v \wedge 0x0f; \)
Changing Register Bits

• Some practice problems:
  – Set bit 0 of \( v \) to a ‘1’
    \[
    v = v \mid 0x01;
    \]
  – Clear the 4 upper bits in \( v \) to ‘0’
    \[
    v = v \& 0x0f;
    \]
  – Flip bits 4 and 5 in \( v \)
    \[
    v = v \^ 0b00110000;
    \]

Note: It is the programmer's choice of writing the "mask" constant in binary, hex, or decimal. However, hex is usually preferable (avoids mistakes of missing a bit in binary and easier than converting to decimal).
Checking Register Bits

- To check for a given set of bits we use a bitwise-AND to isolate just those bits
  - The result will then have 0's in the bit locations not of interest
  - The result will keep the bit values of interest

- Examples
  - Check if bit 7 of \( v = '1' \)
    
    
    
    ```
    if ( (v & 0x80) == 0x80) { code }  
    or
    if (v & 0x80) { code }  
    ```
  - Check if bit 2 of \( v = '0' \)
    
    
    
    ```
    if ( (v & 0x04) == 0x00) { code }  
    or
    if ( ! (v & 0x04) ) { code }  
    ```
  - Check if bit 2:0 of \( v = "101" \)
    
    
    
    ```
    if ( (v & 0b00000111) == 0b00000101) {..}  
    ```
  - Check if bit 5-4 of \( v = "01" \)
    
    
    
    ```
    if ( (v & 0x30) == 0x10) { code }  
    ```
Short Notation for Operations

- In C, assignment statements of the form
  \[ x = x \text{ op} y; \]
- Can be shortened to
  \[ x \text{ op} = y; \]
- Example:
  \[ x = x + 1; \] can be written as \[ x += 1; \]
- The preceding operations can be written as
  \[ v| = 0x01; \]
  \[ v & = 0x0f; \]
  \[ v ^ = 0b00110000; \]
Logical vs. Bitwise Operations

- The C language has two types of logic operations
  - Logical and Bitwise

- Logical Operators (&&, ||, !)
  - Operate on the logical value of a FULL variable (char, int, etc.) interpreting that value as either True (non-zero) or False (zero)
  - char x = 1, y = 2, z = x && y;
    • Result is z = 1; Why?
  - char x = 1;
    if(!x) { /* will NOT execute since !x = !true = false */ }

- Bitwise Operators (&, |, ^, ~)
  - Operate on the logical value of INDIVIDUAL bits in a variable
  - char x = 1, y = 2, z = x & y;
    • Result is z = 0; Why?
  - char x = 1;
    if(~x) { /* will execute since ~x = 0xfe = non-zero = true */ }
ARDUINO BOARD INTRO
Arduino Uno

- The Arduino Uno is a microcomputer development board based on the Atmel ATmega328P 8-bit processor.
- Most microcomputer manufacturers (Atmel, Freescale, etc.) produce small PC boards with their chips on them for engineers to experiment with and hopefully generate sales of the product.

http://arduino.cc/en/Main/ArduinoBoardUno

Printed circuit (PC) board with processor and other circuits for programming the system and interfacing other devices

Atmega328P 8-bit processor
Arduino Uno

- Arduino
  - An Italian company
  - They make numerous boards with different processors
  - Hardware and software are open source.
  - Very popular with hobbyists, due in a large part to their low cost.

http://arduino.cc/en/Main/Products
Arduino Uno

- What’s on an Arduino Uno board?

- Atmel ATmega328P microcontroller
- 16MHz oscillator (i.e. clock signal generator)
- USB interface
- Power connector (for use when not connected to USB)
- Reset button
- Connectors for I/O lines D0 – D13
- I/O lines A0 – A5
- Power and ground pins
Arduino Uno

• Arduino Unos can be stacked with "shield" boards to add additional capabilities (Ethernet, wireless, D/A, LCDs, sensors, motor control, etc.)
Controlling the pins of the Arduino to be digital inputs and outputs

