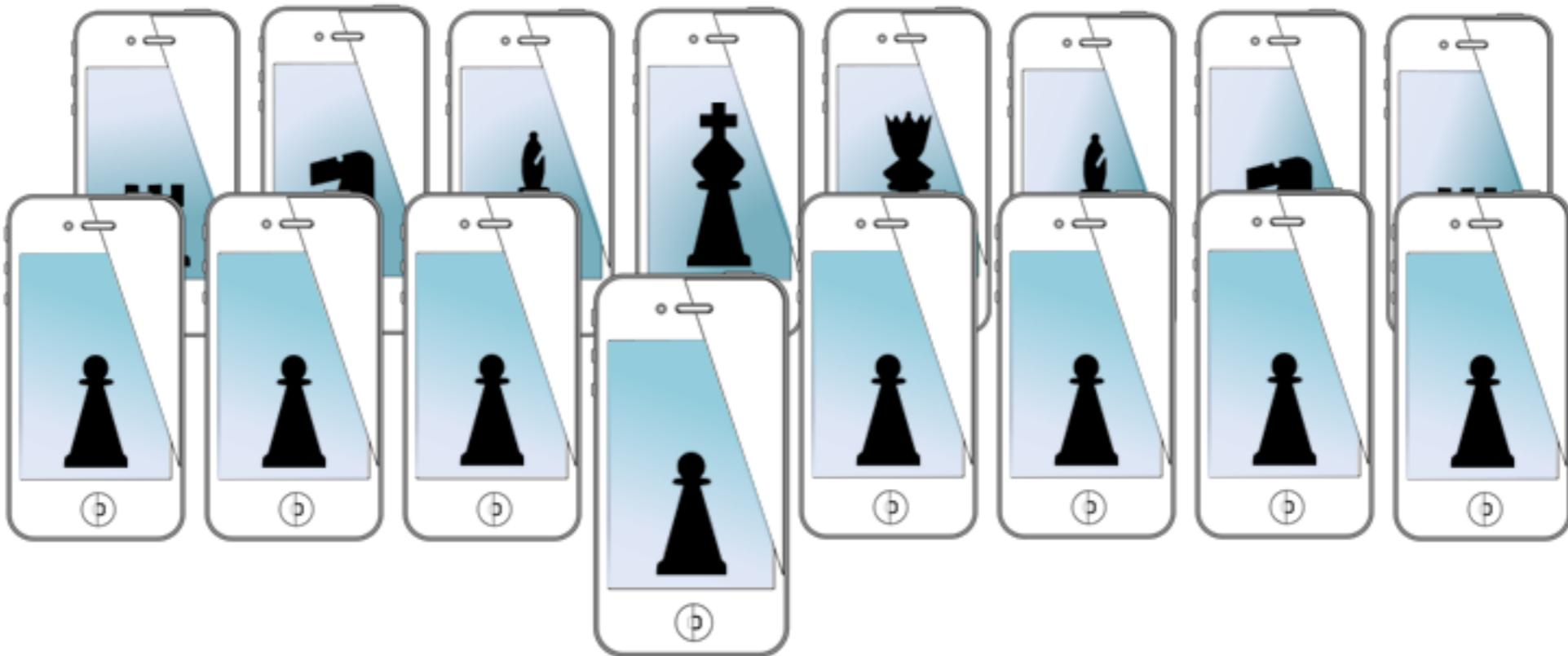


MOBILE SENSING LEARNING & CONTROL



CSE5323 & 7323

Mobile Sensing, Learning, and Control

lecture fourteen: microcontroller basics

Eric C. Larson, Lyle School of Engineering,
Computer Science and Engineering, Southern Methodist University

course logistics

- A3 grades have been up for most of the break
- A4 is due this Friday!

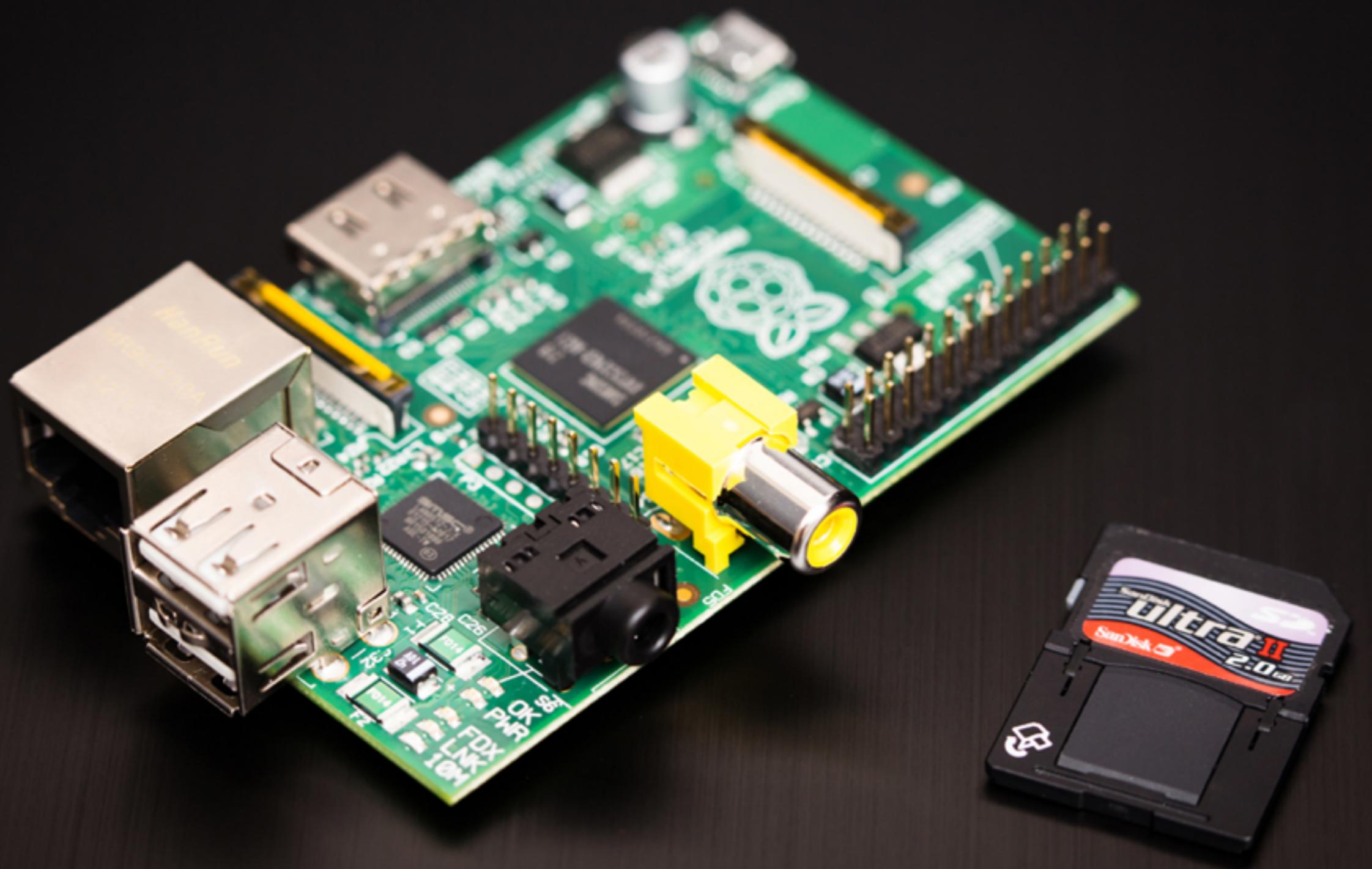
agenda

- microcontrollers
 - architecture
 - peripherals
 - common usage
- Arduino sketches
 - the Atmega328

embedded systems

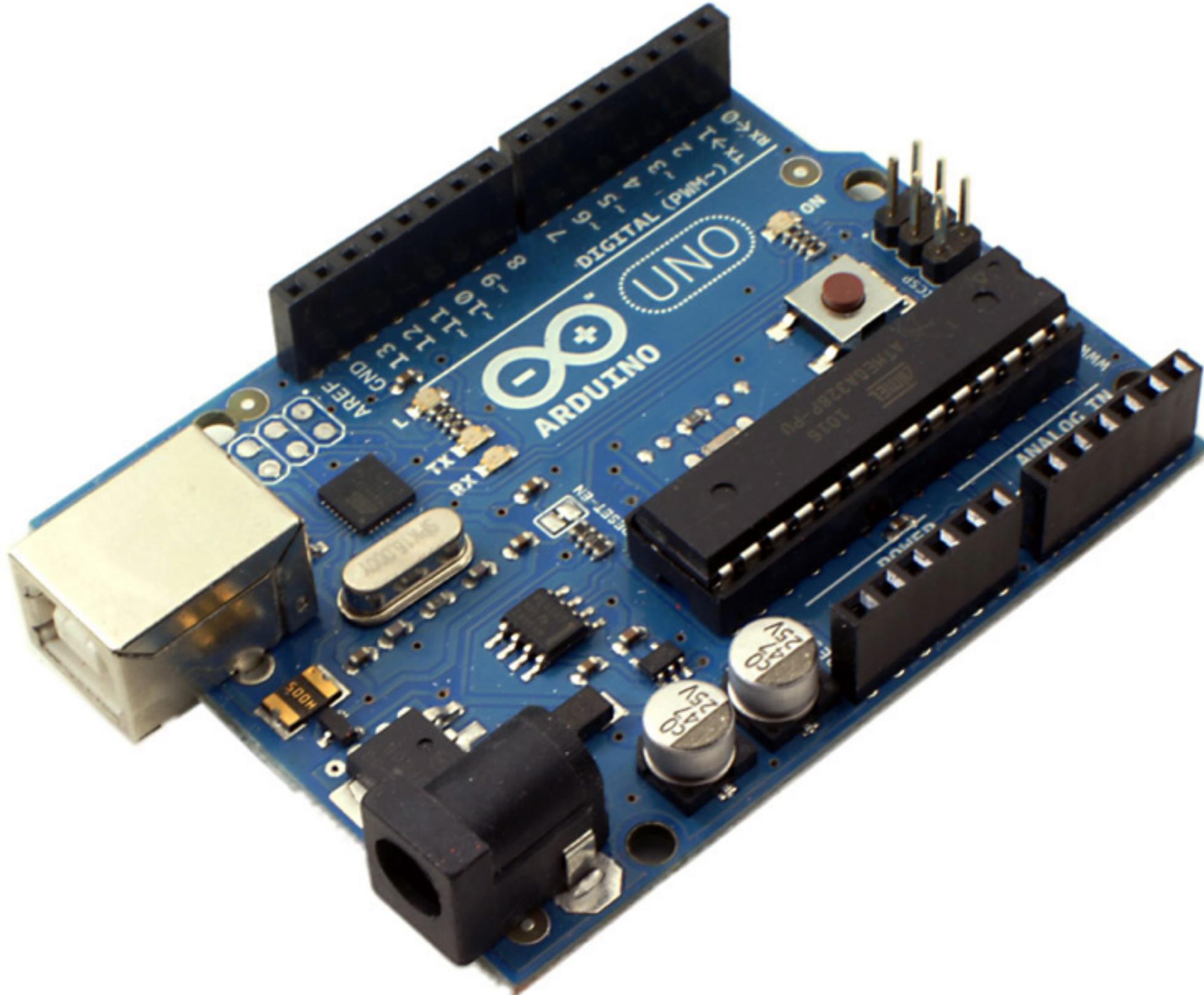
- cameras
- microcontrollers
- FPGAs
- internet of things
- many, many, things

