EE 109 Unit 4

Microcontrollers (Arduino) Overview
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) $v = 58$;
  – 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 C is a byte, manipulating individual bits requires us to use BITWISE LOGICAL 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 isolate a bit(s) value from the others in the register
Bitwise Logical Operations

- ANDs can be used to control whether a bit passes changed or a '0' is produced (i.e. AND's can force a bit to '0')
- ORs can be used to control whether a bit passes unchanged or a '1' is produced (i.e. OR's can force a bit to '1')
- XORs can be used to control whether a bit passes unchanged or is inverted/flipped
Logical Operations

• Logic operations on numbers means performing the operation on each pair of bits

\[ \begin{align*}
0xF0 & \quad \text{AND} \quad 0x3C \\
0x30 & \quad \text{AND} \quad 0x1111 \ 0000 \\
0xF0 & \quad \text{OR} \quad 0x3C \\
0xFC & \quad \text{OR} \quad 0x1111 \ 1100 \\
0xF0 & \quad \text{XOR} \quad 0x3C \\
0xCC & \quad \text{XOR} \quad 0x1100 \ 1100
\end{align*} \]
Logical 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;
    z = x && y;
  – Result is z = 1; Why?

• Bitwise Logical Operators (&, |, ^, ~)
  – Operate on the logical value of INDIVIDUAL bits in a variable
    char x = 1, y = 2, z;
    z = x & y;
  – Result is z = 0; Why?
Logical Operations

• Bitwise logic operations are often used for "bit fiddling"
  – Change the value of a bit in a register w/o affecting other bits
  – C operators: & = AND, | = OR, ^ = XOR, ~ = NOT

• Examples (Assume an 8-bit variable, v)
  – Clear the LSB to '0' w/o affecting other bits
    • v = v & 0xfe; or equivalently
    • v = v & ~(0x01);
  – Set the MSB to '1' w/o affecting other bits
    • v = v | 0x80;
  – Flip the LS 4-bits w/o affecting other bits
    • v = v ^ 0x0f;
Changing Register Bits

- Bitwise logic operations can be used to change the values of individual bits in registers without affecting the other bits in the register.
  - Set bit 0 of v to a ‘1’
    \[ v = v | 0x01; \]
  - Clear the 4 upper bits in v to ‘0’s
    \[ v = v & 0x0f; \]
  - Flip bits 4 and 5 in v
    \[ v = v ^ 0b00110000; \]
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 bits 2:0 of \( v = "101" \)
    if \( ((v \& 0b00000111) == 0b00000101) \) { code }
  – Check if bits 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
  – \( v |= 0x01; \)
  – \( v &= 0x0f; \)
  – \( v ^= 0b00110000; \)
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

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 (can also be powered if connected to USB)
- Reset button
- Connectors for I/O lines D0 – D13
- Power and ground pins
- I/O lines A0 – A5
**Arduino Uno**

- Arduino Unos can be stacked with "shield" boards to add additional capabilities (Ethernet, wireless, D/A, LCDs, sensors, motor control, etc.)
ARDUINO PORTS AND PINS
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

![Diagram of computer architecture with video interface, processor, memory, keyboard interface, and data bus connecting all components.](image-url)
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/I^2^C interfaces
  - A/D converter
The program you write and compile on your laptop is downloaded into the microcontroller on the UNO board.

The code resides in the FLASH memory while the CPU fetches one instruction at a time and executes it. Data sits in the RAM (SRAM).

Your program controls external inputs and outputs primarily through PORTs B, C, and D which effectively control the values of the I/O pins.
Digital I/O Example

This program...

• Checks if the button is being pressed (i.e. the value on Port B bit 7 is '1').

• If so, it sets the value on Port B bit 5 to '1' (which is a high voltage) and connects to an LED to make it light up

• Otherwise it sets PB5 to '0' (low voltage) and the LED does NOT light up

Software code to control the microcontroller

```c
#include <avr/io.h>
int main()
{
    while(true){
        if(PortB[7] == 1)
            PortB[5] = 1;
        else
            PortB[5] = 0;
    }
    return 0;
}
```

Disclaimer: This code & connections are an approximation and should not just be copied.

Main Point: What happens to the hardware (button and LED) is controlled by the software
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
  – **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
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 ports (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[0]</td>
<td>DIG8</td>
<td>PortC[0]</td>
<td>AN0</td>
<td>PortD[0]</td>
<td>DIG0</td>
</tr>
<tr>
<td></td>
<td>(don't use)</td>
<td></td>
<td>(don't use)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(don't use)</td>
<td></td>
<td></td>
<td></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)
Arduino Digital I/O

• 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

• Most I/O pins in a port can be directly controlled by your software for "digital I/O" OR be used for other specific HW functionality integrated on chip
  – PORTC0 can be used as a digital I/O OR as the Analog-to-Digital Conversion input: ADC0
  – PORTD0 can be used as digital I/O OR the serial communication receive input: RXD

• We will discuss these other HW functions later...focus on digital I/O
How to connect LEDs and Switches/Pushbuttons

INPUT AND OUTPUT DEVICES
What Do we Do Now

• Great! We have this Arduino microcontroller with all these pins...what should we connect to them?
  – Outputs: LED's, LCD screens, wired/wireless communication
  – Inputs: Buttons, Switches, temperature sensors, rotary encoders, etc.