ARDUINO DIGITAL I/O
Flashback to Week 1

- Recall the computer interacts with any input or output (I/O) device by simply doing reads/writes to the memory locations (often called registers) in the I/O interfaces...
- The Arduino has many of these I/O interfaces all connected via the data bus
Atmel ATmega328P

- The Arduino Uno is based on an Atmel ATmega328P 8-bit microcontroller
  - 32kb of FLASH ROM
  - 2048 bytes of RAM
  - 23 I/O lines
  - 3 timer/counters
  - Serial/SPI/I²C interfaces
  - A/D converter
Arduino Digital I/O

• ATmega328P has 23 pins on the chip that can be connected to other devices (switches, LEDs, motors, etc.)
  – Other members of the ATmega family may have more or less lines.
  – The Arduino Uno can make use of only 20 of these lines.
• Each pin can be used as a digital input or a digital output
  – **For output pins:** Your code determines what value ('1' or '0') appears on the pin and can connect to another component
  – **For input pins:** Your code senses/reads what value another device is putting on the pin

Main Point: Individual pins on the Arduino can be used as inputs OR outputs
Groups B, C and D

- The Arduino provides around 20 separate digital input/output bits that we can use to interface to external devices
- Recall computers don't access individual bits but instead the byte (8-bits) is the smallest unit of access
- Thus to deal with our digital inputs we will put the bits into 3 groups: Group B, C, and D
  - We often refer to these groups as "ports" but you'll see that "port" is used in multiple places so we'll generally use "group"

Software to Arduino Name Mapping
Group B bit5-bit0 = DIG13-DIG8
Group C bit5-bit0 = A5-A0
Group D bit7-bit0 = DIG7-DIG0
Arduino Port/Pin Mapping

- Since computers usually deal with groups of 8-bits (a.k.a. a byte), all of the 20 I/O pins are split into **three 8-bit I/O groups (B, C and D)**
  - The avr-gcc software (SW) and the Arduino hardware use different names to refer to the bits within each port

<table>
<thead>
<tr>
<th>SW</th>
<th>Arduino</th>
<th>SW</th>
<th>Arduino</th>
<th>SW</th>
<th>Arduino</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortB, bit0</td>
<td>DIG8</td>
<td>PortC, bit0</td>
<td>AN0</td>
<td>PortD, bit0</td>
<td>DIG0</td>
</tr>
<tr>
<td>PortB, bit1</td>
<td>DIG9</td>
<td>PortC, bit1</td>
<td>AN1</td>
<td>PortD, bit1</td>
<td>DIG1</td>
</tr>
<tr>
<td>PortB, bit2</td>
<td>DIG10</td>
<td>PortC, bit2</td>
<td>AN2</td>
<td>PortD, bit2</td>
<td>DIG2</td>
</tr>
<tr>
<td>PortB, bit3</td>
<td>DIG11</td>
<td>PortC, bit3</td>
<td>AN3</td>
<td>PortD, bit3</td>
<td>DIG3</td>
</tr>
<tr>
<td>PortB, bit4</td>
<td>DIG12</td>
<td>PortC, bit4</td>
<td>AN4</td>
<td>PortD, bit4</td>
<td>DIG4</td>
</tr>
<tr>
<td>PortB, bit5</td>
<td>DIG13</td>
<td>PortC, bit5</td>
<td>AN5</td>
<td>PortD, bit5</td>
<td>DIG5</td>
</tr>
<tr>
<td>PortB, bit6</td>
<td>Clock1</td>
<td>PortC, bit6</td>
<td>Reset</td>
<td>PortD, bit6</td>
<td>DIG6</td>
</tr>
<tr>
<td>Clock2</td>
<td>PortC, bit6</td>
<td>Reset (don't use)</td>
<td>PortD, bit7</td>
<td>DIG7</td>
<td></td>
</tr>
<tr>
<td>(don't use)</td>
<td>DIG1</td>
<td>(don't use)</td>
<td>DIG2</td>
<td>DIG3</td>
<td></td>
</tr>
</tbody>
</table>

**Main Point:** Each pin has a name the software uses (Portx) and a name used on the Arduino circuit board (Anx or DIGx)
Using Ports to Interface to HW

- The I/O ports (i.e. groups of pins) are the middle men between your software program and the physical devices connected to the chip.

