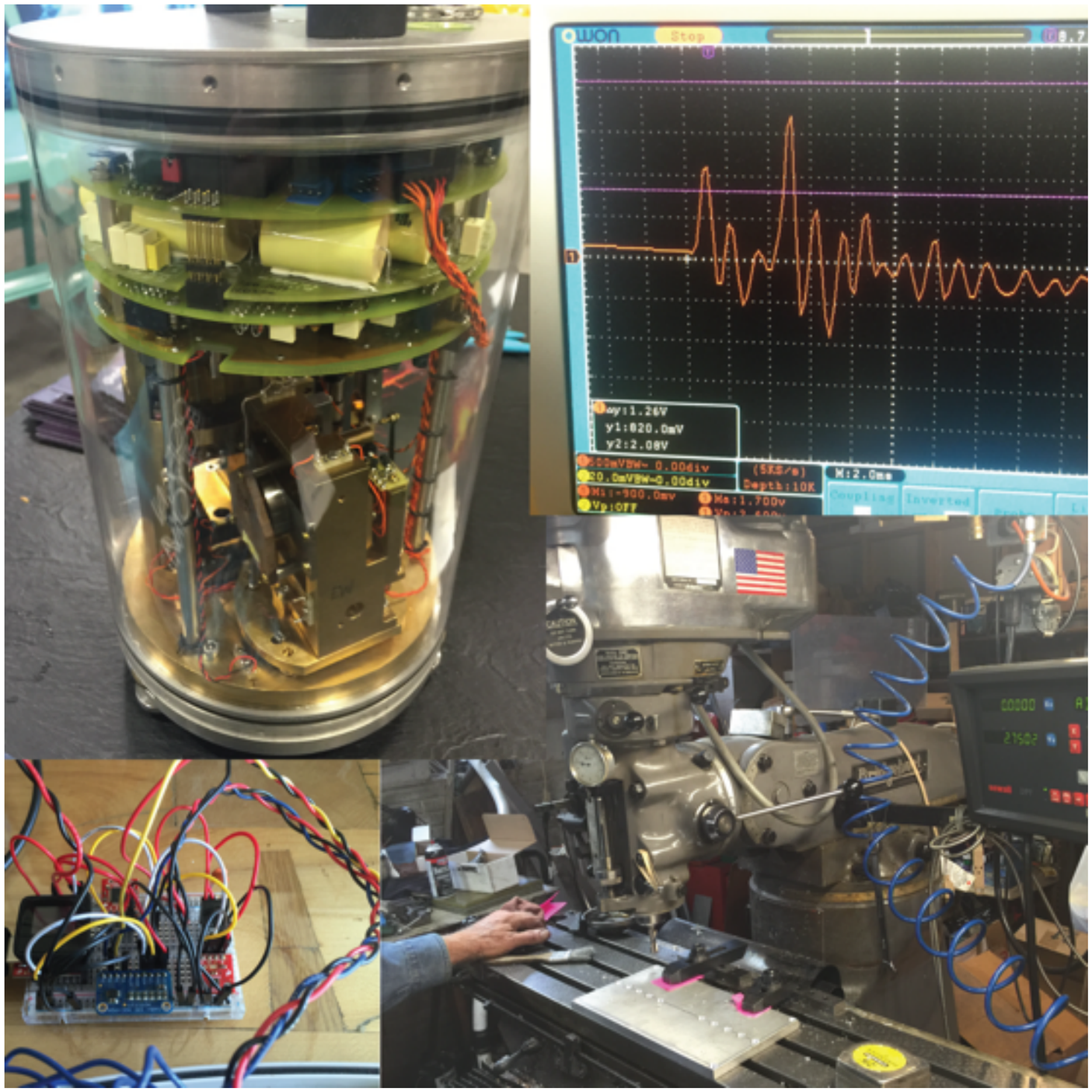
Microcontrollers and Design Patterns

J.R. Leeman and C. Marone

Techniques of Geoscientific Experimentation

September 13, 2016



Today we're going to cover microcontroller interfaces and code design basics

- **PWM**
- **Serial, Parallel**
- **I2C, SPI**
- **Timers and Interrupts**
- **State Machines**
- **Polling**

