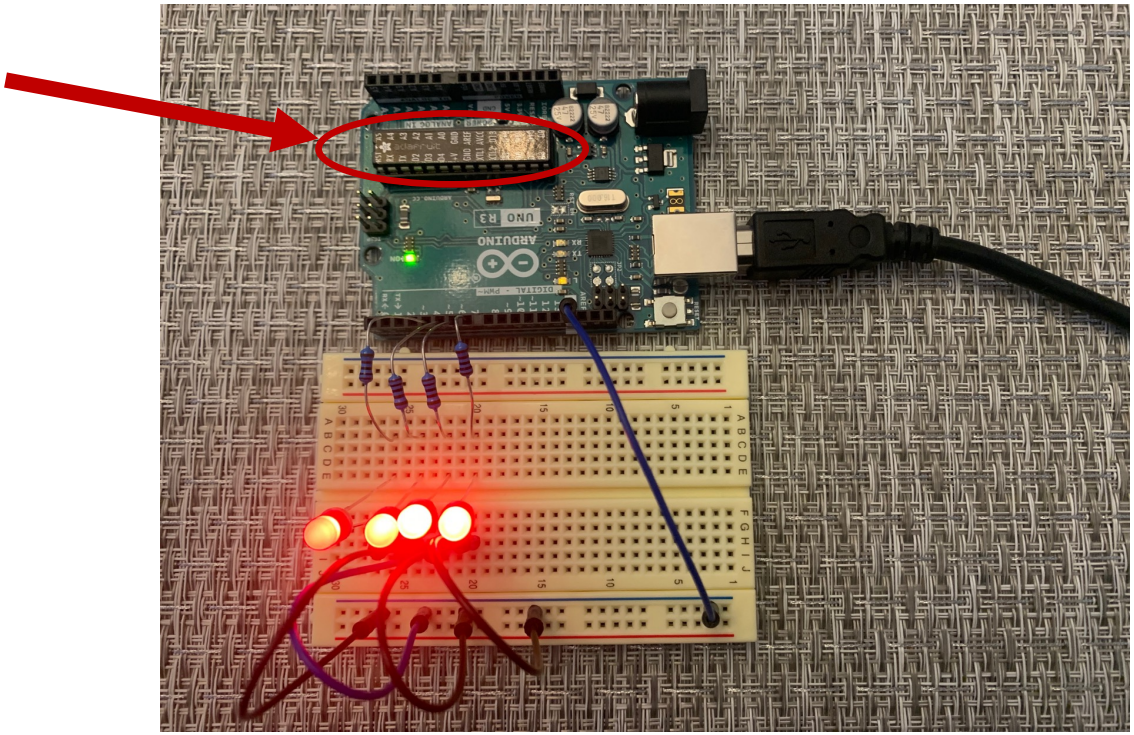
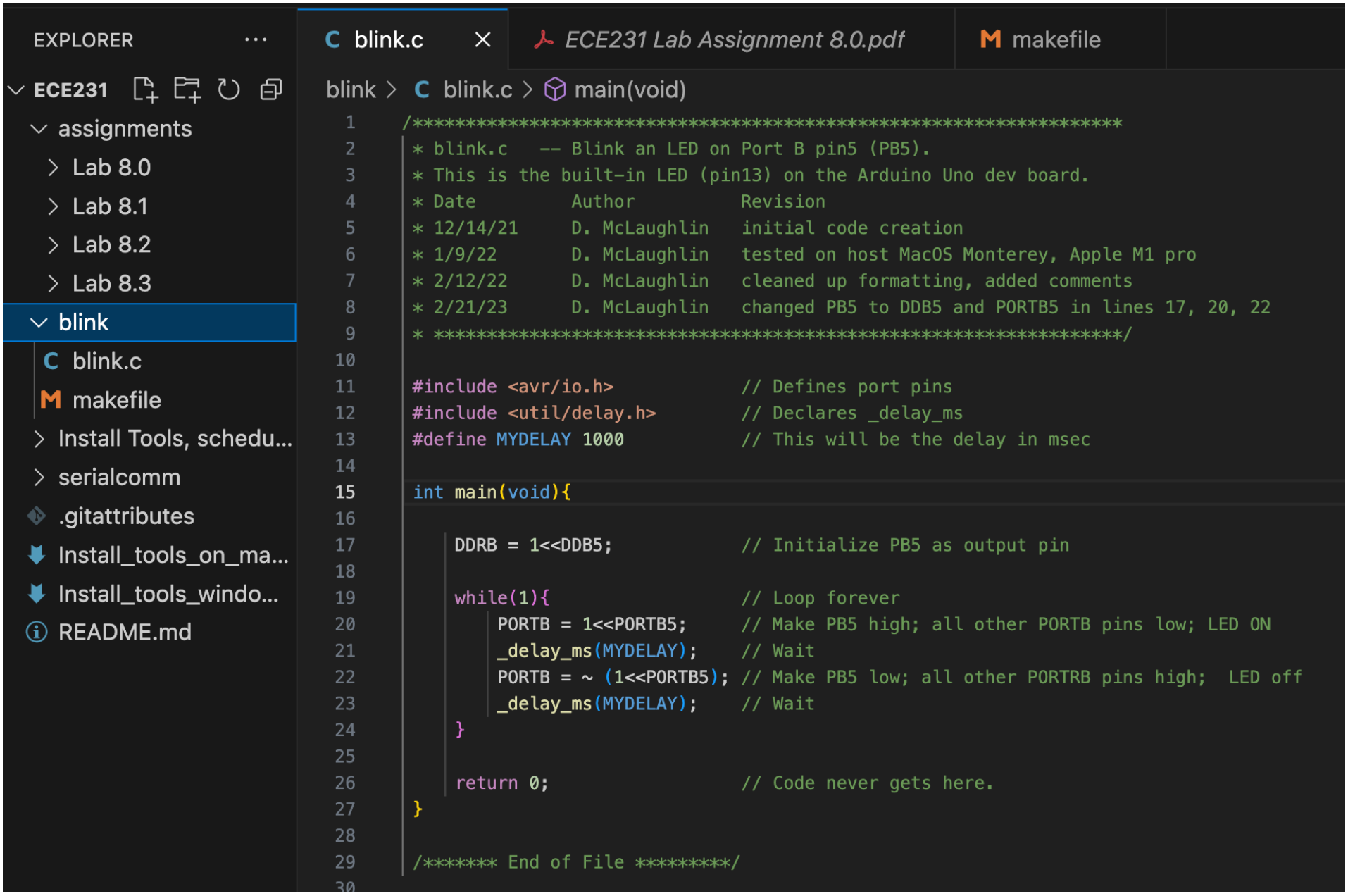
Lectures 8.2 & 8.3
AVR GPIO Programming

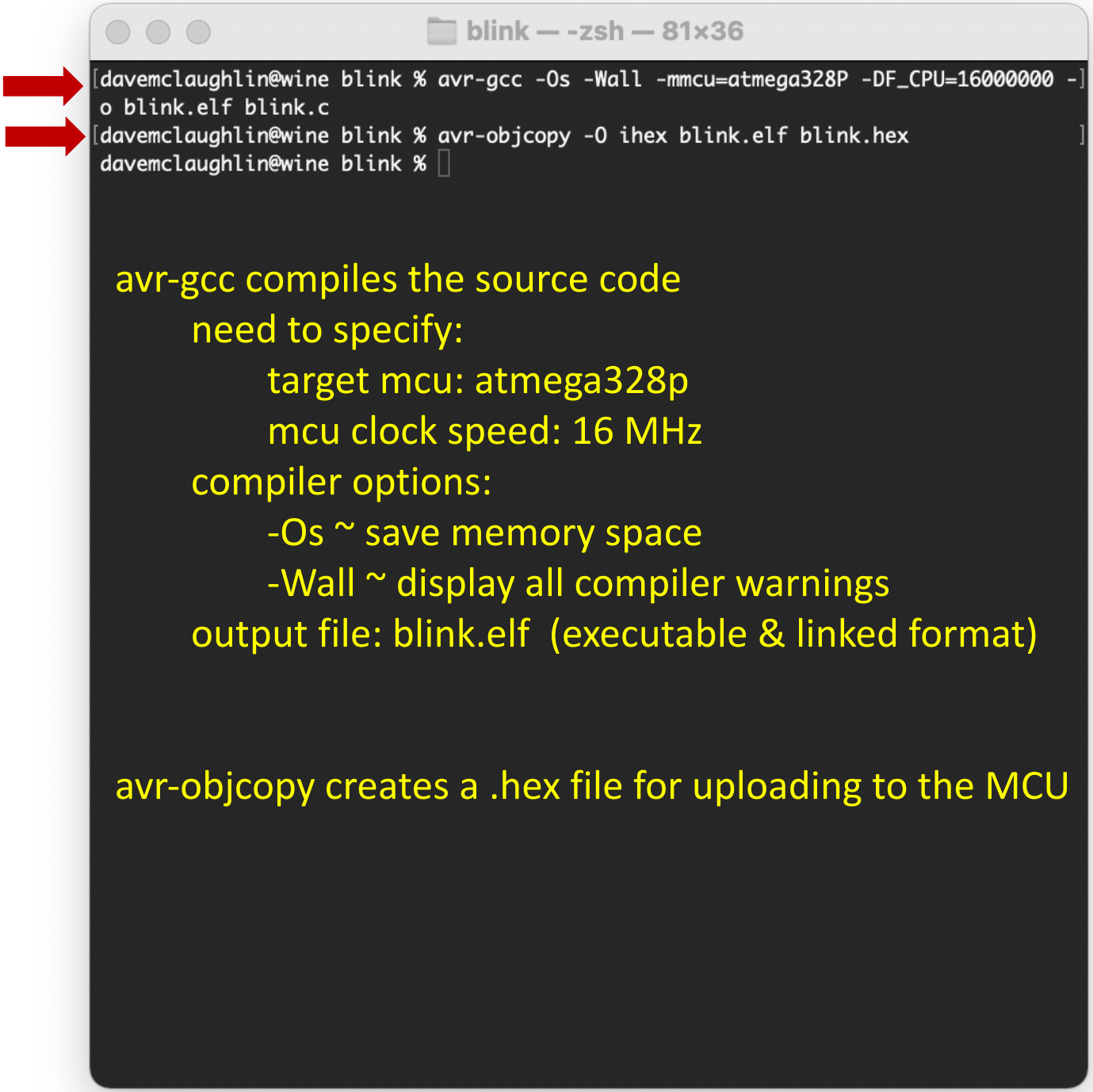
We are not programming an "Arduino Uno". We are programming an AVR ATmega328P microcontroller (MCU) that happens to reside on an Arduino Uno dev board.



Prof. David McLaughlin
ECE Department
University of Massachusetts
Amherst, MA

Spring 2024

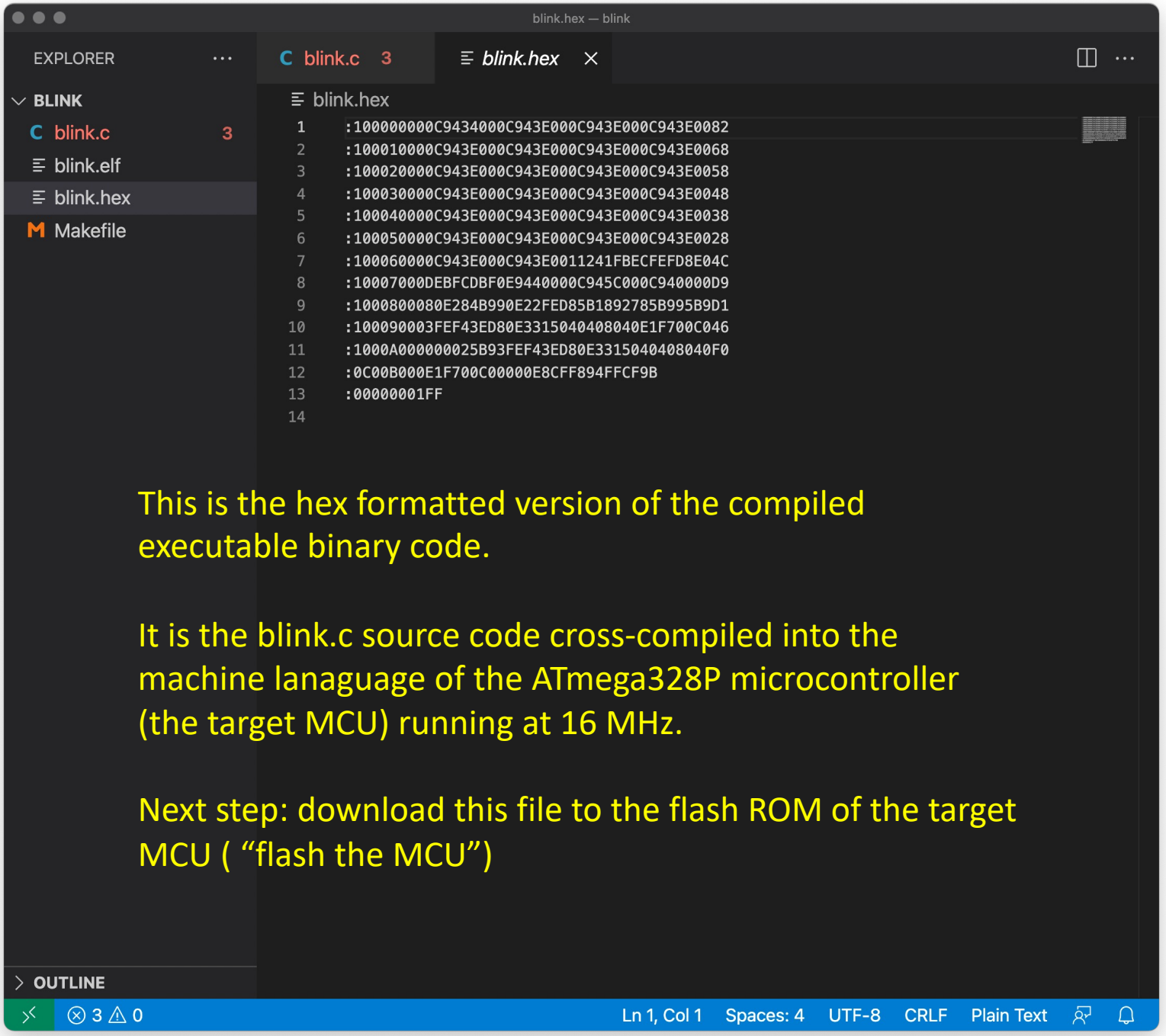

EXPLORER
ECE231
assignments
Lab 8.0
Lab 8.1
Lab 8.2
Lab 8.3
blink
blink.c
makefile
Install Tools, schedu...
serialcomm
.gitattributes
Install_tools_on_ma...
Install_tools_windo...
README.md
blink.c
ECE231 Lab Assignment 8.0.pdf
makefile
blink > blink.c > main(void)
/***
* blink.c -- Blink an LED on Port B pin5 (PB5).
* This is the built-in LED (pin13) on the Arduino Uno dev board.
* Date Author Revision
* 12/14/21 D. McLaughlin initial code creation
* 1/9/22 D. McLaughlin tested on host MacOS Monterey, Apple M1 pro
* 2/12/22 D. McLaughlin cleaned up formatting, added comments
* 2/21/23 D. McLaughlin changed PB5 to DDB5 and PORTB5 in lines 17, 20, 22
* ***/
#include <avr/io.h> // Defines port pins
#include <util/delay.h> // Declares _delay_ms
#define MYDELAY 1000 // This will be the delay in msec
int main(void){
DDRB = 1<<DDB5; // Initialize PB5 as output pin
while(1){ // Loop forever
PORTB = 1<<PORTB5; // Make PB5 high; all other PORTB pins low; LED ON
_delay_ms(MYDELAY); // Wait
PORTB = ~ (1<<PORTB5); // Make PB5 low; all other PORTRB pins high; LED off
_delay_ms(MYDELAY); // Wait
}
return 0; // Code never gets here.
}
/******* End of File *********/