embedded system: a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints





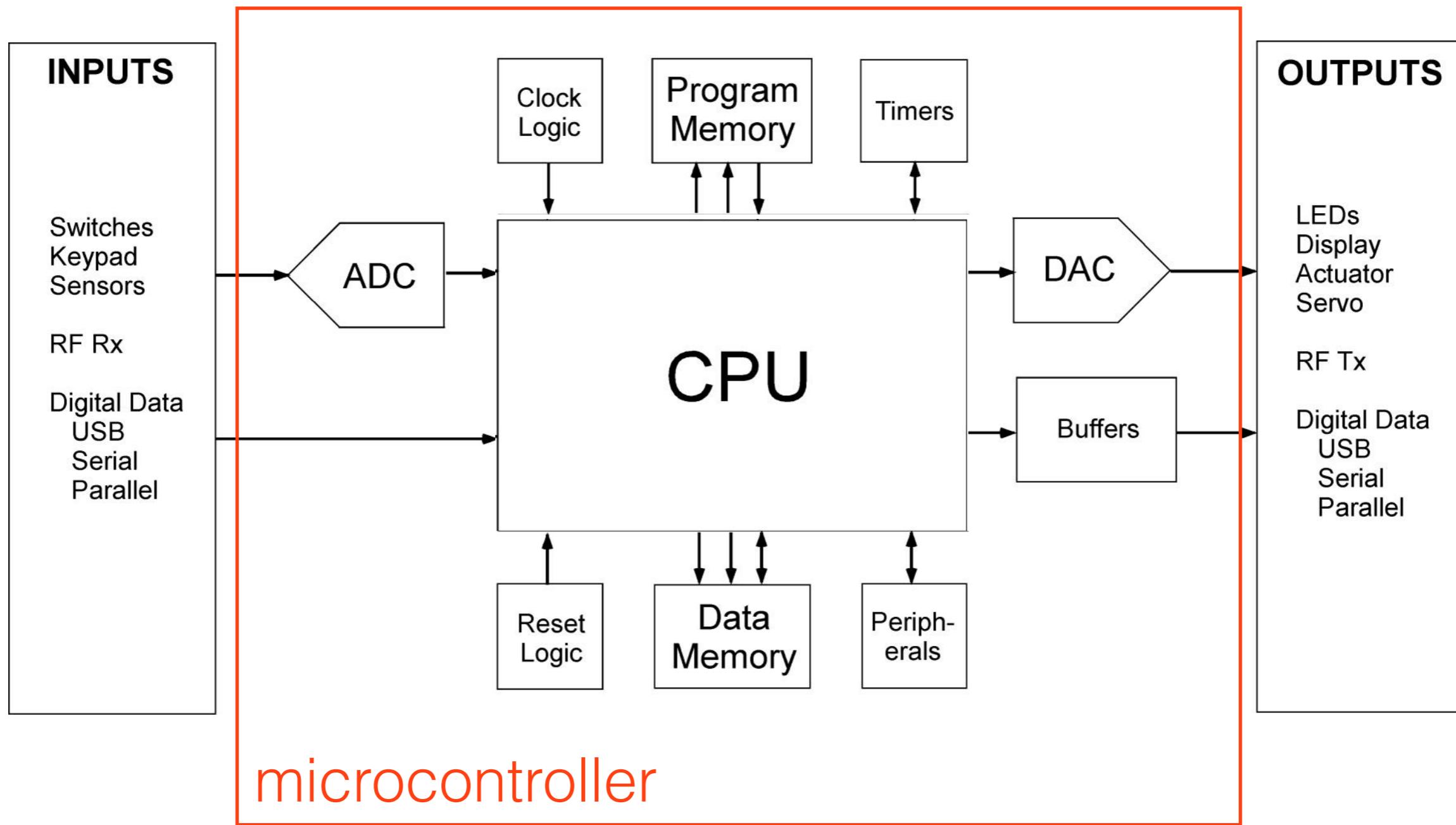




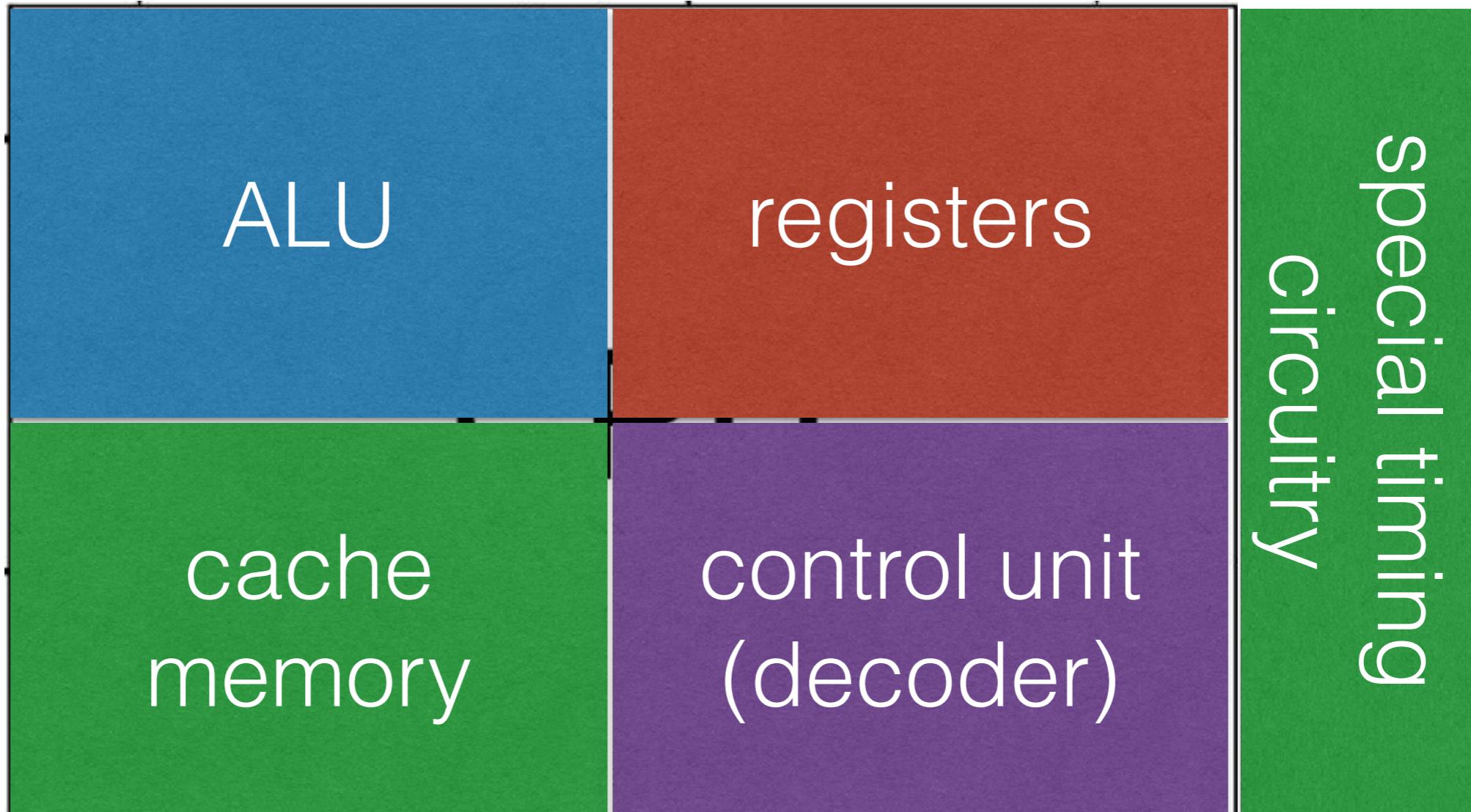


iMore

embedded systems

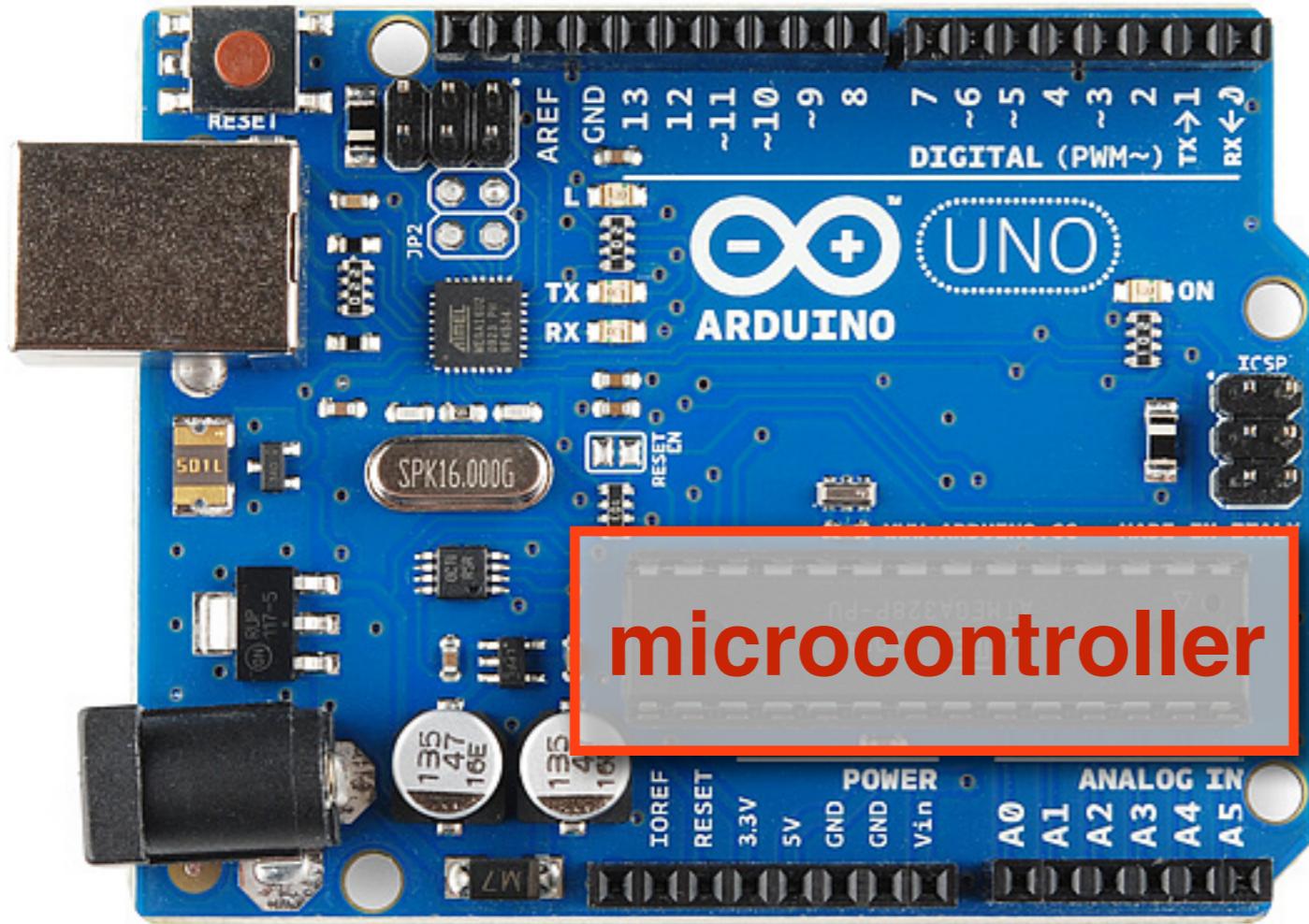


microcontroller



MCU

microcontroller



excellent tutorials:

<http://www.adafruit.com/category/17>

- Atmega328
 - 16MHz
 - 16 bit ALU
 - 2KB SRAM
 - 32KB flash
 - 1KB EEPROM
 - 14 GPIO (6 PWM)
 - 6 analog inputs
 - hardware serial, TTL
 - hardware SPI

arduino sketch

- write in c
- use arduino functions

```
Blink
/*
Blink
Turns on an LED on for one second, then off for one second, repeatedly.

This example code is in the public domain.
*/

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
    // initialize the digital pin as an output.
    pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH);      // turn the LED on (HIGH is the voltage level)
    delay(1000);                // wait for a second
    digitalWrite(led, LOW);     // turn the LED off by making the voltage LOW
    delay(1000);                // wait for a second
}
```

called on startup or
reset

run forever

arduino serial comm

- really, really simple
- the Rx and Tx are on pins 0 and 1

```
Serial.begin(9600);
```

baud rate
9600 pulses/sec

```
float resistance;
```

send a float!

```
Serial.print(resistance);
```

```
Serial.print(" kohms\n");
```

send a string!

atmega CPU

small ALU (8-16 bit typical)

RISC (simple instruction set)

harvard architecture (separate program and data memory)

pipelined load-store architecture (read/write from registers)

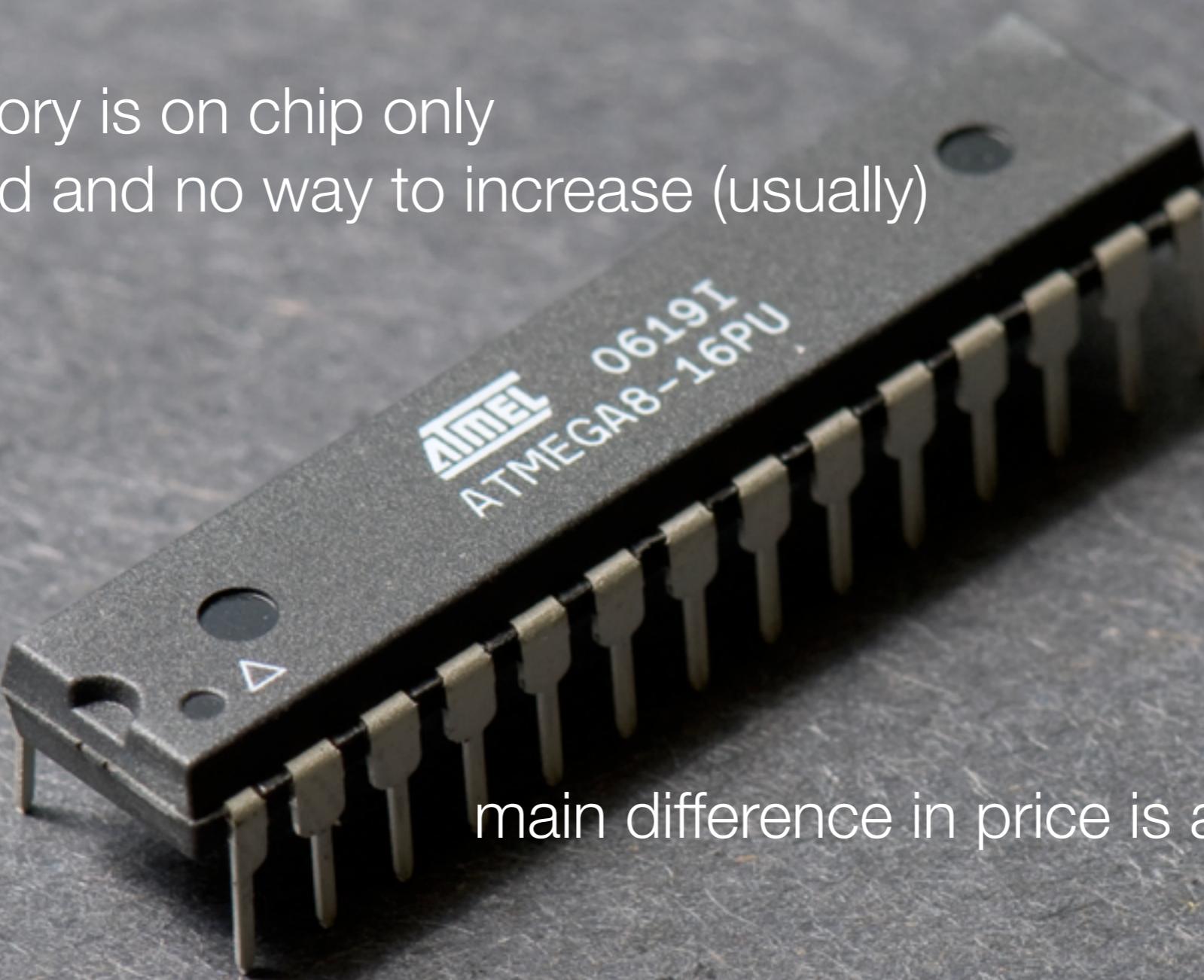
low clock speeds (8-32 MHz)



optimized for low-level compilers like C
typically no OS is used (sometimes RTOS)

atmega memory

memory is on chip only
limited and no way to increase (usually)



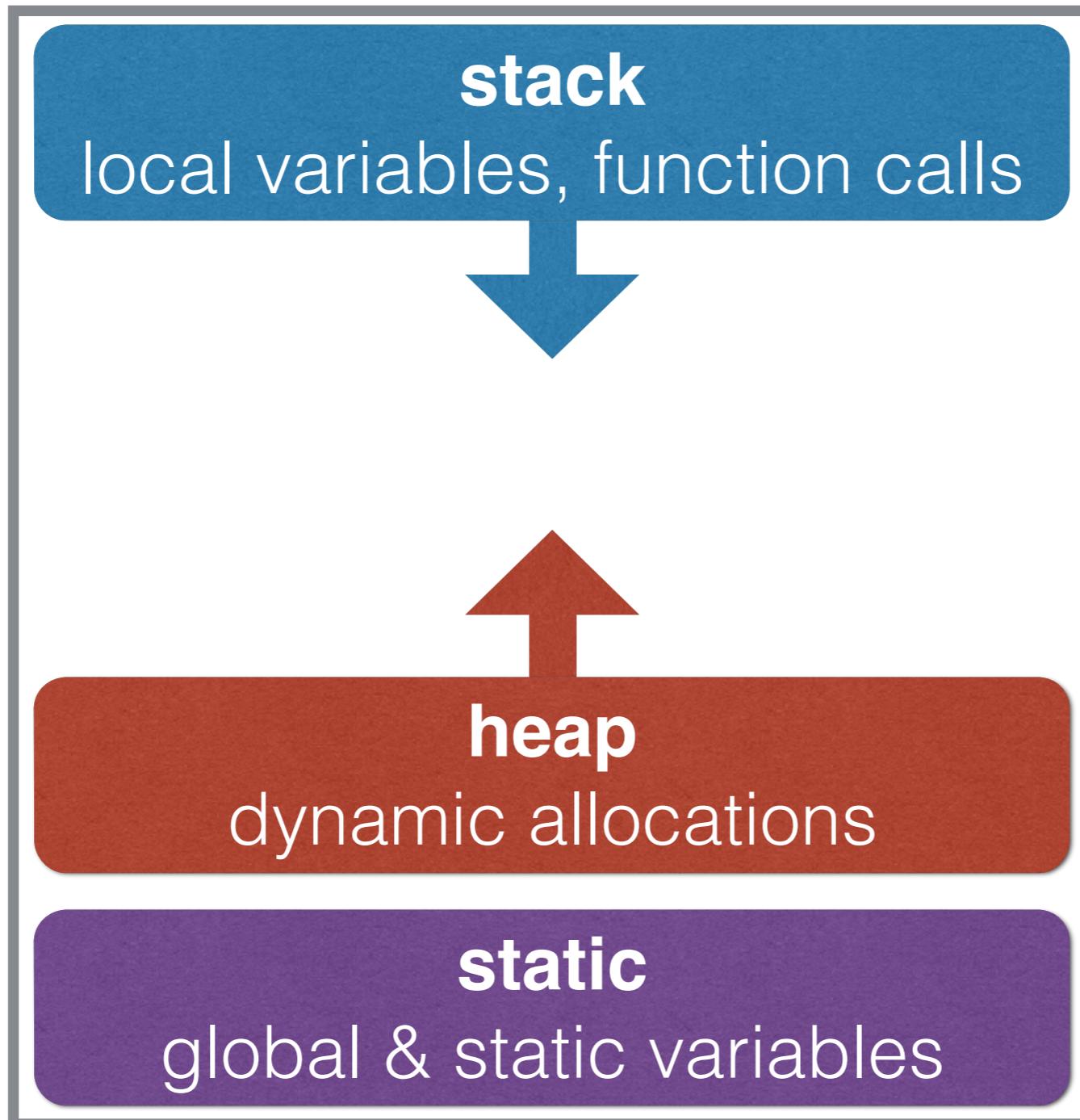
main difference in price is available memory

atmega memory

- three memory spaces
- volatile
 - SRAM
- non-volatile
 - EEPROM
 - flash (program memory)

atmega SRAM

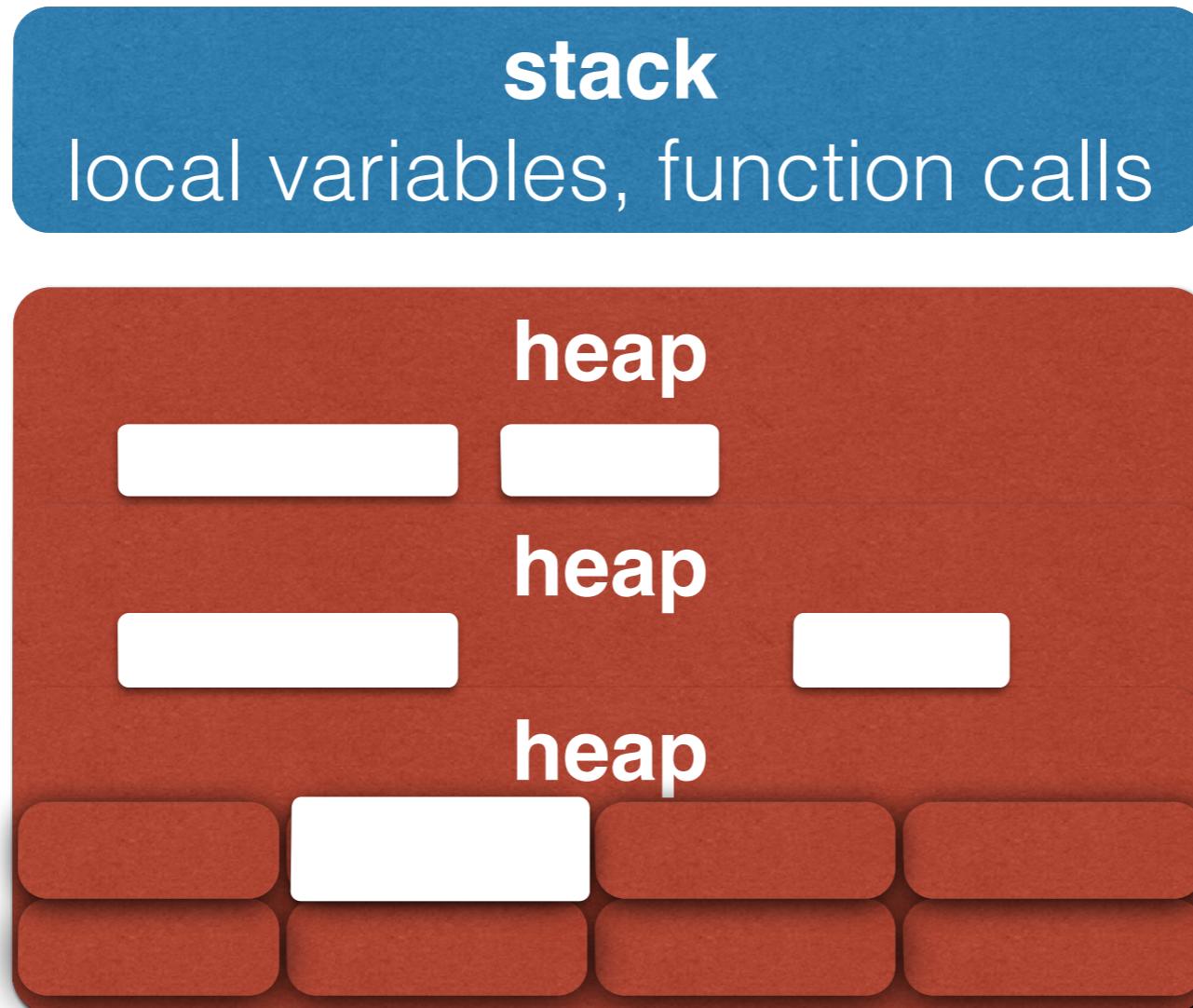
- SRAM divided into three areas





easy to do

- fragmented memory in the heap
 - it not an OS, garbage collection is VERY primitive





atmega SRAM

- guidelines to avoid fragmentation

```
int tmpSensorData[512];
int *moreSensorData;

// the setup routine runs once when you press reset:
void setup() {
    moreSensorData = (int*)malloc(512);
}

// the loop routine runs over and over again forever:
void loop() {
    int f = analogRead(1);

}
```

ased

atmega non-volatile

- EEPROM
 - electronically erasable programmable memory
 - stays in memory when power turns off
 - good for saving a “state”
 - infinite number of reads
 - ~100,000 writes / erases
 - one write takes ~3.3ms, per byte

atmega non-volatile

eprom_write §

```
* turned off and may be retrieved later by another sketch.  
*/  
  
#include <EEPROM.h>  
int addr = 0;  
void setup()  
{  
}  
void loop()  
{  
    // value from 0 to 255.  
    int val = analogRead(0) / 4;  
  
    // write the value to the appropriate byte of the EEPROM.  
    EEPROM.write(addr, val);  
  
    // advance to the next address. there are 512 bytes in  
    // the EEPROM, so go back to 0 when we hit 512.  
    addr = addr + 1;  
    if (addr == 512)  
        addr = 0;  
  
    delay(100);  
  
    byte value = EEPROM.read(addr);  
}
```

write a byte to
“address”, 0-511

read a byte from
“address”

atmega non-volatile

- program memory (flash)
 - where your code instructions are saved
 - flash memory
 - great for variables that do not change
 - like strings!
 - not stored in the SRAM until copied over

```

#include <avr/pgmspace.h>

// save some unsigned ints
PROGMEM prog_uint16_t charSet[] = { 8400, 77, 42};

// save some chars
PROGMEM prog_uchar signMessage[] = {"PROGRAM MEMORY STRING"};

unsigned int displayInt;
int k; // counter variable
char myChar;

// read back a 2-byte int
displayInt = pgm_read_word_near(charSet + k)

// read back a char
myChar = pgm_read_byte_near(signMessage + k);

for (int i = 0; i < 6; i++)
{
    strcpy_P(buffer, (char*)pgm_read_word(&(string_table[i]))); // copy
    Serial.println( buffer );
    delay( 500 );
}

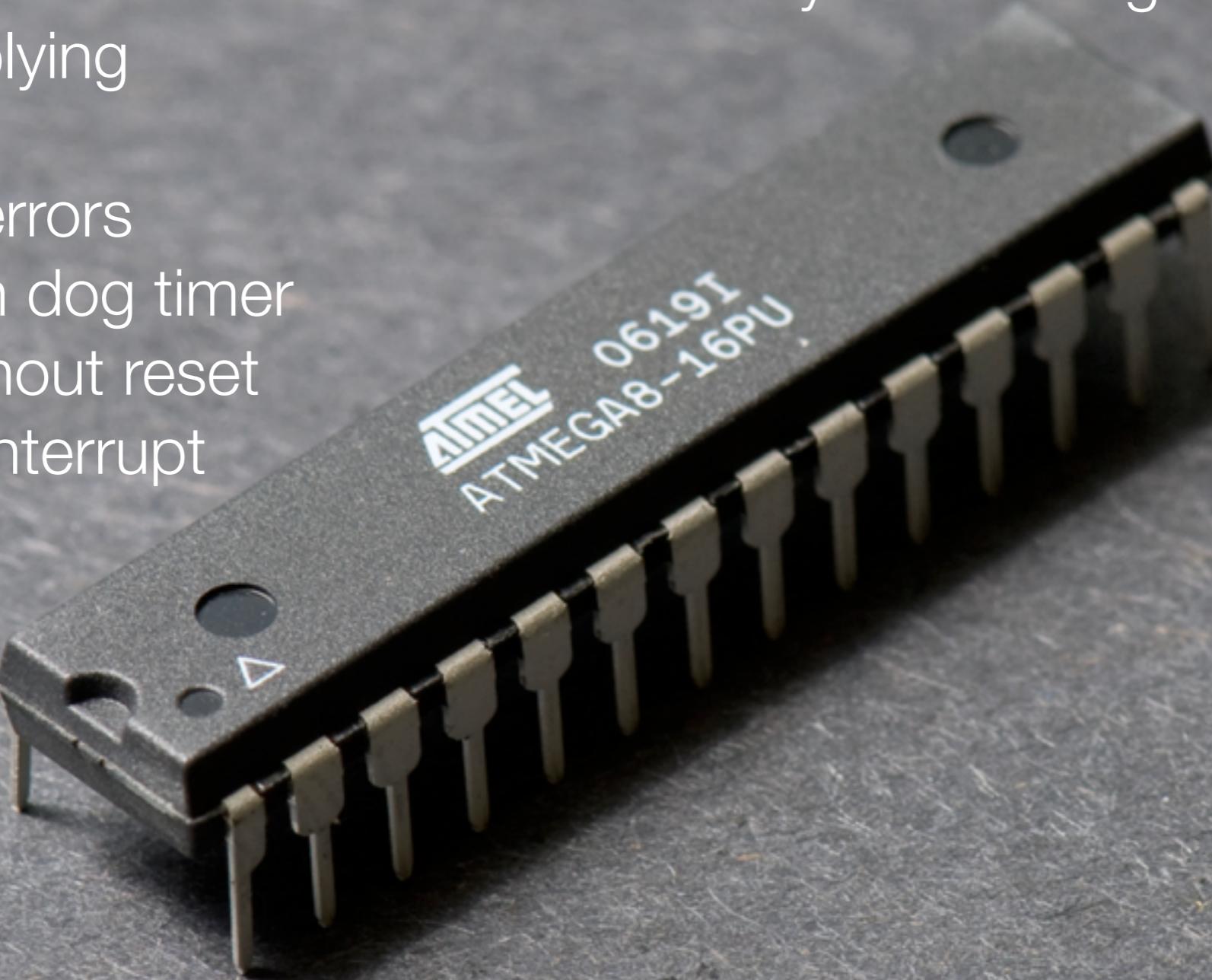
```