- Your program is responsible for managing these ports (groups of I/O pins) in order to make things happen on the outside.

```c
#include <avr/io.h>

int main()
{
    // check input Group C
    // bit 0 value

    // set output Group B
    // bit 2 to Logic 1=5V

    // ...
}
```

- Arduino A0 ⇔ PortC bit 0
- Arduino D10 ⇔ PortB bit 2
- A button
- An LED

Logic 1 = 5V
Logic 0 = 0V
Overview

• In the next few slides you will learn
  – What your software needs to do to setup the pins for use as digital inputs and/or outputs
  – To set bits (to 1) and clear bits (to 0) using bitwise operations (AND, OR, NOT) to control individual I/O pins
  – How to do it in a readable syntax using shift operators (<<, >>)

• Don't be worried if it doesn't make sense the first time...listen, try to make sense of it, and ask a lot of questions.

```c
#include <avr/io.h>
int main()
{
    // check input Group C
    // bit 0 value
    // set output Group B
    // bit 2 to Logic 1=5V
    // ...
}
```

What is the actual code we would write to accomplish these task?

We'll answer that through the next few slides.
Controlling I/O Groups/Ports

- Each port (B, C, and D) has 3 registers in the µC associated with it that control the operation
  - Each bit in the register controls something about the corresponding I/O bit.
  - Data Direction Register (DDRB, DDRC, DDRD)
  - Port Output Register (PORTB, PORTC, PORTD)
  - Port Input Register (PINB, PINC, PIND)

- You'll write a program that sets these bits to 1's or 0's as necessary

What you store in the register bits below affect how the pins on the chip operates.
Register 1: Data Direction Register

- DDRx (Data direction register) \([x=B,C,D...DDRB, DDRC, DDRD]\)
  - Controls whether pins on the chip act as inputs or outputs.
  - Example: If DDRB[5] = 0 -> PB5 (Port B bit 5 = DIG13 pin) will be used as **input**
  - Example: If DDRB[5] = 1 -> PB5 (Port B bit 5) will be used as **output**
  - All I/O lines start out as inputs when the µC is reset or powered up.

**DDRD**

<table>
<thead>
<tr>
<th>DDRD7</th>
<th>DDRD6</th>
<th>DDRD5</th>
<th>DDRD4</th>
<th>DDRD3</th>
<th>DDRD2</th>
<th>DDRD1</th>
<th>DDRD0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**DDRB**

<table>
<thead>
<tr>
<th>DDB7</th>
<th>DDB6</th>
<th>DDB5</th>
<th>DDB4</th>
<th>DDB3</th>
<th>DDB2</th>
<th>DDB1</th>
<th>DDB0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- **PD[7:4] = INPUT**
- **PD[3:0] = OUTPUT**
- **PB[5] = OUTPUT**
- **PB[4:0] = INPUT**

Consider a leaf BLOWER / VACCUM. There must be a switch to select if you want it to blow (output) or produce suction (input)...DDR register is that "switch"

*Notation: [7:4] means bit7-bit4*

Register 2: PORT (Pin-Output) Register

- PORTx (Primarily used if port X is configured as an output)
  - When a pin is used as an output (DDRx[n] = 1), the corresponding bit in PORTx[n] determines the value/voltage of that pin.
  - E.g. By placing a '1' in port B bit 5, pin PB5 will output a high voltage

**Main Point:** For pins configured as outputs, the values you put in the PORT register will be the output voltages
Register 3: PIN (Pin-Input) Register

• PINx[n] (Used if PORT is configured as an input)
  – When a bit is an input (DDxn=0), getting the bit from PINxn reflects the current value at the corresponding input pin

• The program doesn’t have to do anything special to read the digital signals into the PIN register, just use the register name
  – if(PIND == 0x00) // check if all the signals coming into port D are 0's
  – char val = PINB; // read and save all 8 signals coming into port B in a variable 'val'.
  – Referencing PINx produces a ______________ (at the instant the line of code is execute) of the bit values coming in those 8 pins; it does not constantly ____________ the input bits

