# Embedded Systems Design: A Unified Hardware/Software Introduction

## Chapter 4 Standard Single Purpose Processors: Peripherals

# Introduction

- Single-purpose processors
  - Performs specific computation task
  - Custom single-purpose processors
    - Designed by us for a unique task
  - *Standard* single-purpose processors
    - "Off-the-shelf" -- pre-designed for a common task
    - a.k.a., peripherals
    - serial transmission
    - analog/digital conversions
- In this chapter we cover several standard peripherals: Timer/Counters, UART for serial transmission, LCD controller, Keypad controller, Stepper motor controller

# Generating Delay Time

- Timing loops can be used to generate time delays:

*Example of delay function that can be used to program a general purpose processor:*

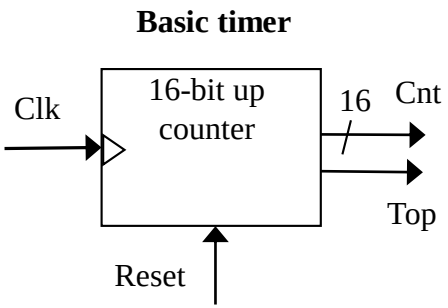
```
void MSDelay (unsigned int itime)
  {
        unsigned int i, j;
        for (i = 0; i < itime; i++)
            for (j = 0; j< 1275; j++);
}
```

*We know the number of clock cycles that each instruction requires, we can calculate the delay time. This is an inefficient method, delay function occupies much of program's run time, leaving little time for other computation.*

- Other option is to use Timers

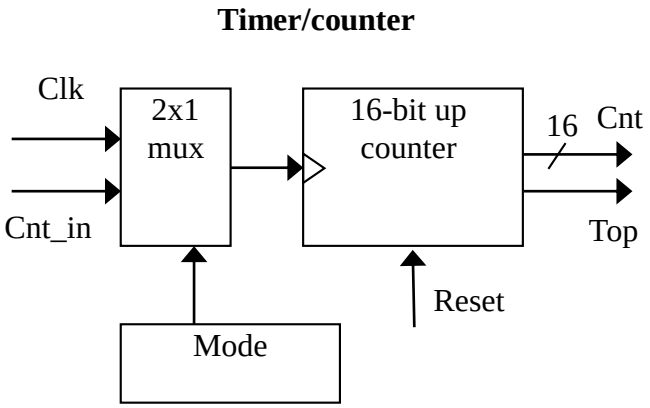
# Timers, counters, watchdog timers

- Timer: measures time intervals
  - To generate timed output events
    - e.g., hold traffic light green for 10 s
  - To measure input events
    - e.g., measure a car's speed
- Based on counting clock pulses
    - E.g., let Clk period be 10 ns
    - And we count 20,000 Clk pulses
    - Then 200 microseconds have passed
    - 16-bit counter would count up to 65,535*10 ns = 655.35 microsec., resolution = 10 ns
    - Top: indicates top count reached, wrap-around



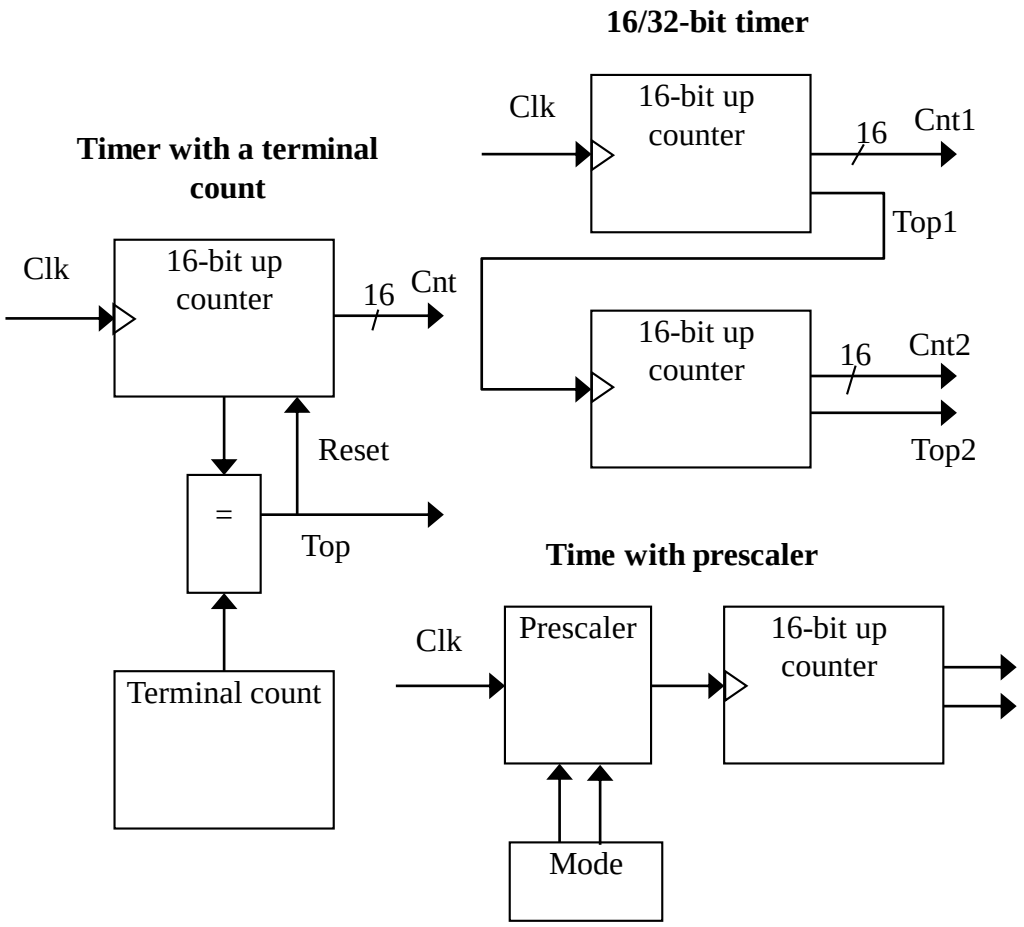
# Counters

- Counter: like a timer, but counts pulses on a general input signal rather than clock
  - e.g., count cars passing over a sensor
  - Can often configure device as either a timer or counter

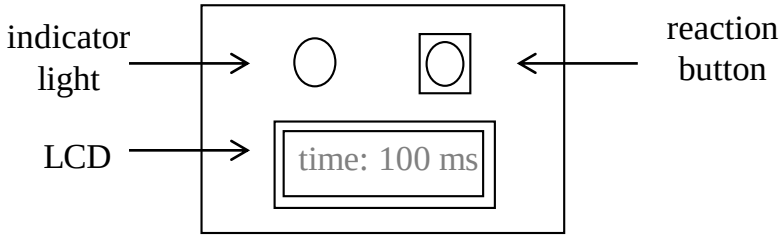


# Other timer structures

- Interval timer
  - Indicates when desired time interval has passed
  - We set terminal count to desired interval
    - *Number of clock cycles = Desired time interval / Clock period*
- Cascaded counters
- Prescaler
  - Divides clock
  - Increases range, decreases resolution