• We'll start simple and try to attach an LED (output) and a pushbutton/switch (input)
(Light-Emitting) Diodes

• The simplest output we can control is an LED (Light-emitting diode) which is like a tiny light bulb
• An LED glows ('on') when the voltage across it is greater than 1.7-2V and is 'off' otherwise
• Voltage/Current Relationship:
  – For a resistor, current flowing through a resistor is proportional to the voltage across it \( I = \frac{1}{R} \times V \)
  – For an LED, current grows exponentially with voltage \( I \approx A e^v \)
  – Since a small change in voltage could cause a large increase in current and possibly blow-out the LED, we need to limit current with a resistor
• LEDs are polarized meaning they only work in one orientation (longer leg must be at higher voltage)
LED Connection Approaches

- Below are some options for connecting an LED
- We need a series resistor to limit current
  - Choose value based on amount of current you want
  - Amount of current will determine brightness of LED
  - \( i = \frac{V_1}{R_1} = \frac{(V_s-V_{\text{LED}})}{R_1} \)
  - Usually \( R_1 \) is a few hundred ohms (330 or 470 ohms)

An Arduino output will serve as our voltage source that can be either '0' (0V) or '1' (5V)

Choose resistor to limit current

Doesn't matter where resistor is placed as long as it is in series

LED Schematic Symbol

Breadboard view
LED Connection Approaches

• When letting a digital output control an LED, the value (i.e. '0' or '1') that causes the LED to light up depends on how the circuit is wired

  – Note: Gates can often "sink" (take in) more current than they can "source" (push out), so option 2 may be preferred…but let's not worry about this now...let's use option 1

Main Point: LED's should always be connected in series with a current-limiting resistor
Switches and Pushbuttons

- Switches and pushbuttons can be in one of two configurations: **open** or **closed**
  - Switches can be opened or closed and then stay in that position until changed
  - Pushbuttons are open by default and require you to push them to close the circuit (they then open when you release)

- **Important Note 1:**
  - When open a SW/PB looks like an infinite resistance (no current can flow)
  - When closed a SW/PB looks like a wire (R=0) and no voltage drops across it

- **Important Note 2:**
  - SW or PBs don't produce digital 0's or 1's on their own, they control what voltage (PWR/GND) is connected to your device
Power & Ground Connections

- Easy mistake when you're just learning to wire up circuits:
  - Wire the inputs & outputs but forget to connect power and ground
- All circuits and chips require a connection to a power source and ground
  - Gates
  - Switches
  - Buttons

Actual connection… …will be drawn like this
Connecting a Switch

- Switches do not produce a 0 (GND) or 1 (VDD) by itself

- Option 1: Attach one side to GND and the other side to the device
  - When the switch=open, nothing is connected to the device (a.k.a. “floating”)
  - A floating input may sometimes appears as zero, and other times as a one.
  - We need the inputs to logic gates to be in either the 0 or 1 state...not floating

- Option 2:
  - SW open => Input = VDD = 1
  - SW closed => Direct wire from both VDD and GND to input = Short Circuit = unknown voltage and possibly LARGE current flow...BAD!!!
Using a Pull-up Resistor

- Solution: Put GND on the far side and a "pull-up" resistor at the input side
  - "Pull-up resistor" used to hold the input high unless something is forcing it to a zero
  - SW open => Arduino input looks like inf. Resistance in series with Rp. Thus no current through Rp and thus no voltage drop across Rp...input = VDD = 1
  - SW closed => Direct wire from GND to input...input = GND = 0...Also current flowing from Vdd to GND is limited by Rp preventing a short circuit.
  - Usually Rp is large (10k ohms) to limit current

\[
\text{Vin} = Vdd - V_{RP} \\
\text{Vin} = Vdd - i_{RP} \times Rp
\]

To calculate Vin:

- \(i_{RP} = 0\) since in series with inf. resistance of Arduino input
- Thus, \(\text{Vin} = Vdd\)

Main Point: Buttons & switches should have GND connected to one side & a pull-up resistor on the other
Alternative Switch/Resistor Connections

- Consider the options to connect PWR & GND to a SW/PB
- Note: A gate input "looks like" an inf. resistance

