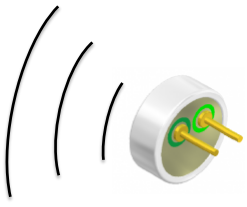
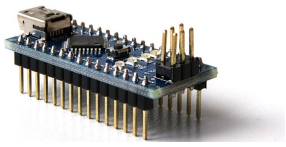
ETH Course 402-0248-00L: Electronics for Physicists II (Digital)

- 1: Setup uC tools, introduction
- 2: Solder SMD Arduino Nano board
- 3: Build application around ATmega328P
- 4: Design your own PCB schematic
- 5: Place and route your PCB
- 6: Start logic design with FPGAs

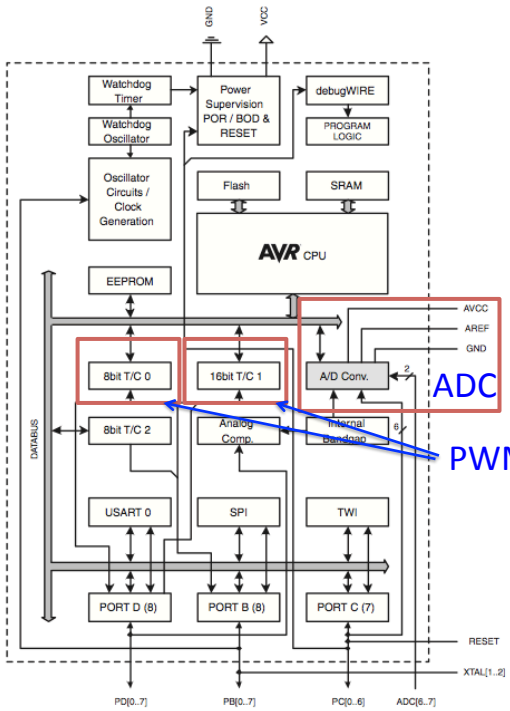
Exercise 3: "Sound volume robot"

- measures sound volume and moves arm to indicate loudness
- microphone -> preamp -> ADC -> uC -> PWM output





(debugging, programming)



Modules you will use

Data Direction Register (DDR)

- If the bit is *zero* -> pin will be an *input*
  - Making a bit to be zero == '**clearing** the bit'
- If the bit is *one* -> pin will be an *output*
  - Making a bit to be one == '**setting** the bit'
- To change the data direction for a set of pins belonging to PORTx at the same time:
  1. Determine which bits need to be set and cleared in DDRx
  2. Store the binary number or its equivalent (in an alternate base, such as hex) into DDRx

PORT Pin and register details

ATmega328 datasheet, pp. 76-94

Figure 13-1. I/O Pin Equivalent Schematic

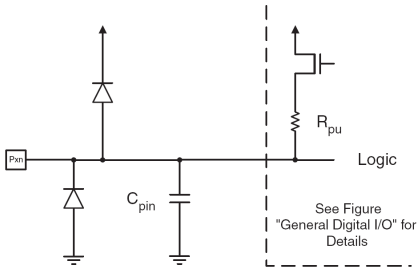
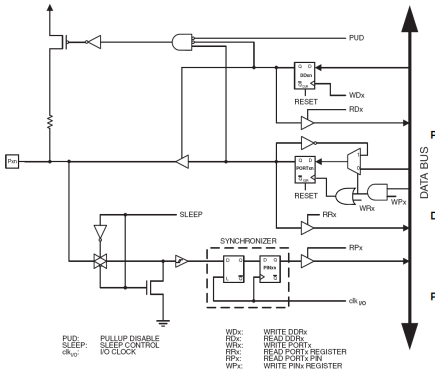


Figure 13-2. General Digital I/O[1]



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
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

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
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

PIND – The Port D Input Pins Address

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

ATmega328 Port Pin Details

- See the ATmega328 data sheet, pp. 76-94
- Port pin functionality is controlled by three *register* (special memory location) bits:
  - DDRx
    - Data Direction bit in DDRx register (read/write)
  - PORTxn
    - PORTxn bit in PORTx data register (read/write)
  - PINxn
    - PINxn bit in PINx register (read only)

Example 1

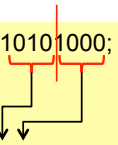
- Make Arduino pins 3, 5, and 7 (PD3, PD5, and PD7) to be outputs

- Arduino approach

```
pinMode(3, OUTPUT);
pinMode(5, OUTPUT);
pinMode(7, OUTPUT);
```

- Alternate approach

```
DDRD = 0b10101000;

or

DDRD = 0xA8;

or

DDRD | = 1<<PD7 | 1<<PD5 | 1<<PD3;
```



Example 2

- Make pins Arduino pins 0 and 1 (PD0 and PD1) inputs, and turn on pull-up resistors