<http://arduino.cc/en/Reference/PROGMEM#.UyKZIdztJg0>

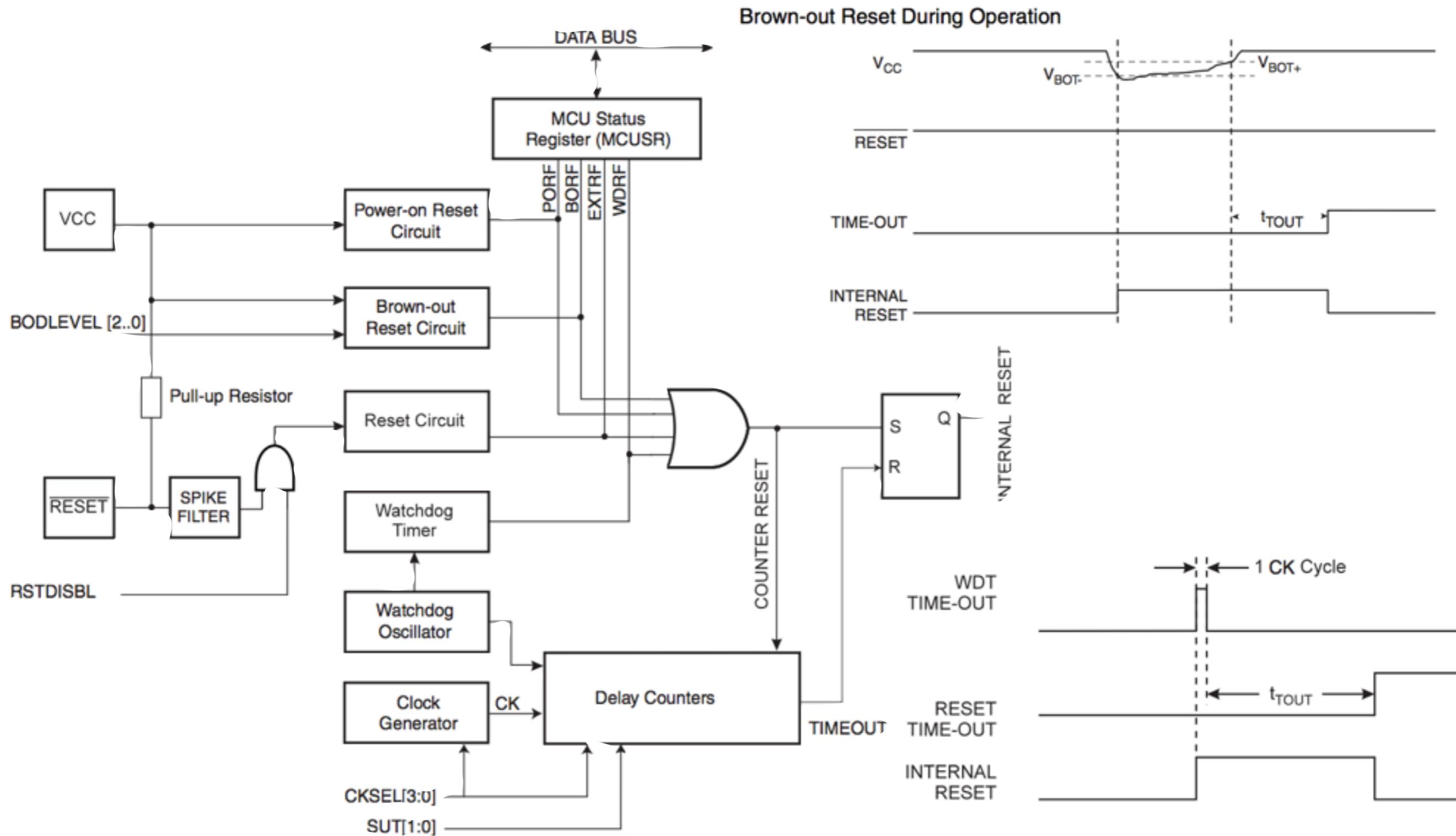
atmega osc & reset

oscillator is the clock to everything
external or internal with various ways of dividing down or
multiplying

reset for errors
watch dog timer
brownout reset
reset via interrupt



atmega reset



watch dog timer

- enable or disable

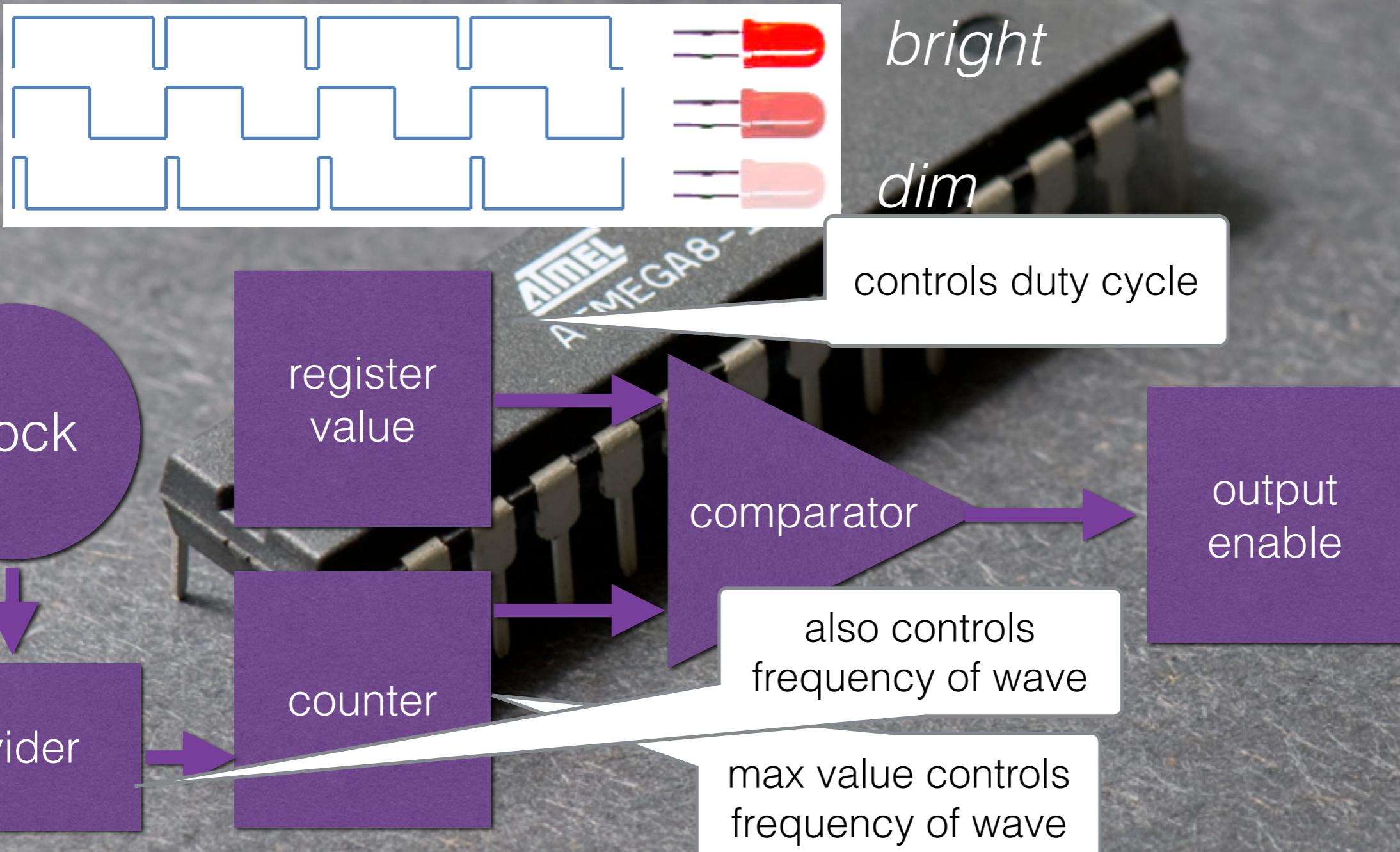
```
if(USE_WDT==1)
    wdt_enable(WDTO_8S); // watch dog set at 8 seconds, don't go much lower
else
    wdt_disable();
```

- must set back to zero periodically or reset occurs

```
void SW_ISR() {
    // if you do not interrupt the WDT, then the board will reset
    wdt_reset();
}
```

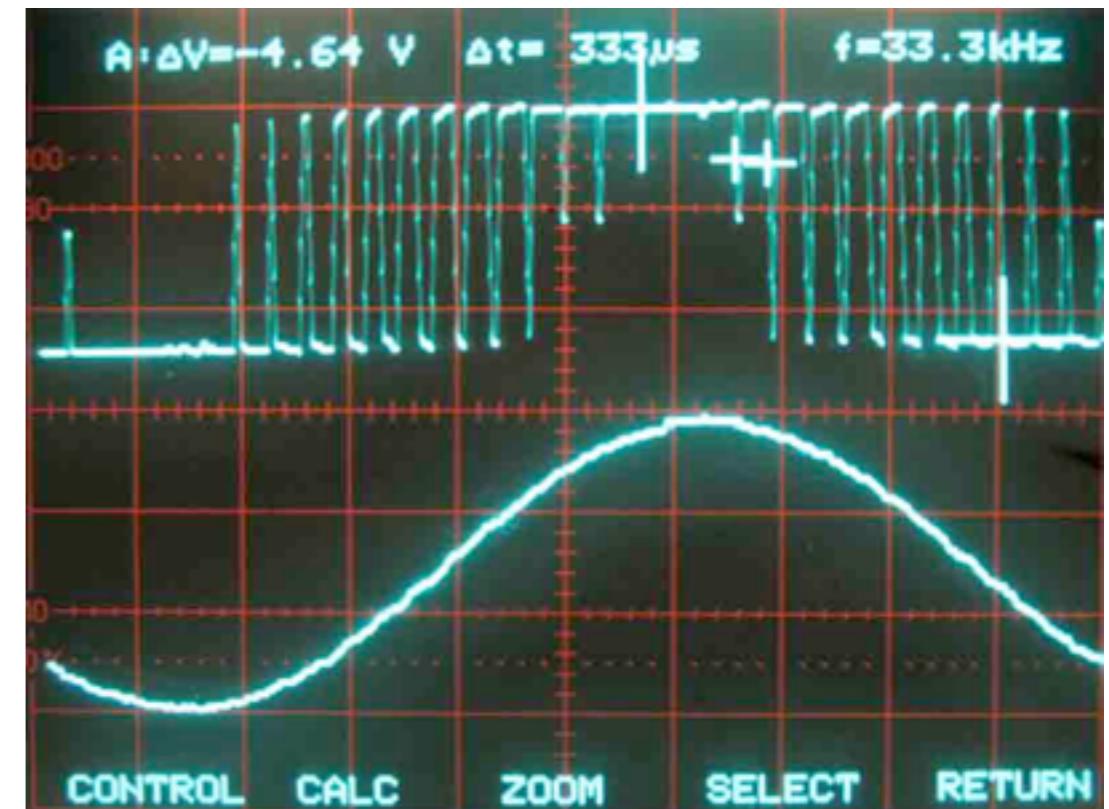
timers

WDT, real-time clock,
pulse width modulation: square wave output



PWM

- servo control and motor control
 - width of pulse matters
- digital to analog conversion
 - smooth a square wave
- LED brightness control
 - “duty cycle” matters



```
ledBrightness += delta; // change brightness setting  
analogWrite(LED_PWM, ledBrightness); // set new brightness
```

pin to write to
(must be PWM)

value to write (0-255)

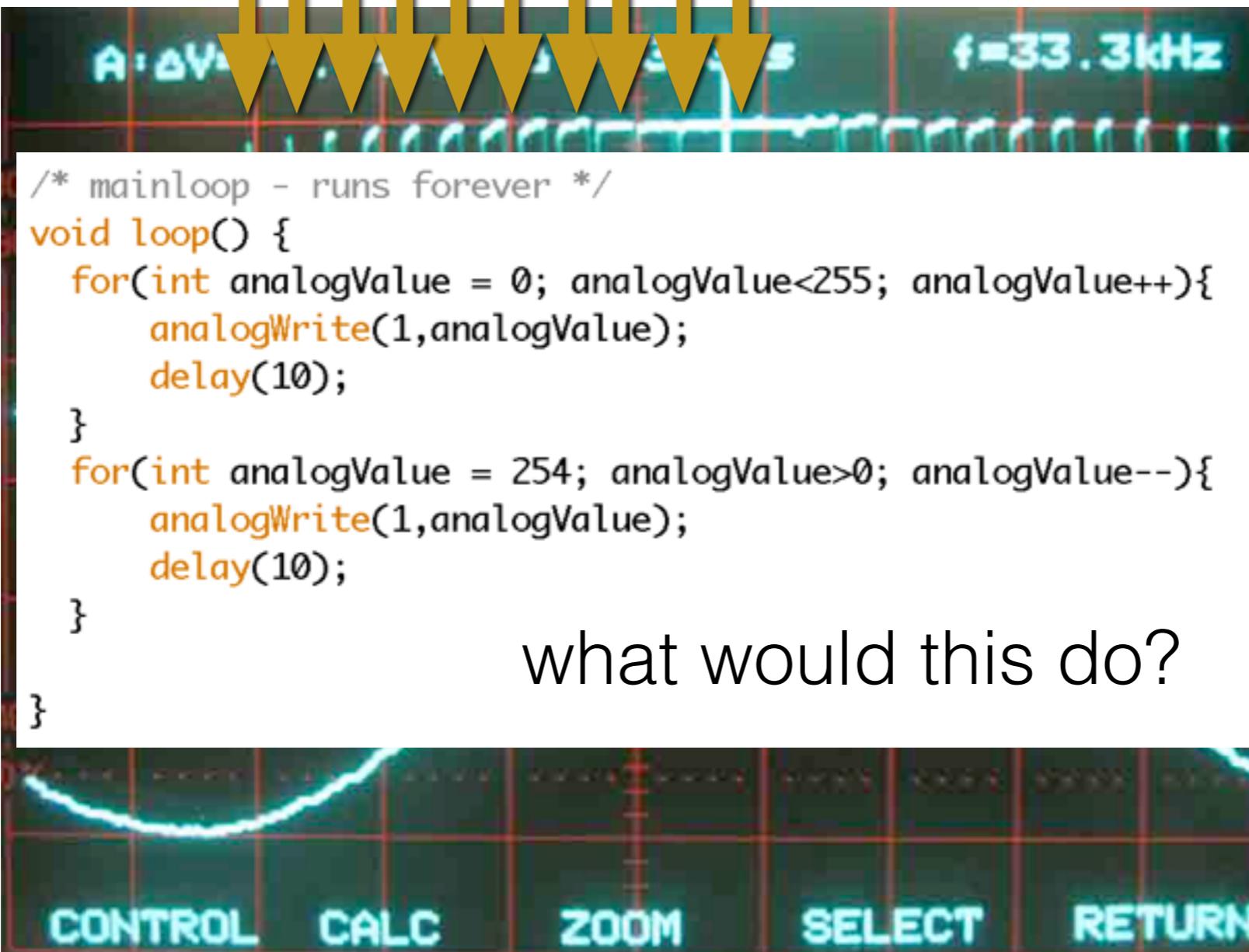
PWM

change in
duty cycle

PWM

after
low pass
filter

15% 35% 75% 95% 100%
10% 25% 50% 85% 99%



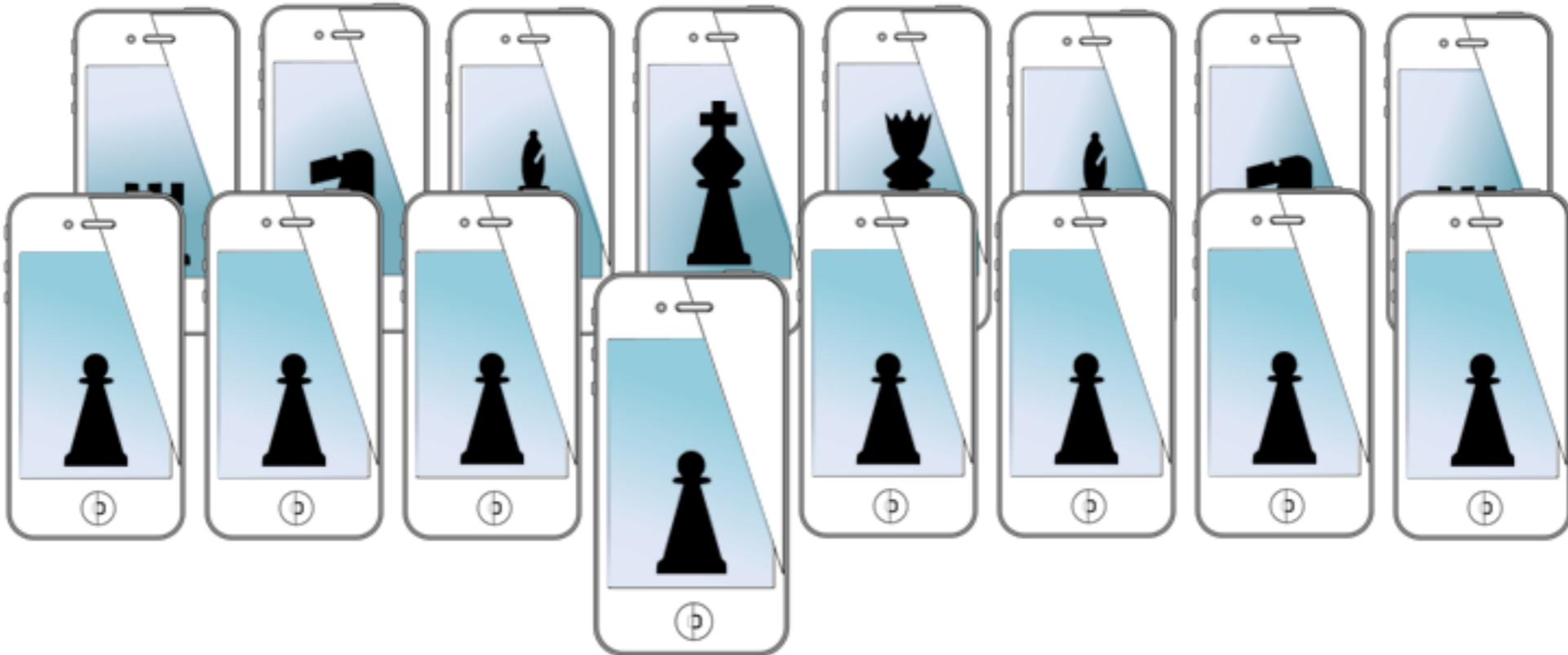
normal PWM:
constant
duty cycle

this example:
time varying
duty cycle

for next time...