avr-gcc compiles the source code

- need to specify:
  - target mcu: atmega328p
  - mcu clock speed: 16 MHz
- compiler options:
  - -Os ~ save memory space
  - -Wall ~ display all compiler warnings
- output file: blink.elf (executable & linked format)

avr-objcopy creates a .hex file for uploading to the MCU



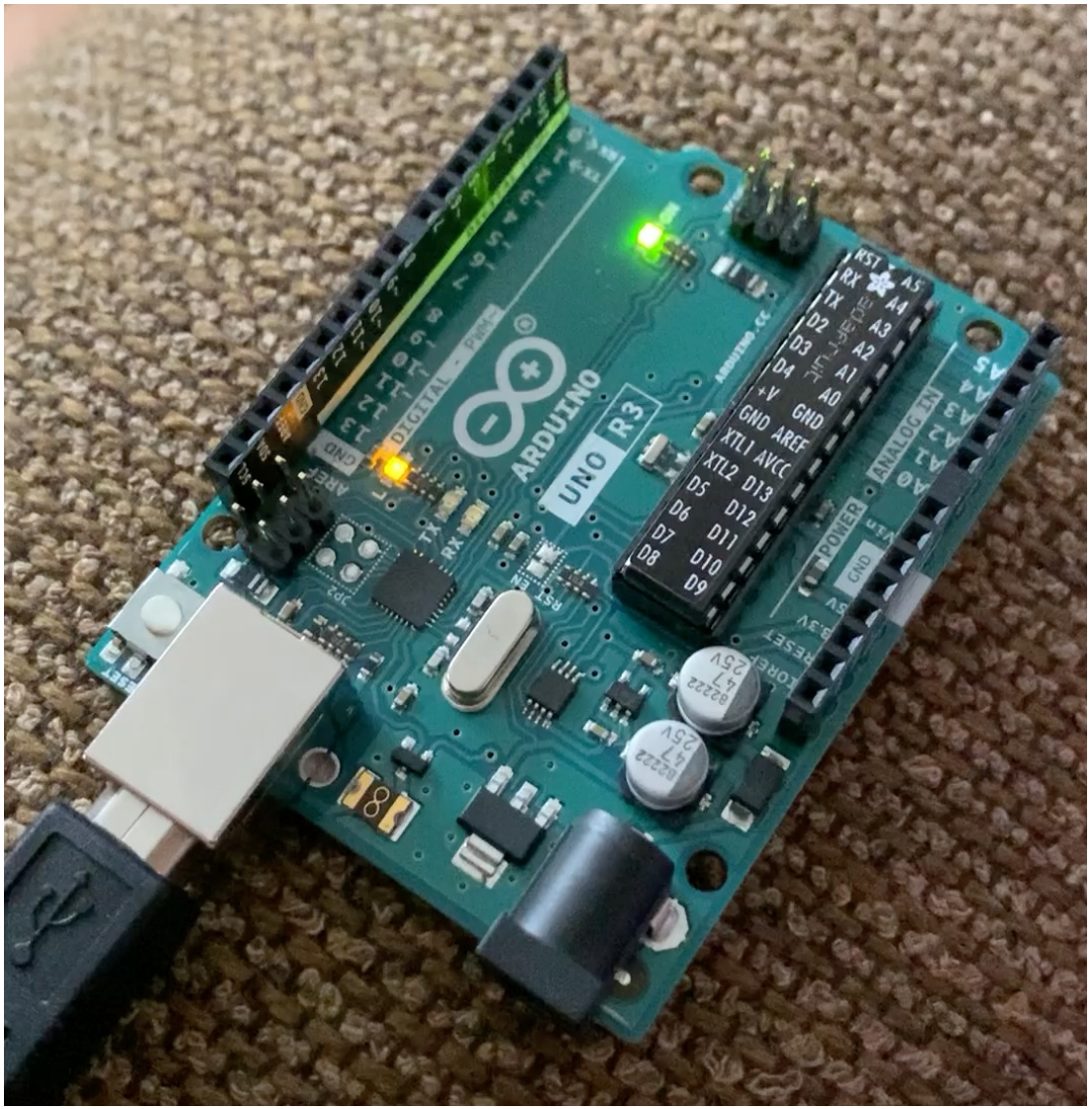
This is the hex formatted version of the compiled executable binary code.

It is the blink.c source code cross-compiled into the machine lanaguage of the ATmega328P microcontroller (the target MCU) running at 16 MHz.

Next step: download this file to the flash ROM of the target MCU (“flash the MCU”)

```
blink — -zsh — 81×36

davemclaughlin@wine blink % avr-gcc -Os -Wall -mmcu=atmega328P -DF_CPU=16000000 -o blink.elf blink.c
davemclaughlin@wine blink % avr-objcopy -O ihex blink.elf blink.hex
davemclaughlin@wine blink % avrdude -c arduino -b 115200 -P /dev/tty.usbmodem2101 -p atmega328p -U flash:w:blink.hex:i

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "blink.hex"
avrdude: writing flash (188 bytes):

Writing | ################################################## | 100% 0.04s

avrdude: 188 bytes of flash written
avrdude: verifying flash memory against blink.hex:
avrdude: load data flash data from input file blink.hex:
avrdude: input file blink.hex contains 188 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.03s

avrdude: verifying ...
avrdude: 188 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.

davemclaughlin@wine blink %
```

Internal LED (PB5) On & Off for 1 Second, repeat

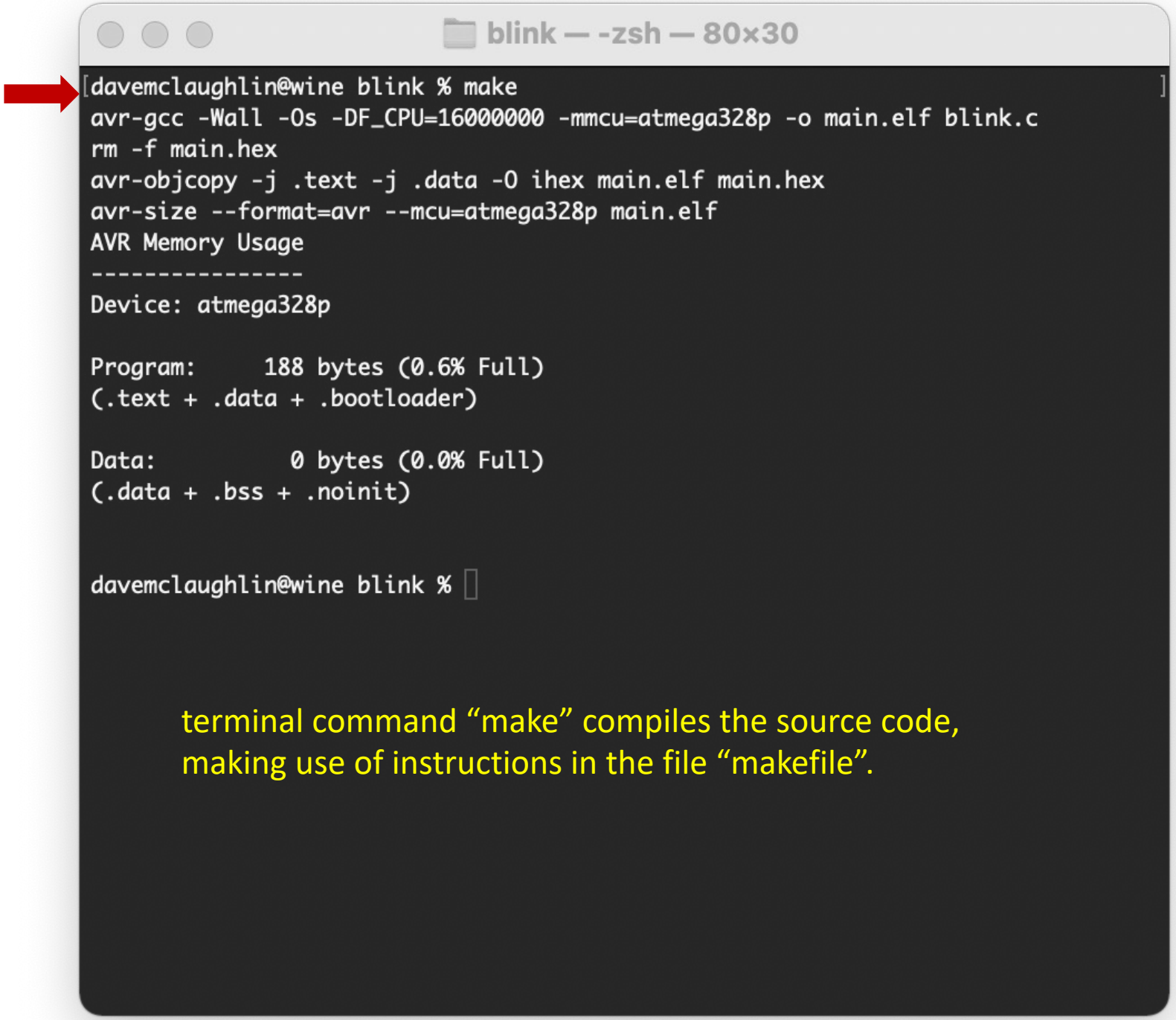


ink.c × | ECE231 Lab Assignment 8.0.pdf | makefile

> C blink.c > main(void)

```
/*****************************************************************
* blink.c   -- Blink an LED on Port B pin5 (PB5).
* This is the built-in LED (pin13) on the Arduino Uno dev board.
* Date          Author          Revision
* 12/14/21      D. McLaughlin   initial code creation
* 1/9/22        D. McLaughlin   tested on host MacOS Monterey, Apple M1 pro
* 2/12/22       D. McLaughlin   cleaned up formatting, added comments
* 2/21/23       D. McLaughlin   changed PB5 to DDB5 and PORTB5 in lines 17, 20, 22
* ****************************************************************/

#include <avr/io.h>             // Defines port pins
#include <util/delay.h>         // Declares _delay_ms
#define MYDELAY 1000            // This will be the delay in msec

int main(void){

    RB = 1<<DDB5;               // Initialize PB5 as output pin

    ile(1){                     // Loop forever
        PORTB = 1<<PORTB5;      // Make PB5 high; all other PORTB pins low; LED ON
        _delay_ms(MYDELAY);     // Wait
        PORTB = ~ (1<<PORTB5); // Make PB5 low; all other PORTRB pins high;  LED off
        _delay_ms(MYDELAY);     // Wait
    }

    return 0;                   // Code never gets here.
}

/******* End of File *********/
```

Change 1000 to 100 for a faster blink...



terminal command “make” compiles the source code, making use of instructions in the file “makefile”.

```
blink — -zsh — 92×35

davemclaughlin@wine blink % make flash
avrdude -c arduino -b 115200 -P /dev/tty.usbmodem101 -p atmega328p -U flash:w:main.hex:i
[                                                                                       ]
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "main.hex"
avrdude: writing flash (188 bytes):

Writing | ################################################## | 100% 0.04s

avrdude: 188 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: load data flash data from input file main.hex:
avrdude: input file main.hex contains 188 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.03s

avrdude: verifying ...
avrdude: 188 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.

davemclaughlin@wine blink %
```

compile & flash the difficult way, specifying everthing @ command line

```
blink — -zsh — 81×36
[davemclaughlin@wine blink % avr-gcc -Os -Wall -mmcu=atmega328P -DF_CPU=16000000 -]
o blink.elf blink.c
[davemclaughlin@wine blink % avr-objcopy -O ihex blink.elf blink.hex              ]
[davemclaughlin@wine blink % avrdude -c arduino -b 115200 -P /dev/tty.usbmodem2101]
  -p atmega328p -U flash:w:blink.hex:i

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be performe
d
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "blink.hex"
avrdude: writing flash (188 bytes):

Writing | ################################################## | 100% 0.04s

avrdude: 188 bytes of flash written
avrdude: verifying flash memory against blink.hex:
avrdude: load data flash data from input file blink.hex:
avrdude: input file blink.hex contains 188 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.03s

avrdude: verifying ...
avrdude: 188 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.

davemclaughlin@wine blink %
```

compile & flash the target MCU using make

```
blink — -zsh — 92×35
davemclaughlin@wine blink % make flash
avrdude -c arduino -b 115200 -P /dev/tty.usbmodem101 -p atmega328p -U flash:w:main.hex:i
[                                                                                      ]
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "main.hex"
avrdude: writing flash (188 bytes):

Writing | ################################################## | 100% 0.04s

avrdude: 188 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: load data flash data from input file main.hex:
avrdude: input file main.hex contains 188 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.03s

avrdude: verifying ...
avrdude: 188 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.

davemclaughlin@wine blink %
```

```
# makefile V2.0. This is a makefile for AVR ATmega328P projects using Arduino Uno Dev board
# ECE-231 Spring 2024.
#
# Instructions:
# Put this file (makefile, no extension) into the source code folder for each
# programming project. You should only need to change the SERIALPORT and SOURCEFILE.
# Options for running this makefile from Command Prompt (Windows) or Terminal (macOS):
# "make" compiles code & creates a hex file for uploading to the MCU
# "make -B" will force compiling even if the source code hasn't changed
# "make flash" will compile, create hex file, and upload the hex file to the MCU
# "make clean" will delete intermediate files make.elf and make.hex

#_____________ MODIFY SERIALPORT AND SOURCEFILE_____________________
# Specify the com port (windows) or USB port (macOS)
# Use Device Manager to identify COM port number for Arduino Uno board in Windows
# In Terminal, type ls /dev/tty.usb* to determine USB port number in macOS
SERIALPORT = /dev/tty.usbmodem143101
# Specify the name of your source code here:
SOURCEFILE = blink.c
#____________________________________________________________________
# Don't change anything below unless you know what you're doing....
CLOCKSPEED = 16000000
PROGRAMMER = Arduino

begin:  main.hex

main.hex: main.elf
    rm -f main.hex
    avr-objcopy -j .text -j .data -O ihex main.elf main.hex
    avr-size --format=avr --mcu=atmega328p main.elf

main.elf: $(SOURCEFILE)
    avr-gcc -Wall -Os -DF_CPU=$(CLOCKSPEED) -mmcu=atmega328p -o main.elf $(SOURCEFILE)

flash:  begin
    avrdude -c $(PROGRAMMER) -b 115200 -P $(SERIALPORT) -p atmega328p -U flash:w:main.hex:i

clean:
    rm -f main.elf main.hex
```

arduino not Arduino

Embedded Programming Involves:

- Reads & writes to memory locations (registers) in the I/O interfaces, to interact with sensors & actuators connected to ports
- Configuring peripherals (timers, input/output ports, ADCs, etc…)
- Loops (to repeat things)
- Conditional branching (if something xxx, else yyy)

Read: Sections 7.1 – 7.3 of textbook:
M. Mazidi, S. Naimi, and S. Naimi, The AVR Microcontroller and Embedded Systems Using Assembly and C, 2nd Edition, 2017. Available in paperback ($23.75) and Kindle ($15.50) editions from Amazon.