- Arduino approach

```
pinMode(0, INPUT);
pinMode(1, INPUT);
digitalWrite(0, HIGH);
digitalWrite(1, HIGH);
```

- Alternate approach

```
DDRD = 0; // all PORTD pins inputs
PORTD = 0b00000011;
or
PORTD = 0x03;

or better yet:
DDRD & = ~(1<<PD1 | 1<<PD0);
PORTD | = (1<<PD1 | 1<<PD0);
```

Structure of an Arduino Program

- An arduino program == '**sketch**'
  - Must have:
    - setup()
    - loop()
  - setup()
    - configures pin modes and registers
  - loop()
    - runs the main body of the program forever
      - like while(1) {…}
  - Where is main() ?
    - Arduino simplifies things
    - Does things for you

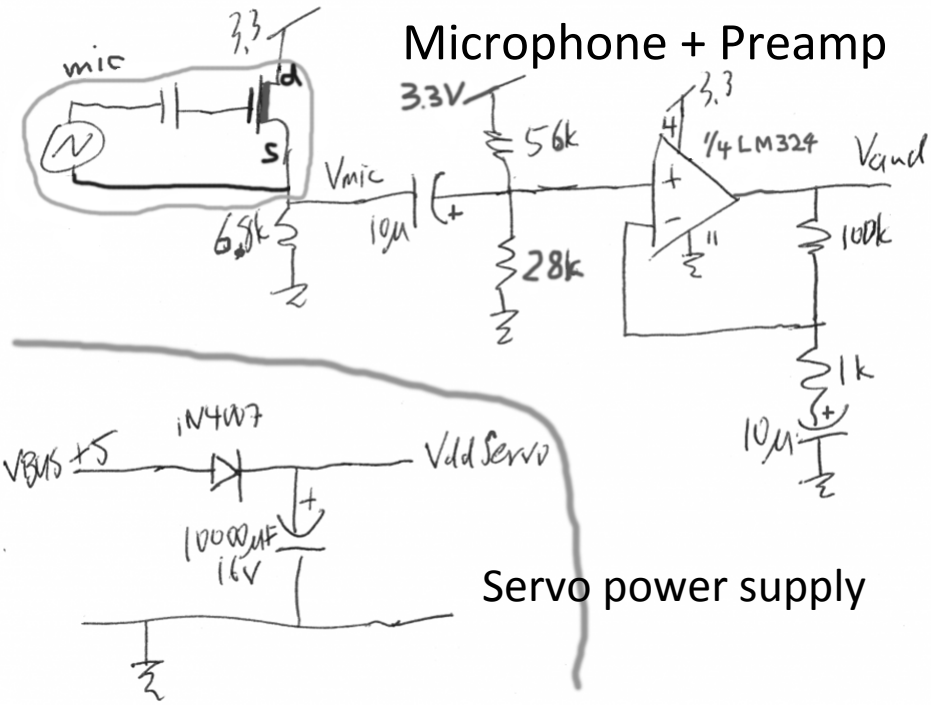
```
/* Blink  -  turns on an LED for DELAY_ON msec,
then off for DELAY_OFF msec, and repeats
BJ Furman rev. 1.1   Last rev: 22JAN2011
*/
#define LED_PIN  13    // LED on digital pin 13
#define DELAY_ON  1000
#define DELAY_OFF 1000

void setup()
{
  // initialize the digital pin as an output:
  pinMode(LED_PIN, OUTPUT);
}

// loop() method runs forever,
// as long as the Arduino has power

void loop()
{
  digitalWrite(LED_PIN, HIGH);    // set the LED on
  delay(DELAY_ON);  // wait for DELAY_ON msec
  digitalWrite(LED_PIN, LOW);   // set the LED off
  delay(DELAY_OFF);  // wait for DELAY_OFF msec
}
```