- PWM, ADC, and GPIO
- sensors and more control
- arduino shields
- blue tooth communication

MOBILE SENSING LEARNING & CONTROL



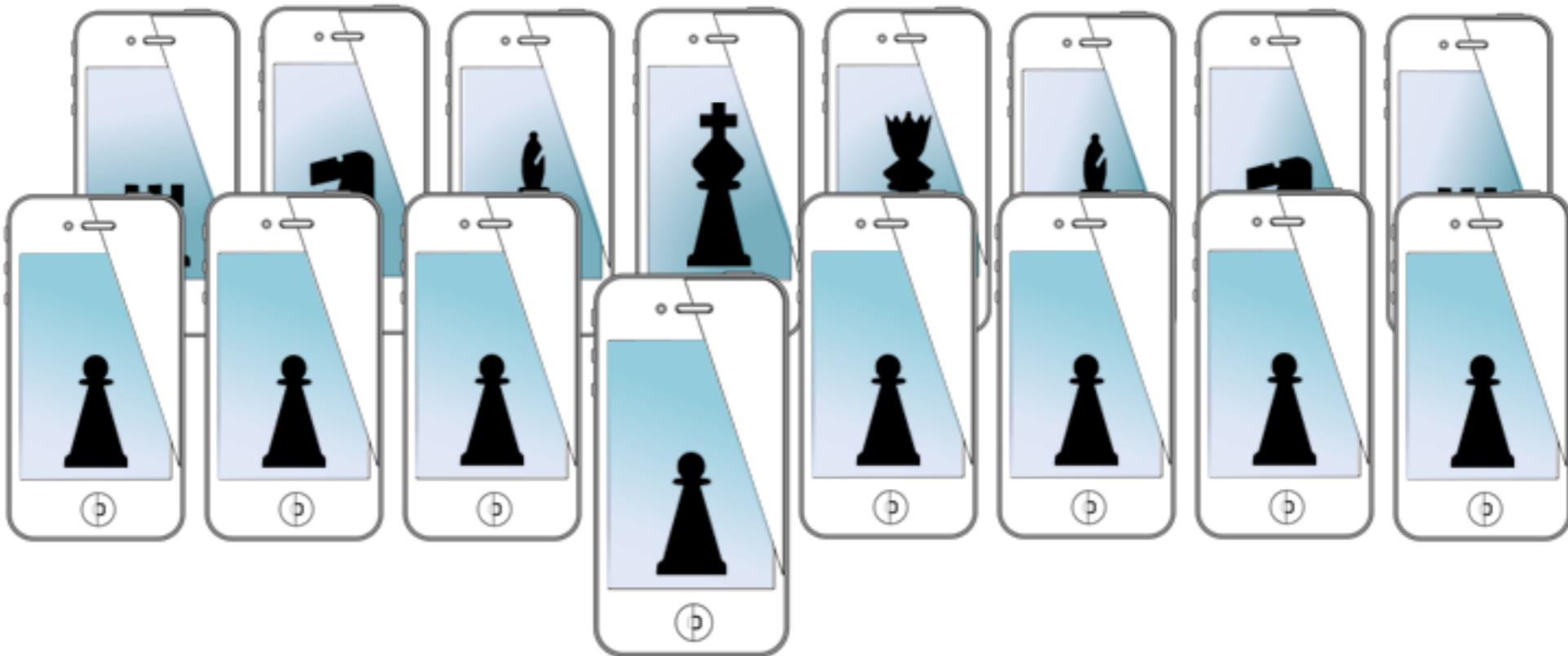
CSE5323 & 7323

Mobile Sensing, Learning, and Control

lecture fourteen: microcontroller basics

Eric C. Larson, Lyle School of Engineering,
Computer Science and Engineering, Southern Methodist University

MOBILE SENSING LEARNING & CONTROL



CSE5323 & 7323

Mobile Sensing, Learning, and Control

lecture fifteen: microcontroller sensing and control

Eric C. Larson, Lyle School of Engineering,
Computer Science and Engineering, Southern Methodist University

course logistics

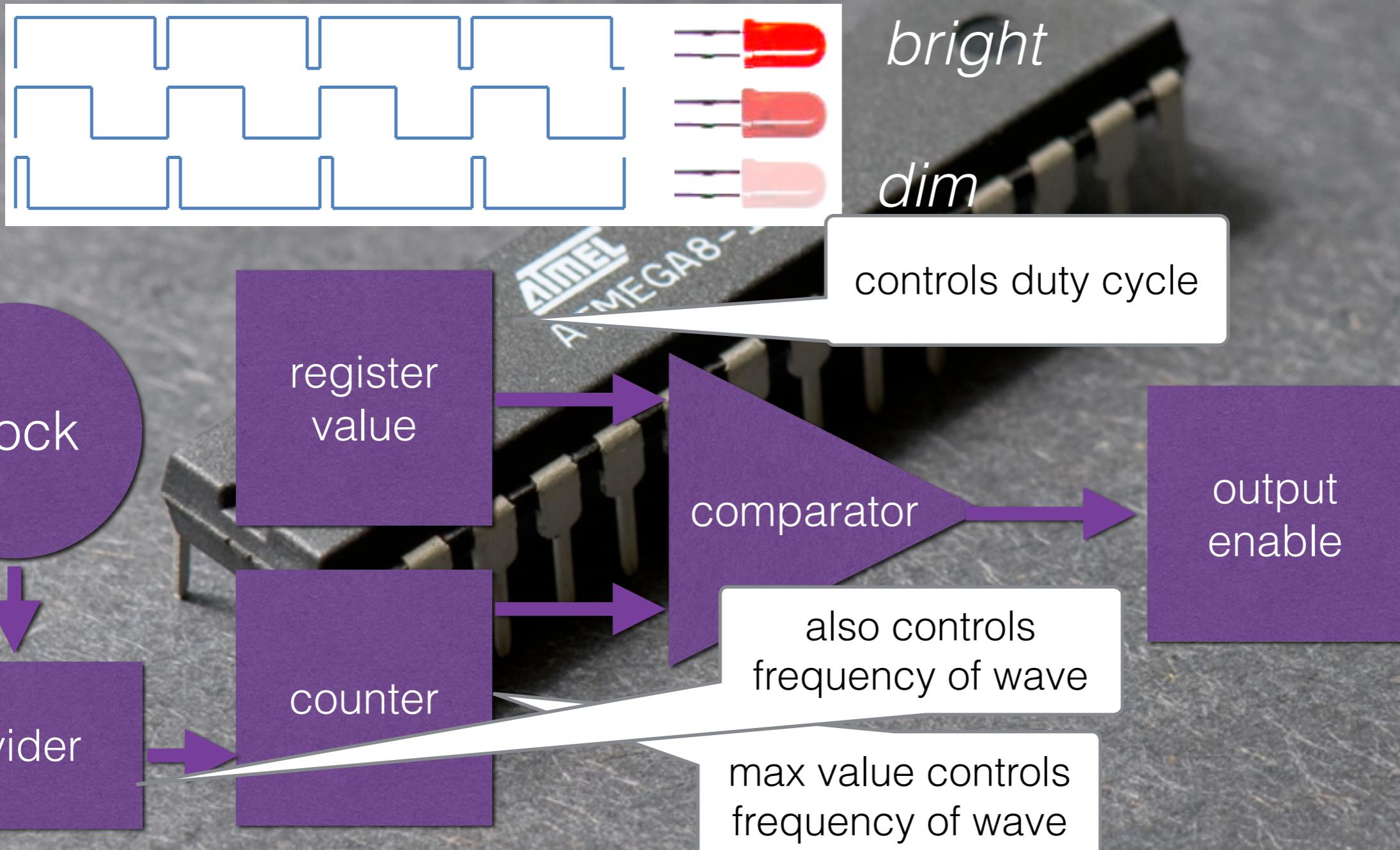
- A4 is due... lets talk about it
- A5 is due two weeks from Friday!
 - and will be due, no extensions!

agenda

- PWM, ADC, GPIO, and interrupts
- hobbyist electronics
 - common sensors / controls
 - potentiometers
 - magnetic switches
 - servos and motors
 - beginnings of bluetooth LE

PWM review

pulse width modulation: square wave output



PWM with registers

- registers control most peripherals on a uC

```
pinMode(3, OUTPUT);  
pinMode(11, OUTPUT);
```

in this register

```
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
```

```
TCCR2B = _BV(CS22);
```

```
OCR2A = 200;
```

```
OCR2B = 120;
```

set bit with
this name

TCCR2A – Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0	
(0xB0)	COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20	TCCR2A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR2B – Timer/Counter Control Register B

Bit	7	6	5	4	3	2	1	0	
(0xB1)	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20	TCCR2B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PWM with registers

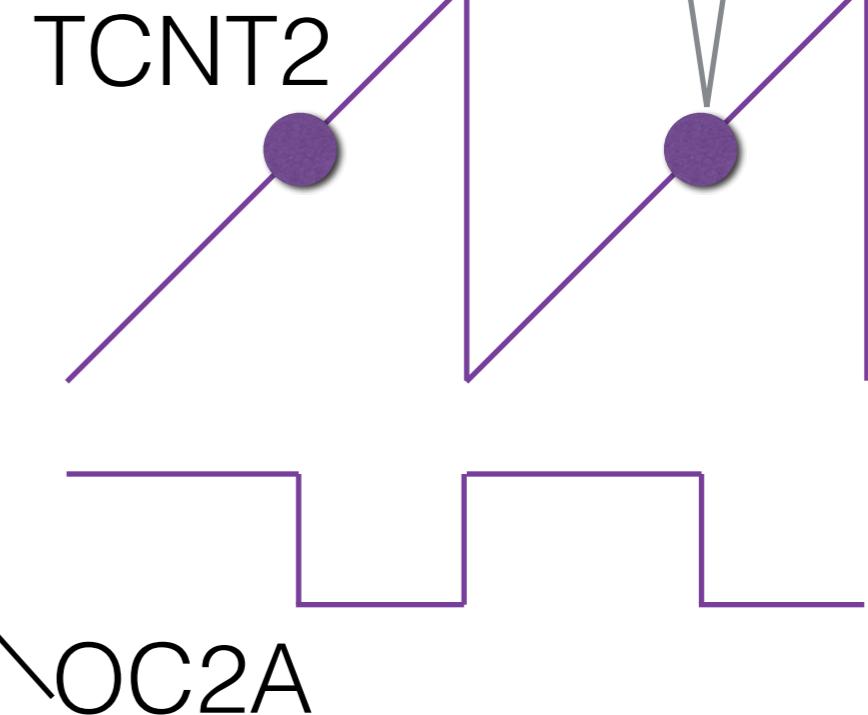
- registers control most peripherals on a uC

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
TCCR2B = _BV(CS22);
OCR2A = 200;
OCR2B = 120;
```

compare value

Table 18-4. Compare Output Mode, Phase Correct PWM Mode⁽¹⁾

COM2A1	COM2A0	Description
0	1	WGM22 = 0: Normal Port Operation, OC2A Disconnected. WGM22 = 1: Toggle OC2A on Compare Match.
1	0	Clear OC2A on Compare Match when up-counting. Set OC2A on Compare Match when down-counting.
1	1	Set OC2A on Compare Match when up-counting. Clear OC2A on Compare Match when down-counting.



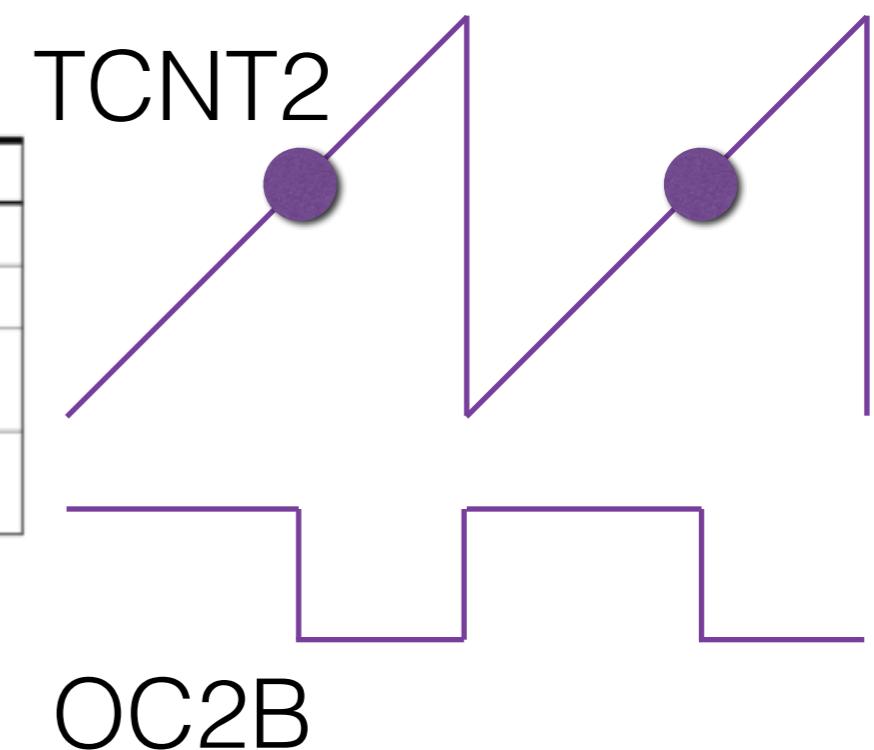
PWM with registers

- registers control most peripherals on a uC

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
TCCR2B = _BV(CS22);
OCR2A = 200;
OCR2B = 120;
```

Table 18-6. Compare Output Mode, Fast PWM Mode⁽¹⁾

COM2B1	COM2B0	Description
0	0	Normal port operation, OC2B disconnected.
0	1	Reserved
1	0	Clear OC2B on Compare Match, set OC2B at BOTTOM, (non-inverting mode).
1	1	Set OC2B on Compare Match, clear OC2B at BOTTOM, (inverting mode).



PWM with registers

- registers control most peripherals on a uC

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
TCCR2B = _BV(CS22);
OCR2A = 200;
OCR2B = 120;
```

Table 18-8. Waveform Generation Mode Bit Description

Mode	WGM22	WGM21	WGM20	Timer/Counter Mode of Operation	TOP	Update of OCR _x at	TOV Flag Set on ⁽¹⁾⁽²⁾
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	-	-	-
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	-	-	-
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

PWM with registers

- registers control most peripherals on a uC

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
TCCR2B = _BV(CS22);
OCR2A = 200;
OCR2B = 120;
```

Table 18-9. Clock Select Bit Description

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$\text{clk}_{\text{T2S}}/(\text{No prescaling})$
0	1	0	$\text{clk}_{\text{T2S}}/8$ (From prescaler)
0	1	1	$\text{clk}_{\text{T2S}}/32$ (From prescaler)
1	0	0	$\text{clk}_{\text{T2S}}/64$ (From prescaler)

two outputs at $16\text{MHz} / 64 / 256 = 976.5625\text{Hz}$

PWM with registers

- registers control most peripherals on a uC

```
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM21) | _BV(WGM20);
TCCR2B = _BV(CS22);
OCR2A = 200;
OCR2B = 120;
```

OCR2A – Output Compare Register A

Bit	7	6	5	4	3	2	1	0	
(0xB3)	OCR2A[7:0]								OCR2A
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	

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value (TCNT2). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC2A pin.

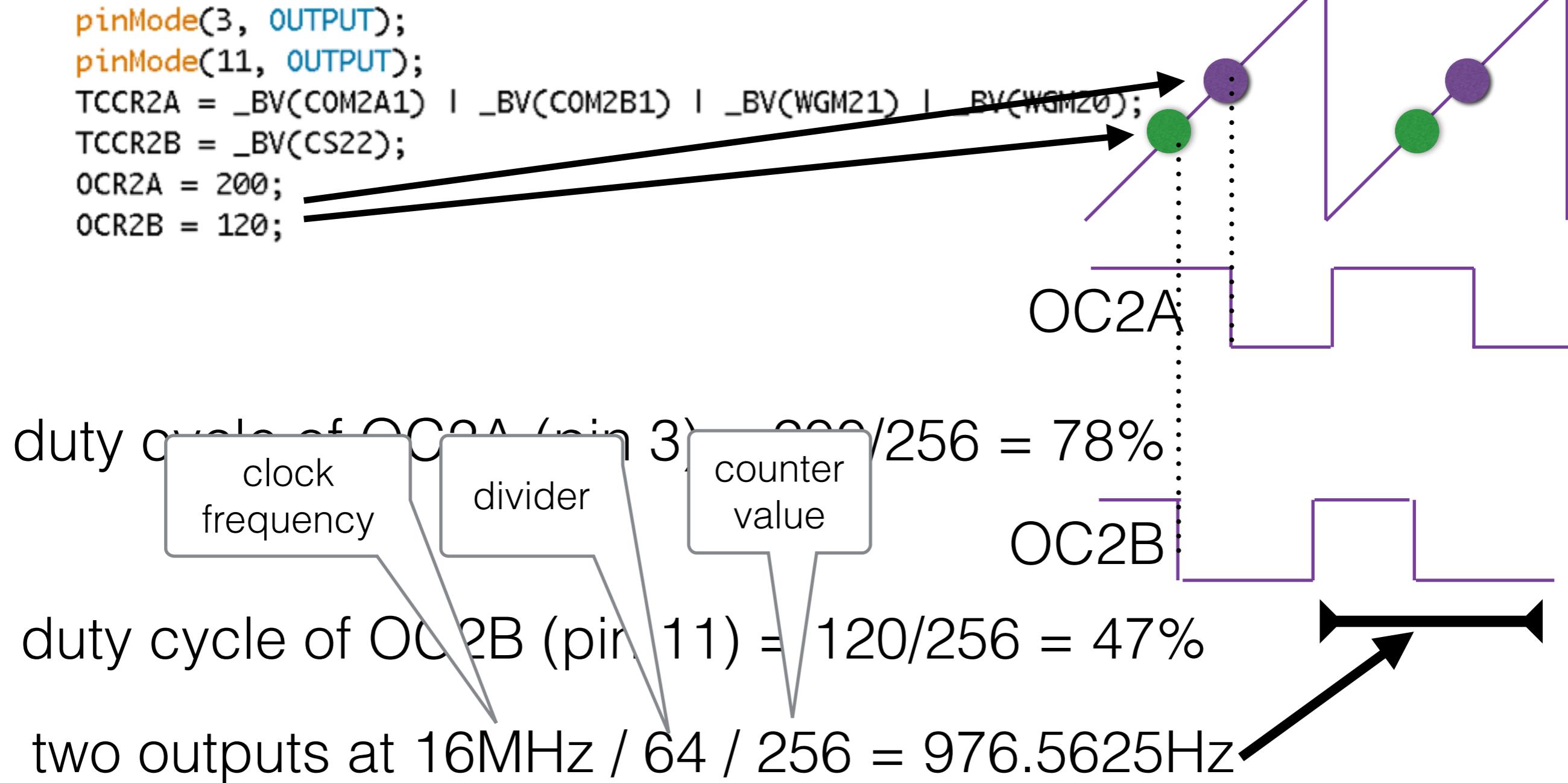
OCR2B – Output Compare Register B

Bit	7	6	5	4	3	2	1	0	
(0xB4)	OCR2B[7:0]								OCR2B
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	

The Output Compare Register B contains an 8-bit value that is continuously compared with the counter value (TCNT2). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC2B pin.