The Embedded Programming Super-Loop Architecture

```
#include ...

int main(void)
{

        initialize();                   // Set things up

        while(1)                        // Loop forever
        {
                doSomething();          // Application task
                doSomethingElse();      // Another application task
        }

        return 0;                       // Execution never gets here
}
```

adapted from https://www.includehelp.com/c/the-super-loop-architecture-for-embedded-c-programming.aspx

ink.c × | ECE231 Lab Assignment 8.0.pdf | makefile

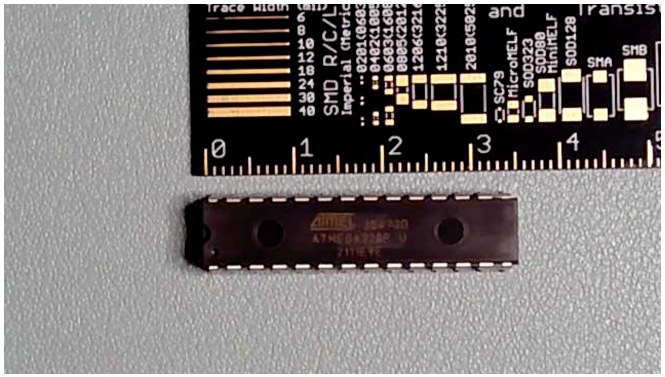
> C blink.c > main(void)

```
/*****************************************************************
* blink.c   -- Blink an LED on Port B pin5 (PB5).
* This is the built-in LED (pin13) on the Arduino Uno dev board.
* Date          Author          Revision
* 12/14/21      D. McLaughlin   initial code creation
* 1/9/22        D. McLaughlin   tested on host MacOS Monterey, Apple M1 pro
* 2/12/22       D. McLaughlin   cleaned up formatting, added comments
* 2/21/23       D. McLaughlin   changed PB5 to DDB5 and PORTB5 in lines 17, 20, 22
* ****************************************************************/

#include <avr/io.h>             // Defines port pins
#include <util/delay.h>         // Declares _delay_ms
#define MYDELAY 1000            // This will be the delay in msec

int main(void){

    DDRB = 1<<DDB5;             // Initialize PB5 as output pin

    while(1){                   // Loop forever
        PORTB = 1<<PORTB5;      // Make PB5 high; all other PORTB pins low; LED ON
        _delay_ms(MYDELAY);     // Wait
        PORTB = ~ (1<<PORTB5); // Make PB5 low; all other PORTRB pins high;  LED off
        _delay_ms(MYDELAY);     // Wait
    }

    return 0;                   // Code never gets here.
}

/******* End of File *********/
```

Observations

- All C statements end in ;
- preprocessor directives begin with #. They are not C instructions. They are commands for the pre-processor for use at compile-time.
- Identifiers must be declared before they are used
- white spaces are optional; they improve readibility
- comments are optional; they improve a programmer & user's life
- combination of decimal (127) and hexadecimal, or hex (0xFF) & binary (0b00000010) is used. You'll find hex to be easiest.
- all C programs have a main() function. (A main "superloop" is a standard way of writing an embedded 8-bit application.)
- user-defined functions make programming easier to debug & understand
- code structures are surrounded by { and }
- embedded C is "nitty-gritty" --- we get down to the register level to toggle bits.

C code, compiled & linked, hex machine code flashed to ROM



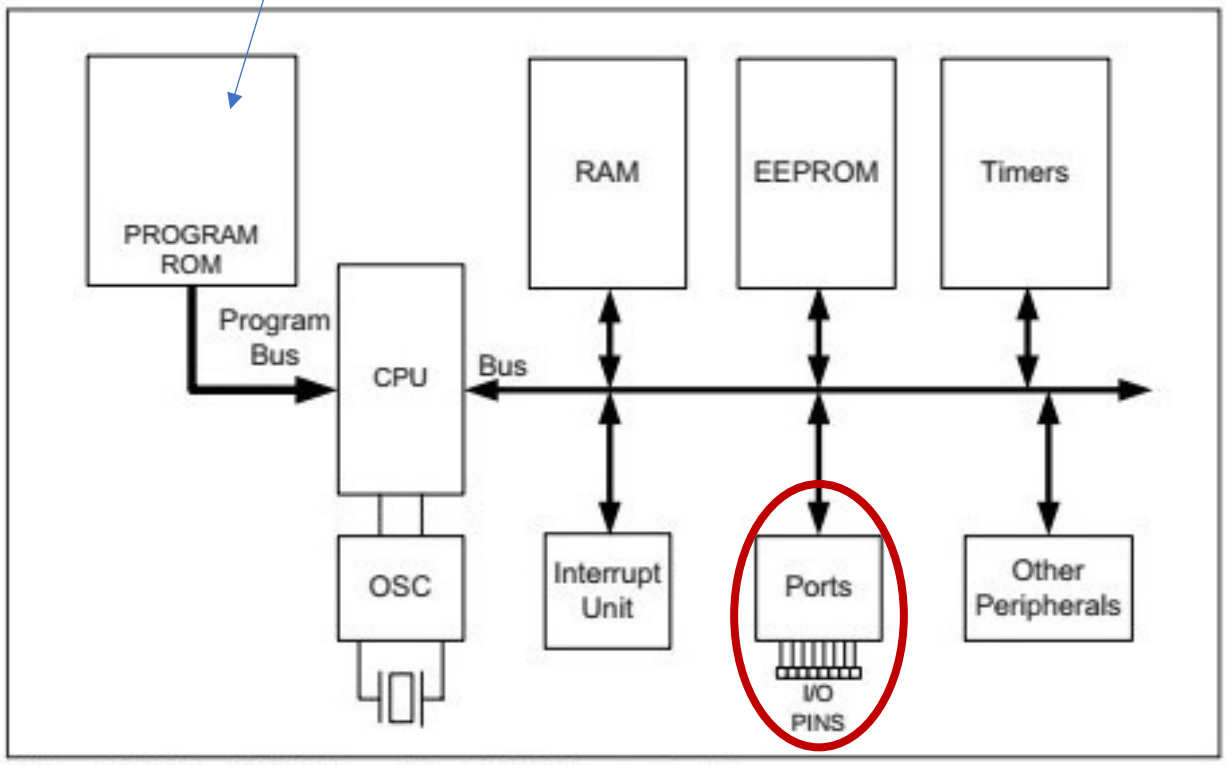


Figure 1-2. Simplified View of an AVR Microcontroller

The I/O ports refer to **hardware:** groups of pins on the ATmega328p DIP that are set high (5V) or low (0V), internally or externally

software: registers – memory locations that are written to, or read from

ATmega328P MCU (28 or 32 pin package)

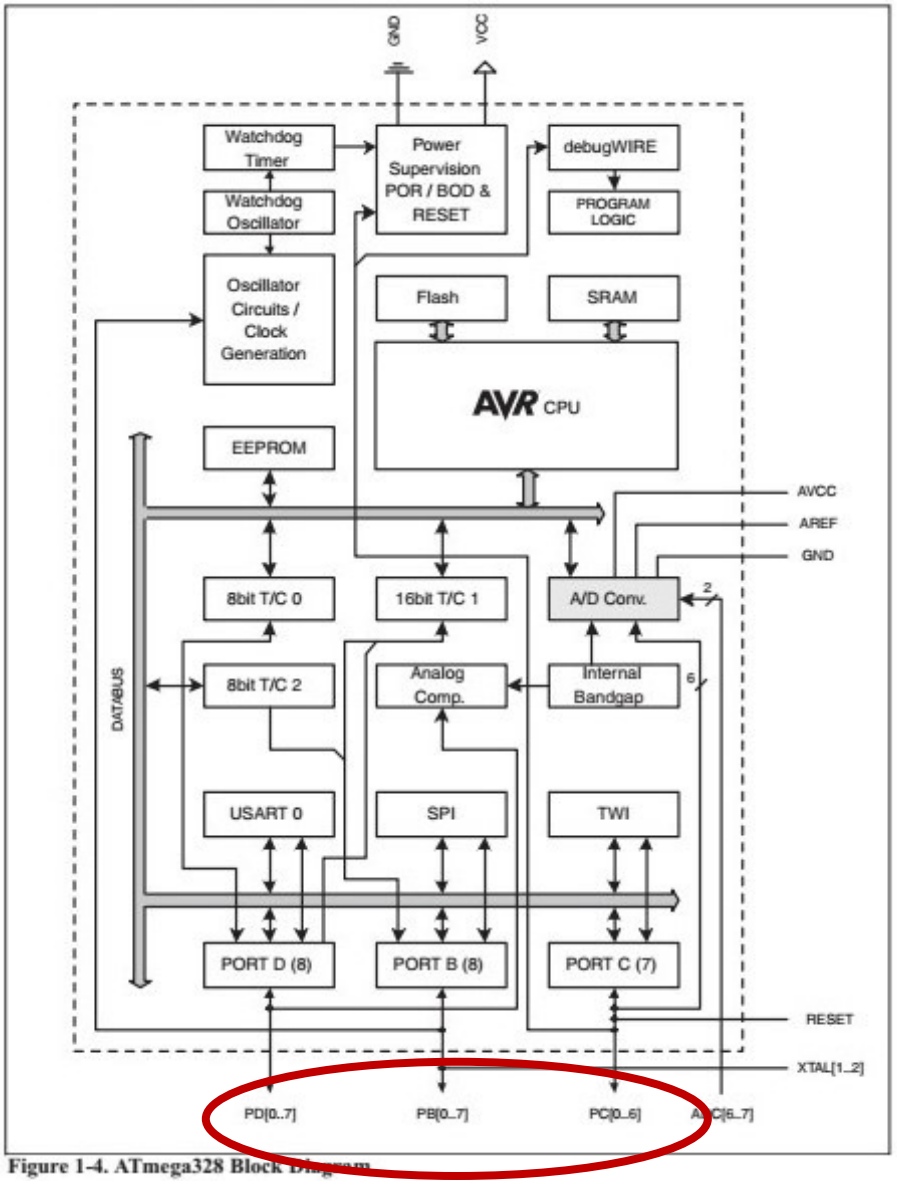


Figure 1-4. ATmega328 Block Diagram

$3.51 Digikey

28 pin PDIP

32 pin Thin Quad Flat Package (TQFP)

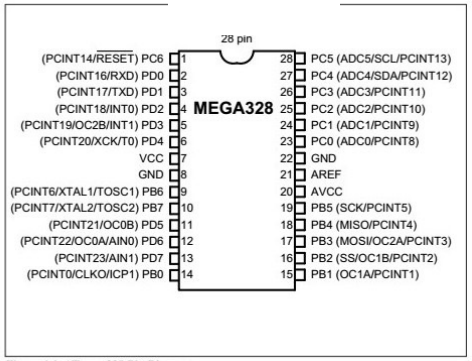


Figure 4-1. ATmega328 Pin Diagram

- 23 General Purpose I/O (GPIO) pins
  - PORTB Pins PB0-PB7
  - PORTD Pins PD0-PD7
  - PORTC Pins PC0-PC6
- 8 Analog to Digital Converter (ADC) pins
- 3 Timers
- UART for serial communication

- 32K Byte Flash ROM for Code
- 2K Byte data RAM
- 1K Byte data EEPROM

ATtiny25 MCU (8 pin package)

$1.19 from Digikey

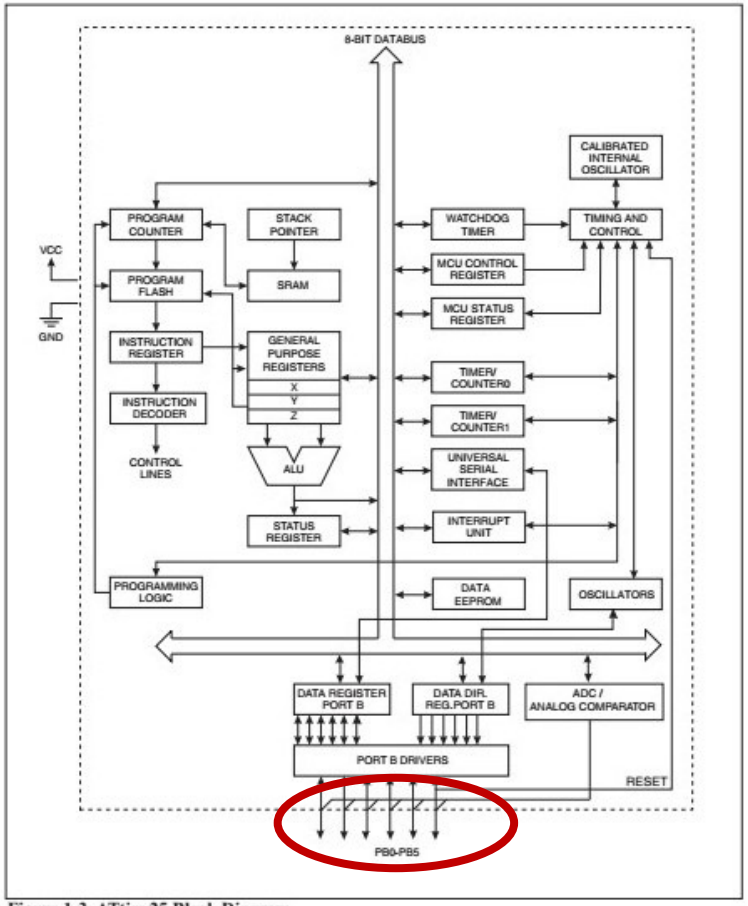
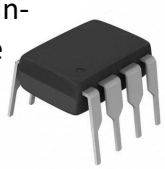