# Example: Reaction Timer

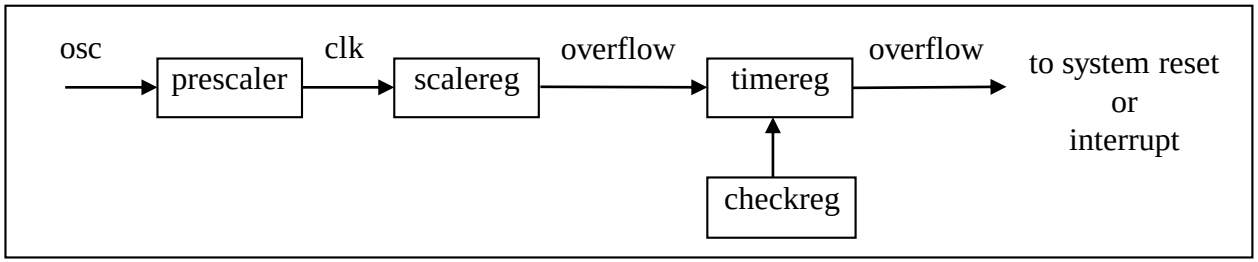



- Measure time between turning light on and user pushing button
  - 16-bit timer, clk period is 83.33 ns, counter increments every 6 cycles
  - Resolution = 6*83.33=0.5 microsec.
  - Range = 65535*0.5 microseconds = 32.77 milliseconds
  - Want program to count millisec., so initialize counter to 65535 – 1000/0.5 = 63535

```
/* main.c */

#define MS_INIT       63535
void main(void){
   int count_milliseconds = 0;

   configure timer mode
   set Cnt to MS_INIT

   wait a random amount of time
   turn on indicator light
   start timer

while (user has not pushed reaction button){
   if(Top) {
      stop timer
      set Cnt to MS_INIT
      start timer
      reset Top
      count_milliseconds++;
   }
}
turn light off
printf("time:  %i ms", count_milliseconds);
}
```

# Watchdog timer

- Must reset timer every X time unit, else timer generates a signal
- Common use: detect failure, self-reset
- Another use: timeouts
  - e.g., ATM machine
  - 16-bit timer, 2 microsec. resolution
  - *timereg* value = $2*(2^{16}-1)–X = 131070–X$
  - For 2 min., X = 120,000 microsec.




```
/* main.c */

main(){
  wait until card inserted
  call watchdog_reset_routine

  while(transaction in progress){
    if(button pressed){
      perform corresponding action
      call watchdog_reset_routine
    }

/* if watchdog_reset_routine not called every
< 2 minutes, interrupt_service_routine is
called */
}
```

```
watchdog_reset_routine(){
/* checkreg is set so we can load value into
timereg.  Zero is loaded into scalereg and
11070 is loaded into timereg */

  checkreg = 1
  scalereg = 0
  timereg  = 11070
}

void interrupt_service_routine(){
  eject card
  reset screen
}
```

# Arduino Uno Timers

- ATmega328 has two timers:

| | |
|---|---|
| Timers | 2 x 8-bit, 1 x 16-bit |

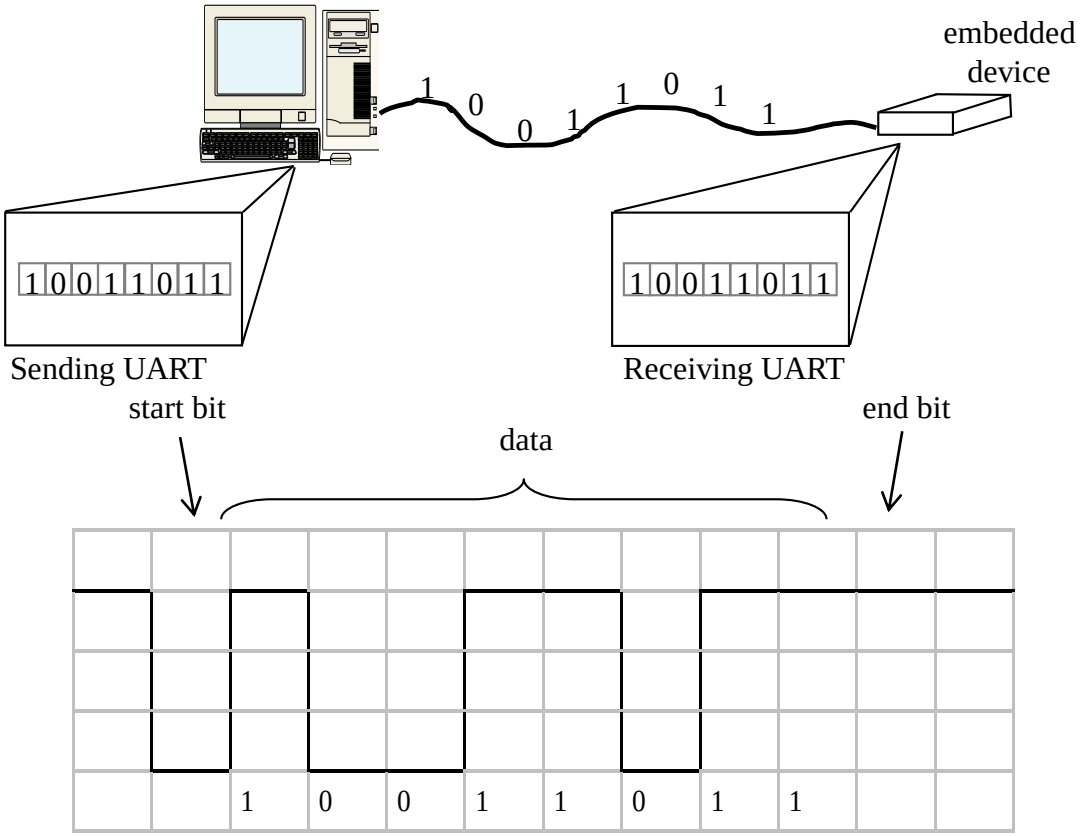
- As an Arduino programmer you will use timers and interrupts without detailed knowledge, because all the low level hardware is hidden by the Arduino API.
- Many Arduino functions use timers, for example the time functions: delay(), millis() and micros() and delayMicroseconds(). The PWM function analogWrite() uses timers, as do the tone() and the noTone() functions.

# Arduino Uno Timers

- **Timer0: 8bit timer.**
  In the Arduino world timer0 is been used for the software Sketch timer functions, like __delay()__, __millis()__ and __micros()__. If you change timer0 registers, this may influence the Arduino timer function. So you should know what you are doing.
- **Timer1: 16bit timer.**
  In the Arduino world the __Servo library__ uses timer1 on Arduino Uno.