PWM with registers

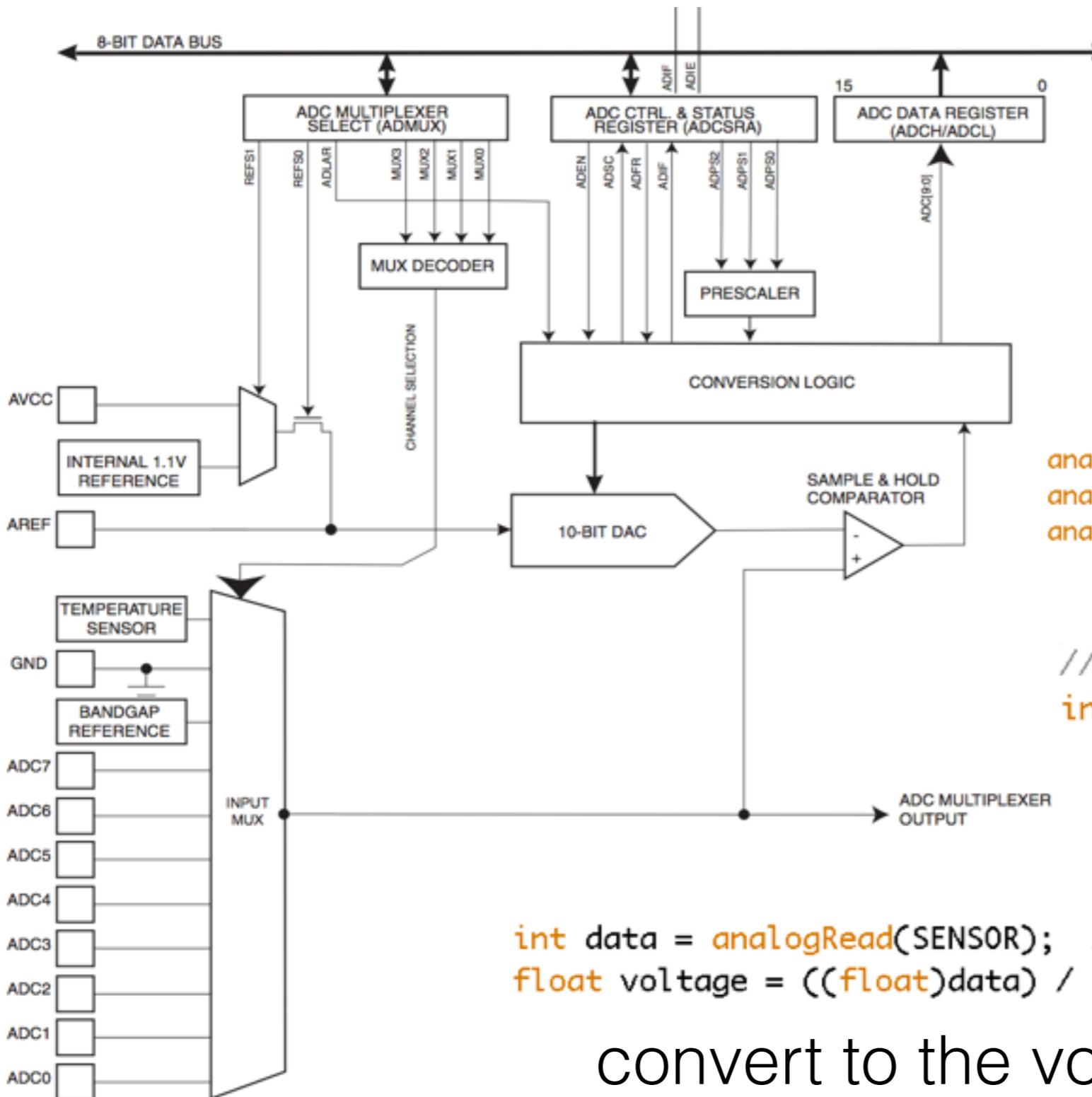
- registers control most peripherals on a uC



atmega ADC

- conversion from analog to digital is more explicit
 - dependent on reference voltage
 - sampling rate
 - dynamic range
- arduino: convert analog value into integer number
 - has 6 analog pins!
 - but they are muxed to one ADC \$\$
 - can run at 76.9kHz (15 kHz full resolution)
 - max resolution is 10 bits

atmega ADC



```
analogReference(DEFAULT); //5V reference  
analogReference(INTERNAL1V1); //1.1V  
analogReference(EXTERNAL); //Voltage from pin
```

```
// read the input on analog pin 0:  
int sensorValue = analogRead(A0);
```

```
int data = analogRead(SENSOR); // read value from ADC  
float voltage = ((float)data) / 1024.0*VREF;
```

convert to the voltage



atmega ADC

- fairly fast conversion: use prescaler
 - then use analogRead as normal

```
const unsigned char PS_16 = (1 << ADPS2);
const unsigned char PS_32 = (1 << ADPS2) | (1 << ADPS0);
const unsigned char PS_64 = (1 << ADPS2) | (1 << ADPS1);
const unsigned char PS_128 = (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

// set up the ADC
ADCSRA &= ~PS_128; // remove bits set by Arduino library
```

// you can choose a prescaler from
// PS_16, PS_32, PS_64 or PS_128

ADCSRA |= PS_64; // set our own prescal

full resolution, ADC < 200kHz

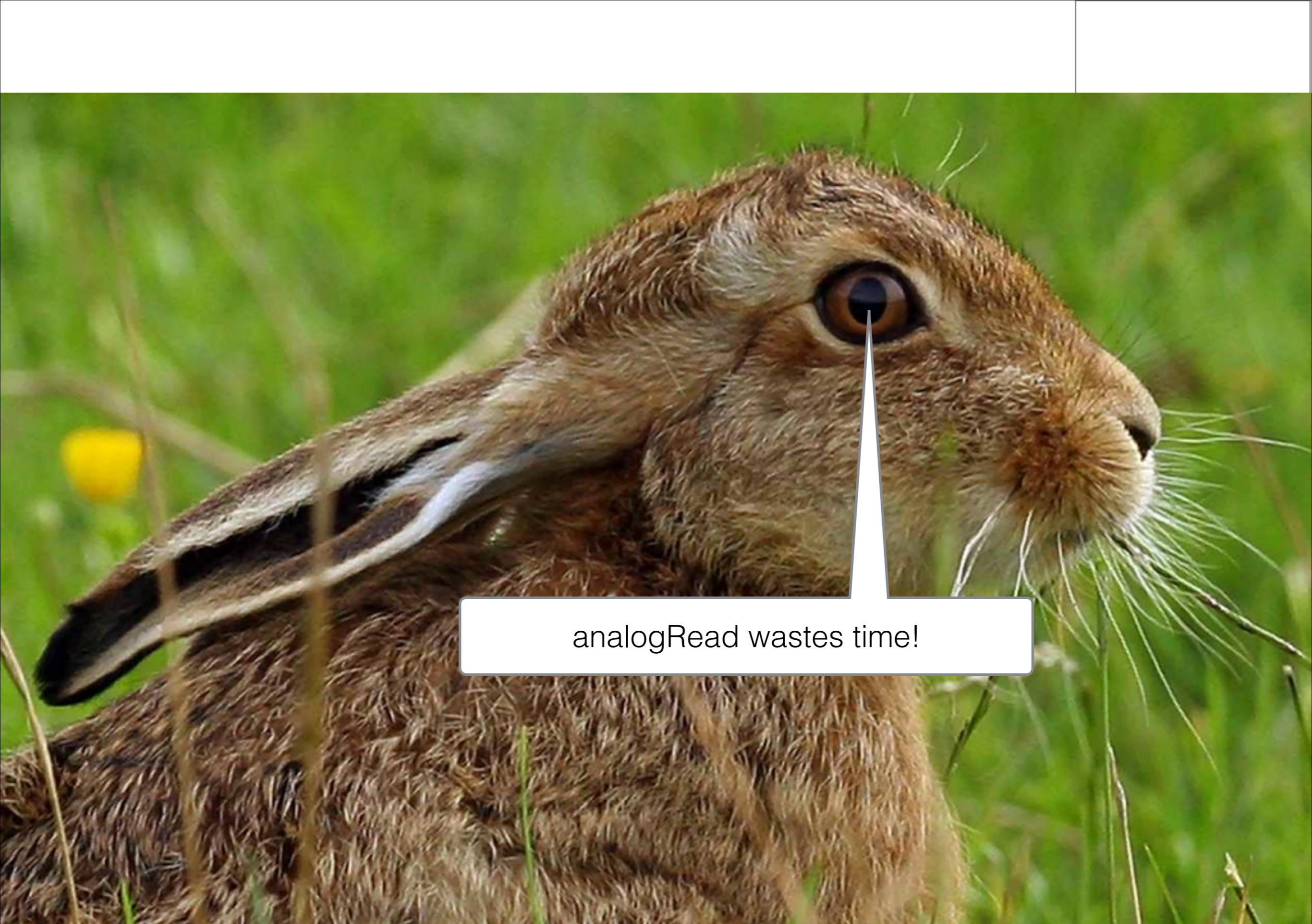
to 64

ADC sampling is 16MHz / 64 ~ 250kHz

			Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	ADCSRA
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	
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	



atmega ADC

- fastest ADC possible
 - do not use the arduino library

```
// set up the ADC
ADCSRA &= ~PS_128; // remove bits set by Arduino library

// you can choose a prescaler from above.
// PS_16, PS_32, PS_64 or PS_128
ADCSRA |= PS_64; // set our own prescaler to 64

/* Enable the ADC */
ADCSRA |= _BV(ADEN);

/* Set the PIN pin as an output. */
pinMode(A0, INPUT);
```

then use the registers directly to read ADC

atmega ADC

```
int adc_read(byte adcx) {  
    /* adcx is the analog pin we want to use. ADMUX's first few bits are  
     * the binary representations of the numbers of the pins so we can  
     * just 'OR' the pin's number with ADMUX to select that  
     * We first zero the four bits by setting ADMUX equal to  
     * four bits. */  
    ADMUX &= 0xf0;  
    ADMUX |= adcx;  
  
    /* This starts the conversion. */  
    ADCSRA |= _BV(ADSC);  
  
    /* wait around until the conversion is finished. */  
    while ( (ADCSRA & _BV(ADSC)) );  
  
    /* return the converted value */  
    return ADC;  
}
```

set reference

select analog pin

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	ADMUX
(0x7C)	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0	
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – REFS[1:0]: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in [Table 24-3](#). If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

- **Bits 3:0 – MUX[3:0]: Analog Channel Selection Bits**

The value of these bits selects which analog inputs are connected to the ADC. See [Table 24-4](#) for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

atmega ADC

```
int adc_read(byte adcx) {  
    /* adcx is the analog pin we want to use. ADMUX's first few bits are  
     * the binary representations of the numbers of the pins so we can  
     * just 'OR' the pin's number with ADMUX to select that pin.  
     * We first zero the four bits by setting ADMUX equal to its higher  
     * four bits. */  
    ADMUX = 0xf0;  
    ADMUX |= adcx;  
  
    /* This starts the conversion. */  
    ADCSRA |= _BV(ADSC); ——————  
    /* wait around until the conversion is finished. */  
    while ( (ADCSRA & _BV(ADSC)) );  
  
    /* return the converted value */  
    return ADC;  
}
```

start conversion

wait to be done

ADCSRA – ADC Control

Bit	7	6	5	4	3	2	1	0	ADCSRA
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	
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	

- **Bit 7 – ADEN: ADC Enable**

Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- **Bit 6 – ADSC: ADC Start Conversion**

In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC.

ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

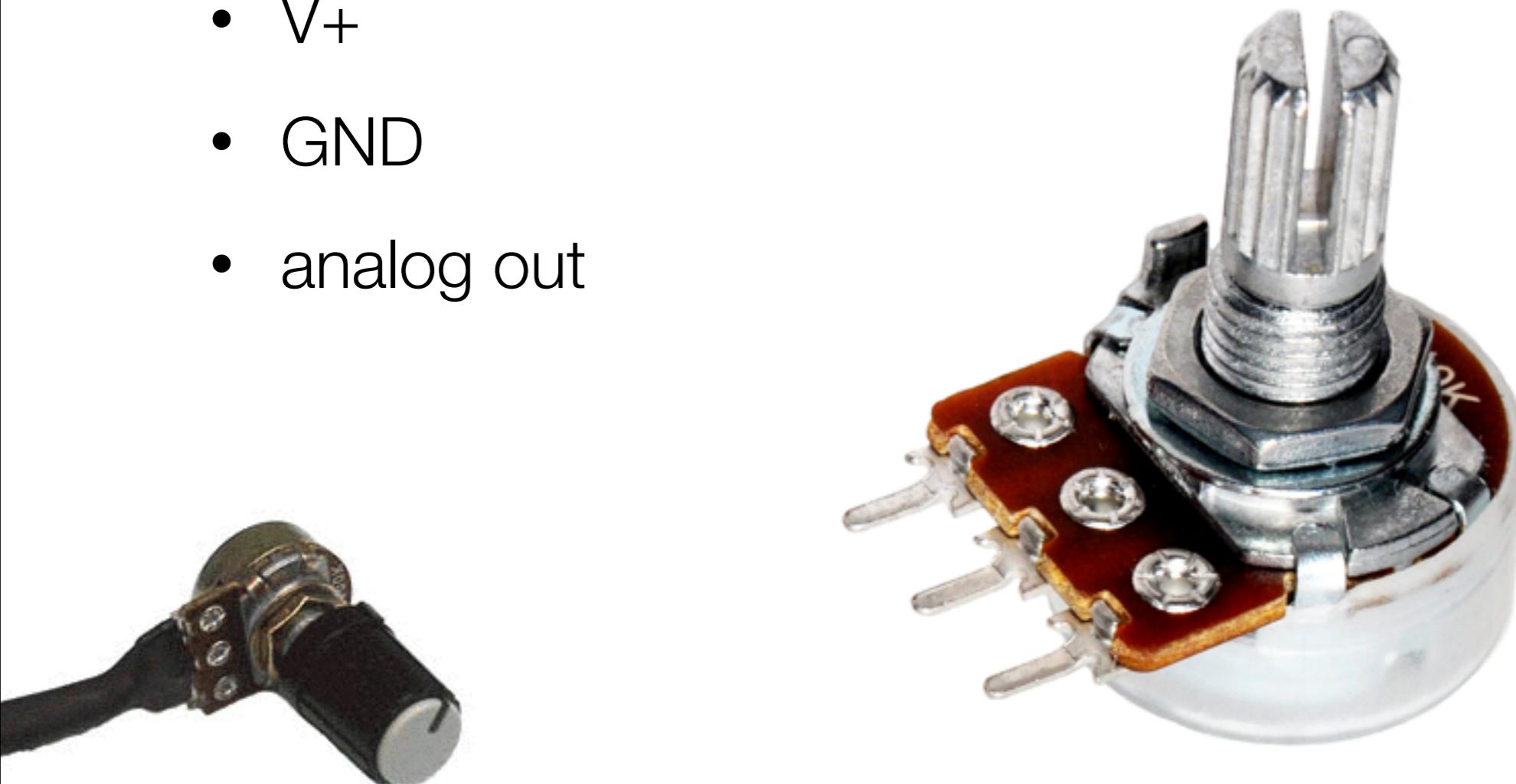


ADC summary

- try using analogRead without no adjustments
- if not fast enough, use the prescaler
- if you still need to shave time,
 - write your own adc_read using the registers

potentiometers

- controls voltage, divided by two resistors
- three leads
 - V+
 - GND
 - analog out