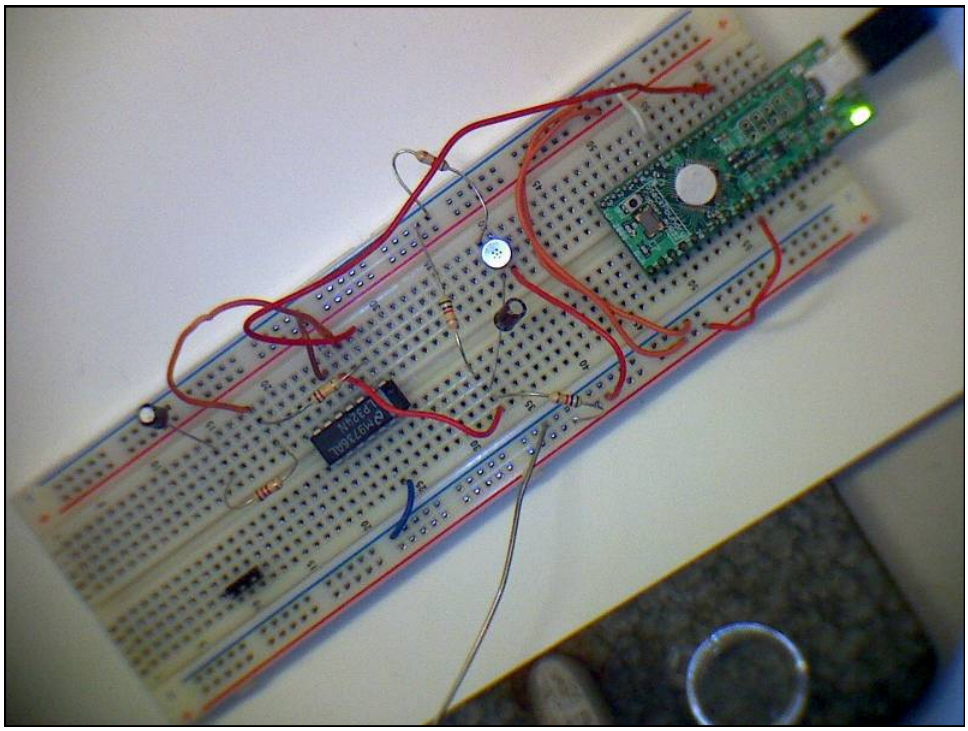
Example 3

```
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

What you should achieve today

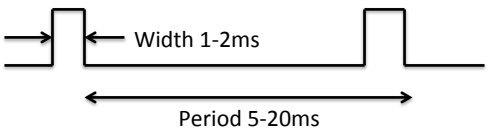
- Check and fix the **microphone preamplifier** and **servo power supply** from last week and verify that they work.
- Control a servo by writing and using a function **analogWrite(pin,pw)** that sets the PWM output pulse width to a duticycle value between 0(0%) to 255 (100%). See http://arduino.cc/en/Reference/AnalogWrite

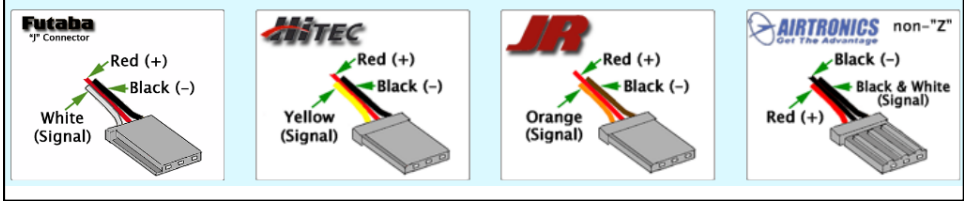




"RC" servos (Radio-Control Servo-Motors)

- Position controlled – Servo has internal position measurement and controller
- Rotation angle 120 degrees
- Pulse width from 1-2ms sets desired position
- Pulses must be sent at frequency 50-200Hz
- Pulse height >2V





ATmega328P timers

Figure 14-1. 8-bit Timer/Counter Block Diagram

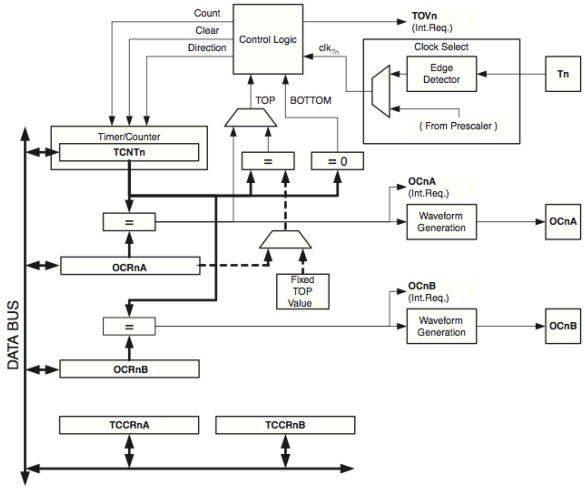
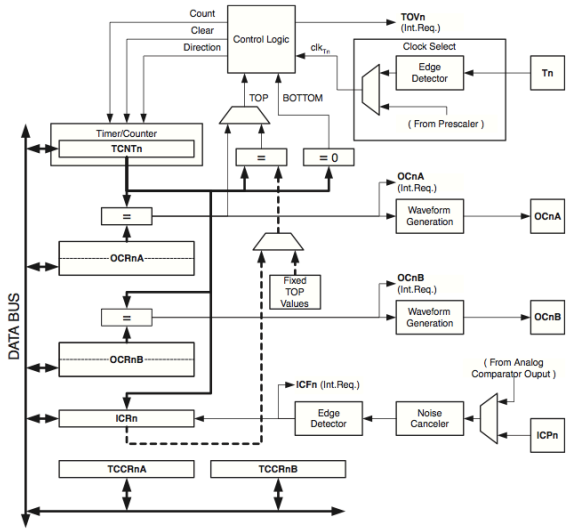


Figure 15-1. 16-bit Timer/Counter Block Diagram[(1)]



ATmega328P PWM implementation

We have two possibilities: 8-bit PWM or 16-bit PWM.
8-bit PWM:

- It uses an internal 8-bit timer/counter (timer0 or timer2)
- Time resolution is limited both in period and duticycle. For 16MHz clock, Tpwm = 490Hz (2.04ms) and steps in duticycle of 7.97us
- Easiest to program. It just need the use of analogWrite(pin,dc), where dc is the duticycle in the range of 0(0%) to 255(100%).
- Its resolution is not enough for our Servo application.

16-bit PWM:

- It uses the 16-bit internal timer/counter (timer1)
- Time resolution is bigger both for Tpwm and duticycle.
- Still easy to program, but not so much as 8-bit.
- Arduino provides a Servo library that adjust the period to 20 ms and you can control PWM pin location and duticycle.

Arduino Servo Library

- #include<Servo.h>
- Variables:

  Servo myservo; //Tpwm is 20ms
- Setup():

  myservo.attach(9,1000,2000);
- Loop():

  myservo.write(deg); //deg is char (0 to 180)
  myservo.writeMicroseconds(us); // us is int
  a=myservo.read(); //a is char from 0 to 180
  if (myservo.attached()) {myservo.deattach();}

Inside Servo library