# Serial Transmission Using UARTs

- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
- Parity: extra bit for simple error checking
- Start bit, stop bit
- Baud rate
  - signal changes per second
  - bit rate usually higher



# Asynchronous Serial Communication

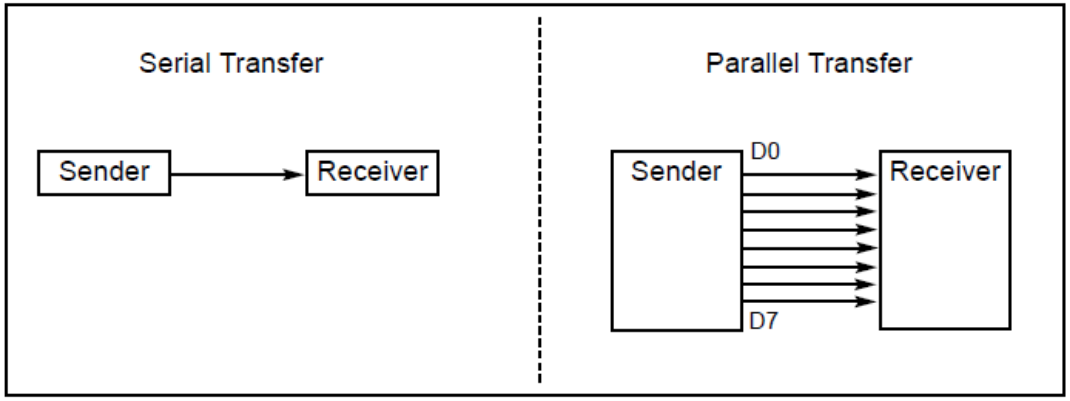



**Figure 10-1. Serial versus Parallel Data Transfer**

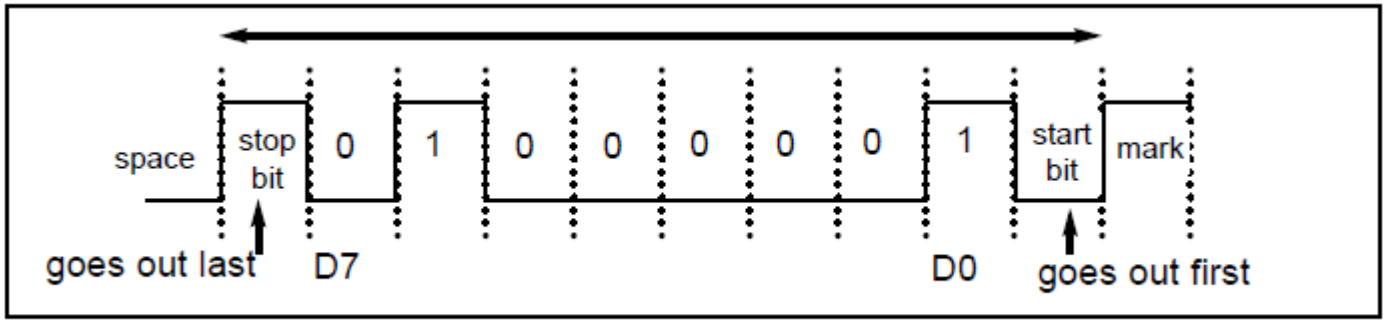
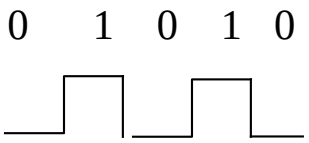



**Figure 10-3. Framing ASCII "A" (41H)**

# Baud rate and Bit rate

In case of Binary encoding:

Baud = Bit rate

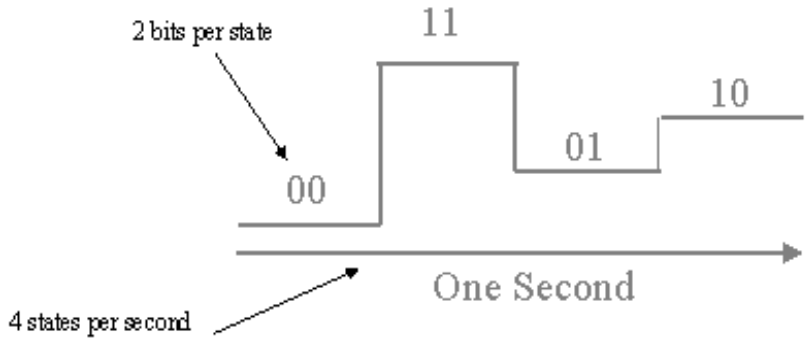



Otherwise:

Baud $\neq$ Bit rate

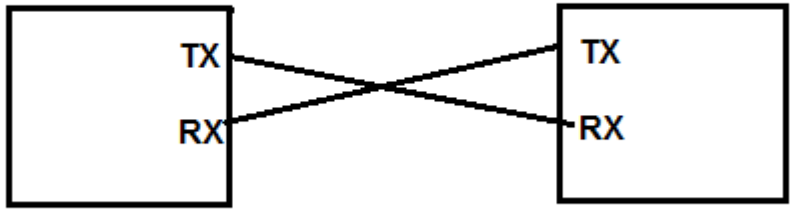
## Baud Rate and Bit Rate

- **Baud Rate**
  - Number of times line changes per second
  - Let baud rate be 4 (4 changes per second)
  - Let bits per line change be 2
  - Therefore the Bit rate = 8 bits per second
  - Bit rate = x2 Baud rate in this example



# Arduino Serial Port

- Arduino Uno has one UART and as a result one serial port: It communicates on digital pin 0 (RX) and digital pin1 (TX) UART as well as with the computer via USB.
- UART is a full-duplex protocol




  - Baud rate is programmable

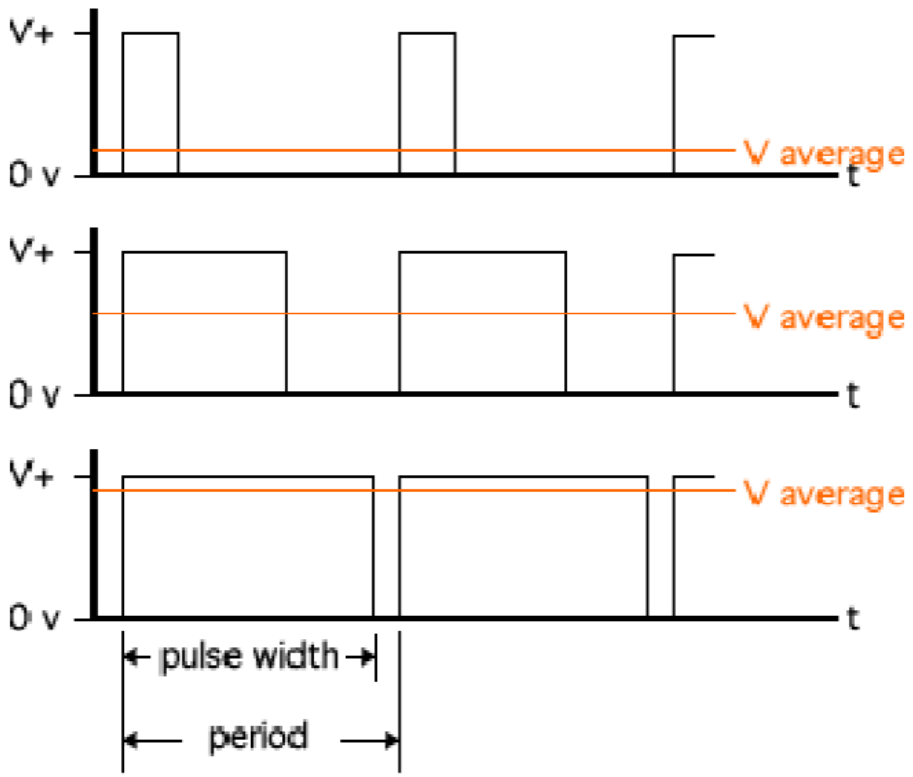
# Arduino Serial Port

Used for communication between the Arduino board and a computer or other devices. All Arduino boards have at least one serial port (also known as a UART or USART): Serial.