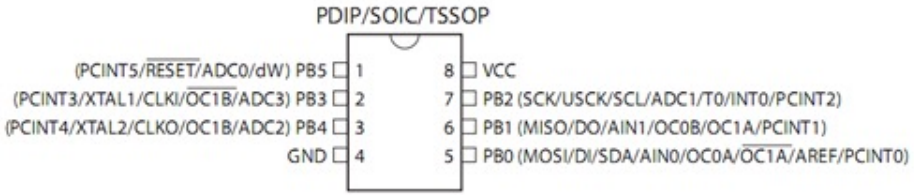
Figure 1-3. ATtiny25 Block Diagram

Plastic Dual In-Line Package (PDIP)



Small Outline Integrated Circuit Package (SOIC)

Pinout ATtiny25/45/85



- 6 General Purpose I/O (GPIO) pins
  - PORTB Pins PB0-PB5
- 4 Analog to Digital Converter (ADC) pins
- 2 Timers
- no communication

- 2K Byte Flash ROM for Code
- 128 Byte data RAM
- 128 Byte data EEPROM

ATmega328P vs. Arduino Uno

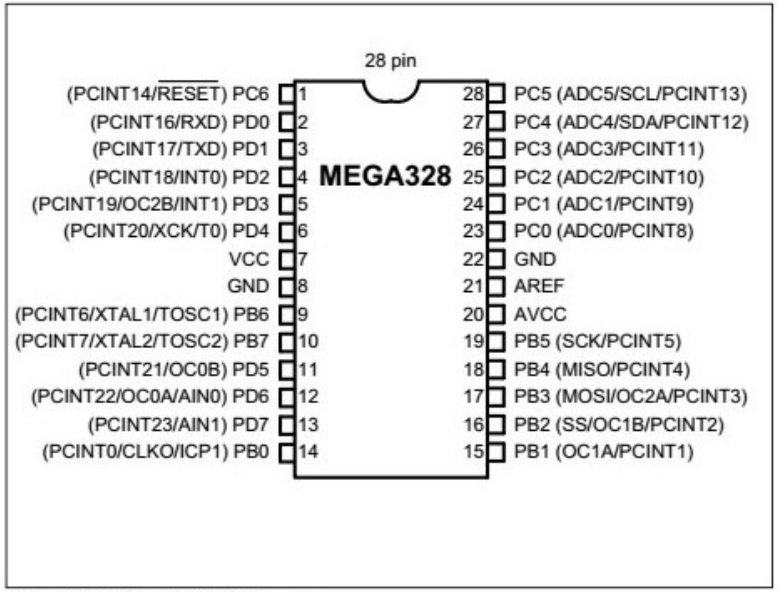


Figure 4-1. ATmega328 Pin Diagram

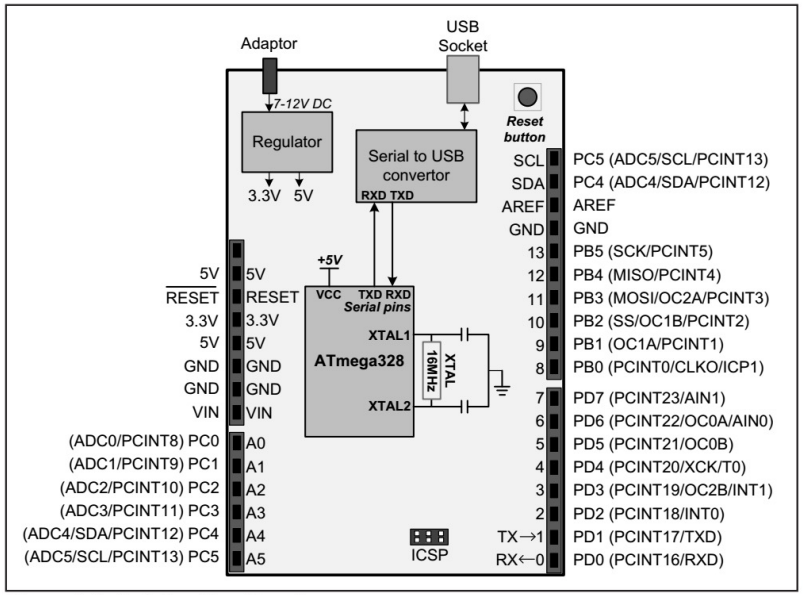
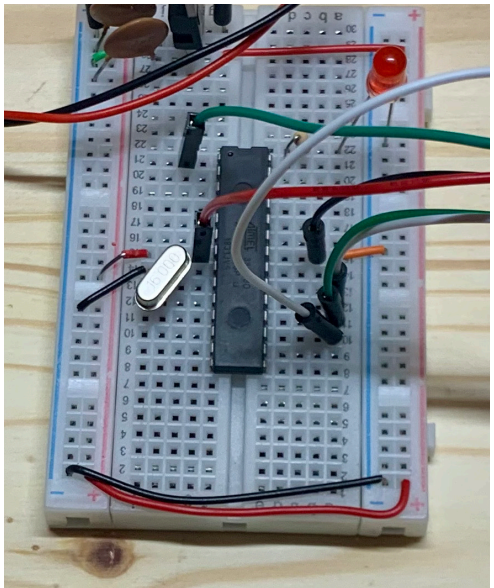
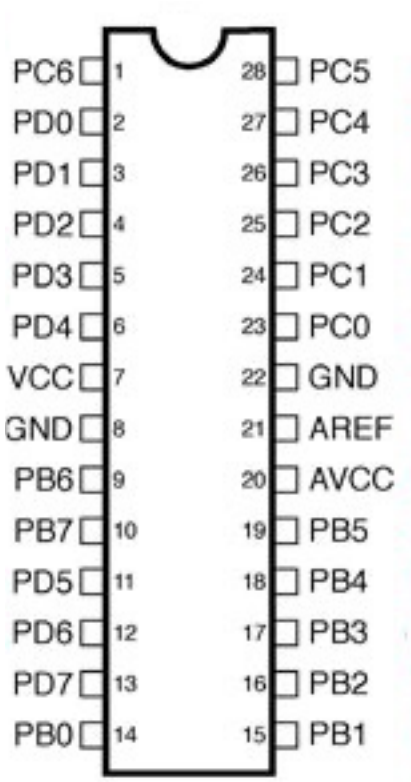
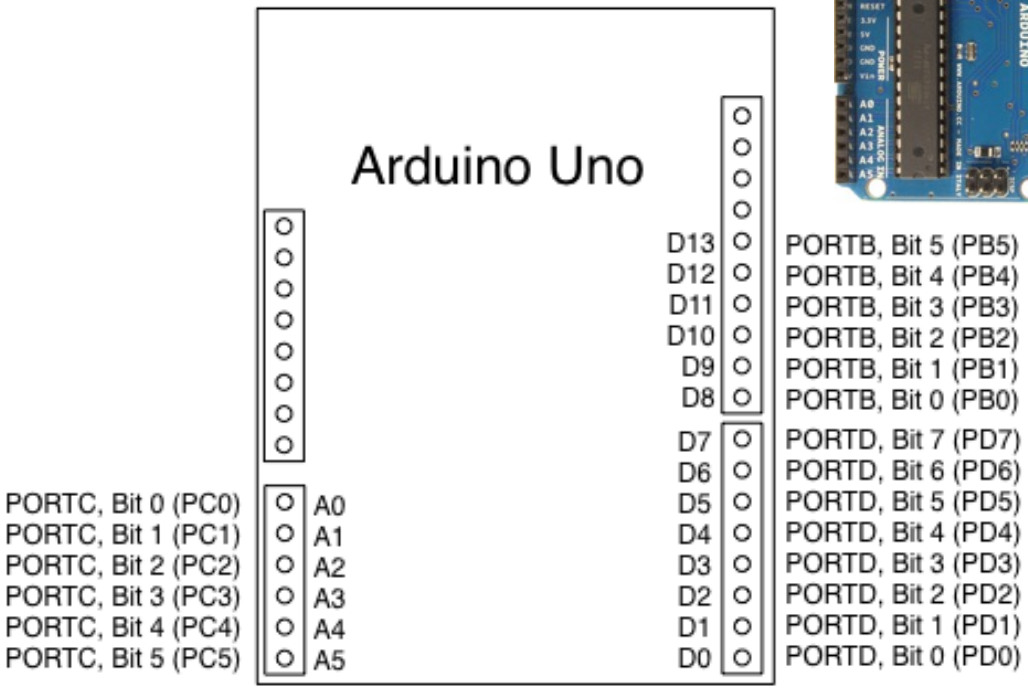


Figure 8-5. Ardu[illegible]



Pin Correspondence Between ATmega328p DIP (bare IC) and Arduino Uno (Dev Board)





PORTB	
5	D13
4	D12
3	D11
2	D10
1	D9
0	D8

PORTC	
5	A5
4	A4
3	A3
2	A2
1	A1
0	A0

PORTD	
7	D7
6	D6
5	D5
4	D4
3	D3
2	D2
1	D1
0	D0

Source: https://bytes.usc.edu/ee109/arduino-ports/

The structure of I/O ports

3 8-bit memory registers per port

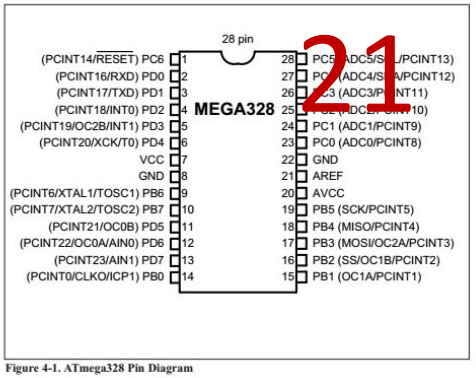
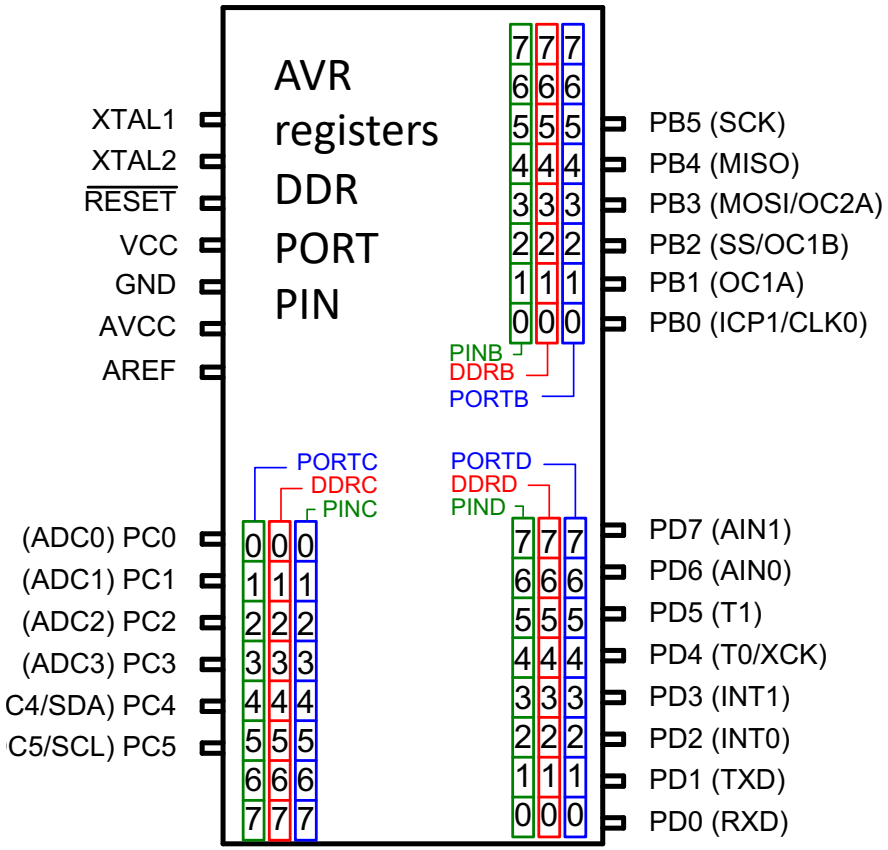


Figure 4-1. ATmega328 Pin Diagram

DDRx = Data Direction Register
- DDRB = Port B Data Direction Register
- DDRC = Port C Data Direction Register
- DDRD = Port D Data Direction Register

PORTx = Data Register
- PORTB = Port B Data Register
- PORTC = Port C Data Register
- PORTD = Port D Data Register

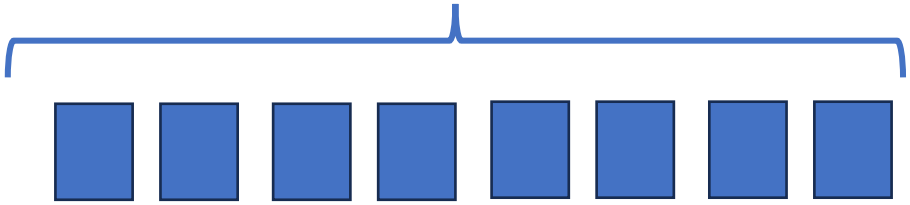
PINX = Input Pins Register
- PINB = Port B input Pin register
- PINC = Port C input pins register
- PIND = Port D input pins register



Px refers to the I/O pins.
PB, PC, PD

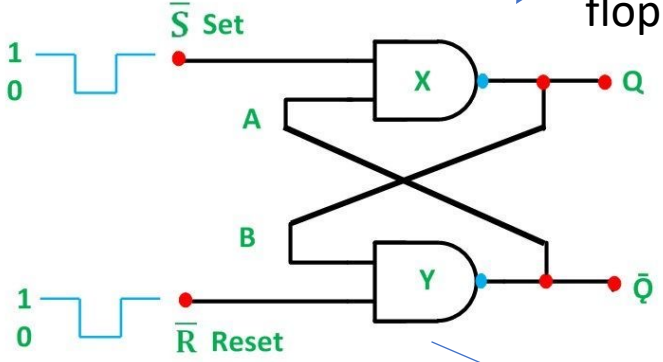
what is a memory register?