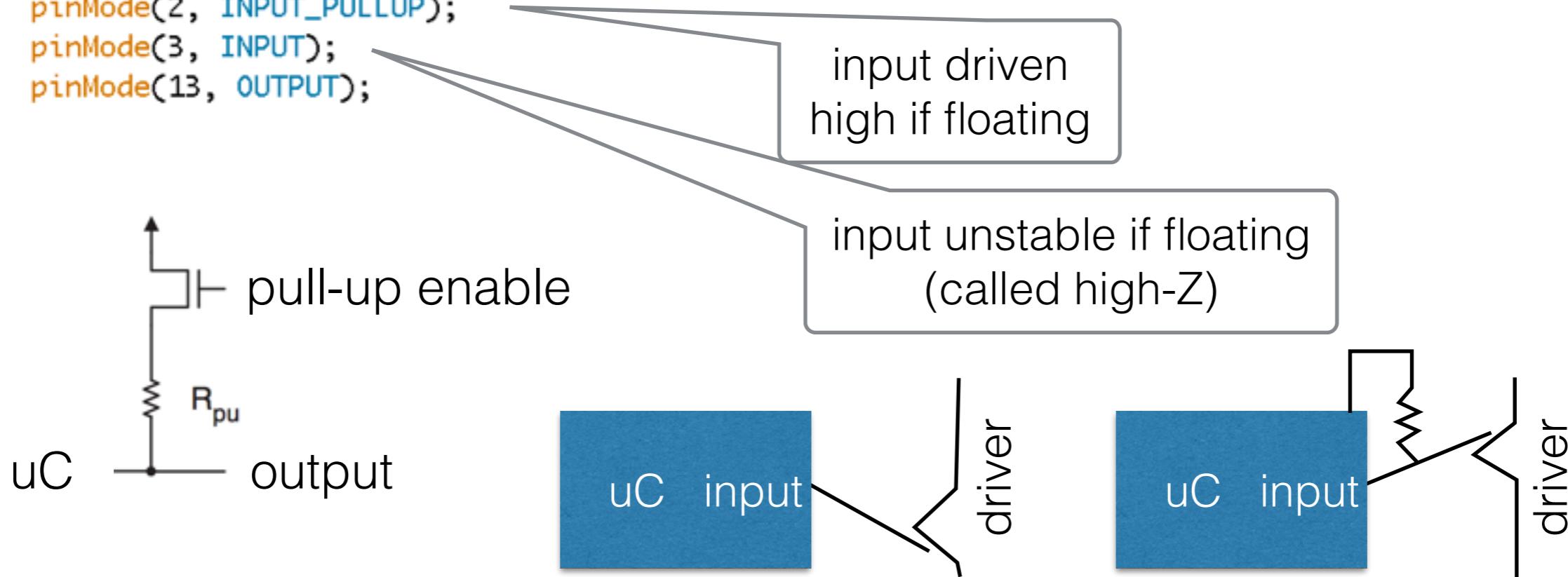
analog read demo

atmega GPIO

- digital inputs and outputs
 - buffered to drive ~40mA loads max
 - about the current needed to drive an LED, max brightness
- multiplexed for different functions
- internal pull-up resistors (20k)
 - can enable or disable
 - arduino auto enables pull-ups for input
 - disables for output
- edge detection (as interrupts)

atmega GPIO

```
//configure pin2 as an input and enable the internal pull-up resistor  
pinMode(2, INPUT_PULLUP);  
pinMode(3, INPUT);  
pinMode(13, OUTPUT);
```



arduino read or write

```
int sensorVal = digitalRead(2);  
digitalWrite(13, LOW);
```



atmega GPIO

- fast access

Table 14-1. Port Pin Configurations

DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

```
PORTB = (1<<PORTB7)|(1<<PORTB6)|(1<<PORTB1)|(1<<PORTB0);
```

```
DDRB = (1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0);
```

byte pinValues = PINB;

read register

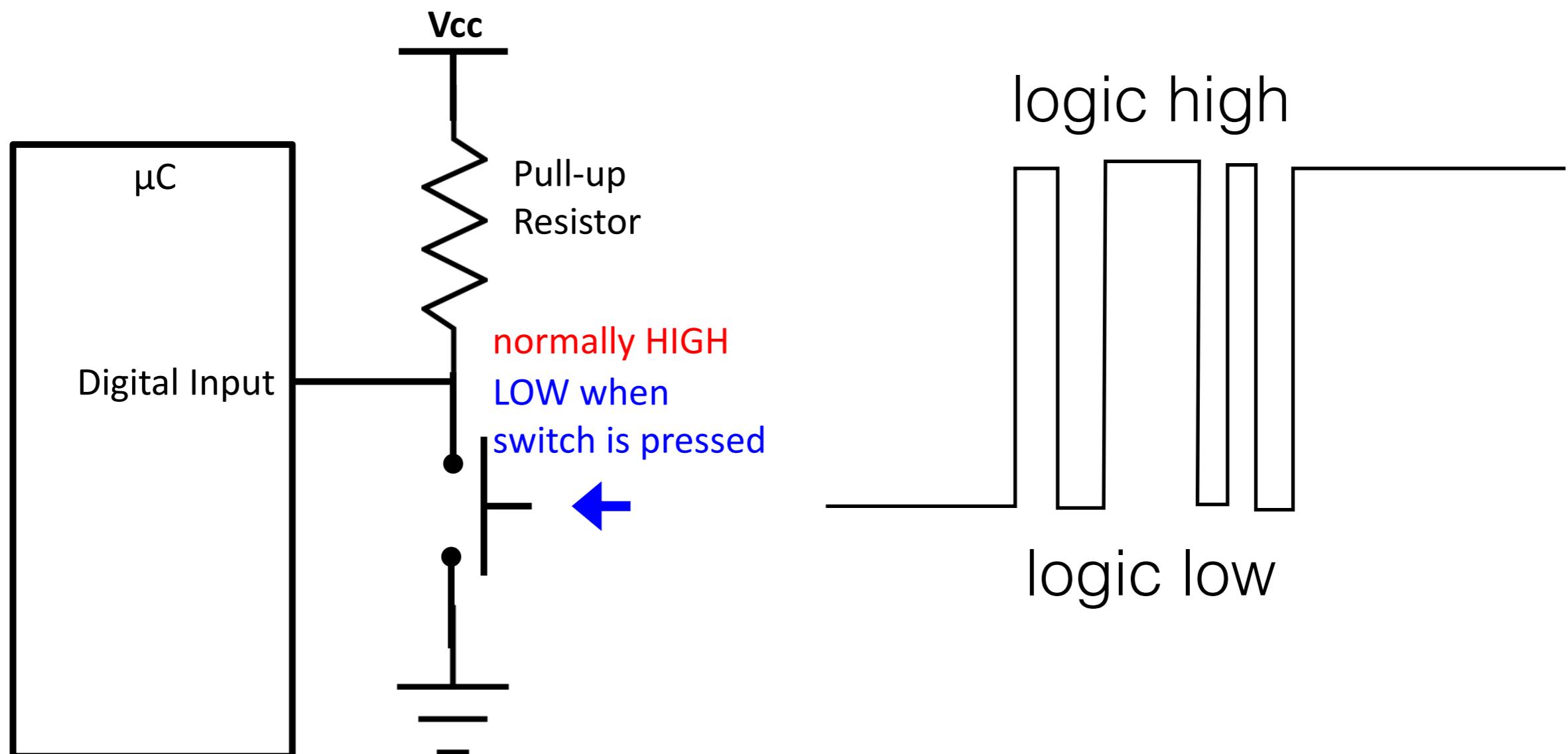
what does this do?

PORTB = 0b11000011
DDRB = 0b00001111
in in lo hi



atmega GPIO

- connecting to mechanical devices
 - like a switch



atmega GPIO

- connecting to mechanical devices
 - like a switch

```
// If the switch changed, due to noise or pressing:  
if (reading != lastButtonState) {  
    // reset the debouncing timer  
    lastDebounceTime = millis();  
}  
  
if ((millis() - lastDebounceTime) > debounceDelay) {  
    // whatever the reading is at, it's been there for longer  
    // than the debounce delay, so take it as the actual current state:  
  
    // if the button state has changed:  
    if (reading != buttonState) {  
        buttonState = reading;  
  
        // only toggle the LED if the new button state is HIGH  
        if (buttonState == HIGH) {  
            ledState = !ledState;  
        }  
    }  
}
```

arduino interrupts

- interrupts are from external or internal triggers
 - dedicated hardware “looks” for a trigger
 - main loop stops execution
 - local variables and instruction location copied to stack
 - **other interrupts turned off**
 - interrupt service routine called
 - after execution, stack reloaded, main loop continues

arduino interrupts

```
volatile boolean blinkLED;
```

select pin 2

call this routine
(not function)

```
// configure interrupt when switch is pressed  
attachInterrupt(0, SW_ISR, FALLING);
```

Uno: pin2=0, pin3=1

could be rising,
falling, change, low

```
/* switch press interrupt service routine */  
void SW_ISR() {  
    blinkLED = !blinkLED; // toggle blink state  
}
```

switch off/on interrupts

```
noInterrupts();
```

```
// code that needs continuous execution
```

```
interrupts();
```

AVR interrupts

- arduino does not give full access to interrupts
- so timer interrupts must use the AVR c-code

```
TCCR1A = 0; // clear control register A
TCCR1B = _BV(WGM13); // set PWM phase correct mode

// get the prescaler amount
long cycles = (F_CPU / 2000000) * MICROSECONDS;
// the counter runs backwards after TOP, interrupt is at
if(cycles < RESOLUTION)
    clockSelectBits = _BV(CS10); // no prescale, full xtal
else if((cycles >>= 3) < RESOLUTION) clockSelectBits = _BV(CS11); // prescale by /8
else if((cycles >>= 3) < RESOLUTION) clockSelectBits = _BV(CS11) | _BV(CS10); // prescale by /64
else if((cycles >>= 2) < RESOLUTION) clockSelectBits = _BV(CS12); // prescale by /256
else if((cycles >>= 2) < RESOLUTION) clockSelectBits = _BV(CS12) | _BV(CS10); // prescale by /1024
else
    cycles = RESOLUTION - 1, clockSelectBits = _BV(CS12) | _BV(CS10); // request was out of bounds, set as max

// set the timer period
ICR1 = cycles; // number of cycles before overflow (when TCR = 1111)

// set the prescaler
TCCR1B &= ~(_BV(CS10) | _BV(CS11) | _BV(CS12)); // clear bits
TCCR1B |= clockSelectBits; // set scalar bits
// enable the interrupt
TIMSK1 = _BV(TOIE1); // sets the timer overflow interrupt
```

max value

call ISR

max value

call ISR

max value

call ISR

count up and then down

timer value
like PWM!
call ISR

convert u-seconds to cycles

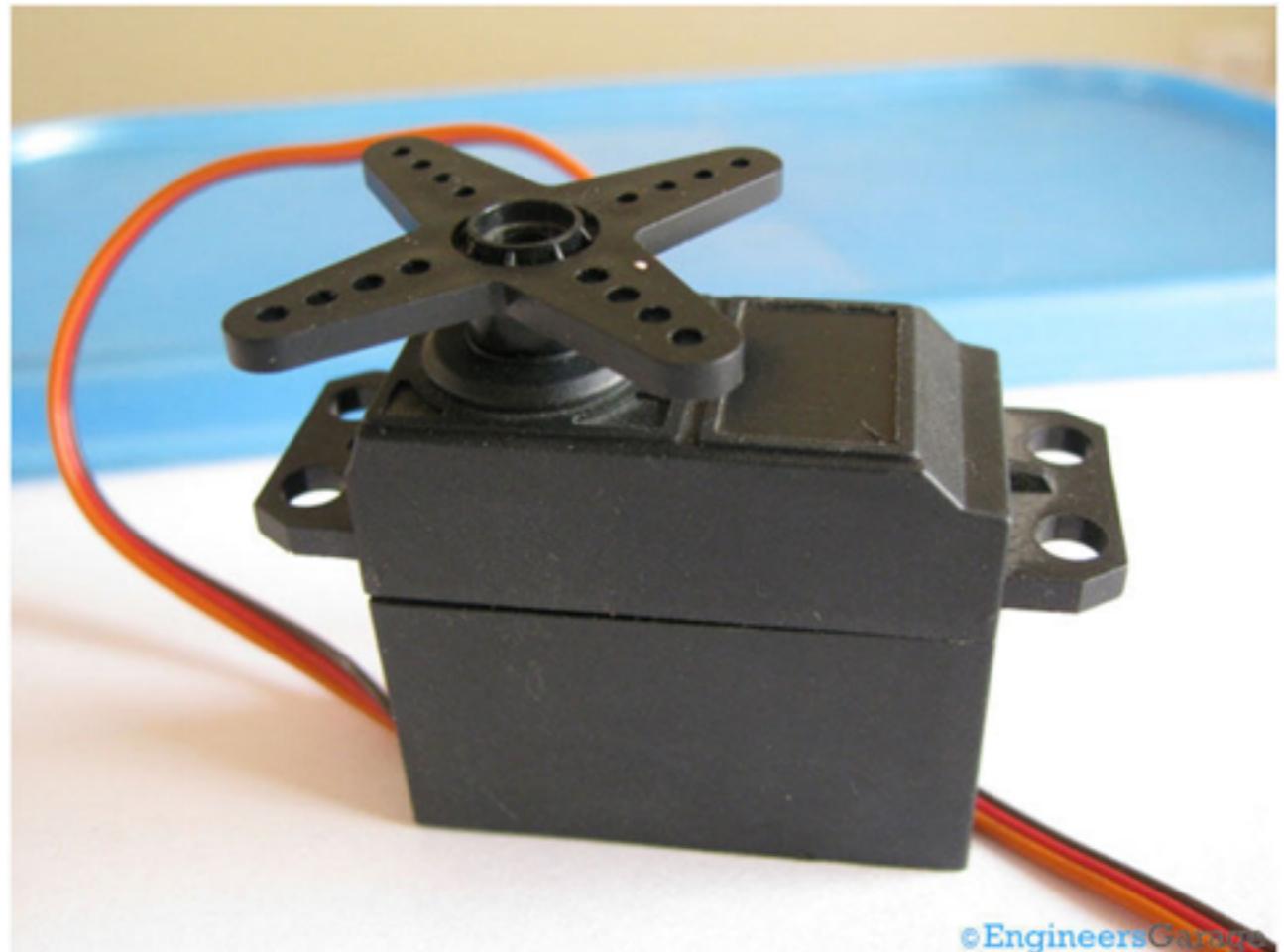
max value of counter

set pre-scale

enable interrupt from PWM

servos

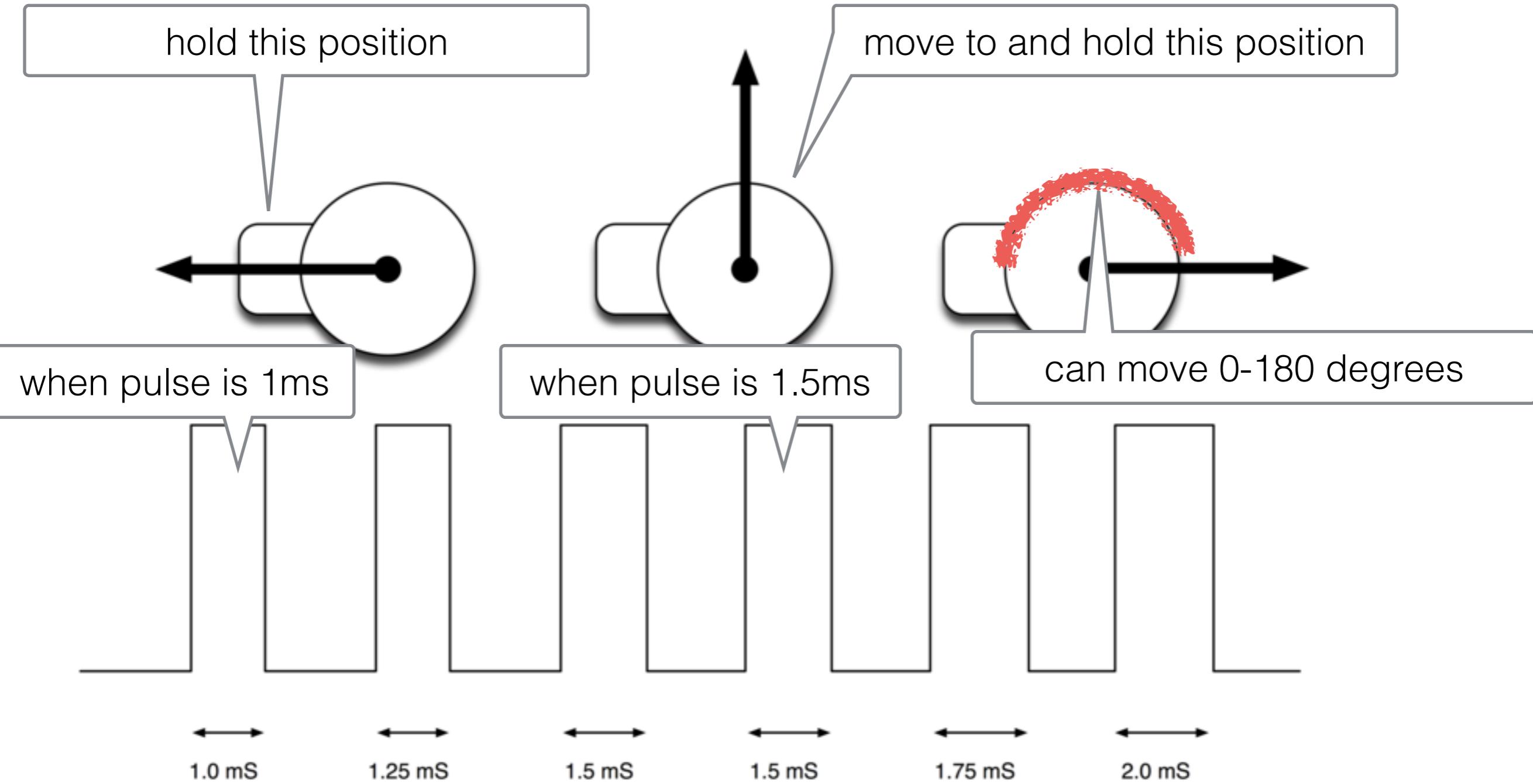
- two types
 - position: rotate and stay
 - continuous: delta movements
- controlled through pulses



©EngineersGarage

rotational servo

- needs repeating pulse train (usually 1-2ms)



arduino servo

- makes it really easy for 1-2ms pulsed servos
- #include <Servo.h>

```
Servo myservo; // create servo object to control a servo  
int potpin = 0; // analog pin used to connect the potentiometer  
int val; // variable to read the value from the analog pin  
  
void setup()  
{  
    myservo.attach(9); // set PWM from a  
} // resistor (pot) to the servo object  
  
void loop()  
{  
    val = analogRead(potpin); // reads the value of the potentiometer (value between 0 and 1023)  
    val = map(val, 0, 1023, 0, 179); // scale it to use it with the servo (value between 0 and 180)  
    myservo.write(val); // sets the servo position according to the scaled value  
    delay(15); // waits for the servo to get there  
}
```

set pwm from a resistor (pot)

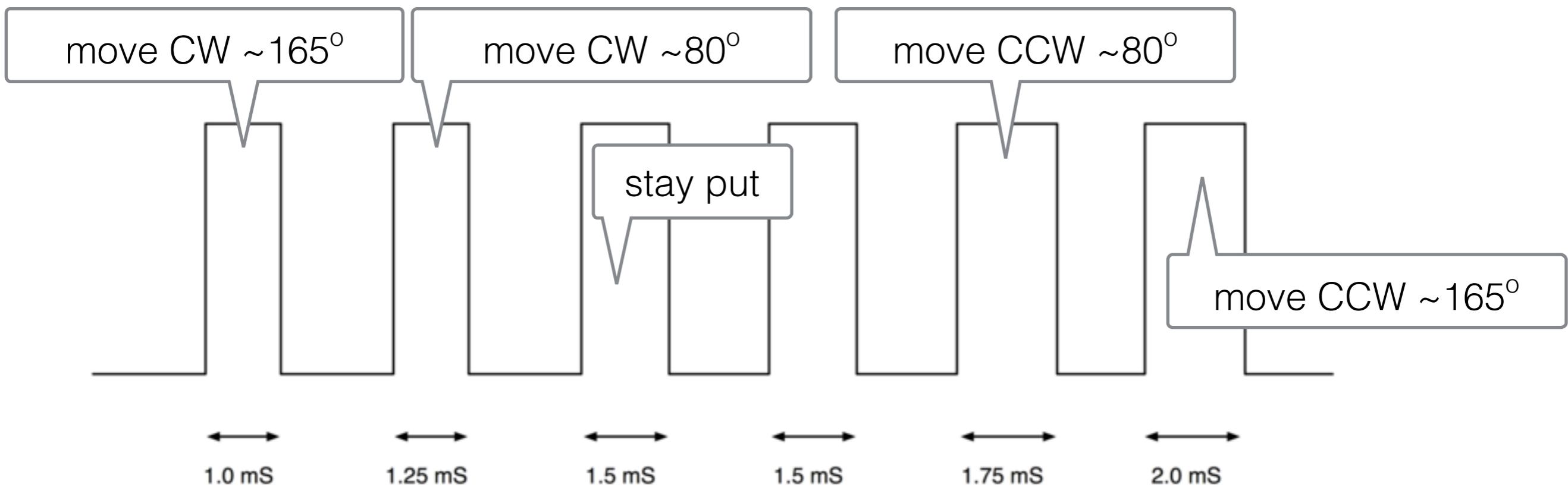
use pin 9 (must be PWM)

set the new angle (and forget it)

instance of class (up to 6 on the uno)

continuous servos

- each pulse is a new command
- pulse width is angle to move from current angle
 - servo can rotate all the way around
 - counter clockwise (CCW) or clockwise (CW)