- It communicates on digital pins 0 (RX) and 1 (TX) as well as with the computer via USB.
- Thus, if you use these functions, you cannot also use pins 0 and 1 for digital input or output.
- You can use the Arduino environment's built-in serial monitor to communicate with an Arduino board. Click the serial monitor button in the toolbar and select the same baud rate used in the call to begin().
- Serial communication on pins TX/RX uses TTL logic levels (5V or 3.3V depending on the board).

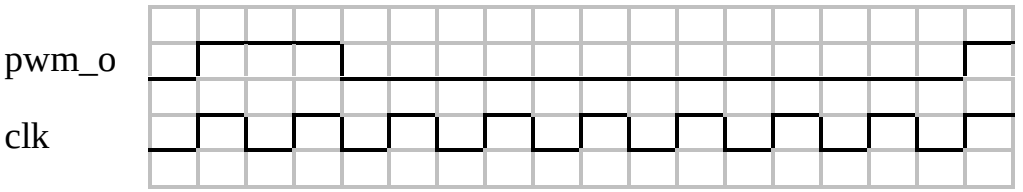
# Pulse Width Modulation (PWM)

PWM is a way to provide a variable signal from a given set signal. PWM does this by changing the pulse width, which in turn, changes the duty cycle of a square wave to alter how much power is supplied to the attached component. It varies because the signal takes the duty cycle and averages the power signal that is output.

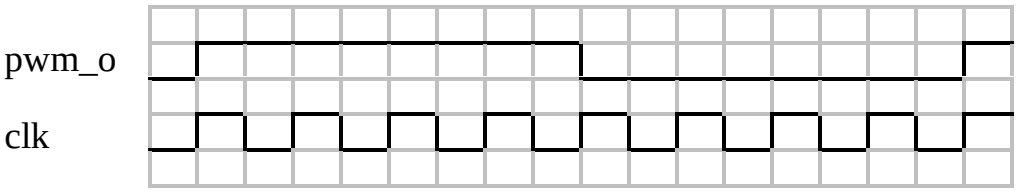


# Pulse width modulator

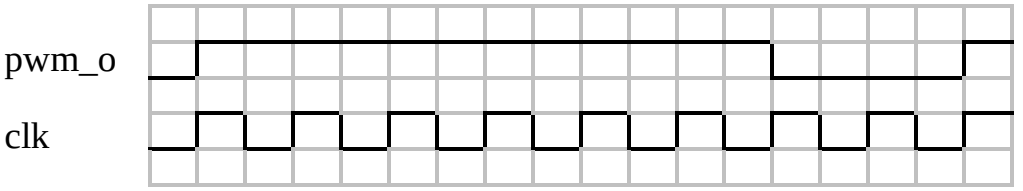
- Generates pulses with specific high/low times
- Duty cycle: % time high
  - Square wave: 50% duty cycle
- Common use: control average voltage to electric device
  - Simpler than DC-DC converter or digital-analog converter
  - DC motor speed, dimmer lights
- Another use: encode commands, receiver uses timer to decode




25% duty cycle – average pwm_o is 1.25V




50% duty cycle – average pwm_o is 2.5V.




75% duty cycle – average pwm_o is 3.75V.

# PWM on Arduino Uno

- PWM function works on pins 3, 5, 6, 9, 10, and 11 on Arduino Uno.
- After a call to analogWrite(), the pin will generate a steady square wave of the specified duty cycle.

Example Code

Sets the output to the LED proportional to the value read from the potentiometer.

```
int ledPin = 9;      // LED connected to digital pin 9
int analogPin = 3;   // potentiometer connected to analog pin 3
int val = 0;         // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);   // sets the pin as output
}

void loop()
{
  val = analogRead(analogPin);   // read the input pin
  analogWrite(ledPin, val / 4);  // analogRead values go from 0 to 1023, analogWrite values from 0 to 255
}
```

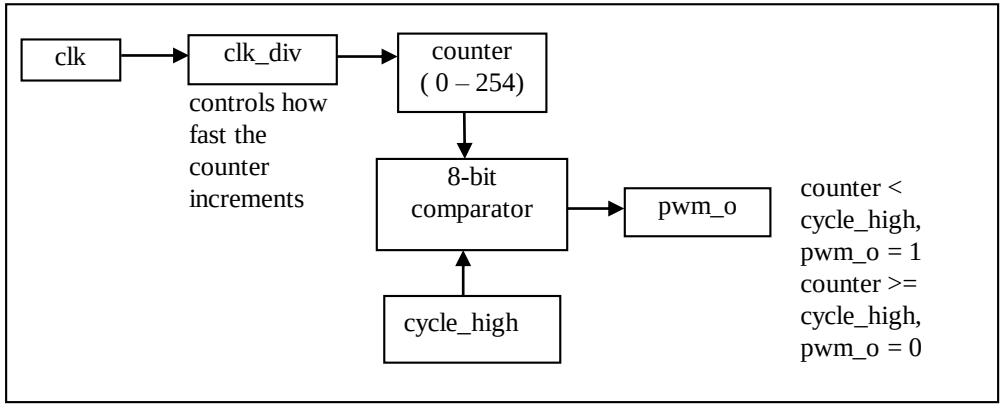
# Arduino PWM Duty Cycle

- AnalogWrite function needs pin number and a value to set the duty cycle.
- The value that gets input for this function has to be between 0 and 255.
- PWM duty cycle is calculated using:

$$duty\ cycle * 255 = analogWrite\ value$$

- For example, a duty cycle of 20% would need to be set at a value of 51.

# Controlling a DC motor with a PWM



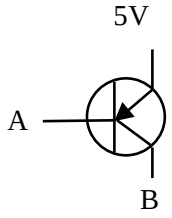

Internal Structure of PWM

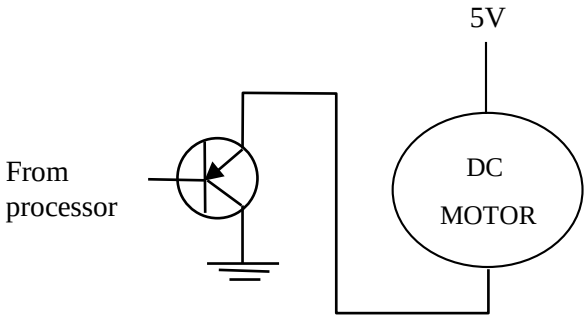
| Input Voltage | % of Maximum Voltage Applied | RPM of DC Motor |
|---|---|---|
| 0 | 0 | 0 |
| 2.5 | 50 | 1840 |
| 3.75 | 75 | 6900 |
| 5.0 | 100 | 9200 |

Relationship between applied voltage and speed of the DC Motor

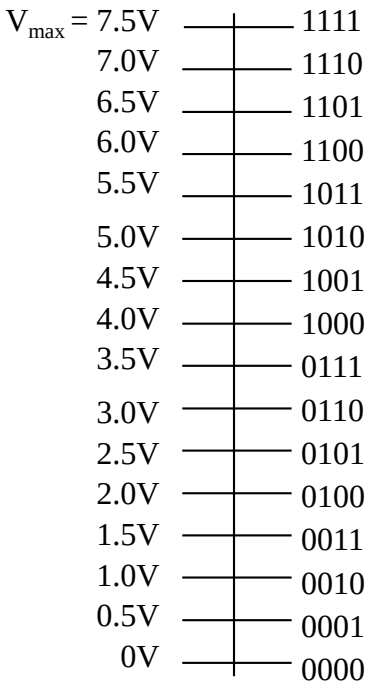
```
void main(void){

   /* controls period */
   PWMP = 0xff;
   /* controls duty cycle */
   PWM1 = 0x7f;

   while(1){};
 }
```

The PWM alone cannot drive the DC motor, a possible way to implement a driver is shown below using an MJE3055T NPN transistor.