**Option 1** Preferred

**Option 2** Less Preferred (just take our word)

**Option 3** Gate input always '1'

**Option 4** Gate input always '0'
Controlling the pins of the Arduino to be digital inputs and outputs

**ARDUINO DIGITAL I/O**
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.
## Controlling I/O 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

<table>
<thead>
<tr>
<th>DDRD</th>
<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>PORTD</td>
<td>PORTD7</td>
<td>PORTD6</td>
<td>PORTD5</td>
<td>PORTD4</td>
<td>PORTD3</td>
<td>PORTD2</td>
<td>PORTD1</td>
<td>PORTD0</td>
</tr>
<tr>
<td>PIND</td>
<td>PINB7</td>
<td>PINB6</td>
<td>PINB5</td>
<td>PINB4</td>
<td>PINB3</td>
<td>PINB2</td>
<td>PINB1</td>
<td>PINB0</td>
</tr>
<tr>
<td>DDRB</td>
<td>DDRB7</td>
<td>DDRB5</td>
<td>DDRB4</td>
<td>DDRB3</td>
<td>DDRB2</td>
<td>DDRB1</td>
<td>DDRB0</td>
<td></td>
</tr>
<tr>
<td>PORTB</td>
<td>PORTB7</td>
<td>PORTB5</td>
<td>PORTB4</td>
<td>PORTB3</td>
<td>PORTB2</td>
<td>PORTB1</td>
<td>PORTB0</td>
<td></td>
</tr>
<tr>
<td>PINB</td>
<td>PINB7</td>
<td>PINB5</td>
<td>PINB4</td>
<td>PINB3</td>
<td>PINB2</td>
<td>PINB1</td>
<td>PINB0</td>
<td></td>
</tr>
</tbody>
</table>
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 \(\rightarrow\) PB5 (Port B bit 5 = DIG13 pin) will be used as **input**
  - Example: If DDRB[5] = 1 \(\rightarrow\) 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>1</td>
</tr>
</tbody>
</table>

### DDRB

<table>
<thead>
<tr>
<th>DDRB7</th>
<th>DDRB5</th>
<th>DDRB4</th>
<th>DDRB3</th>
<th>DDRB2</th>
<th>DDRB1</th>
<th>DDRB0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### PD[7:4] = INPUT  

<table>
<thead>
<tr>
<th>PORTB</th>
<th>PIN</th>
<th>PORTC</th>
<th>PIN</th>
<th>PORTD</th>
<th>PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTB[0]</td>
<td>DIG8</td>
<td>PORTC[0]</td>
<td>AN0</td>
<td>PORTD[0]</td>
<td>DIG0</td>
</tr>
<tr>
<td><strong>PORTB[5]</strong></td>
<td><strong>DIG13</strong></td>
<td><strong>PORTC[5]</strong></td>
<td><strong>AN5</strong></td>
<td><strong>PORTD[5]</strong></td>
<td><strong>DIG5</strong></td>
</tr>
</tbody>
</table>

#### 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"

Register 2: PORT 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 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.
  - The action of referencing PINx causes all the signals to be acquired.
  - 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'.

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

- If a port is an input but has no signal connected to it, it will “float” and could be read as either zero or one.

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 logic 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'
    
    \[
    DDRB = DDRB \text{ } | \text{ } 0b00001000; \quad // \text{ DDRB } |= \text{ 0x08;}
    \]
  - Clear the 2 upper bits in PORTC to ‘0’s
    
    \[
    PORTC = PORTC \text{ } & \text{ } 0x3f;
    \quad // \text{ PORTC } &= \text{ } ~(0b11000000)
    \]
  - Flip bits 7 and 1 in DDRC
    
    \[
    DDRC = DDRC \text{ } ^\text{ } 0b10000010; \quad // \text{ DDRC } ^= \text{ 0x82;}
    \]
  - Check if PIND, bit 4 = '1'
    
    \[
    \text{if (PIND } & \text{ 0x10) } \{ \text{ code } \}
    \]
Review

• To use a pin(s) as output:
  – If you want to use Port B, bit 5 as an output, make DDRB, bit 5 = 1
    • Ex. DDRB |= 0b00100000;
  – Then perform operations on PORTB register to place the desired output value into bit 5
    • Ex. PORTB |= 0b00100000; // make pin on B5 = Vdd (5V)
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**


&

| 0 | 0 | 0 | ? | 0 | 0 | 0 | 0 |
Pull Up Resistors

- Adding and wiring pull-up resistors for input buttons can be time consuming...
- Thankfully, each Arduino input bit has an internal optional “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.