Pulse width modulation changes the duty cycle of a square wave

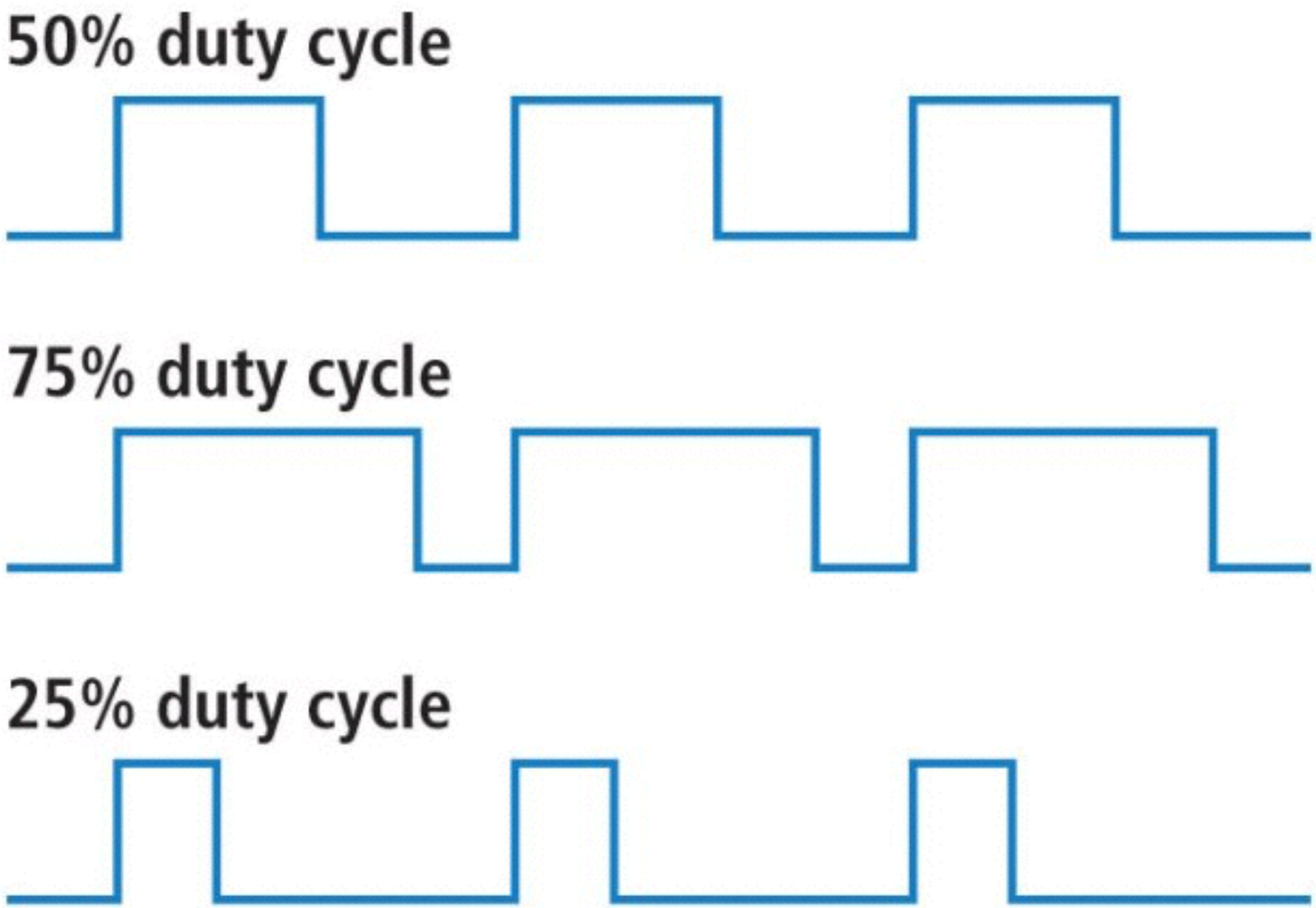


Image: sparkfun.com

```
int led = 9;           // the PWM pin the LED is attached to
int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by

// the setup routine runs once when you press reset:
void setup() {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  // set the brightness of pin 9:
  analogWrite(led, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness <= 0 || brightness >= 255) {
    fadeAmount = -fadeAmount;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

PWM is also used to drive servos