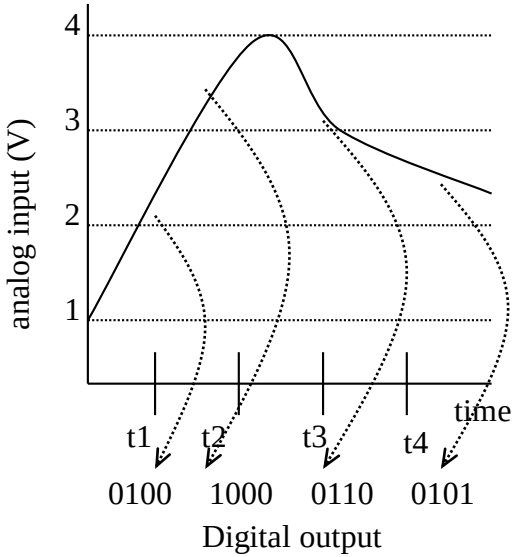


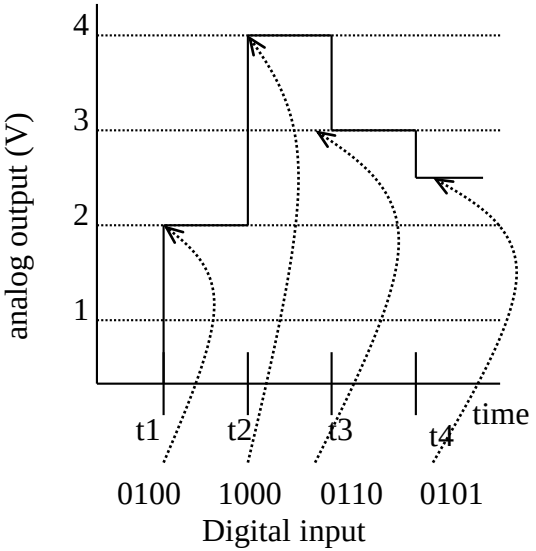
# Analog-to-digital converters




**proportionality**




**analog to digital**




**digital to analog**

# Analog-to-digital converters

- Ratio equation

  $e/V_{max} = d/(2^n - 1)$

  e: sampled voltage

  $V_{max}$: analog signal maximum voltage

  d: decimal equivalent of encoded voltage

  n: number bits available for digital encoding

- Resolution

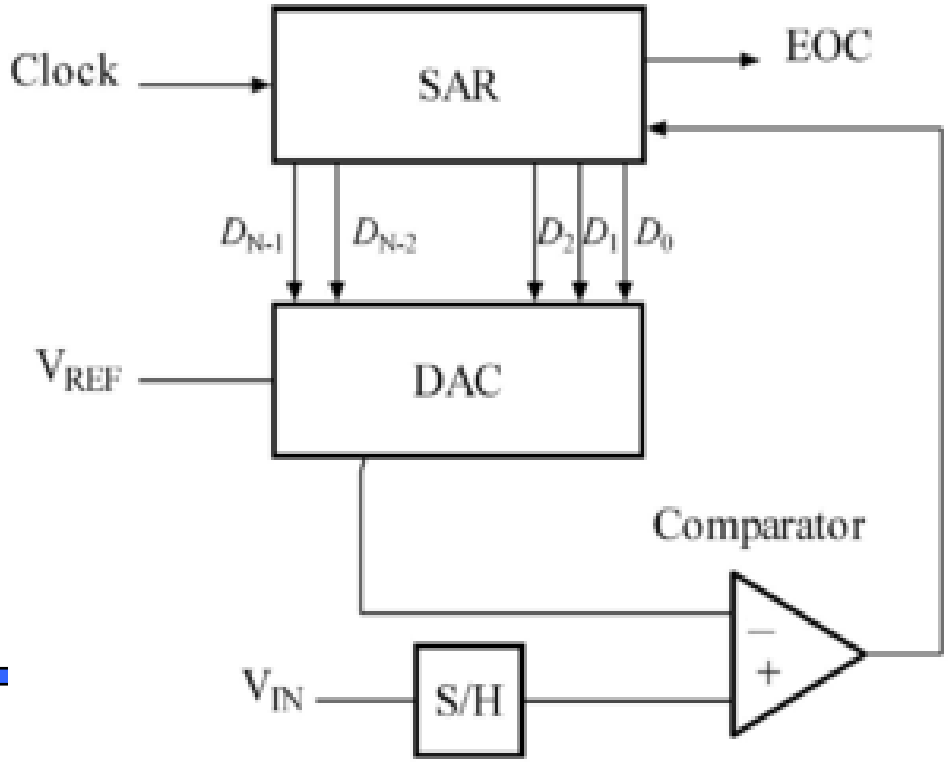
  Resolution = $V_{max} / (2^n - 1)$

# Digital-to-analog conversion using successive approximation

Successive approximation ADC converts analog waveform into digital representation via a binary search through all possible quantization levels before finally converging upon

a digital output for

each conversion



# Digital-to-analog conversion using successive approximation

Given an analog input signal whose voltage should range from 0 to 15 volts, and an 8-bit digital encoding, calculate the correct encoding for 5 volts. Then trace the successive-approximation approach to find the correct encoding.

5/15 = d/(28-1)
d= 85

Encoding: 01010101

***Successive-approximation method***

| | | |
|---|---|---|
| ½($V_{max}$ – $V_{min}$) = 7.5 volts<br>$V_{max}$ = 7.5 volts. | 0 0 0 0 0 0 0 0 | |
| ½(7.5 + 0) = 3.75 volts<br>$V_{min}$ = 3.75 volts. | 0 1 0 0 0 0 0 0 | |
| ½(7.5 + 3.75) = 5.63 volts<br>$V_{max}$ = 5.63 volts | 0 1 0 0 0 0 0 0 | |
| ½(5.63 + 3.75) = 4.69 volts<br>$V_{min}$ = 4.69 volts. | 0 1 0 1 0 0 0 0 | |
| ½(5.63 + 4.69) = 5.16 volts<br>$V_{max}$ = 5.16 volts. | | 0 1 0 1 0 0 0 0 |
| ½(5.16 + 4.69) = 4.93 volts<br>$V_{min}$ = 4.93 volts. | | 0 1 0 1 0 1 0 0 |
| ½(5.16 + 4.93) = 5.05 volts<br>$V_{max}$ = 5.05 volts. | | 0 1 0 1 0 1 0 0 |
| ½(5.05 + 4.93) = 4.99 volts | | 0 1 0 1 0 1 0 1 |

# Analog to Digital Conversion in ARDUINO

- Arduino has a 10-bit ADC on board.
- The analog reading in the analog inputs are converted into corresponding 10bit(0-1023).
- Analog to digital conversion module of ARDUINO UNO has 6 input ports.