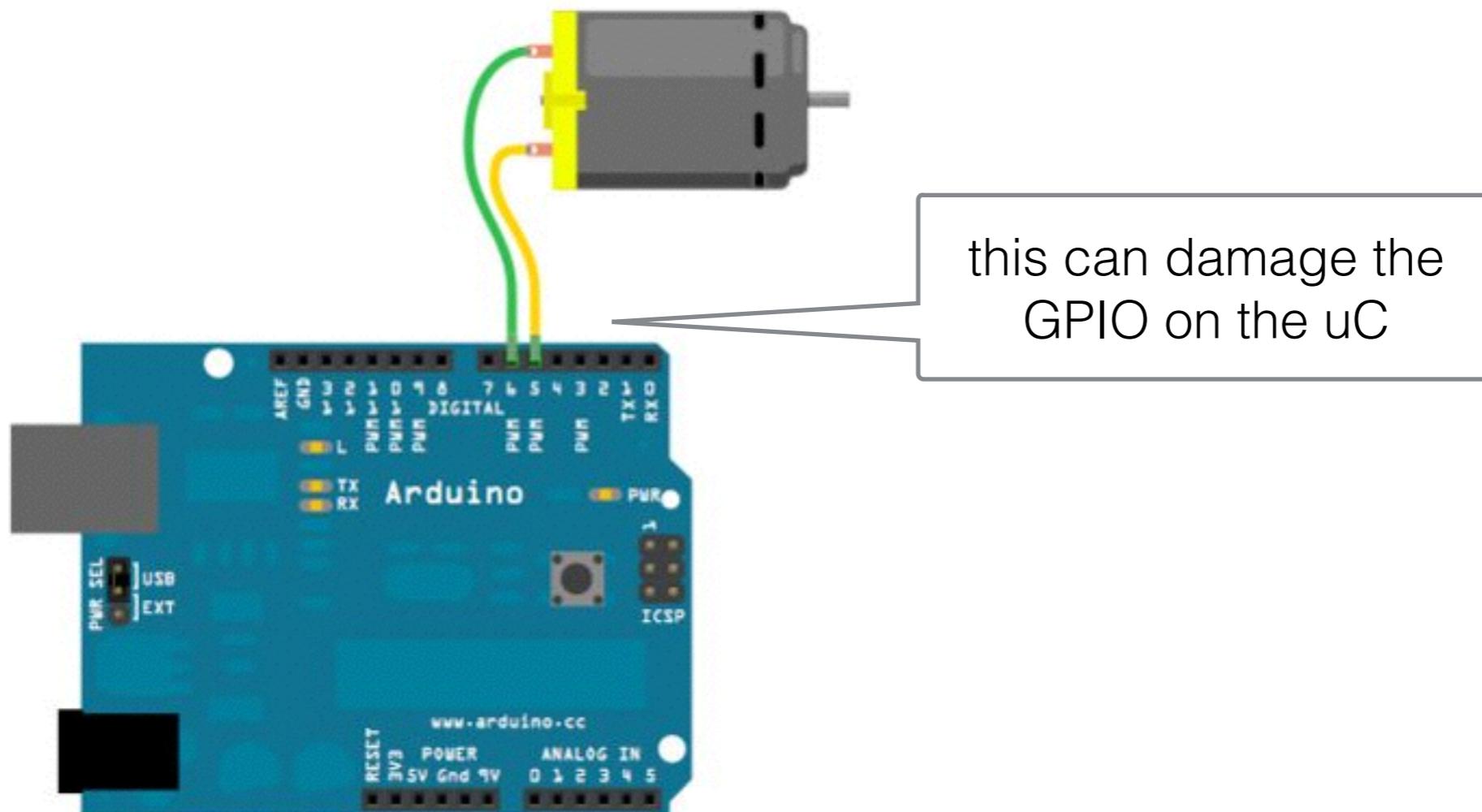


continuous servo demo

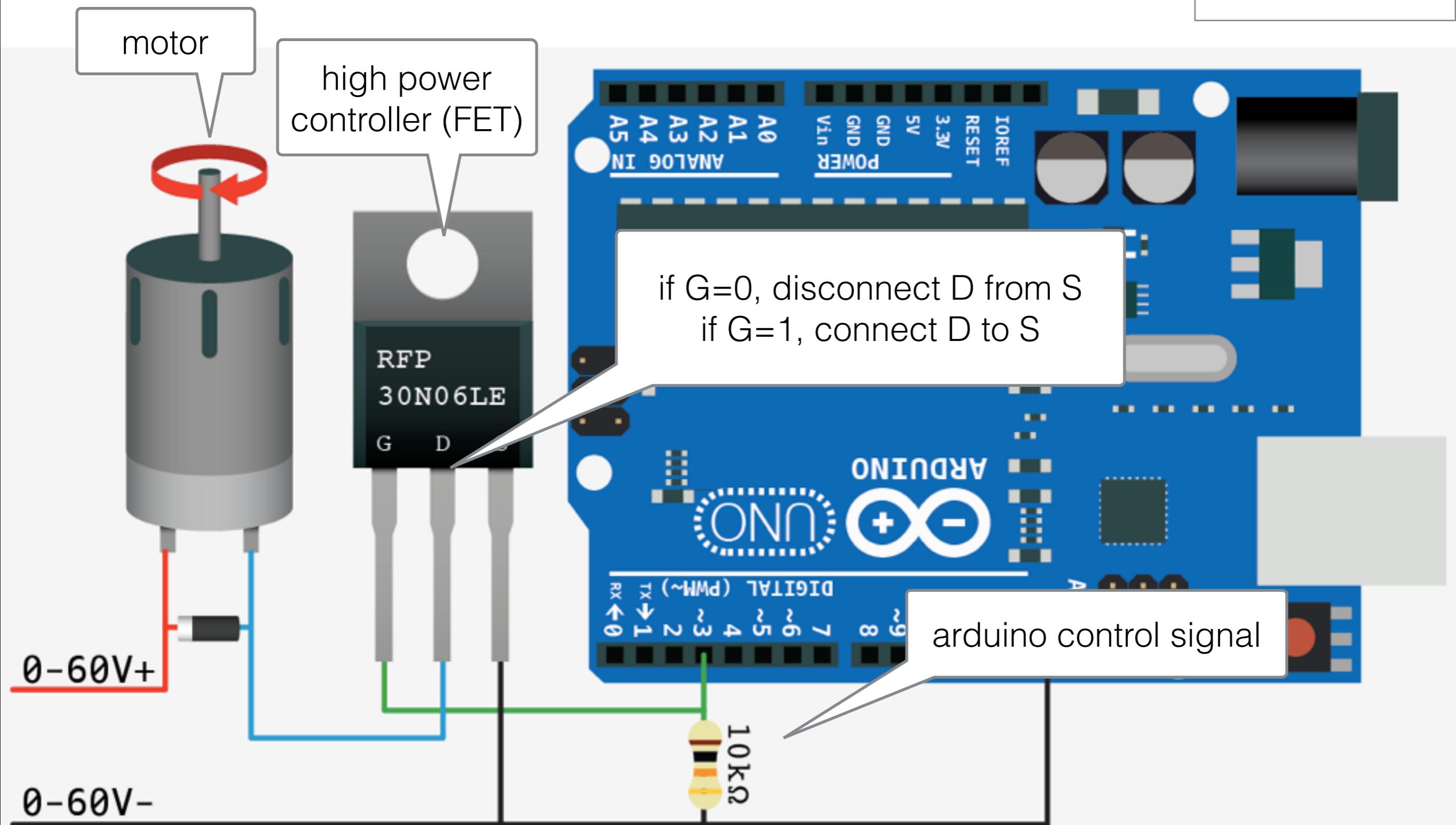
DC motor control

- give power to the motor in bursts
 - longer bursts == more power!
- just like LED brightness
- ...or like the dimmer switch in your home
- motors use a lot of power!
 - arduino board will not cut it
 - arduino just provides the control signal to motor

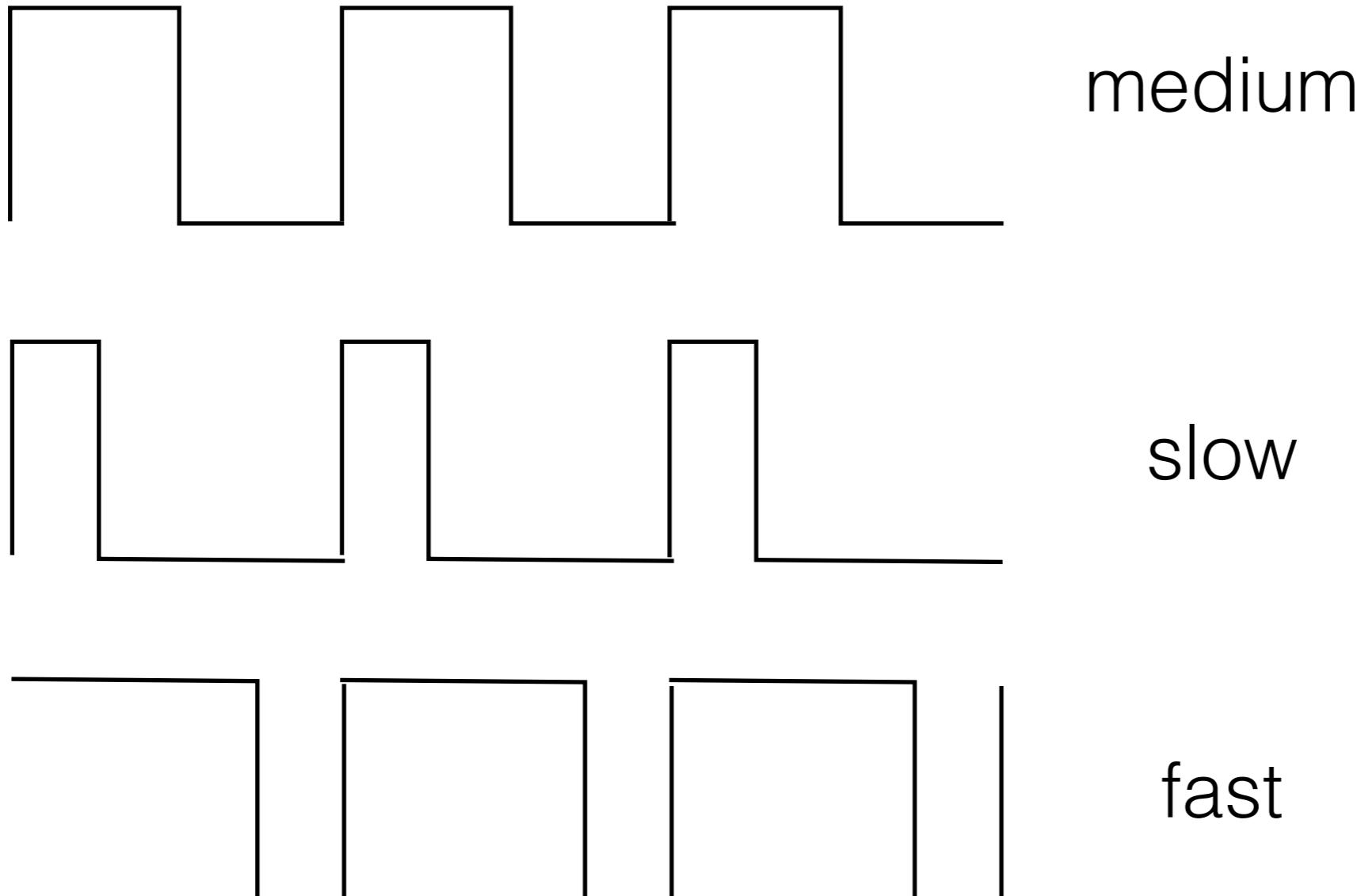
DC motor control



DC motor control

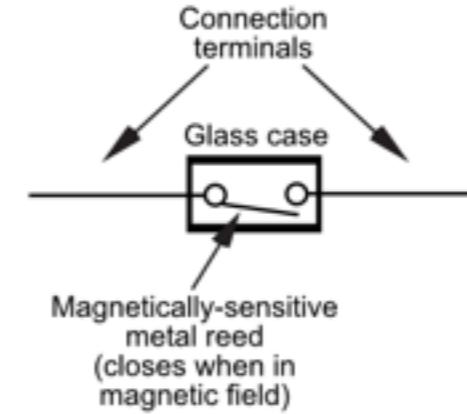


DC motor control



other sensors

- binary
 - reed switches
 - buttons
 - ball switches
- analog
 - hall effect sensors
 - capacitive touch
 - accelerometers
- <http://playground.arduino.cc/Main/InterfacingWithHardware#arstat>

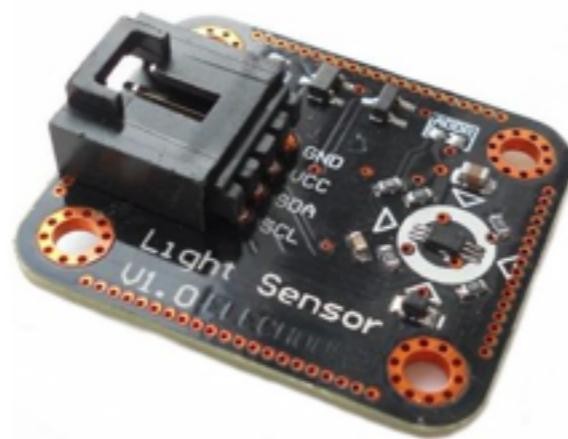
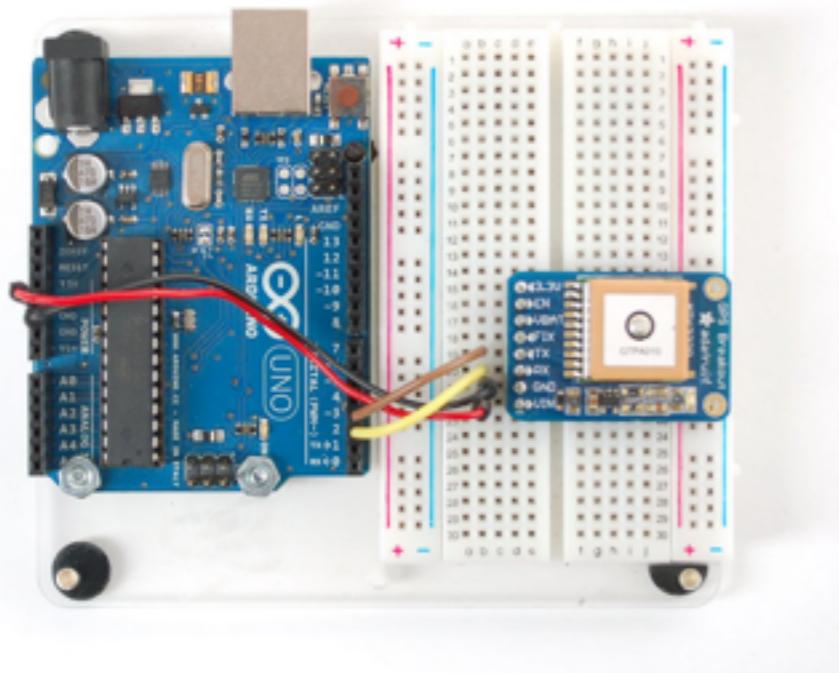
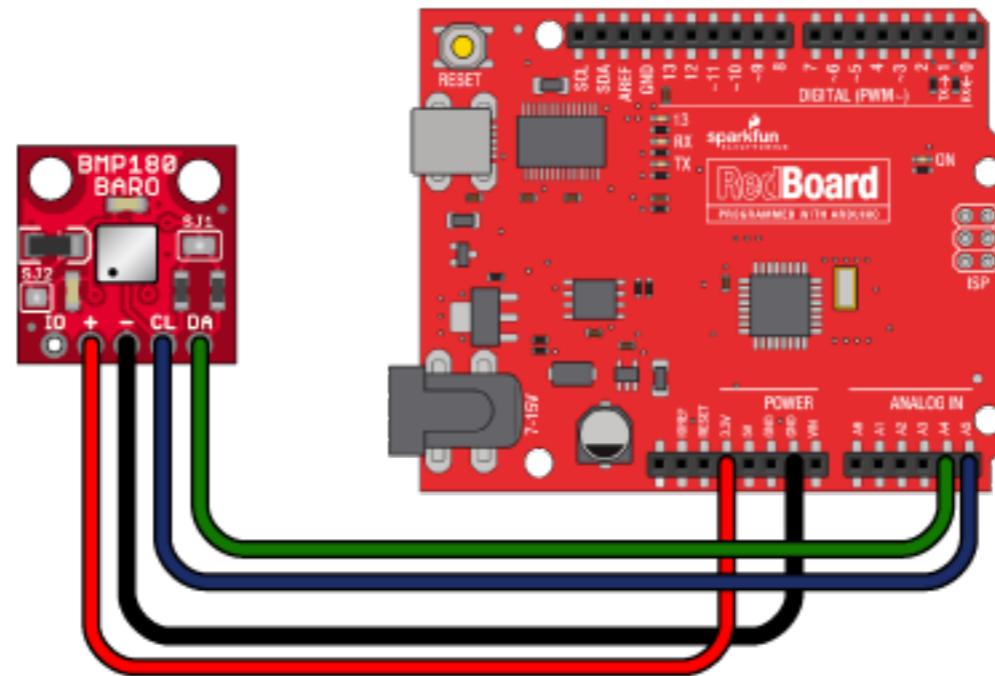


other sensors

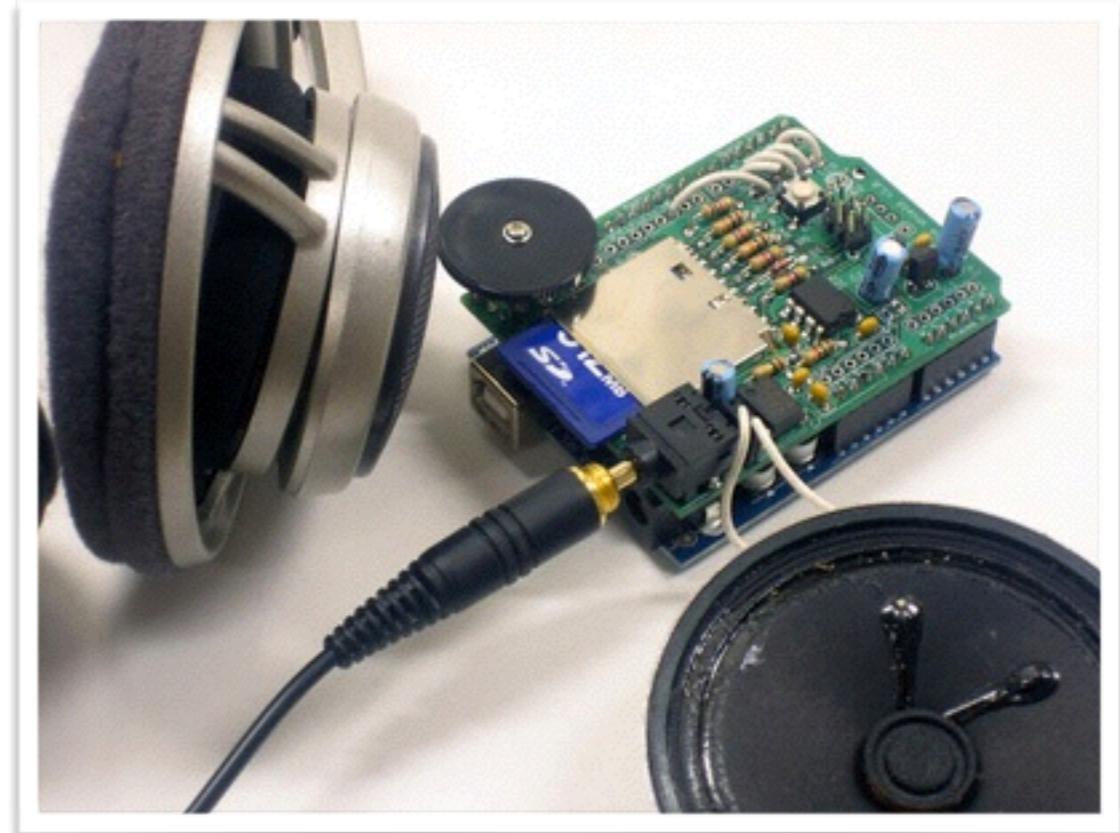
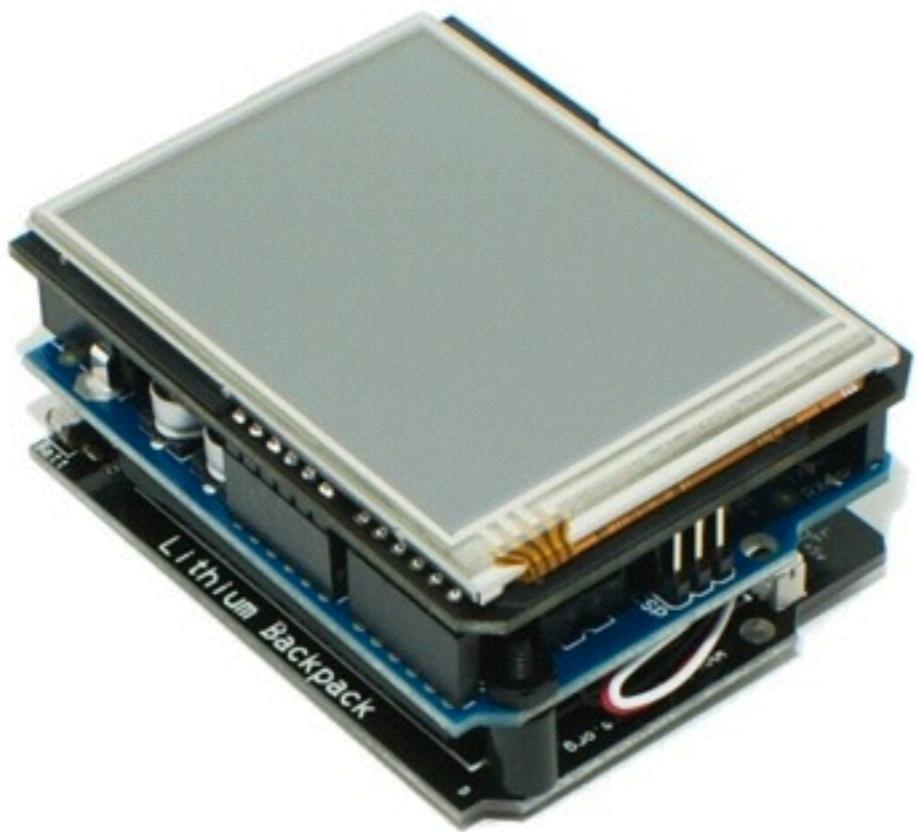
- digital sensors