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.
Enabling Pull Up Resistors

- When DDRx[n] is '0' (i.e. a pin is used as input), the value in the PORTx[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

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

<table>
<thead>
<tr>
<th>PORTD0</th>
<th>PORTD1</th>
<th>PORTD2</th>
<th>PORTD3</th>
<th>PORTD4</th>
<th>PORTD5</th>
<th>PORTD6</th>
<th>PORTD7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Enable Pull-Up Resistors

<table>
<thead>
<tr>
<th>PIN0</th>
<th>PIN1</th>
<th>PIN2</th>
<th>PIN3</th>
<th>PIN4</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>PIND3</td>
<td>PIND4</td>
</tr>
</tbody>
</table>

Made-up values read from the push-buttons (which don't require you to wire up external pull-up resistors)

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td></td>
</tr>
</tbody>
</table>

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
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;
}
```

<table>
<thead>
<tr>
<th>DDRD (starts at 0's on reset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PORTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>
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 (DD7) or bit 4 (PD4)...

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

int main()
{
    // Init. D7 to output
    DDRD |= (1 << DD7);

    // Enable pull-up on PD4
    PORTD |= (1 << PD4);

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

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

```
x = x >> 2;
(Right Shift by 2 bits)
```
```
x = x << 2;
(Left Shift by 2 bits):
```
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…
#define macros

- Can be used for simple find/replace scenarios
  - `#define find_pat replace_pat`
- Makes constants more readable and easier to change (if you have the same constant in 10 places and you realize you need to change it, just change the one `#define` statement)
Register Bit Names

- Symbolic names for the positions of bits in each register are defined as constants in `<avr/io.h>`
  - `#include <avr/io.h>`
- Each PORTx register has their own constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PB5</td>
<td>PB4</td>
<td>PB3</td>
<td>PB2</td>
<td>PB1</td>
<td>PB0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PORTC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC5</td>
<td>PC4</td>
<td>PC3</td>
<td>PC2</td>
<td>PC1</td>
<td>PC0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PORTD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PD7</td>
<td>PD6</td>
<td>PD5</td>
<td>PD4</td>
<td>PD3</td>
<td>PD2</td>
<td>PD1</td>
<td>PD0</td>
</tr>
</tbody>
</table>

- All DDRx registers share the same constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDRB,C,D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DD7</td>
<td>DD6</td>
<td>DD5</td>
<td>DD4</td>
<td>DD3</td>
<td>DD2</td>
<td>DD1</td>
<td>DD0</td>
</tr>
</tbody>
</table>

- All PINx registers share the same constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PINB,C,D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PIN7</td>
<td>PIN6</td>
<td>PIN5</td>
<td>PIN4</td>
<td>PIN3</td>
<td>PIN2</td>
<td>PIN1</td>
<td>PIN0</td>
</tr>
</tbody>
</table>
Putting it All Together

• Values for working with bits can be made using the ‘<<‘ shift operator
  – OK: PORTB |= 0x20;  Better: PORTB |= (1 << PB5) ;
  – OK: DDRD |= 0x04;  Better: DDRD |= (1 << 2);
• This makes the code more readable and your intention easier to understand...
• More examples
  – DDRC |= (1 << DD5)  set DDRC, bit 5
  – PORTB ^= (1 << PB2)  invert PORTB, bit 2
  – PORTD &= (1 << PD3)  clear PORTD, bit 3 <- WRONG!
    Why?
Clearing Bits...A Common Mistake

• When using the ‘&=‘ operation to clear bits, remember to invert the bits.
• This won’t work to clear PD3 to ‘0’
  – PORTD &= (1 << PD3);
  – is the same as
  – PORTD &= 0b0001000;
  – which clears everything but bit PD3
• Use the ‘~’ operator to complement the bits.
  – PORTD &= ~(1 << PD3);
  – is the same as
  – PORTD &= 0b11110111;
  – and now PD3 gets cleared.
Defining Your Own Symbolic Names

• You can make your own more meaningful symbolic names
  – #define LEDBIT  (1 << PB6)
  – #define CLKBIT  (1 << PC3)
  – PORTB |= LEDBIT;
  – PORTC |= CLKBIT;
  – PORTC &^= ~CLKBIT;

• Can combine multiple bits into one defined value
  – #define MULTIBITS  ((1 << PB3) | (1 << PB4) | (1 << PB5))
  – is the same as
  – #define MULTIBITS  0b00111000
  
  \[
  \begin{array}{c|c}
  1 << PB3 & 00001000 \\
  1 << PB4 & 00010000 \\
  1 << PB5 & 00100000 \\
  \hline
  & 00111000 \\
  \end{array}
  \]
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);
    ```
Do We Need Step 2...Yes!!!

- Can't we just do step 1 and 3 and OR the bits of x into PORTB
  
  ```c
  #define MASKBITS 0x0f
  PORTB |= (x & MASKBITS);
  ```

- No, because what if the destination (PORTB) already had some 1's where 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
    
    ```c
    PORTB &= ~MASKBITS;
    ```
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);
  ```
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 ){
                {}
                cnt++;
            }
        }
        return 0;
    }
}
```
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.
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;
}
```