8 flip flops



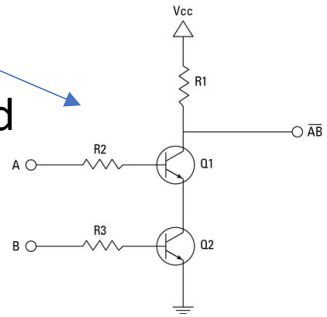
data (contents)
address

each flip flop

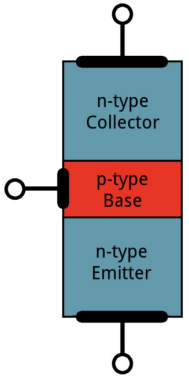


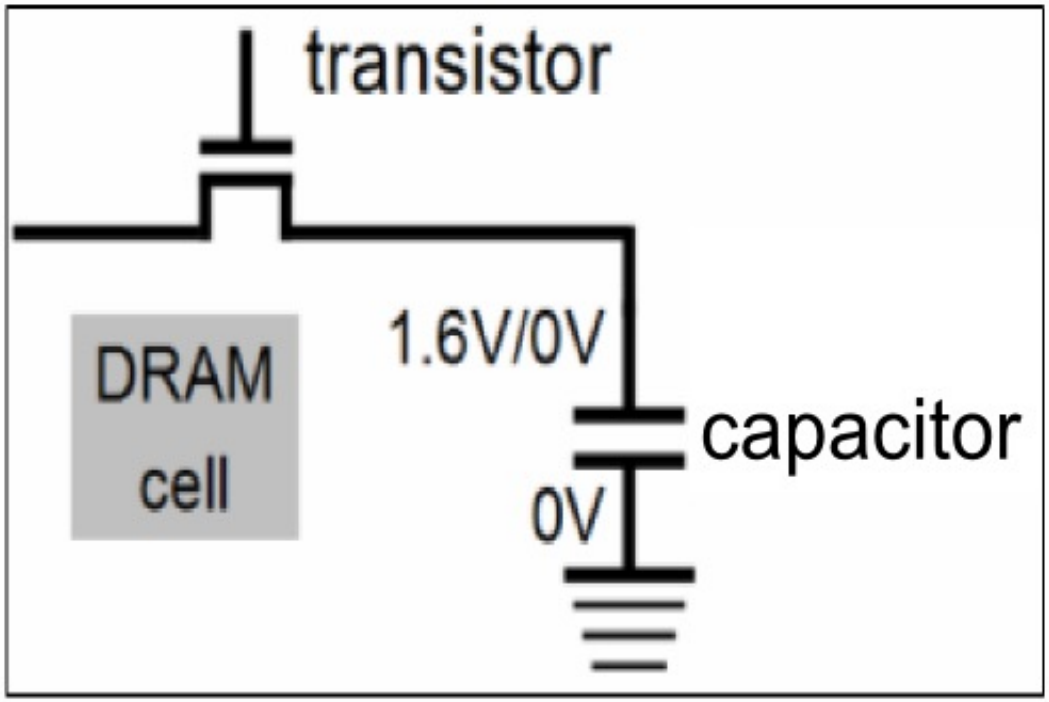
A rough calculation:
1 8-bit register requires 8x2x2 = 32 transistors.
Our ATmega328p has 32Kbyte ROM, so ~ 32 x 32 x 1000 = 1 x 10^6

each nand gate



each transistor



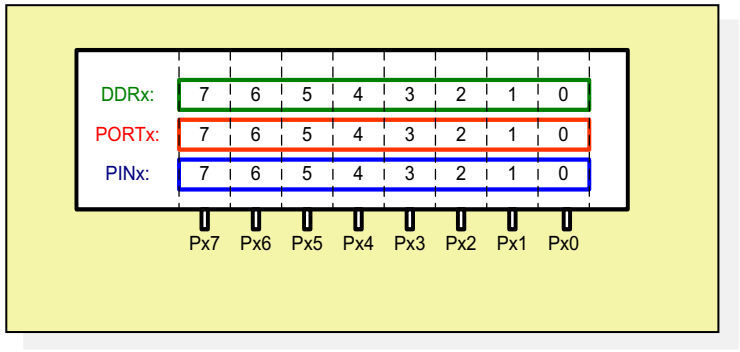


Another calculation:
1 8-bit register requires 8 transistors.
Our ATmega328p has 32Kbyte ROM, so 256,000 transistors

The answer is likely a few hundred k transistors for the 328P

Apple M1 Pro has 33,700,000,000 (33.7 B transistors)
Xilinx Versal VP1802 FPGA has 92,000,000,000
Nvidia's Grace Brackwell (deep learning) has 208B MOSFETS

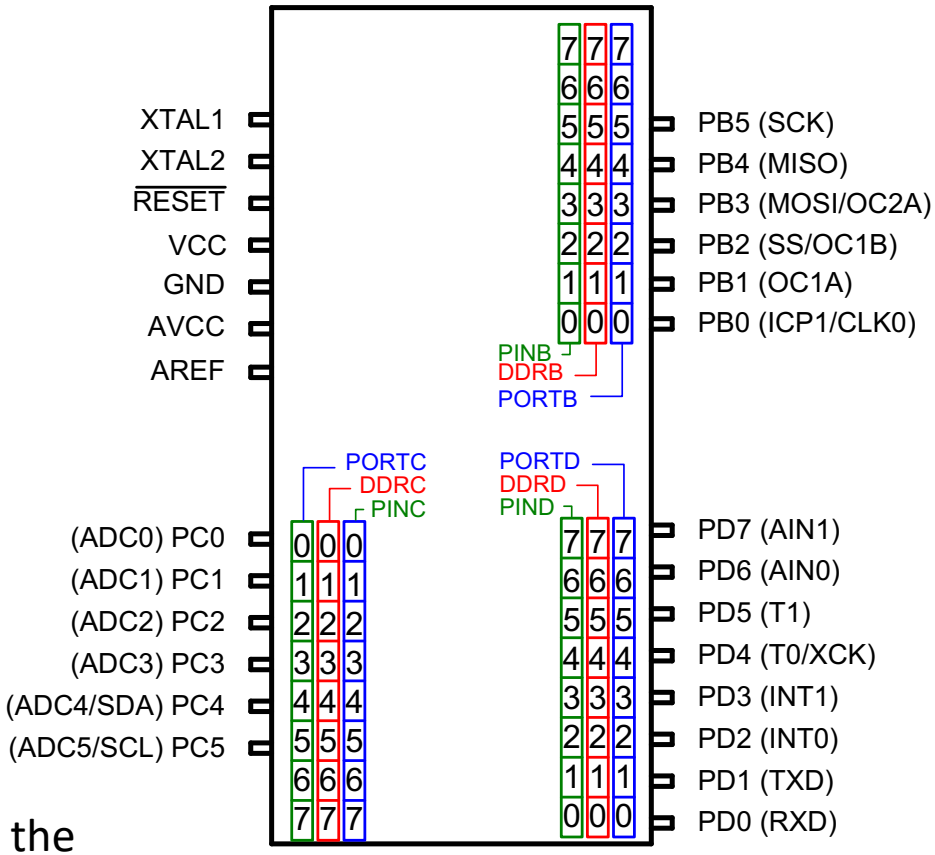
The structure of I/O pins & registers



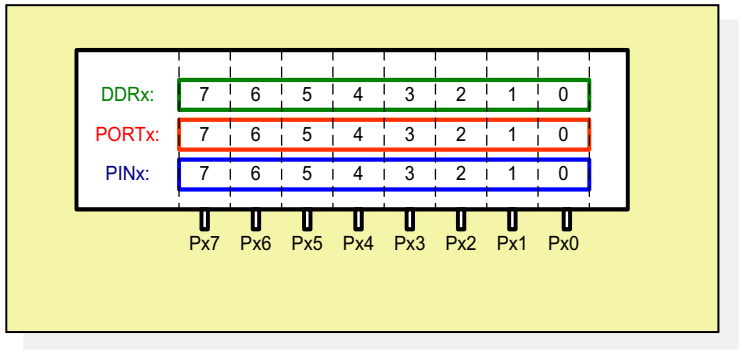
software: we access the ports by writing to and reading from these registers. Example:

```
DDRB =  0xFF; //set as outputs
DDRB =  0x00; //set as inputs
PORTB = 0x00; //set pins low
input_value = PINC; //read pins
```

hardware: we physically connect things to the port pins (LEDs, motors, sensors, ...)



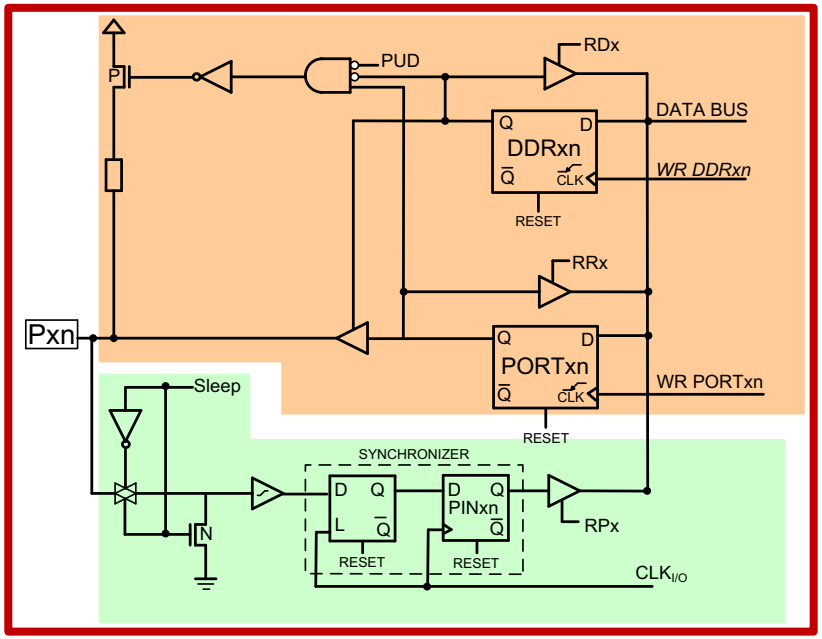
The structure of I/O pins & registers



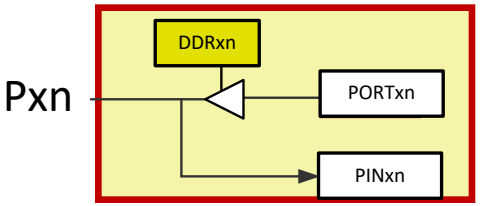
we access the ports by writing to and reading from these registers.

Example:

```
DDRB =  0xFF; //set as outputs
DDRB =  0x00; //set as inputs
PORTB = 0x00; //set pins low
input_value = PINC; //read pins
```

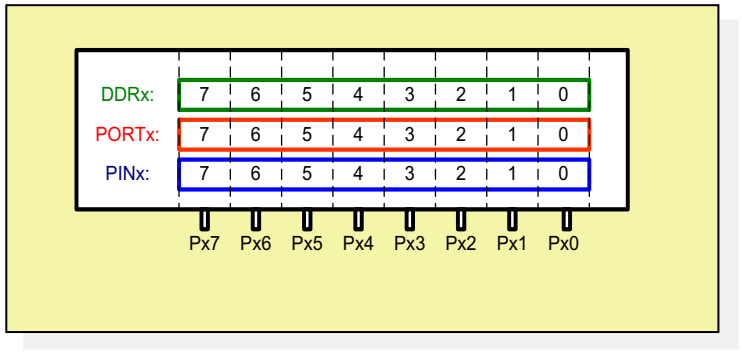


circuit views of the configuration of each bit in a port



simplified view

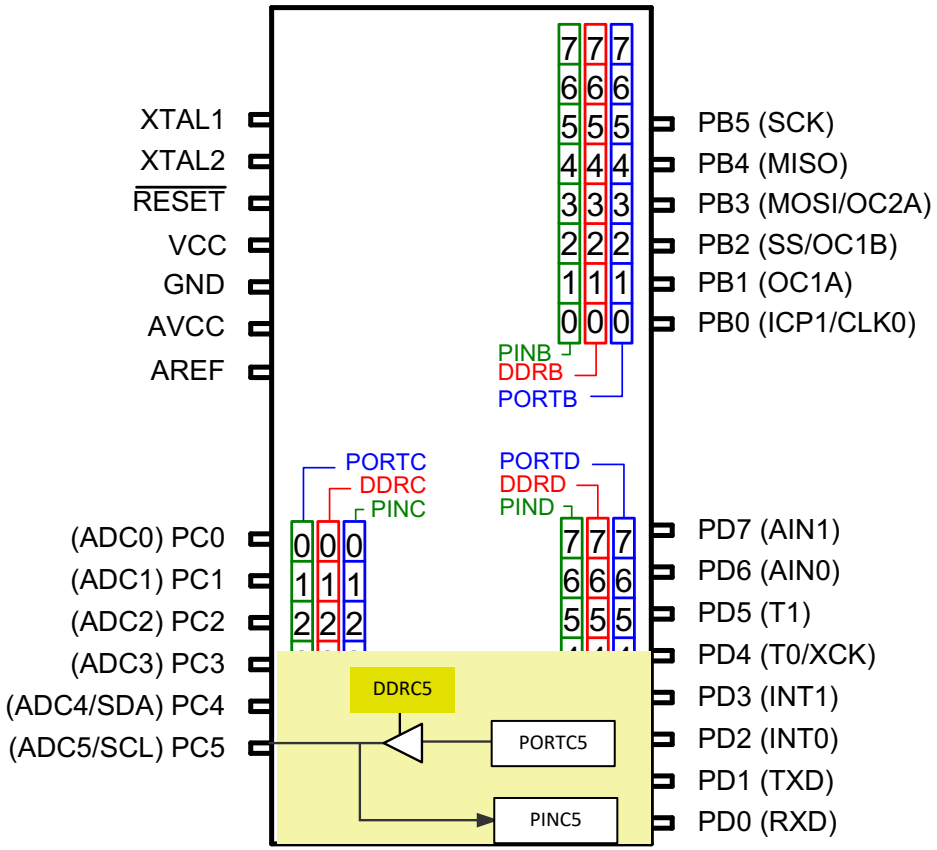
The structure of I/O pins & registers



we access the ports by writing to and reading from these registers.