```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo
// twelve servo objects can be created on most boards

int pos = 0;    // variable to store the servo position

void setup() {
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop() {
  for (pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 180 degrees
    // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
  for (pos = 180; pos >= 0; pos -= 1) { // goes from 180 degrees to 0 degrees
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
}
```

Parallel interfaces transfer many data bits at once

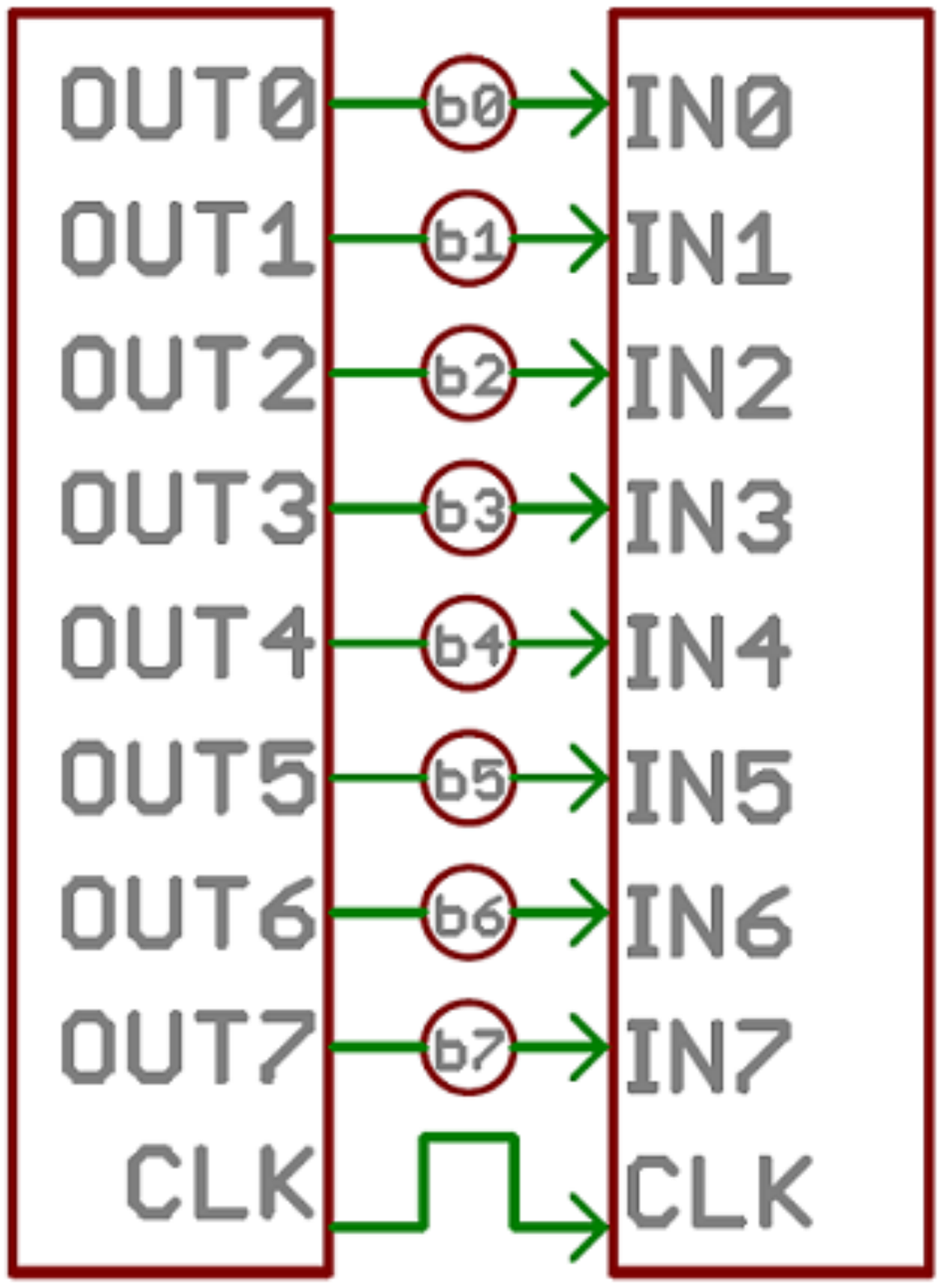


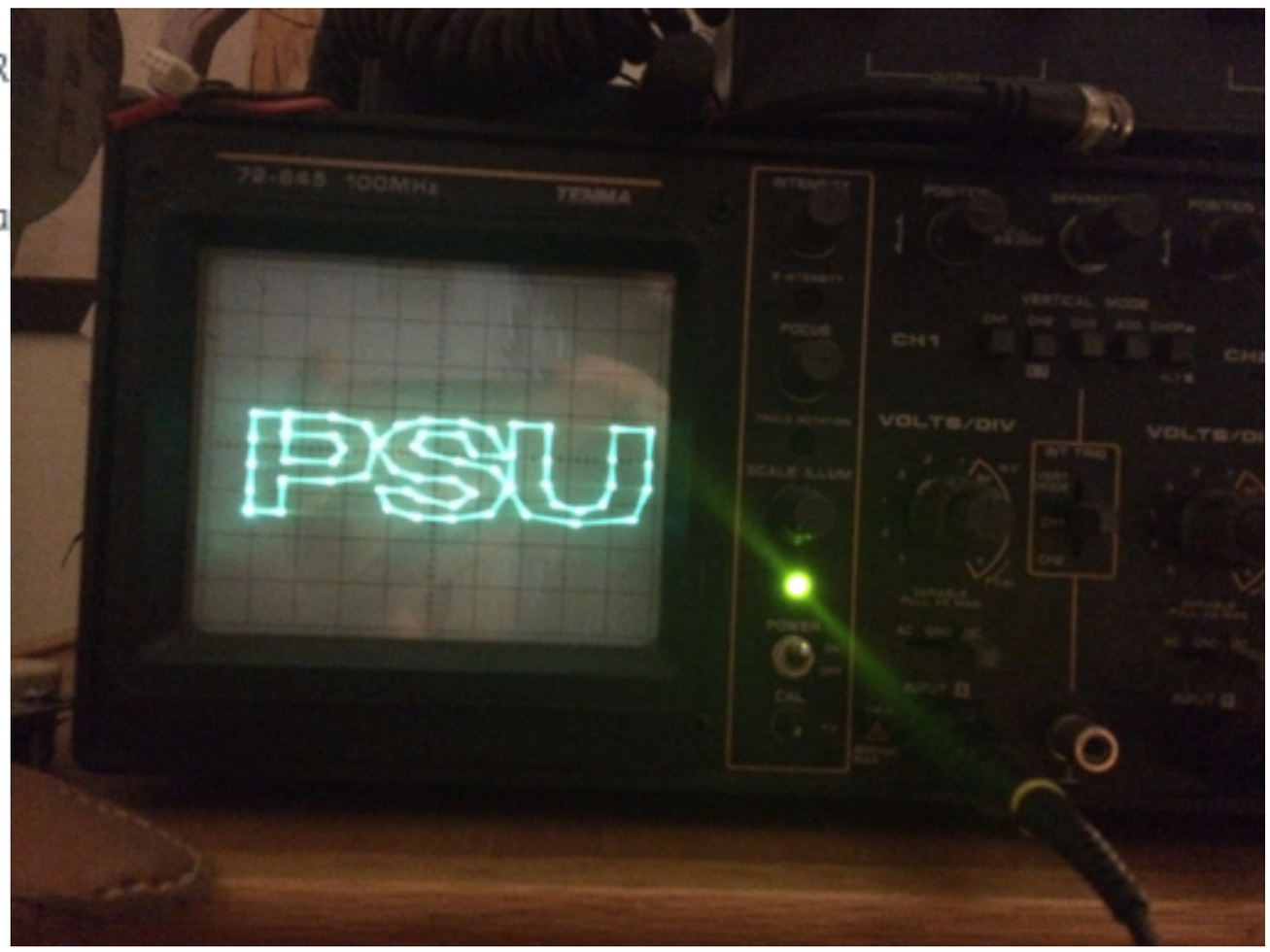
Image: sparkfun.com

These were once very common on printers and other computer peripherals

Image: karbosguide.com

Arduino port manipulation can be done!