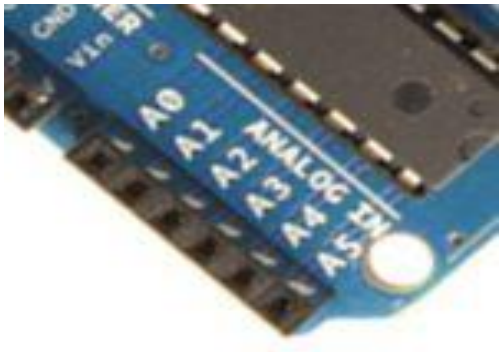
$$d = (2^n - 1) * e/V_{max}$$

e: sampled voltage

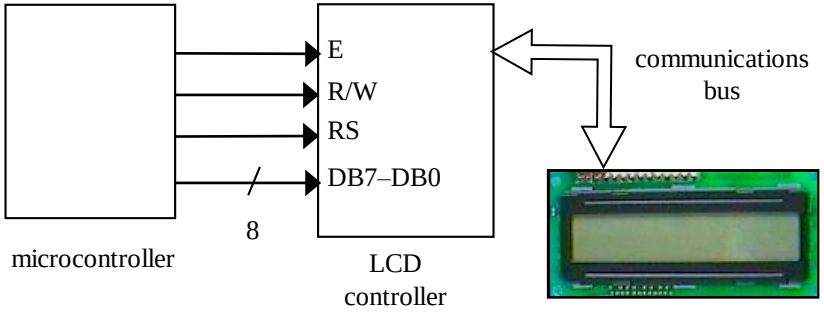
$V_{max}$: analog signal maximum voltage

d: decimal equivalent of encoded voltage

n: number bits available for digital encoding



# LCD controller

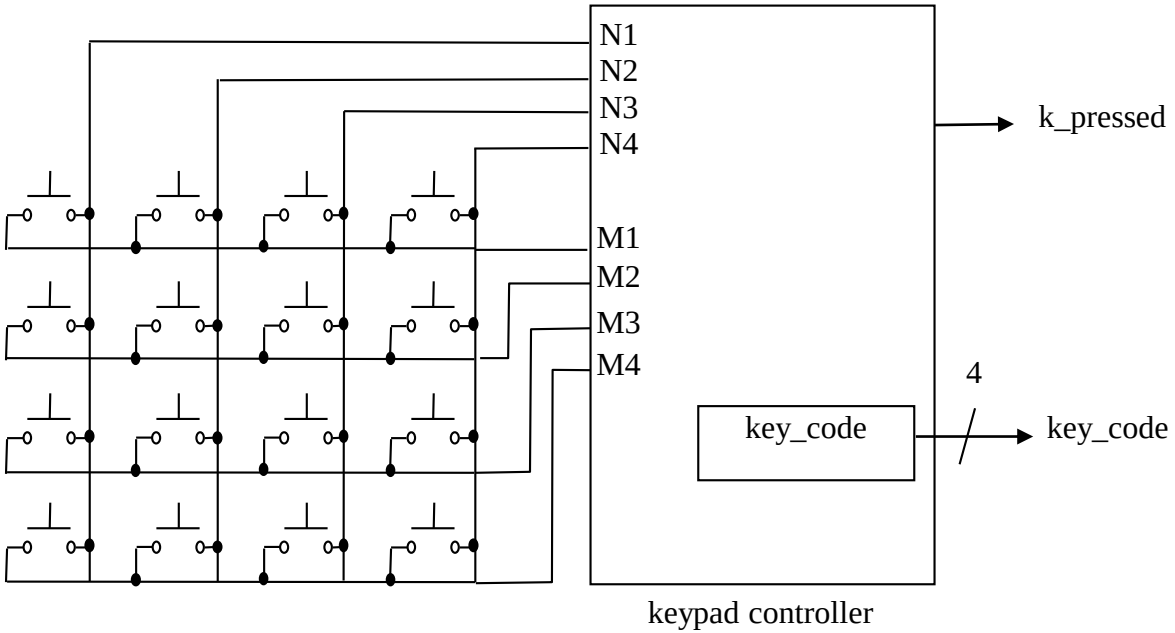



```
void WriteChar(char c){

  RS = 1;                 /* indicate data being sent */
  DATA_BUS = c;           /* send data to LCD */
  EnableLCD(45);          /* toggle the LCD with appropriate delay */
}
```

| CODES | |
|---|---|
| I/D = 1 cursor moves left | DL = 1 8-bit |
| I/D = 0 cursor moves right | DL = 0 4-bit |
| S = 1 with display shift | N = 1 2 rows |
| S/C =1 display shift | N = 0 1 row |
| S/C = 0 cursor movement | F = 1 5x10 dots |
| R/L = 1 shift to right | F = 0 5x7 dots |
| R/L = 0 shift to left | |

| RS | R/W | $DB_7$ | $DB_6$ | $DB_5$ | $DB_4$ | $DB_3$ | $DB_2$ | $DB_1$ | $DB_0$ | Description |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Clears all display, return cursor home |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | * | Returns cursor home |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | I/D | S | Sets cursor move direction and/or specifies not to shift display |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | ON/OFF of all display(D), cursor ON/OFF (C), and blink position (B) |
| 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | * | * | Move cursor and shifts display |
| 0 | 0 | 0 | 0 | 1 | DL | N | F | * | * | Sets interface data length, number of display lines, and character font |
| 1 | 0 | WRITE DATA | | | | | | | | Writes Data |

# Keypad controller




N=4, M=4

# Stepper Motor

Stepper motor translates electrical pulses to mechanical movement.

Used for position control in robotics disk drives, dot matrix printer.