Example:

```
DDRB =  0xFF; //set as outputs
DDRB =  0x00; //set as inputs
PORTB = 0x00; //set pins low
input_value = PINC; //read pins
```



ink.c × ECE231 Lab Assignment 8.0.pdf makefile

> C blink.c > main(void)

```c
/*****************************************************************
* blink.c   -- Blink an LED on Port B pin5 (PB5).
* This is the built-in LED (pin13) on the Arduino Uno dev board.
* Date          Author          Revision
* 12/14/21      D. McLaughlin   initial code creation
* 1/9/22        D. McLaughlin   tested on host MacOS Monterey, Apple M1 pro
* 2/12/22       D. McLaughlin   cleaned up formatting, added comments
* 2/21/23       D. McLaughlin   changed PB5 to DDB5 and PORTB5 in lines 17, 20, 22
* ****************************************************************/

#include <avr/io.h>             // Defines port pins
#include <util/delay.h>         // Declares _delay_ms
#define MYDELAY 1000            // This will be the delay in msec

int main(void){

    DDRB = 1<<DDB5;             // Initialize PB5 as output pin

    while(1){                   // Loop forever
        PORTB = 1<<PORTB5;      // Make PB5 high; all other PORTB pins low; LED ON
        _delay_ms(MYDELAY);     // Wait
        PORTB = ~ (1<<PORTB5); // Make PB5 low; all other PORTRB pins high;  LED off
        _delay_ms(MYDELAY);     // Wait
    }

    return 0;                   // Code never gets here.
}

/******* End of File *********/
```

#include <avr/io.h>

io.h

```
#  include <avr/iom325pa.h>
#elif defined (__AVR_ATmega3250__)
#  include <avr/iom3250.h>
#elif (defined __AVR_ATmega3250A__)
#include <avr/iom3250a.h>
#elif defined (__AVR_ATmega3250P__)
#  include <avr/iom3250p.h>
#elif defined (__AVR_ATmega3250PA__)
#  include <avr/iom3250pa.h>
#elif defined (__AVR_ATmega328P__)
#  include <avr/iom328p.h>
#elif (defined __AVR_ATmega328__)
#include <avr/iom328.h>
#elif defined (__AVR_ATmega329__)
```

full path to this header file:
/Applications/Arduino.app/Contents/Java/hardware/tools/avr/avr/include/avr

iom328p.h

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

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

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

#define PINC _SFR_IO8(0x06)
#define PINC0 0
#define PINC1 1
```

these are the actual memory addresses of the I/O port registers.

We don't need to know these.
#include<avr/io.h> defines symbolic constants that allow us to use PINB, DDRB, PORTB, etc...instead of needing to know the addresses.

recall: the machine code interacts with the physical world by doing reads & writes to registers (memory locations) associated with I/O ports

C blink.c > main(void)

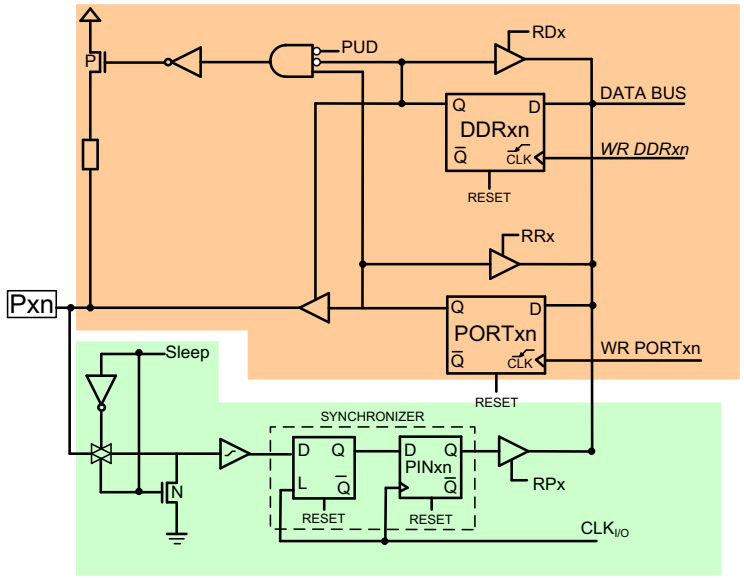
```
/*************************************************************************
* blink.c   -- Blink an LED on Port B pin5 (PB5).
* This is the built-in LED (pin13) on the Arduino Uno dev board.
* Date          Author          Revision
* 12/14/21      D. McLaughlin   initial code creation
* 1/9/22        D. McLaughlin   tested on host MacOS Monterey, Apple M1 pro
* 2/12/22       D. McLaughlin   cleaned up formatting, added comments
* 2/21/23       D. McLaughlin   changed PB5 to DDB5 and PORTB5 in lines 17, 20, 22
* ************************************************************************/

#include <avr/io.h>             // Header file for I/O pins
#include <util/delay.h>         // Declares _delay_ms
#define MYDELAY 1000            // This will be the delay in msec

int main(void){

    DDRB = 1<<DDB5;             // Initialize PB5 as output pin

    while(1){                   // Loop forever
        PORTB = 1<<PORTB5;      // Make PB5 high; all other PORTB pins low; LED ON
        _delay_ms(MYDELAY);     // Wait
        PORTB = ~ (1<<PORTB5); // Make PB5 low; all other PORTRB pins high;  LED off
        _delay_ms(MYDELAY);     // Wait
    }

    return 0;                   // Code never gets here.
}
```

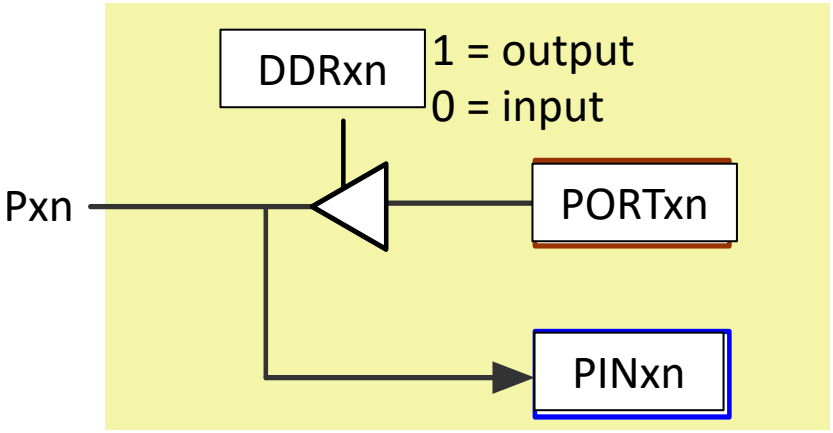
#define DDRB _SFR_IO8 (0x04)
#define PORTB _SFR_IO8 (0x05)
#define PB5 5
#define PINB5 5

I/O pins and port register logic

DDRXn =1 (pin set as output)		
PORTxn	Pxn (output voltage)	Pxn (output logic)
0 (clear)	0V	0 (low)
1 (set)	5V	1 (high)

DDRXn =0 (pin set as input)		
PORTxn	Pxn (input voltage)	PINxn (input logic)
0, high Z	0V	0
0, high Z	5V	1
0, high Z	float	don't use
1, pullup	0V	0
1, pullup	5V	1
1, pullup	float	1





Example 1

- Write a program segment that sets all the pins of PORTD as high (5V) output.

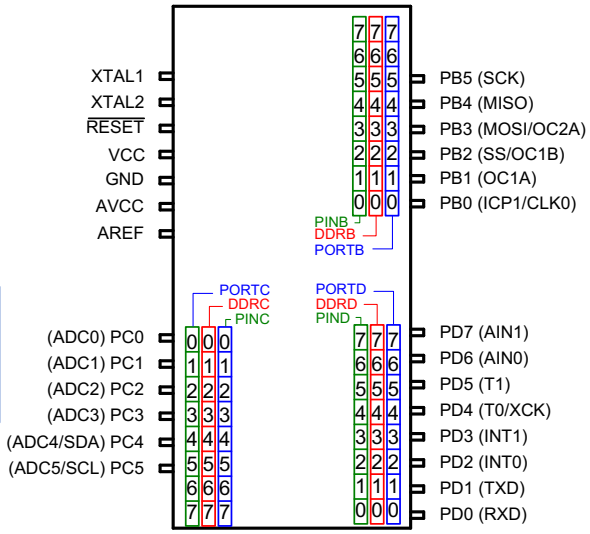
DDRXn =1 (pin set as output)		
PORTxn	Pxn (output voltage)	Pxn (output logic)
0 (clear)	0V	0 (low)
1 (set)	5V	1 (high)

DDRB:	1	1	1	1	1	1	1	1
PORTB:	1	1	1	1	1	1	1	1

```
DDRD = 0b11111111;  // binary
PORTD = 0b11111111;
```

```
DDRD = 0xFF;        //hex
PORTD = 0xFF;
```

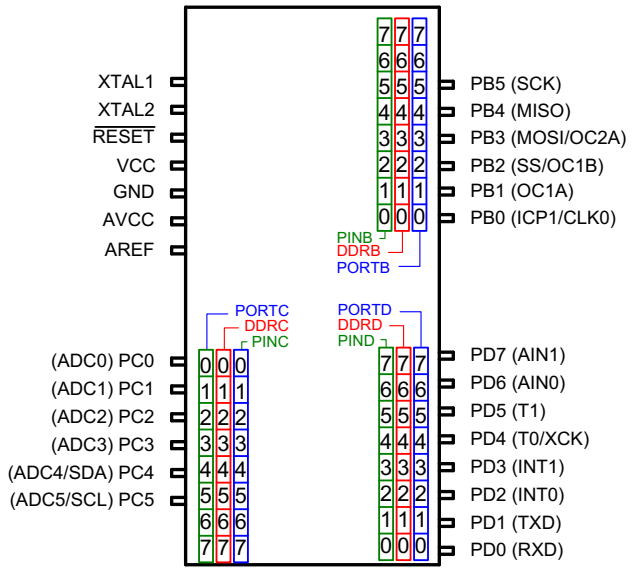
```
DDRD = 255;  //decimal. More difficult
PORTD = 255;
```



Example 1, cont.

- Write a complete program that makes PD0-PD7 output pins and sets PD0-PD3 low and PD4-PD7 high.

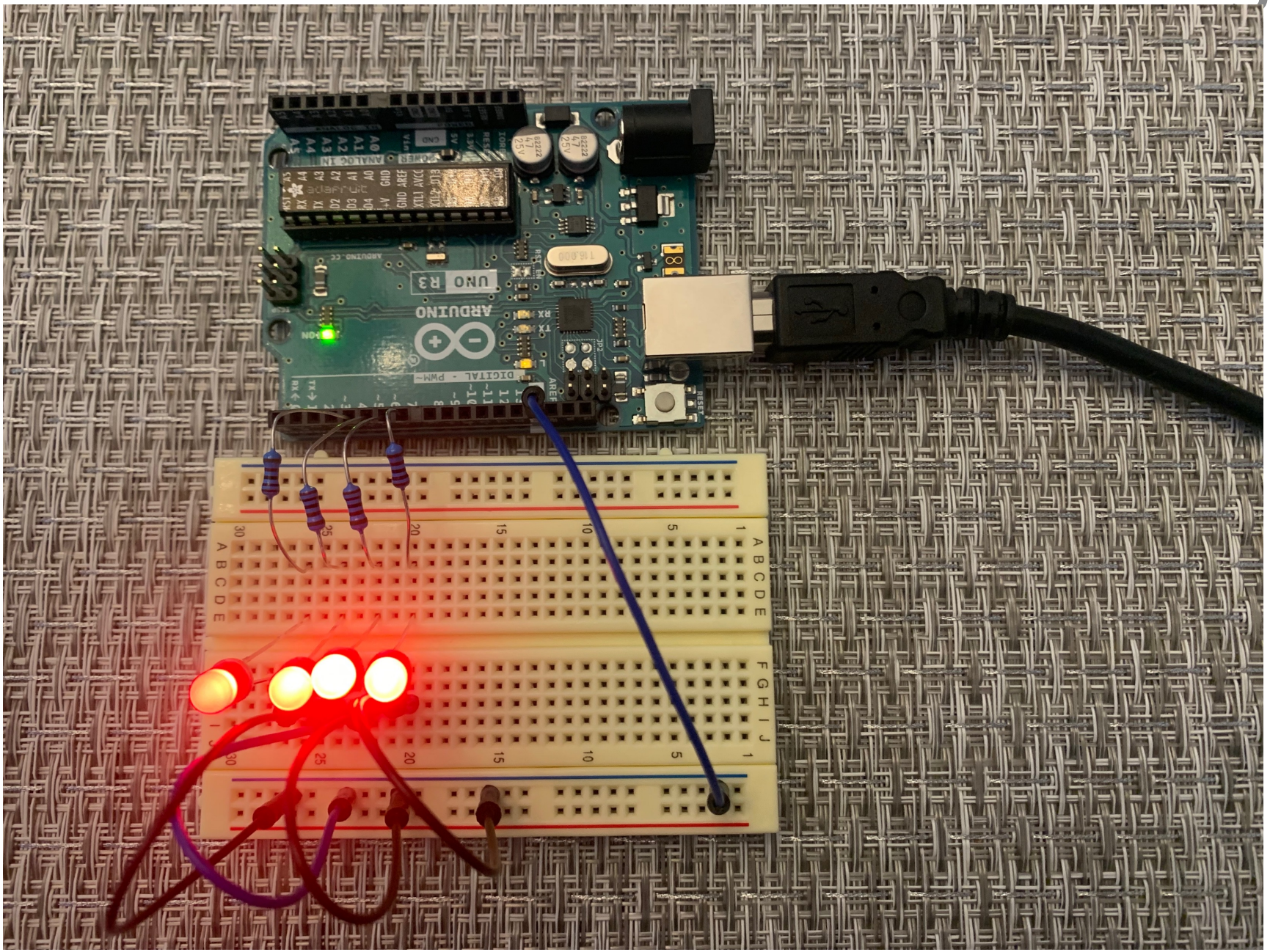
DDRB:	1	1	1	1	1	1	1	1
PORTB:	1	1	1	1	0	0	0	0




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

int main(void)
{
    DDRD = 0xFF;   // Hex for 1111 1111
    PORTD = 0xF0; // Hex for 1111 0000

    while(1);     // Sit here forever

    return(0);
}
```

~/Documents/GitHub/ECE231/example1

```
/*********************************
 * example1.c makes PC4-PC7 high and
 * glows LEDs connected to these pins.
 * These are pins 4-7 on Arduino Uno
 * Date          Author          Revision
 * 2/14/22       D. McLaughlin   initial release
 * ********************************/

#include <avr/io.h>

int main(void){
    DDRD = 0xFF;     // Corresponds to 1111 1111
    PORTD = 0xF0;    // Corresponds to 1111 0000

    while(1);        // Wait forever
}
```