• Programs must read the full eight bits in the PIN register, but can then use bitwise logical operations to check individual bits

Main Point: For pins configured as inputs, referencing the PINx register samples the input voltages at all the pins
Review of Accessing Control Registers in C

• Control registers have names and act just like variables in a C program

• To put values into a control register you can assign to them like any C variable or perform bitwise operations
  – `DDRD = 0xff;` // 0b11111111 or 255
  – `DDRB = 255;`
  – `PORTD |= 0xc0;` // 0b11000000 or 192
  – `PORTD |= 0b01110000;`

• To read the value of a control register you can write expressions with them
  – `unsigned char myvar = PIND;` // grabs all 8-inputs on the port D
  – `myvar = PINB & 0x0f;` // you will see this grabs just the lower 4 inputs
Practice: Changing Register Bits

• Use your knowledge of the bitwise operations to change the values of individual bits in registers without affecting the other bits in the register.
  – Set DDRB, bit 3 to a '1'
    \[
    \text{DDRB} |= 0b00001000; \quad \text{// DDRB} |= 0x08;
    \]
  – Clear the 2 upper bits in PORTC to ‘0’s
    \[
    \text{PORTC} &= 0x3f; \\
    \quad \text{// PORTC} &= ~(0b11000000)
    \]
  – Flip bits 7 and 1 in DDRC
    \[
    \text{DDRC} ^= 0b10000010; \quad \text{// DDRC} ^= 0x82;
    \]
  – Check if PIND, bit 4 = '1'
    \[
    \text{if} (\text{PIND} \& 0x10) \{ \text{code} \} 
    \]
EXAMPLES
LED Outputs Review

• Recall we can connect LEDs to the outputs of a digital signal
  – The digital output value that will turn the LED on varies based on how we wire the LED

• Be sure to use a current-limiting resistor (few hundred ohms ~330-470 ohm)

---

Option 1

LED is on when gate outputs '1'

Option 2

LED is on when gate outputs '0'

Can be discrete gate or Arduino output pin

Can be discrete gate or Arduino output pin
Switch & Button Input Review

• Recall: Switches/buttons alone do not produce 1's and 0's; they must be connected to voltage sources

• Preferred connection:
  – Connect one side of switch to GND (ground)
  – Connect other side of switch to digital input AND to a pull-up resistor (around 10Kohms) whose other side is connected to Vdd

• Switch/button will produce a:
  – 0 when pressed
  – 1 when open (not-pressed)

Main Point: Buttons & switches should have GND connected to one side & a pull-up resistor on the other
Blinking an LED

- Hardware and software to make an LED connected to D7 blink

```c
#include<avr/io.h>
#include<util/delay.h>

int main()
{
    // Init. D7 to output
    DDRD |= 0x80;

    // Repeat forever
    while(1){
        // PD7 = 1 (LED on)
        PORTD |= 0x80;
        _delay_ms(500);

        // PD7 = 0 (LED off)
        PORTD &= ~(0x80);
        _delay_ms(500);
    }
    // Never reached
    return 0;
}
```
Turning an LED on/off with PB

- Hardware to turn an LED connected to D7 on/off when pressing a pushbutton connected to D4
Turning on an LED from a Button

• Note: When the button is pressed a '0' is produced at the PD4 input

```c
#include<avr/io.h>

int main()
{
    // Init. D7 to output
    DDRD |= 0x80;
    // All pins start as input
    // on reset, so no need to
    // clear DDRD bit 4

    // Repeat forever
    while(1){
        // Is PD4 pressed?
        if( (PIND & 0x10) == 0){
            // PD7 = 1 (LED on)
            PORTD |= 0x80;
        }
        else {
            // PD7 = 0 (LED off)
            PORTD &= ~(0x80);
        }
        // Never reached
        return 0;
    }
}
```

DDRD
(starts at 0's on reset)

| 0 0 0 0 0 0 0 0 |

PIND

| ????????? |

&

| ????????? |
Pull Up Resistors

- Adding and wiring pull-up resistors for input buttons can be time consuming...
- Thankfully, each Arduino input bit has an optional internal “pull-up resistor” associated with it.
  - If the pull-up is enabled, in the absence of an input signal, the input bit will be “pulled” up to a logical one.
  - The pull-up has no effect on the input if an active signal is attached.