This is a four phase stepper motor

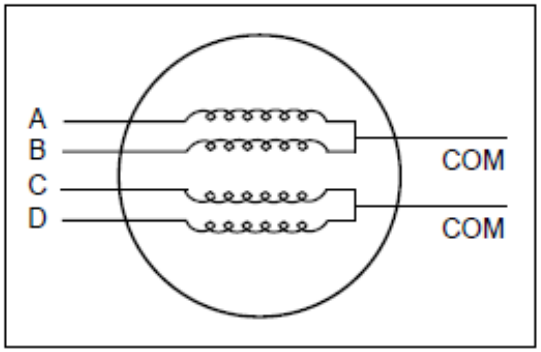



Figure 15-8. Stator Windings Configuration

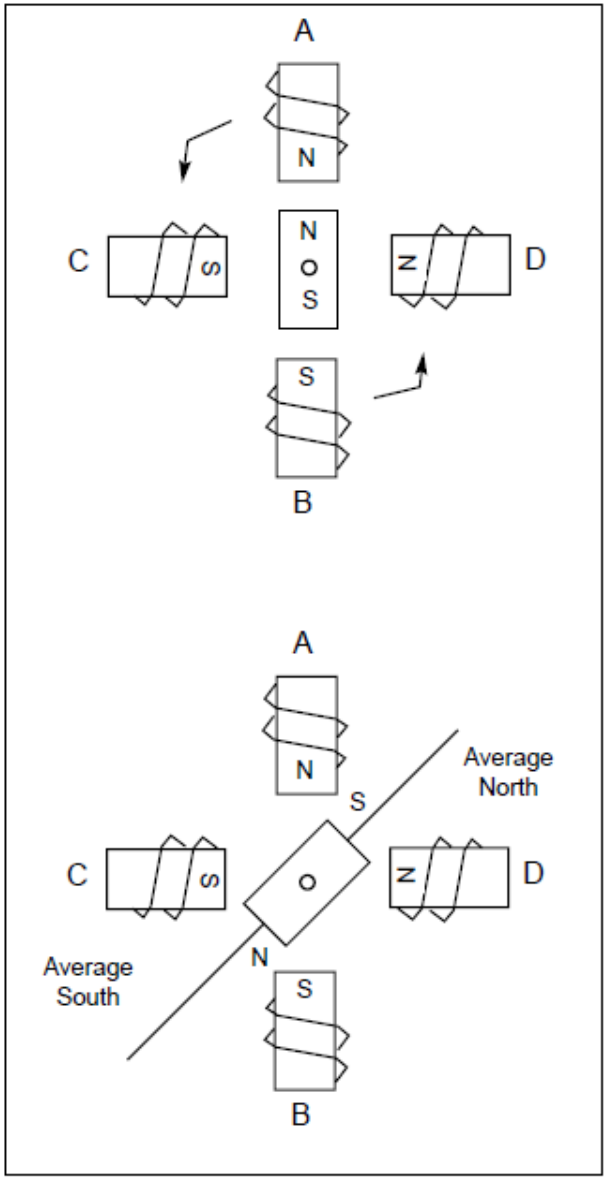



Figure 15-7. Rotor Alignment

# Common Stepper Motor Types

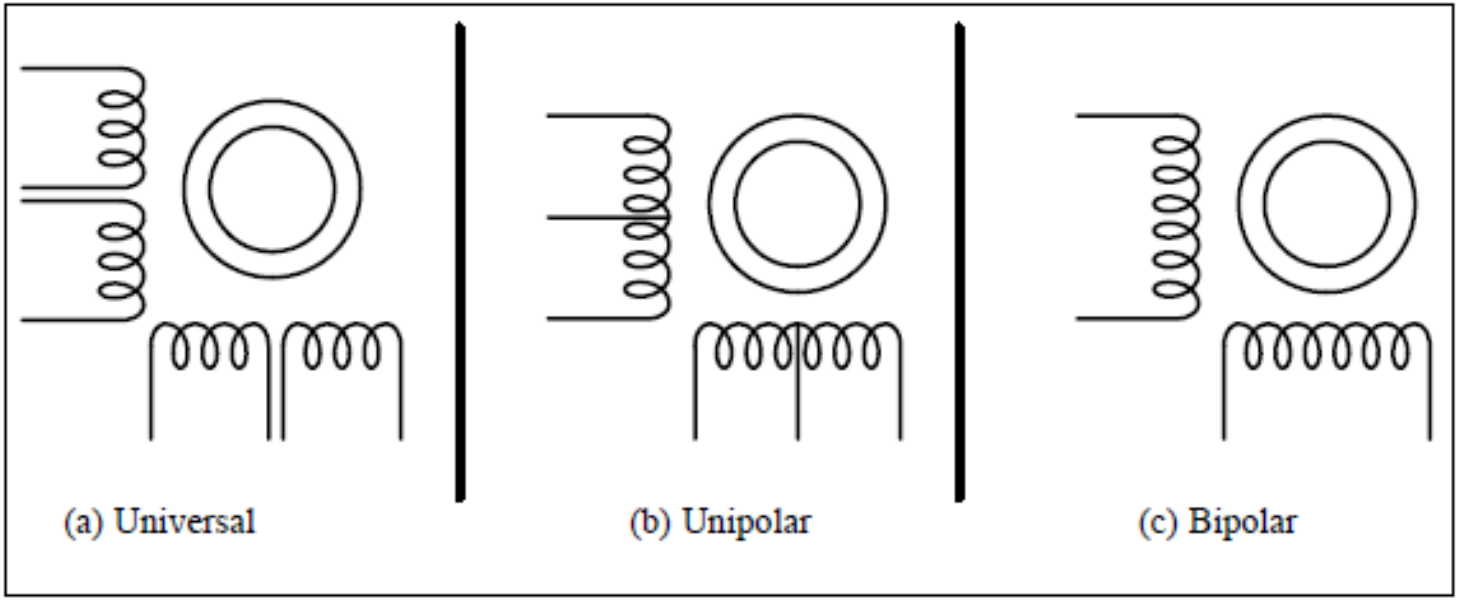
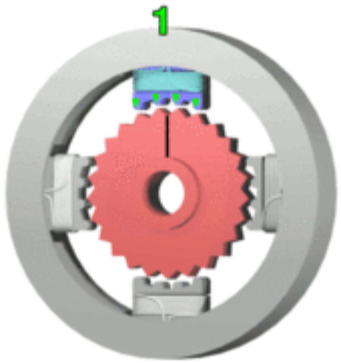
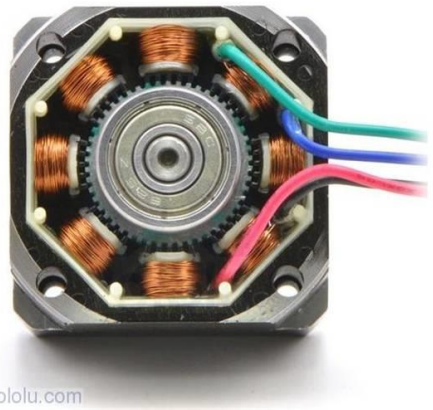



**Figure 15-10. Common Stepper Motor Types**

# Animation of a Unipolar Stepper Motor

**Frame 1:** The top electromagnet (1) is turned on, attracting the nearest teeth of the gear-shaped iron rotor. With the teeth aligned to electromagnet 1, they will be slightly offset from right electromagnet (2).
**Frame 2:** The top electromagnet (1) is turned off, and the right electromagnet (2) is energized, pulling the teeth into alignment with it. This results in a rotation of 3.6° in this example.
**Frame 3:** The bottom electromagnet (3) is energized; another 3.6° rotation occurs.
**Frame 4:** The left electromagnet (4) is energized, rotating again by 3.6°. When the top electromagnet (1) is again enabled, the rotor will have rotated by one tooth position; since there are 25 teeth, it will take 100 steps to make a full rotation in this example.