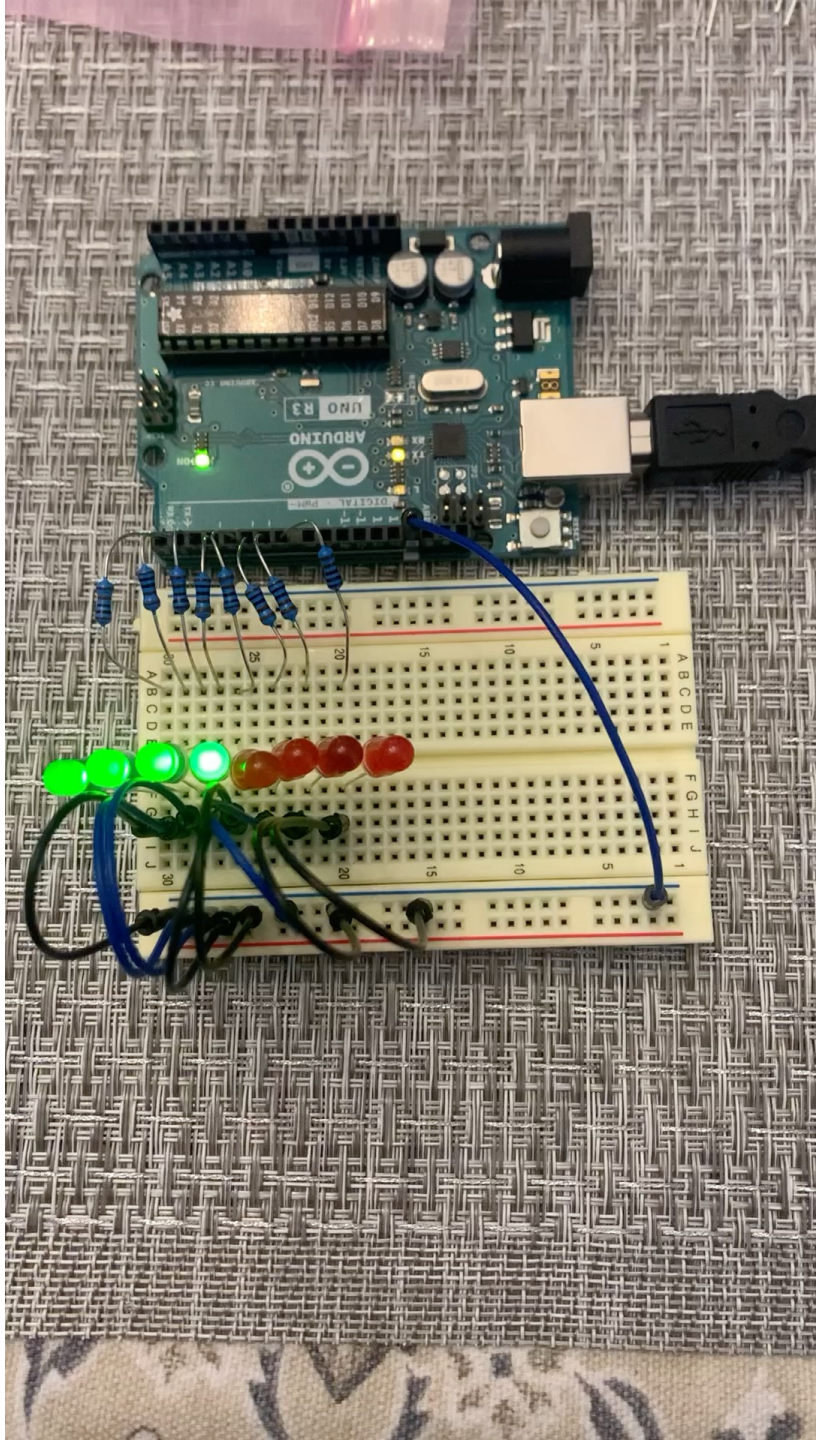

UNO R3
ARDUINO
DIGITAL - PWM~
ANALOG IN
POWER

Example 2

- The following code will alternately blink LEDS on the upper and lower 4 bits of Port D forever with 1/4 sec time delay between blinks.

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

int main(void)
{
    DDRD = 0xFF;          //set all pins of PORTD as output

    while(1)
    {
        PORTD = 0xF0;     // PD0-PD3 low; PD4-PD7 high
        _delay_ms(250);   // 250 msec delay
        PORTD = 0x0F;     //  PD0-PD3 high; PD4-PD7 low
        _delay_ms(250);   // 250 msec delay
    }
    return (0);
}
```

```
/*******************************
 * example2.c Alternately blinks LEDS on PD0-PD3 and
 * PD4-PD7 with 250 mS beteeen blinks.
 * Uses pins 0-3 (PD0-PD3) and 4-7 (PD4-PD7) on Arduino Uno
 * Date          Author          Revision
 * 2/14/22       D. McLaughlin   initial release
 * *******************************/

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

int main(void){

    DDRD = 0xFF;          // Make all pins output

    while(1){             // Loop forever
        PORTD = 0xF0;     // PD0-PD3 low; PD4-PD7 high
        _delay_ms(250); // Wait 250 mSec
        PORTD = 0x0F;     // PD0-PD3 high; PD4-PD7 low
        _delay_ms(250);
    }

    return(0);
}
```

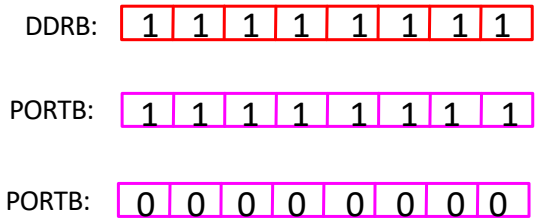

UNO R3
ARDUINO

avr-gcc compiler supports byte size I/O programming in C (we can't write to an individual bit in a register)

Suppose we want to toggle an LED on PB5

```
DDRB = 0xFF;           // FF=1111 1111 all output
while (1)
{
     PORTB = 0xFF;     // FF=1111 1111 all bits high
     _delay_ms(500);
     PORTB = 0x00;     // 00=0000 0000 all bits low
     _delay_ms(500);
}
```

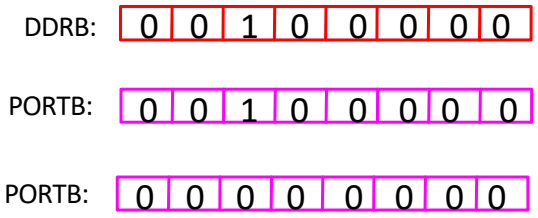
This is a brute force approach:



Suppose we want to toggle *only* the built-in LED (PB5, or Arduino Uno pin 13)

```
DDRB = 0b00100000; // PB5 output; others input
while (1)
{
     PORTB = 0b00100000;  // PB5 high;others low
     _delay_ms(500);
     PORTB = 0b00000000;  // all PORTB pins low
     _delay_ms(500);
}
```

Still brute force: we're still programming *all* the pins in register B at once



Q: How can we configure individual pins (set, clear, or toggle individual bits) without impacting the other pins in a register?

A: Configure registers using bit-wise logical operations.

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

int main(void)
{
    DDRB|=(1<<5); // DDB5 high. No impact on other bits of DDDRB
    PORTB|=(1<<5); // PORTB5 high. No impact on other PORTB bits
    PORTB&=~(1<<5);      // PORTB5 low. No impact on other PORTB bits
    PORTB^=(1<<5);       // Toggle PORTB5. No impact on other PORTB bits.
}
```

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

int main(void)
{
    DDRB|=(1<<DDB5); // DDB5 high. No impact on other DDRB bits
    PORTB|=(1<<PORTB5);    // PORTB5 high. No impact on other PORTB bits
    PORTB&=~(1<<PORTB5);   // PORTB5 low. No impact on other PORTB bits
    PORTB^=(1<<PORTB5); // Toggle PORTB5. No impact on other PORTB bits.
}
```

Recall C logical operators: &&, ||, !
They operate on logical objects and return a boolean result (1 or 0)

ex: 4>5 && 10 returns 0 4>5 || 10 returns 1 !0 returns 1

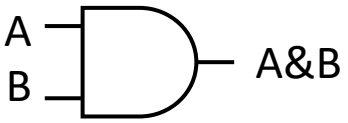
We will now look at bitwise operators: &, |, ^, ~
They operate on the individual bits in an integer (char, int, long) and return a set of bits.

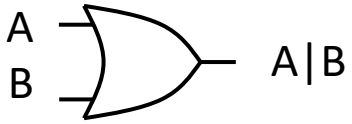
ex: 11110000 & 00110000 returns 00110000
11110000 | 00001111 returns 11111111

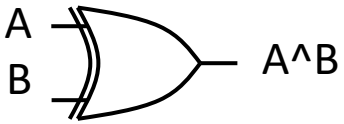
Bit-wise logical operators

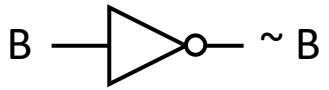
Table 7-3: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	









Bit-wise logical operators applied to 8 bit Bytes:

Table 7-3: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Byte-wise AND (&) example

```
  1111 1000
& 1101 1111
-------------
  1101 1000
```

Byte-wise OR (|) example

```
  0000 0110
| 0010 0010
-------------
  0010 0110
```

Byte-Wise EX-OR (^) example

```
  1110 1011
^ 1111 1100
-------------
  0001 0111
```

AND with 1, get the original bit value
AND with 0, get 0

OR with 1, get 1.
OR with 0, get original bit value

EX-OR with 1, toggle: original 1 becomes 0; original 0 becomes 1.
EX-OR with 0, get original bit value

Bit-wise logical operators

Table 7-3: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Clearing a bit using & mask

```
unsigned char x,y,z;
x = 0b11111111;
y = 0b11011111;
z = x&y; // = 1101 1111
```

Clears bit 5; doesn't impact other bits

```
  1111 1111
& 1101 1111
-------------
  1101 1111
```

Setting a bit using | mask

```
unsigned char x,y,z;
x = 0b00001111;
y = 0b00100000;
z = x|y; // = 0010 1111
```

Sets bit 5; doesn't impact other bits

```
  0000 1111
| 0010 0000
-------------
  0010 1111
```

Bit-wise logical operators

Table 7-3: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Toggling a bit using ^ mask

```
unsigned char x,y,z;
x = 0b11110000;
y = 0b00100000;
z = x^y; // = 1101 0000
```

Toggles bit 5; doesn't impact other bits

```
  1111 0000
^ 0010 0000
  ----------
  1101 0000
```

Bit-wise logical operators

Table 7-3: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

binary operators (operate on two operands)

A&B

A|B

A^B

unary operator (operate on one operand)

~B

Inverting all the bits of a variable

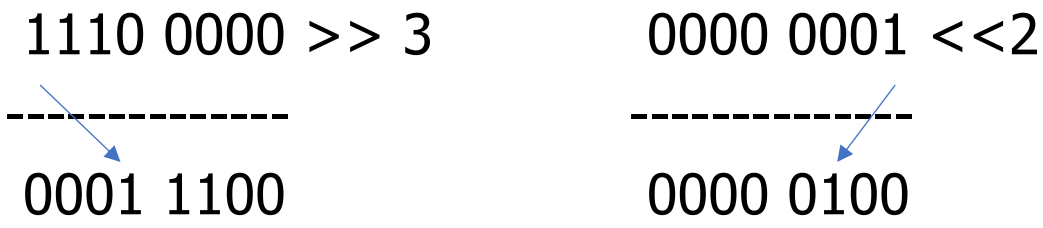
```
unsigned char x,z;
x = 0b11101011;
z = ~y;  // = 0001 0100
```

~ 1110 1011

0001 0100

Shift operations in C

- data >> number of bits to be shifted right
- data << number of bits to be shifted left



What does 1<<3 represent?
0b00000001 << 3 = 0b00001000

SO:
1<<3 IS THE SAME AS 0B00001000
1<<7 IS THE SAME AS 0B10000000
1<<0 IS THE SAME AS 0B00000001

What does 0<<3 represent?
0b00000000 << 3 = 0b00000000

SO:
0<<3 IS THE SAME AS 0B00000000
0<<7 IS THE SAME AS 0B00000000
0<<0 IS THE SAME AS 0B00000000

Setting a bit in a Byte
("Setting a bit" means make it go high)

- We can use | operator to set a bit of a byte to 1

```
 xxxx xxxx                 xxxx xxxx
|0001 0000      OR        |1 << 4
 ------------              ------------
 xxx1 xxxx                 xxx1 xxxx
```

```
PORTB = PORTB | 0b00010000; // Set bit 4 (5th bit) of PORTB
```

```
PORTB | =  0b00010000; // Set bit 4 (5th bit) of PORTB
```

```
PORTB = PORTB | (1<<4); // Set bit 4 (5th bit) of PORTB
```

```
PORTB |= (1<<4); // Set bit 4 (5th bit) of PORTB
```

Sets PORTB4=1. Leaves all other bits of PORTB untouched.

```
PORTB |= (1<<4)|(1<<3)|(1<<2) ; // Set bits 4, 3, and 2 of PORTB
```

Sets PORTB2, PORTB3, PORTB4=1. Other bits of PORTB untouched.

Clearing a bit in a Byte
("Clearing a bit" means make it go low)

- We can use & operator to clear a bit of a byte to 0

```
  xxxx  xxxx
& 1110 1111
------------
  xxx0  xxxx
```

```
  xxxx xxxx
& ~(1 << 4)
------------
  xxx0 xxxx
```

```
  xxxx xxxx
& ~(0001 0000)
------------
  xxx0 xxxx
```

```
  xxxx xxxx
& (1110 1111)
------------
  xxx0 xxxx
```

```
PORTB = PORTB & 0b11101111; // Clear bit 4 (5th bit) of PORTB
```

```
PORTB &= 0b11101111; // Clear bit 4 (5th bit) of PORTB
```

```
PORTB &= ~(1<<4); // Clear bit 4 (5th bit) of PORTB
```

Clears PORTB4 (Sets PORTB4=0). Leaves all other bits of PORTB untouched.

Toggling a bit in a Byte
("Toggling a bit" means 1→0 or 0 →1)

- We can use ^ operator to toggle a bit of a byte