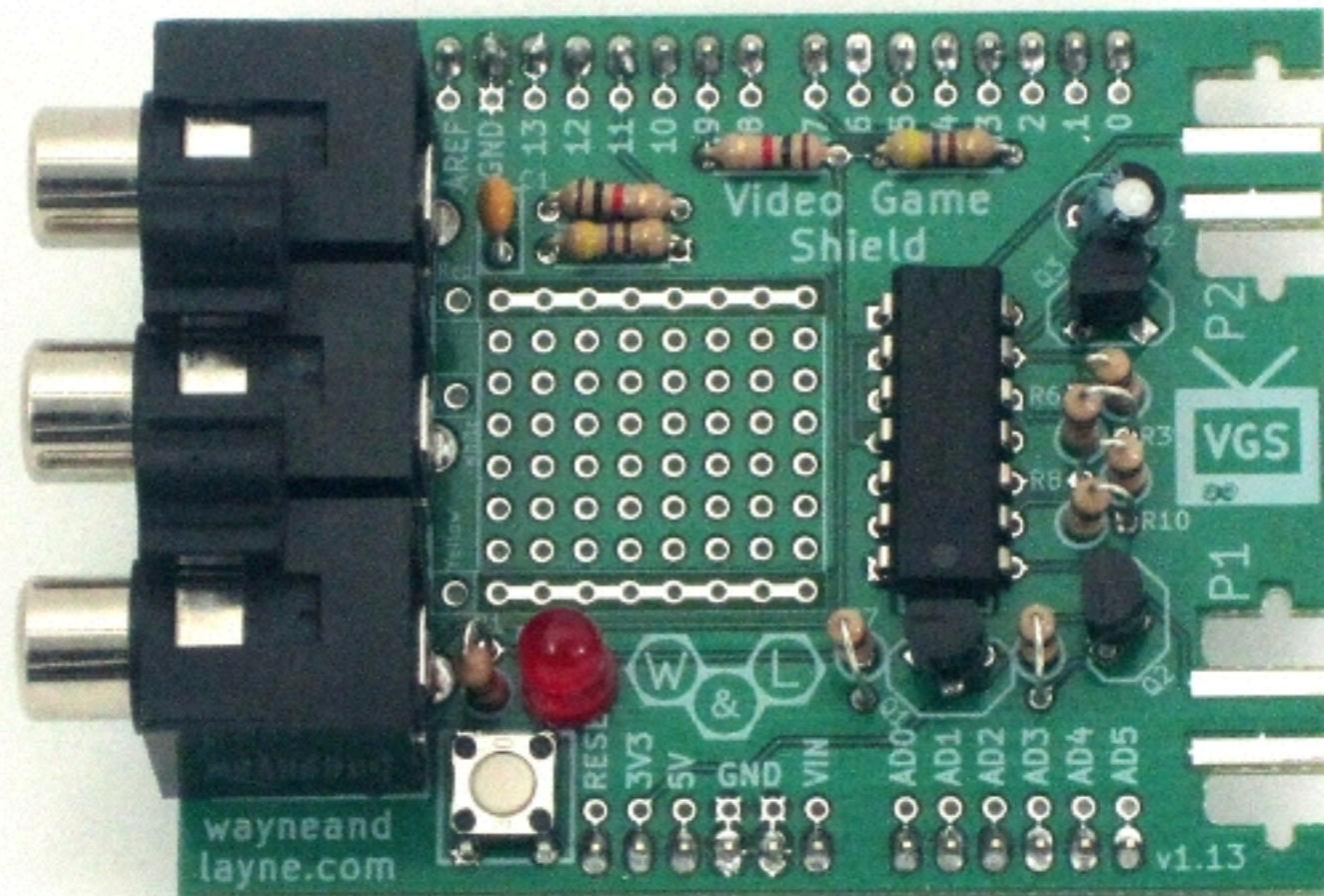
TMP04FS: Serial Digital Output Thermometer



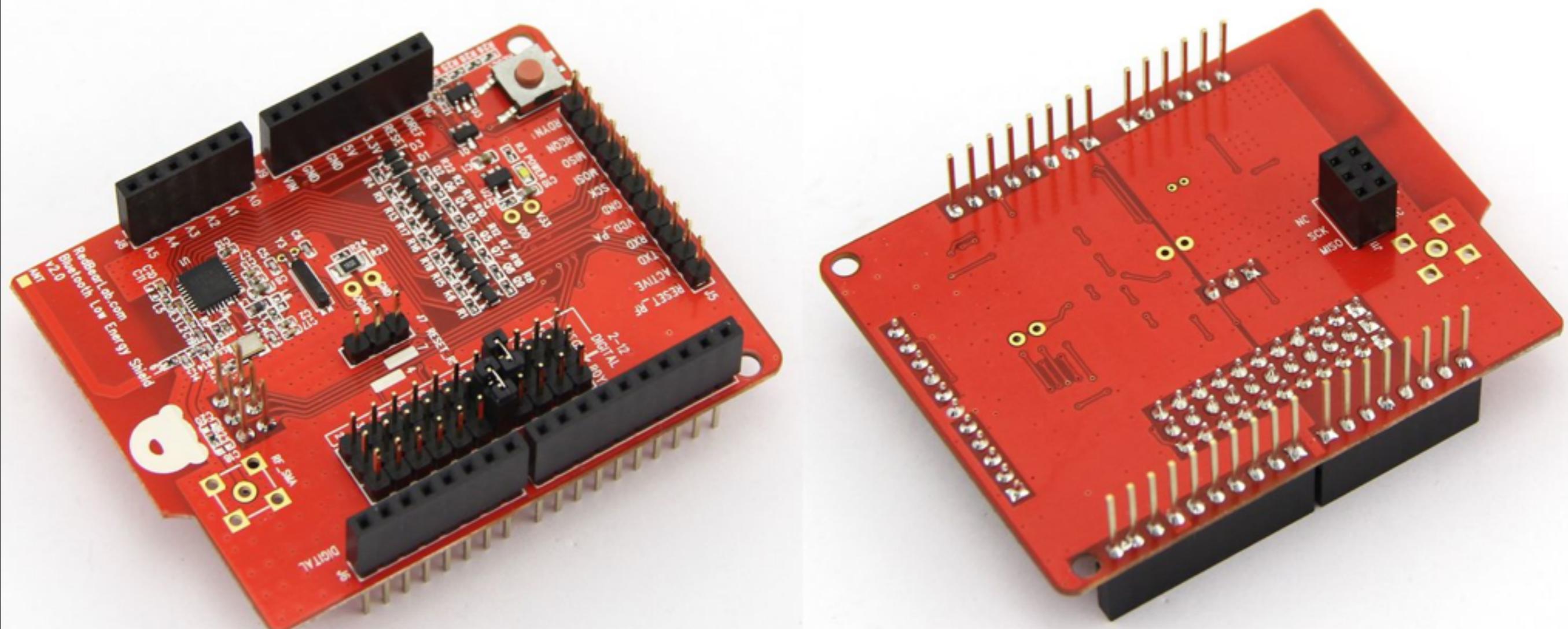
arduino shields

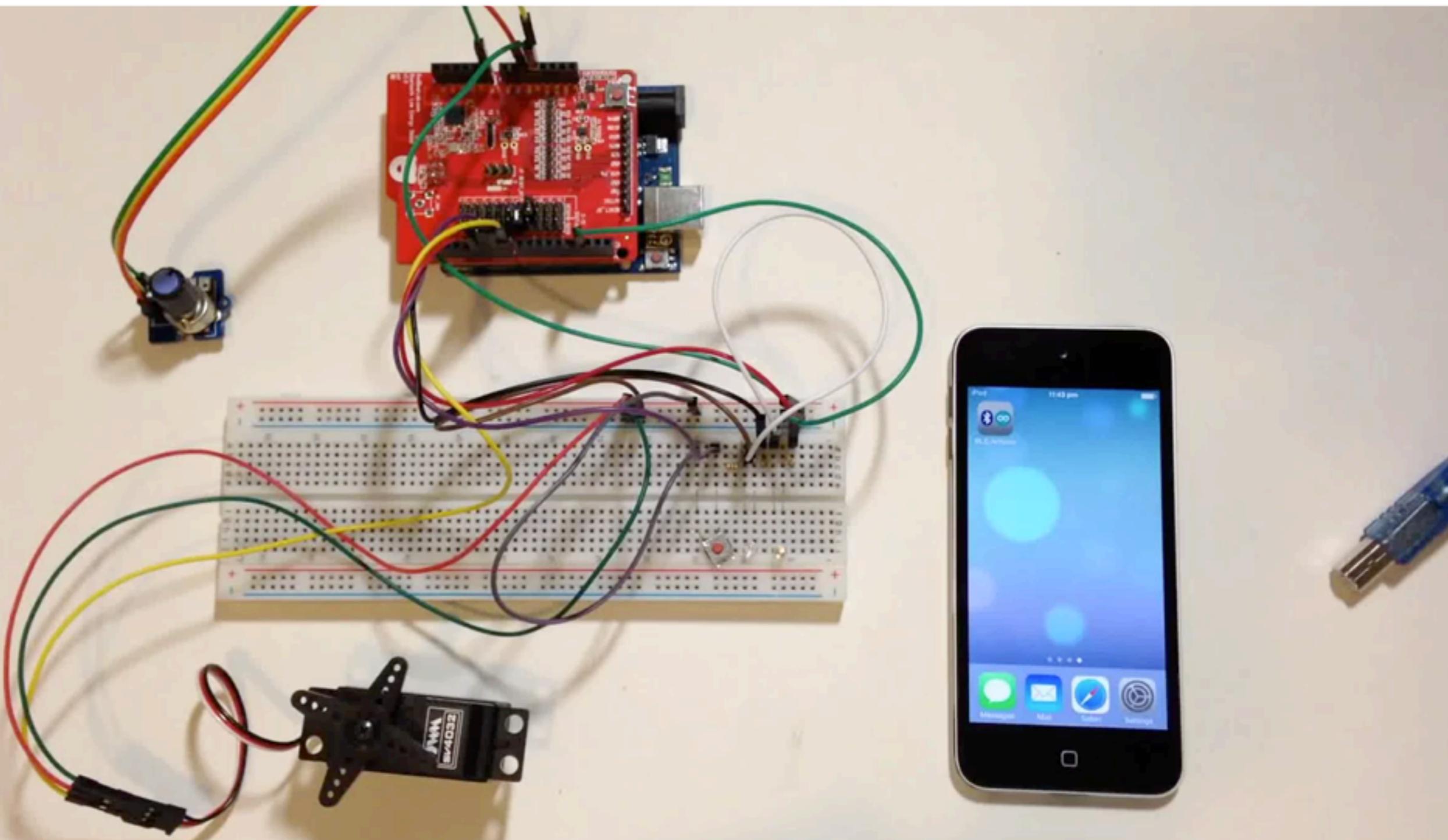


arduino shields



arduino shields





BLE shield

- bluetooth made really easy
- takes care of all the protocol of bluetooth
- uses pins 8-13 of the arduino (SPI)
 - 10 (SS), 11 (MOSI), 12 (MISO), and 13 (SCK)
 - 8 and 9 are RDYN and REQN for handshaking
 - remaining pins are A0-A5 and 0-7
- all communication is setup through `<ble_shield.h>`

BLE shield

setup

```
// Init. and start BLE library.  
ble_begin();  
  
// Enable serial debug  
Serial.begin(57600);
```

The diagram illustrates the code structure. The first two lines of code, `ble_begin();` and `Serial.begin(57600);`, are grouped together and connected by a bracket to a callout box labeled "setup SPI & BLE shield controls". The remaining code block, `while (ble_available()) { Serial.write(ble_read()); }`, is also bracketed and connected to another callout box labeled "if data available on SPI". A third callout box, "read the byte", points to the `ble_read()` function within the loop.

read from BLE, write to serial port

```
while ( ble_available() )  
  Serial.write(ble_read());
```

The diagram shows the main loop logic. It starts with a condition `while (ble_available())`. If true, it executes the block `Serial.write(ble_read());`. This block is connected to a callout box labeled "if data available on SPI". Inside this block, the function `ble_read()` is connected to another callout box labeled "read the byte".

BLE shield

send text that we read from the serial port

```
while ( Serial.available() )  
{  
    unsigned char c = Serial.read(); // read from debug computer  
    if (c != 0x0A)  
    {  
        if (len < 16)  
            buf[len++] = c; // pack in up to 16 bytes  
    }  
    else  
    {  
        buf[len++] = 0x0A;  
  
        for (int i = 0; i < len; i++)  
            ble_write(buf[i]); // add to transmit buffer  
        len = 0;  
    }  
}  
  
ble_do_events(); // perform transmits and acks
```

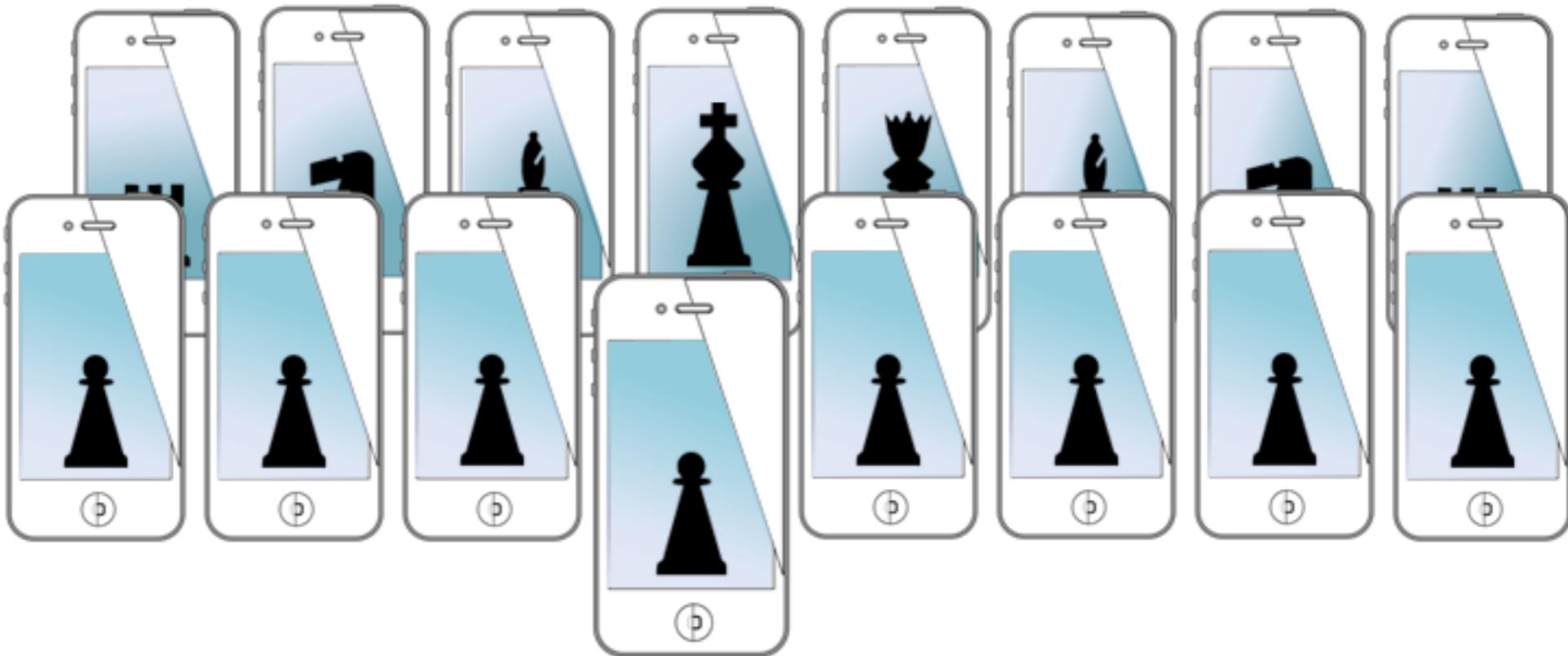
BLE firmata

- enables custom firmware
 - set pins' output (analog or digital)
 - read input (analog or digital)
 - and essentially tell the arduino to do anything
- its a protocol adopted
 - the BLE shield has already implemented this
 - the code can perform almost any operation that includes
 - reads, writes, ADC sampling, PWM, timers, serial comm, dedicated servo control
- <http://redbearlab.com/bleshield/>

for next time...

- all about bluetooth
 - overview of specs
 - using the API and libraries to transmit data between iPhone and arduino

MOBILE SENSING LEARNING & CONTROL



CSE5323 & 7323

Mobile Sensing, Learning, and Control

lecture fifteen: microcontroller sensing and control

Eric C. Larson, Lyle School of Engineering,
Computer Science and Engineering, Southern Methodist University