1) Built Separately

This pull-up resistor can be built separately on your circuit board OR there is one on each pin of the Arduino that can be enabled

2) Enabled in the Arduino

Arduino
Enabling Pull Up Resistors

- When DDRx bit n is '0' (i.e. a pin is used as input), the value in the PORTx bit n registers determines whether the internal pull-up is enabled
  - Remember, the PORT register is normally used when a pin is an output, but here its value helps enable the internal pull-up resistor

A pin being used as an input (DDR bits = 0) whose corresponding PORT bit = 1 will enable the pull up resistors on the PIN bit

<table>
<thead>
<tr>
<th>DDRD</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDD7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>DDD6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>DDD5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>DDD4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>DDD3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DDD2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DDD1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DDD0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PORTD</th>
<th>Enable Pull-Up Resistors</th>
<th>Actual output values from PD3-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTD7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PORTD6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PORTD5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PORTD4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PORTD3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PORTD2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PORTD1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PORTD0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIND</th>
<th>Made-up values read from the push-buttons (which don't require you to wire up external pull-up resistors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIND7</td>
<td>0</td>
</tr>
<tr>
<td>PIND6</td>
<td>1</td>
</tr>
<tr>
<td>PIND5</td>
<td>1</td>
</tr>
<tr>
<td>PIND4</td>
<td>0</td>
</tr>
<tr>
<td>PIND3</td>
<td>?</td>
</tr>
<tr>
<td>PIND2</td>
<td>?</td>
</tr>
<tr>
<td>PIND1</td>
<td>?</td>
</tr>
<tr>
<td>PIND0</td>
<td>?</td>
</tr>
</tbody>
</table>

PD[7:4] (connected to buttons): 0011
Using Internal Pull-up Resistors

• Let's simplify our wiring and use the internal pull-up resistors.
Turning on an LED from a Button

• Note: When the button is pressed a '0' is produced at the PD4 input

```c
#include<avr/io.h>

int main()
{
    // Init. D7 to output
    DDRD |= 0x80;

    // Enable pull-up on PD4
    PORTD |= 0x10;

    // Repeat forever
    while(1){
        // Is PD4 pressed?
        if( (PIND & 0x10) == 0){
            // PD7 = 1 (LED on)
            PORTD |= 0x80;
        } else {
            // PD7 = 0 (LED off)
            PORTD &= ~(0x80);
        }
    }
    // Never reached
    return 0;
}
```
Using "good" syntax/style when performing logic operations

FIDDLING WITH STYLE!
Code Read-ability Tip #1

- Try to replace hex and binary constants with shifted constants

```c
#include<avr/io.h>

int main()
{
    // Init. D7 to output
    DDRD |= 0x80;

    // Enable pull-up on PD4
    PORTD |= 0x10;

    // Repeat forever
    while(1){
        // Is PD4 pressed?
        if( (PIND & 0x10) == 0){
            // PD7 = 1 (LED on)
            PORTD |= 0x80;
        }
        else {
            // PD7 = 0 (LED off)
            PORTD &= ~(0x80);
        }
    }
    // Never reached
    return 0;
}
```

This syntax tells us we are putting a '1' in bit 7 or bit 4...

We will teach you what all this means in the next slides...
Shift Operations

- In C, operators '<<' and '>>' are the shift operators
  - '<<' = Left shift
  - '>>' = Right shift
- Format: `data << bit_places_to_shift_by`
- Bits shifted out and dropped on one side
- Usually (but not always) 0’s are shifted in on the other side

![Shift Operation Diagram]

- Example:
  - Right Shift by 2 bits:
    - Original `x`: 000011
    - Shifted: 00000101
  - Left Shift by 2 bits:
    - Original `x`: 001010
    - Shifted: 00010100
Another Example

- To get a 1 in a particular bit location it is easier to shift the constant 1 some number of places than try to think of the hex or binary constant.