```
  xxxx  xxxx
^ 0001 0000
  ------------
  xxx̄x  xxxx
```

```
  xxxx xxxx
^  (1 << 4)
  ------------
  xxx̄x xxxx
```

```
  xxxx xxxx
^(0001 0000)
  ------------
  xxx̄x xxxx
```

```
PORTB = PORTB ^ 0b00010000; // Toggle bit 4 (5th bit) of PORTB
```

```
PORTB ^= 0b00010000; // Toggle bit 4 (5th bit) of PORTB
```

```
PORTB ^= (1<<4); // Toggle bit 4 (5th bit) of PORTB
```

Clears PORTB4 (Sets PORTB4=0). Leaves all other bits of PORTB untouched.

Example 9-41

Write a C program to toggle only the PORTB.4 bit continuously every 1 ms. Use Timer1, Normal mode, and no prescaler to create the delay. Assume XTAL = 16 MHz.

Solution:

XTAL = 16 MHz ➔ $T_{machine\ cycle}$ = 1/16 MHz = 0.0625 µs
Prescaler = 1:1 ➔ T_{clock} = 0.0625 µs
1 ms/0.0625 µs = 16,000 clocks = 0x3E80 clocks

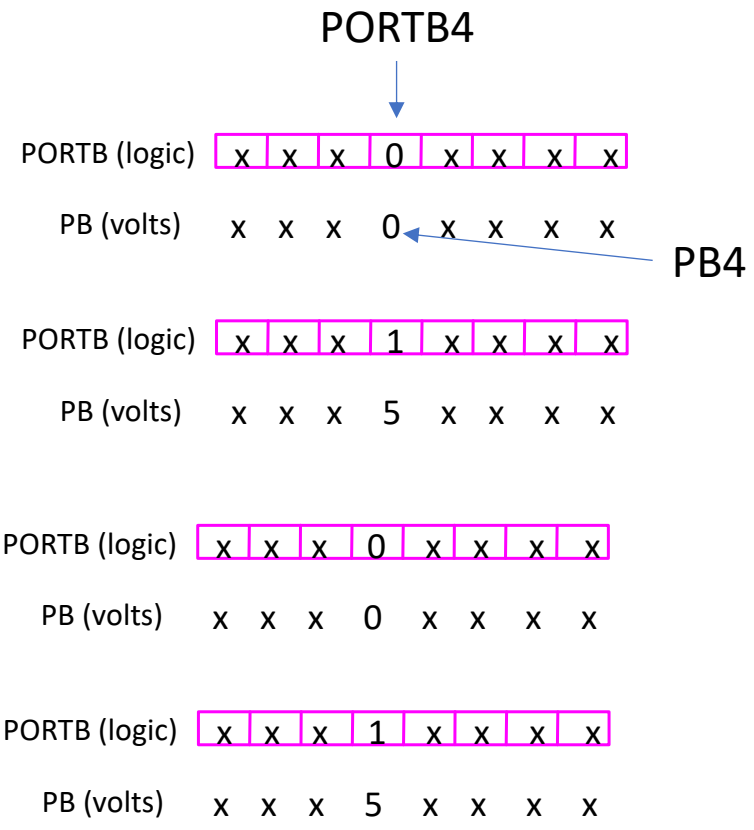
1 + 0xFFFF – 0x3E80 = 0xC180

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

void T1Delay ( );

int main ( )
{
      DDRB = 0xFF;            //PORTB output port

      while (1)
      {
            PORTB ^= (1<<4);    //toggle PB4
            T1Delay ( );        //delay size unknown
      }
}


void T1Delay ( )
{
      TCNT1H = 0xC1;    //TEMP = 0xC1
      TCNT1L = 0x80;

      TCCR1A = 0x00;    //Normal mode
      TCCR1B = 0x01;    //Normal mode, no prescaler

      while ((TIFR1&(0x1<<TOV1))==0);     //wait for TOV1 to roll over

      TCCR1B = 0;
      TIFR1 = 0x1<<TOV1;      //clear TOV1
}
```

Example problem from Chapter 9 of the text. Ignore TIDelay() for now.

Notice the different ways that bit 4 of PORTB is referred to:

PORTB.4
PB4
(1<<4)



ATmega328P full datasheet.pdf (SECURED)

Home Tools ATmega328P full... ×

1 / 662 73.1%

Bookmarks

MICROCHIP

ATmega48A/PA/88A/PA/168A/PA/328/P

megaAVR® Data Sheet

Introduction

The ATmega48A/PA/88A/PA/168A/PA/328/P is a low power, CMOS 8-bit microcontrollers based on the AVR® enhanced RISC architecture. By executing instructions in a single clock cycle, the devices achieve CPU throughput approaching one million instructions per second (MIPS) per megahertz, allowing the system designer to optimize power consumption versus processing speed.

Features

- High Performance, Low Power AVR® 8-Bit Microcontroller Family
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 20 MIPS Throughput at 20MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory Segments
  - 4/8/16/32KBytes of In-System Self-Programmable Flash program memory
  - 256/512/512/1KBytes EEPROM
  - 512/1K/1K/2KBytes Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C(1)
  - Optional Boot Code Section with Independent Lock Bits
    - In-System Programming by On-chip Boot Program
    - True Read-While-Write Operation
  - Programming Lock for Software Security
- QTouch® library support
  - Capacitive touch buttons, sliders and wheels
  - QTouch and QMatrix™ acquisition
  - Up to 64 sense channels
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode

Data Sheet for the ATmega328P series of MCUS

(posted to Moodle)

are configured to enable the pull-ups ({DDxn, PORTxn} = 0b01). See "Configuring the Pin" on page 85 for more details about this feature.

14.4.2 PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.4 PINB – The Port B Input Pins Address[1]

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

14.4.5 PORTC – The Port C Data Register

Bit	7	6	5	4	3	2	1	0	
0x08 (0x28)	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	PORTC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.6 DDRC – The Port C Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x07 (0x27)	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	DDRC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.7 PINC – The Port C Input Pins Address[1]

Bit	7	6	5	4	3	2	1	0	
0x06 (0x26)	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	PINC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

14.4.8 PORTD – The Port D Data Register

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.9 DDRD – The Port D Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.10 PIND – The Port D Input Pins Address[1]

Bit	7	6	5	4	3	2	1	0	
0x09 (0x29)	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Note: 1. Writing to the pin register provides toggle functionality for IO (see "Toggling the Pin" on page 85)

14.4.2 PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.4 PINB – The Port B Input Pins Address[(1)]

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Terminology:

Pin	Data	Data Direction	Input Pin	Bit Position
PB0	PORTB0	DDB0	PINB0	0
PB1	PORTB1	DDB1	PINB1	1
PB2	PORTB2	DDB2	PINB2	2
PB3	PORTB3	DDB3	PINB3	3
PB4	PORTB4	DDB4	PINB4	4
PB5	PORTB5	DDB5	PINB5	5
PB6	PORTB6	DDB6	PINB6	6
PB7	PORTB7	DDB7	PINB7	7

#include <avr/io.h> results in:

```
#define DDB0 0
#define DDB1 1
...
#define DDB7 7

#define PORTB0 0
#define PORTB1 1
...
#define PORTB7 7

#define PINB0 0
#define PINB1 1
...
#define PINB7 7

#define PB0 0
#define PB1 1
```

likewise for PORTC & PORTD

Significance:

```
DDRB = 1<<5;
PORTB = 1<<5;
```

same as

```
DDRB = 1<DDB5;
PORTB = 1<<PORTB5;
```

and

```
DDRB = 1<<PB5
PORTB = 1<<PB5
```

Because:

DDB5 is the same as 5
PORTB5 is the same as 5
PB5 is the same as 5

The compiler doesn't care.
The ATmega328P doesn't care.

This is a programming style/documentation/understandability issue.

ink.c × | ECE231 Lab Assignment 8.0.pdf | makefile

blink.c > main(void)

```c
/*****************************************************************
* blink.c   -- Blink an LED on Port B pin5 (PB5).
* This is the built-in LED (pin13) on the Arduino Uno dev board.
* Date          Author          Revision
* 12/14/21      D. McLaughlin   initial code creation
* 1/9/22        D. McLaughlin   tested on host MacOS Monterey, Apple M1 pro
* 2/12/22       D. McLaughlin   cleaned up formatting, added comments
* 2/21/23       D. McLaughlin   changed PB5 to DDB5 and PORTB5 in lines 17, 20, 22
* ****************************************************************/

#include <avr/io.h>             // Defines port pins
#include <util/delay.h>         // Declares _delay_ms
#define MYDELAY 1000            // This will be the delay in msec

int main(void){

    DDRB = 1<<DDB5;             // Initialize PB5 as output pin

    while(1){                   // Loop forever
        PORTB = 1<<PORTB5;      // Make PB5 high; all other PORTB pins low; LED ON
        _delay_ms(MYDELAY);     // Wait
        PORTB = ~ (1<<PORTB5); // Make PB5 low; all other PORTRB pins high;  LED off
        _delay_ms(MYDELAY);     // Wait
    }

    return 0;                   // Code never gets here.
}

/******* End of File *********/
```

C blink.c > MYDELAY

```c
/*****************************************************************
* blink.c   -- Blink an LED on Port B pin5 (PB5).
* This is the built-in LED (pin13) on the Arduino Uno.
* Date          Author          Revision
* 12/14/21      D. McLaughlin   initial code creation
* 1/9/22        D. McLaughlin   tested on host MacOS Monterey, Apple M1 pro
* 2/12/22       D. McLaughlin   cleaned up formatting, added comments
* ****************************************************************/

#include <avr/io.h>             // Defines PB5
#include <util/delay.h>         // Declares _delay_ms
#define MYDELAY 100            // This will be the delay in msec

int main(void){

    DDRB = 1<<PB5;              // Initialize PB5 as output pin

    while(1){                   // Loop forever
        PORTB = 1<<PB5;         // Make PB5 high; LED ON
        _delay_ms(MYDELAY);     // Wait
        PORTB = ~ (1<<PB5);     // Make PB5 low; LED off
        _delay_ms(MYDELAY);     // Wait
    }

    return 0;                   // Code never gets here.
}

/******* End of File *********/
```

Setting a bit in a Byte
("Setting a bit" means make it go high)

- We can use | operator to set a bit of a byte to 1

```
 xxxx xxxx                xxxx xxxx
|0001 0000      OR       |1 << 4
 ------------             ------------
 xxx1 xxxx                xxx1 xxxx
```

```
PORTB =  PORTB | 0b00010000; //set bit 4 (5th bit) of PORTB
```

```
PORTB | =  0b00010000; //set bit 4 (5th bit) of PORTB
```

```
PORTB =  PORTB | (1<<4); //set bit 4 (5th bit) of PORTB
```

```
PORTB |= (1<<4); //set bit 4 (5th bit) of PORTB
```

```
PORTB |= (1<<PORTB4); //set bit 4 (5th bit) of PORTB (PORTB4 is #defined 4)
```

Sets PORTB4=1. Leaves all other bits of PORTB untouched.

```
PORTB |= (1<<4)|(1<<3)|(1<<2) ; //set bits 4, 3, and 2 of PORTB
```

```
PORTB |= (1<<PORTB4)|(1<<PORTB3)|(1<<PORTB2) ;
```

Sets PORTB2, PORTB3, PORTB4=1. Other bits of PORTB untouched.

Clearing a bit in a Byte
("Clearing a bit" means make it go low)

- We can use & operator to clear a bit of a byte to 0

```
  xxxx  xxxx
& 1110 1111
------------
xxx0  xxxx
```

```
xxxx xxxx
& ~(1 << 4)
------------
xxx0 xxxx
```

```
xxxx xxxx
& ~(0001 0000)
------------
xxx0 xxxx
```

```
xxxx xxxx
& (1110 1111)
------------
xxx0 xxxx
```

```
PORTB &= 0b11101111; //clear bit 4 (5th bit) of PORTB
```

```
PORTB &= ~(1<<4); //clear bit 4 (5th bit) of PORTB
```

```
PORTB &= ~(1<<PORTB4); //clear bit 4 (5th bit) of PORTB
```

Clears PORTB4 (Sets PORTB4=0). Leaves all other bits of PORTB untouched.

Toggling a bit in a Byte
("Toggling a bit" means 1→0 or 0 →1)

- We can use ^ operator to toggle a bit of a byte

```
  xxxx  xxxx
^ 0001 0000
  ------------
  xxx̄x  xxxx
```

```
  xxxx xxxx
^  (1 << 4)
  ------------
  xxx̄x xxxx
```

```
  xxxx xxxx
^(0001 0000)
  ------------
  xxx̄x xxxx
```

```
PORTB ^= 0b00010000; // Toggle bit 4 (5th bit) of PORTB
```

```
PORTB ^= (1<<4); // Toggle bit 4 (5th bit) of PORTB
```

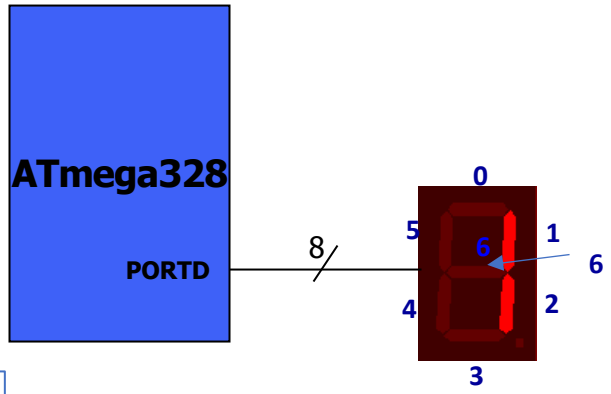
```
PORTB ^= (1<<PORTB4); // Toggle bit 4 (5th bit) of PORTB
```

Clears PORTB4 (Sets PORTB4=0). Leaves all other bits of PORTB untouched.

Example 3

A 7-segment LED is connected to PORTD. Display “1” on the 7-segment.

DDRD:	1	1	1	1	1	1	1	1
PORTD:	0	0	0	0	0	1	1	0




```
DDRD = 0b11111111; // binary
PORTD = 0b00000110;
```

```
DDRD = 0xFF;              // Hex
PORTD = (1<<1)|(1<<2);    // Easier
```

```
DDRD = 0xFF;      // Hex
PORTD = (1<<PORTD1)|(1<<PORTD2); // Preferred
```

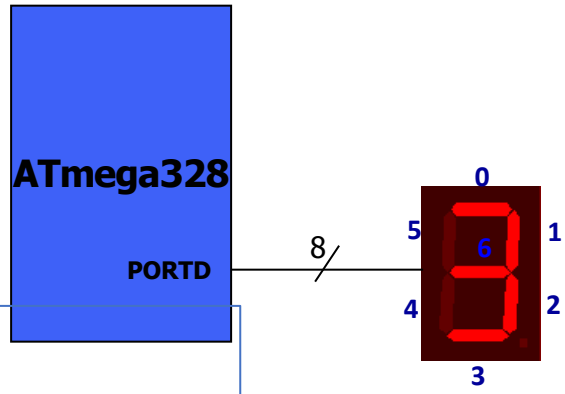
1 = 0 0 0 0 0 0 0 1
1<<1 = 0 0 0 0 0 0 1 0
1<<2 = 0 0 0 0 0 1 0 0
(1<<1)|(1<<2) = 0 0 0 0 0 1 1 0

DDRXn =1 (pin set as output)		
PORTxn	Pxn (output voltage)	Pxn (output logic)
0 (clear)	0V	0 (low)
1 (set)	5V	1 (high)

Example 4

- A 7-segment LED is connected to PORTD. Display "3" on the 7-segment.

DDRD:	1	1	1	1	1	1	1	1
PORTD:	0	0	0	0	1	1	1	1




```
DDRD = 0b11111111;  // binary
PORTD=(1<<0)|(1<<1)|(1<<2)|(1<<3);
or
DDRD = 0xFF;
PORTD= 1<<PORTD0|1<<PORTD1|1<<PORTD2|1<<PORTD3;
```

remember: #include <avr/io.h> results in these definitions:

```
#define PORTD0 0
#define PORTD1 1
...
#define PORTD7 7
```

The compiler and the ATmega328p don't care whether you use PORTD0 or 0. This is a matter of coding style.

DDRXn =1 (pin set as output)		
PORTxn	Pxn (output voltage)	Pxn (output logic)
0 (clear)	0V	0 (low)
1 (set)	5V	1 (high)