```
class Servo
{
public:
  Servo();
  uint8_t attach(int pin);           // attach the given pin to the next free channel, sets pinMode, returns channel number or 0 if failure
  uint8_t attach(int pin, int min, int max); // as above but also sets min and max values for writes.
  void detach();
  void write(int value);             // if value is < 200 its treated as an angle, otherwise as pulse width in microseconds
  void writeMicroseconds(int value); // Write pulse width in microseconds
  int read();                        // returns current pulse width as an angle between 0 and 180 degrees
  int readMicroseconds();            // returns current pulse width in microseconds for this servo (was read_us() in first release)
  bool attached();                   // return true if this servo is attached, otherwise false
private:
   uint8_t servoIndex;               // index into the channel data for this servo
   int8_t min;                       // minimum is this value times 4 added to MIN_PULSE_WIDTH
   int8_t max;                       // maximum is this value times 4 added to MAX_PULSE_WIDTH
};
```

```
Servo::Servo()
{
  if( ServoCount < MAX_SERVOS) {
    this->servoIndex = ServoCount++;                    // assign a servo index to th
    servos[this->servoIndex].ticks = usToTicks(DEFAULT_PULSE_WIDTH);   // store defau
  }
  else
    this->servoIndex = INVALID_SERVO ;  // too many servos
}

uint8_t Servo::attach(int pin)
{
  return this->attach(pin, MIN_PULSE_WIDTH, MAX_PULSE_WIDTH);
}

uint8_t Servo::attach(int pin, int min, int max)
{
  if(this->servoIndex < MAX_SERVOS ) {
    pinMode( pin, OUTPUT) ;                                  // set servo pin to out
    servos[this->servoIndex].Pin.nbr = pin;
    // todo min/max check: abs(min - MIN_PULSE_WIDTH) /4 < 128
    this->min  = (MIN_PULSE_WIDTH - min)/4; //resolution of min/max is 4 uS
    this->max  = (MAX_PULSE_WIDTH - max)/4;
    // initialize the timer if it has not already been initialized
    timer16_Sequence_t timer = SERVO_INDEX_TO_TIMER(servoIndex);
    if(isTimerActive(timer) == false)
      initISR(timer);
    servos[this->servoIndex].Pin.isActive = true;  // this must be set after the chec
  }
  return this->servoIndex ;
}
```

```
void Servo::write(int value)
{
  if(value < MIN_PULSE_WIDTH)
  {  // treat values less than 544 as angles in degrees (valid
    if(value < 0) value = 0;
    if(value > 180) value = 180;
    value = map(value, 0, 180, SERVO_MIN(),  SERVO_MAX());
  }
  this->writeMicroseconds(value);
}
```

```
void Servo::writeMicroseconds(int value)
{
  // calculate and store the values for the given
  byte channel = this->servoIndex;
  if( (channel >= 0) && (channel < MAX_SERVOS) )
  {
    if( value < SERVO_MIN() )          // ensure
      value = SERVO_MIN();
    else if( value > SERVO_MAX() )
      value = SERVO_MAX();

    value = value - TRIM_DURATION;
    value = usToTicks(value);  // convert to tick

    uint8_t oldSREG = SREG;
    cli();
    servos[channel].ticks = value;
    SREG = oldSREG;
  }
}
```