- Suppose we want a 1 in bit location 3. Just take the value 1 and shift it 3 spots to the left.

- Suppose we want a 1 in bit location 5. Shift 1 5 spots to the left. Easier than coming up with 0x20…
Putting it All Together

- Values for working with bits can be made using the ‘<<’ shift operator
  - OK: PORTB |= 0x20;   Better: PORTB |= (1 << 5);
  - OK: DDRD |= 0x04;   Better: DDRD |= (1 << 2);
- This makes the code more readable and your intention easier to understand...
- More examples
  - Set DDRC, bit 5: DDRC |= (1 << 5)
  - Invert PORTB, bit 2: PORTB ^= (1 << 2)
  - Clear PORTD, bit 3: PORTD &= (1 << 3)
    - WRONG! Why?
  - Clear PORTD, bit 3: PORTD &= (0 << 3)
    - WRONG! Why?
  - Clear PORTD, bit 3: PORTD &= ~(1 << 3)
    - RIGHT! Why?
Clearing Bits...A Common Mistake

• When using the ‘&=’ operation to clear bits, remember to invert the bits.

• This won’t work to clear 3 to ‘0’
  – PORTD &= (1 << 3);
  – is the same as
  – PORTD &= 0b0001000;
  – which clears everything but bit 3

• Use the ‘~’ operator to complement the bits.
  – PORTD &= ~(1 << 3);
  – is the same as
  – PORTD &= 0b11110111;
  – and now 3 gets cleared.

• And NEVER use a mask of all 0's
  – PORTD &= (0 << 3); // 0 shifted by any amount is 0 in all bit places
Setting/Clearing Multiple bits

• Can combine multiple bits into one defined value
  – PORTB |= ((1 << 3) | (1 << 4) | (1 << 5));
  – is the same as PORTB |= 0b00111000
  – PORTB &= ~ ((1 << 3) | (1 << 4) | (1 << 5));
  – is the same as PORTB &= 0b11000111;
DEBOUNCING SWITCHES
Counting Presses

- Consider trying to build a system that counted button presses on PC2 (increment once per button press)
- We can write code to check if the button is pressed (==0) and then increment 'cnt'
- But remember, your code executes extremely fast...what will happen?

```c
#include<avr/io.h>
int main()
{
    PORTC |= (1 << PC2);
    int cnt = 0;
    while(1){
        char pressed = (PINC & 0x04);
        if( pressed == 0 ){
            cnt++;
        }
    }
    return 0;
}
```
Waiting Through a Press

- Consider trying to build a system that counted button presses on PC2 (increment once per button press)
- We can write code to check if the button is pressed (==0) and then increment 'cnt'
- But remember, your code executes extremely fast...what will happen?

```c
#include<avr/io.h>
int main()
{
    PORTC |= (1 << PC2);
    int cnt = 0;
    while(1){
        char pressed = (PINC & 0x04);
        if( pressed == 0 ){
            while( (PINC & 0x04) == 0 ){
                {} // This infinite loop will never be reached.
            }
            cnt++;
        }
    }
    return 0;
}
```

---

![Diagram showing the connection between Arduino and a button labeled PC2, with variables PC2 and cnt showing 0-1 transitions](Diagram.png)
Interfacing Mechanical Switches/Buttons

- Mechanical switches and buttons do not make solid, steady contact immediately after being pressed/changed.
- For a short (few ms) time, “bouncing” will ensue and can cause spurious SW operation (one press of a button may look like multiple presses).
- Need to “debounce” switches with your software.
  - Usually waiting around 5 ms from the first detection of a press will get you past the bouncing and into the stable period.