# Stepper Motor Step Angle

How much movement with a single step, depends on the number of teeth on the stator and rotor

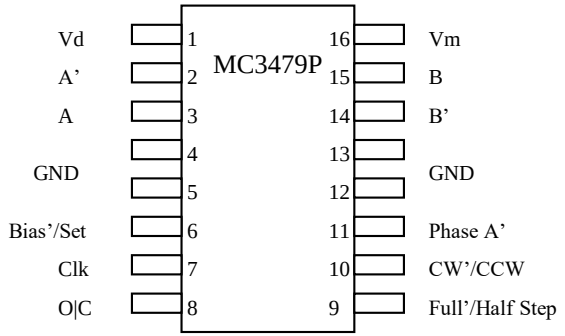
**Table 15-4: Stepper Motor Step Angles**

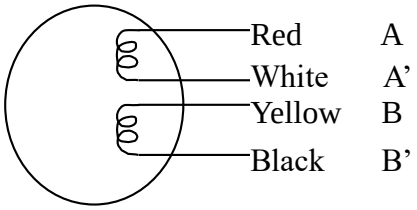
| Step Angle | Steps per Revolution |
|---|---|
| 0.72 | 500 |
| 1.8 | 200 |
| 2.0 | 180 |
| 2.5 | 144 |
| 5.0 | 72 |
| 7.5 | 48 |
| 15 | 24 |

# Stepper motor controller

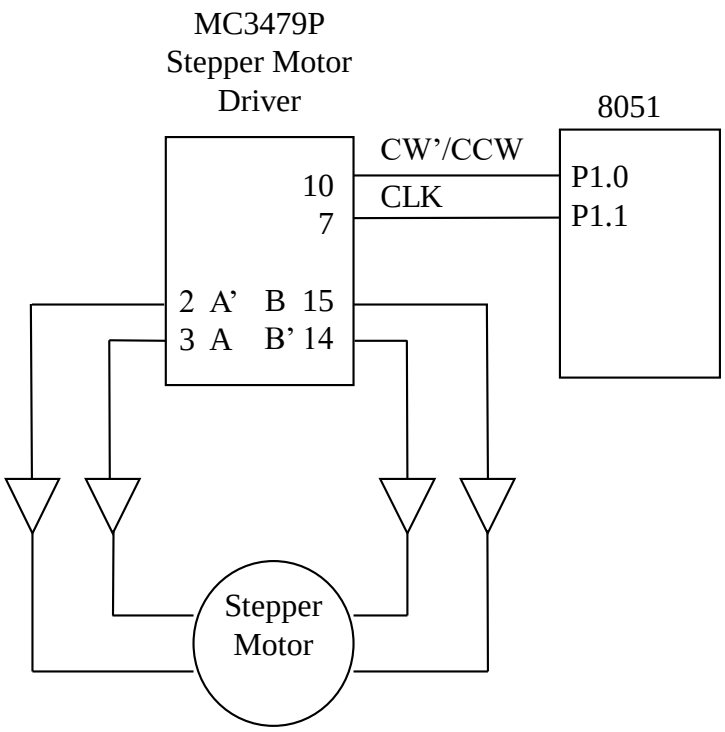
- Stepper motor: rotates fixed number of degrees when given a “step” signal
  - In contrast, DC motor just rotates when power applied, coasts to stop
- Rotation achieved by applying specific voltage sequence to coils
- Controller greatly simplifies this

| Sequence | A | B | A’ | B’ |
|---|---|---|---|---|
| 1 | + | + | - | - |
| 2 | - | + | + | - |
| 3 | - | - | + | + |
| 4 | + | - | - | + |
| 5 | + | + | - | - |



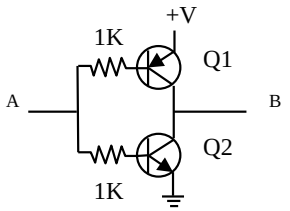
# Stepper motor with controller (driver)



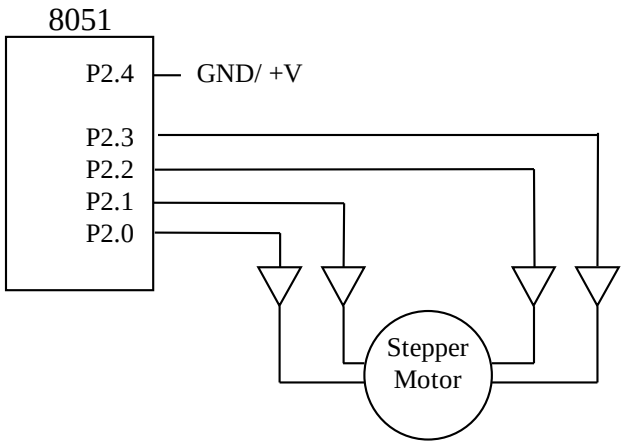

```
/* main.c */

sbit clk=P1^1;
sbit cw=P1^0;

void delay(void){
  int i, j;
  for (i=0; i<1000; i++)
    for ( j=0; j<50; j++)
      i = i + 0;
}
```

```
void main(void){

  */turn the motor forward */
  cw=0;            /* set direction */
  clk=0;           /* pulse clock */
  delay();
  clk=1;

  /*turn the motor backwards */
  cw=1;            /* set direction */
  clk=0;           /* pulse clock */
  delay();
  clk=1;

}
```

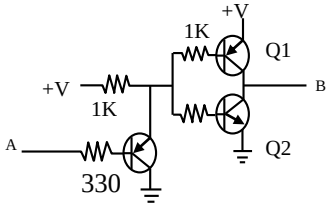
The output pins on the stepper motor driver do not provide enough current to drive the stepper motor. To amplify the current, a buffer is needed. One possible implementation of the buffers is pictured to the left. Q1 is an MJE3055T NPN transistor and Q2 is an MJE2955T PNP transistor. A is connected to the 8051 microcontroller and B is connected to the stepper motor.



# Stepper motor without controller (driver)




A possible way to implement the buffers is located below. The 8051 alone cannot drive the stepper motor, so several transistors were added to increase the current going to the stepper motor. Q1 are MJE3055T NPN transistors and Q3 is an MJE2955T PNP transistor. A is connected to the 8051 microcontroller and B is connected to the stepper motor.




```
/*main.c*/
sbit notA=P2^0;
sbit isA=P2^1;
sbit notB=P2^2;
sbit isB=P2^3;
sbit dir=P2^4;

void delay(){
  int a, b;
  for(a=0; a<5000; a++)
    for(b=0; b<10000; b++)
      a=a+0;
}

void move(int dir, int steps) {
int y, z;
  /* clockwise movement */
  if(dir == 1){
    for(y=0; y<=steps; y++){
      for(z=0; z<=19; z+4){
        isA=lookup[z];
        isB=lookup[z+1];
        notA=lookup[z+2];
        notB=lookup[z+3];
        delay();
      }
    }
  }
```

```
/* counter clockwise movement */
  if(dir==0){
    for(y=0;  y<=step; y++){
      for(z=19; z>=0; z - 4){
        isA=lookup[z];
        isB=lookup[z-1];
        notA=lookup[z -2];
        notB=lookup[z-3];
        delay( );
      }
    }
  }
}
void main( ){
  int z;
  int lookup[20] = {
    1,  1,  0,  0,
    0,  1,  1,  0,
    0,  0,  1,  1,
    1,  0,  0,  1,
    1,  1,  0,  0  };
  while(1){
     /*move forward, 15 degrees (2 steps) */
    move(1, 2);
    /* move backwards,  7.5 degrees (1step)*/
    move(0, 1);
  }
}
```

# Summary

- Numerous single-purpose processors are used to fulfill a specific function in embedded systems.
- A timer informs us when a particular interval of time has passed.
- A counter informs us when a particular number of pulses have occurred on a signal.
- UART converts parallel data to serial data and vice versa..
- A PWM generates pulses on an output signal with specific high and low times.
- LCD controller simplifies writing characters to an LCD.
- Keypad controller simplifies capture and decoding of a button press.
- A stepper-motor controller simplifies the process of rotating the stepper motor a fixed amount forward or backward.
- ADC peripherals are used to convert analog signals to digital.