![Diagram of mechanical switch interfacing with Arduino](image)
Waiting Through a Press

• Consider trying to build a system that counted button presses on PC2 (increment once per button press)
• We can write code to check if the button is pressed (==0) and then increment 'cnt'
• But remember, your code executes extremely fast...what will happen?

```c
#include<avr/io.h>
int main()
{
    PORTC |= (1 << PC2);
    int cnt = 0;
    while(1){
        char pressed = (PINC & 0x04);
        if( pressed == 0 ){
            _delay_ms(5);
            while( (PINC & 0x04) == 0 )
                {}  
            _delay_ms(5);
            cnt++;
        }
    return 0;
    }
}
```
What's Your Function

- Because there is a fair amount of work to do just to recognize a button press, you may want to extract those to functions you can call over and over again.

```c
#include<avr/io.h>

char pc2Pressed()
{
    char pressed = (PINC & 0x04);
    if( pressed == 0 ){
        _delay_ms(5);
        while( (PINC & 0x04) == 0 ) { }
        _delay_ms(5);
        return 1;
    }
    else
        return 0;
}

int main()
{
    PORTC |= (1 << PC2);
    int cnt = 0;
    while(1){
        if( pc2Pressed() )
            cnt++;
    }
    return 0;
}
```
SUMMARY AND EXERCISES
Exercise 1

• We want to use Group C (Port C) bit 5 as an output. Show how you should initialize your program then write a statement to turn the output 'ON' to 5V.
Exercise 2

• Now turn write a statement to turn that same output ‘OFF’ (i.e. to output 0V)
Exercise 3

• We want to use Group B (Port B) bit 3 as an input connected to a switch. You have no separate resistors available to you. Show how you should initialize your program and then write an if statement to check if the input voltage is HIGH (5V).
Common Mistakes

• Don't make these mistakes
• Instead remember:
  – Never shift a 0 when trying to do an AND or OR (e.g. 0 << x)
  – Correctly parenthesize your comparisons
  – To check if a bit is 1, check if the result of the ANDing is not-equal to 0

```c
// Clearing a bit to 0
// Wrong
PORTD &= (0 << 3);
PORTD |= (0 << 3);

// Right
PORTD &= ~(1 << 3);
```

```c
// Checking a bit
// Wrong
if(PIND & (1 << 3) == 0)
// Right
if( (PIND & (1 << 3)) == 0)
```

```c
// Checking if a bit is 1
// Wrong
if( (PIND & (1 << 3)) == 1)
// Right
if( (PIND & (1 << 3)) != 0)
```
Summary – Cheat Sheet

- Refer to the cheat sheet in your lecture notes
- Below are the methods to accomplish the 6 basic I/O tasks
  - Memorize these, practice these, recite these!!!

<table>
<thead>
<tr>
<th>Set Pin as Output</th>
<th>Set Pin as Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDRB</td>
<td>= (1 &lt;&lt; DDB4);</td>
</tr>
<tr>
<td>PORTB</td>
<td>= (1 &lt;&lt; PB4);;</td>
</tr>
<tr>
<td>PORTB &amp;=(~(1 &lt;&lt; PB4));</td>
<td>Check Pin is 0 (PINB &amp; (1 &lt;&lt; PB4)) == 0</td>
</tr>
</tbody>
</table>

Note: DDRB |= (1 << DDB4); is used to set a pin as an output. PORTB |= (1 << PB4); is used to set the value to 1. PORTB &=(~(1 << PB4)); is used to clear the output value to 0. CHECK PIN IS 1 takes the value of the pin and checks if it is 1. Check Pin is 0 checks if the pin value is 0.

Pin Change Interrupts
- PCINT0 / PCINT1
- PCINT2 / PCINT3
- PCINT4 / PCINT5
- PCINT6 / PCINT7
- PCINT8 / PCINT9
- PCINT10 / PCINT11
- PCINT12 / PCINT13
- PCINT14 / PCINT15
- PCINT16 / PCINT17
- PCINT18 / PCINT19
- PCINT20 / PCINT21
- PCINT22 / PCINT23
- PCINT24 / PCINT25
- PCINT26 / PCINT27
- PCINT28 / PCINT29
- PCINT30 / PCINT31

Helpful Arduino Library Functions
- charbuf(char* buf, int max, char* fmt, ...);
- snprintf(chars, max, "%d", digits);
- delay(); // delay 10 milliseconds
- delayMicroseconds(); // delay 10 microseconds
- delayMicroseconds(); // turn on global interrupt enable
- sei(); // turn on global interrupt enable
- cli(); // turn off global interrupt enable