```
void loop()
{
  byte t;
  byte sx;
  float p;
  byte flip;
  for(p=0; p<314; p++)
  {
    sx = ( sin(p/100) * x_half);                // half-sine for X scaling to simulate rotation
    for(t = 0; t < NUM_POINTS; t++)   // run through the X points
    {
      PORTD = map(x_points[t], x_min, x_max, x_mid-sx, x_mid+sx); // scale X with half-sine about mid point
      PORTB = y_points[t];
      delayMicroseconds(FIGURE_DELAY);     // wait FIGUR
    }
  }
  flip = x_min; // flip X min and max values so next ma
  x_min = x_max;
  x_max = flip;
}
```



Serial interfaces transfer data one bit at a time

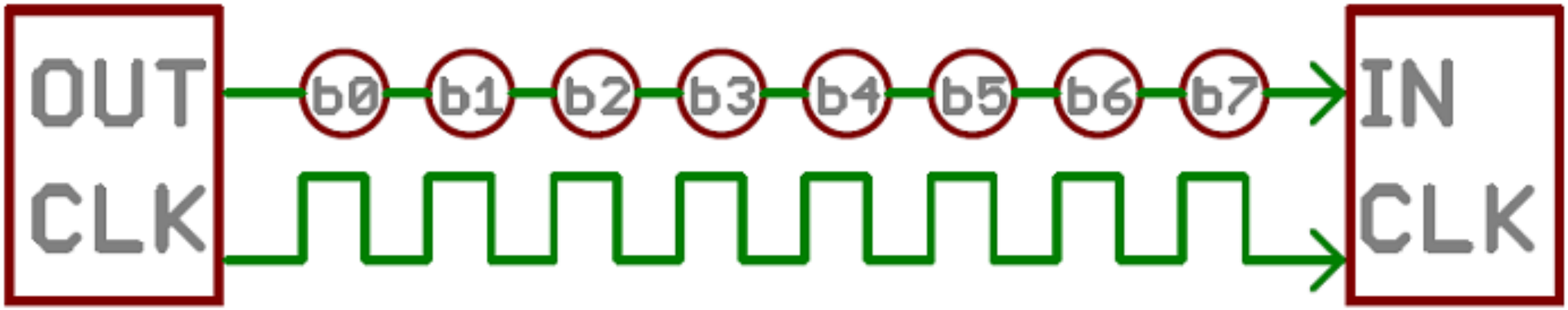


Image: sparkfun.com

The baud rate must be agreed upon, as well as a number of other factors

Image: sparkfun.com

There are cost/data-rate tradeoffs as well as pin count

Serial communications is easy with the Arduino

```
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

// the loop routine runs over and over again forever:
void loop() {
  // read the input on analog pin 0:
  int sensorValue = analogRead(A0);
  // print out the value you read:
  Serial.println(sensorValue);
  delay(1);        // delay in between reads for stability
}
```

SPI is a high throughput four-wire bus

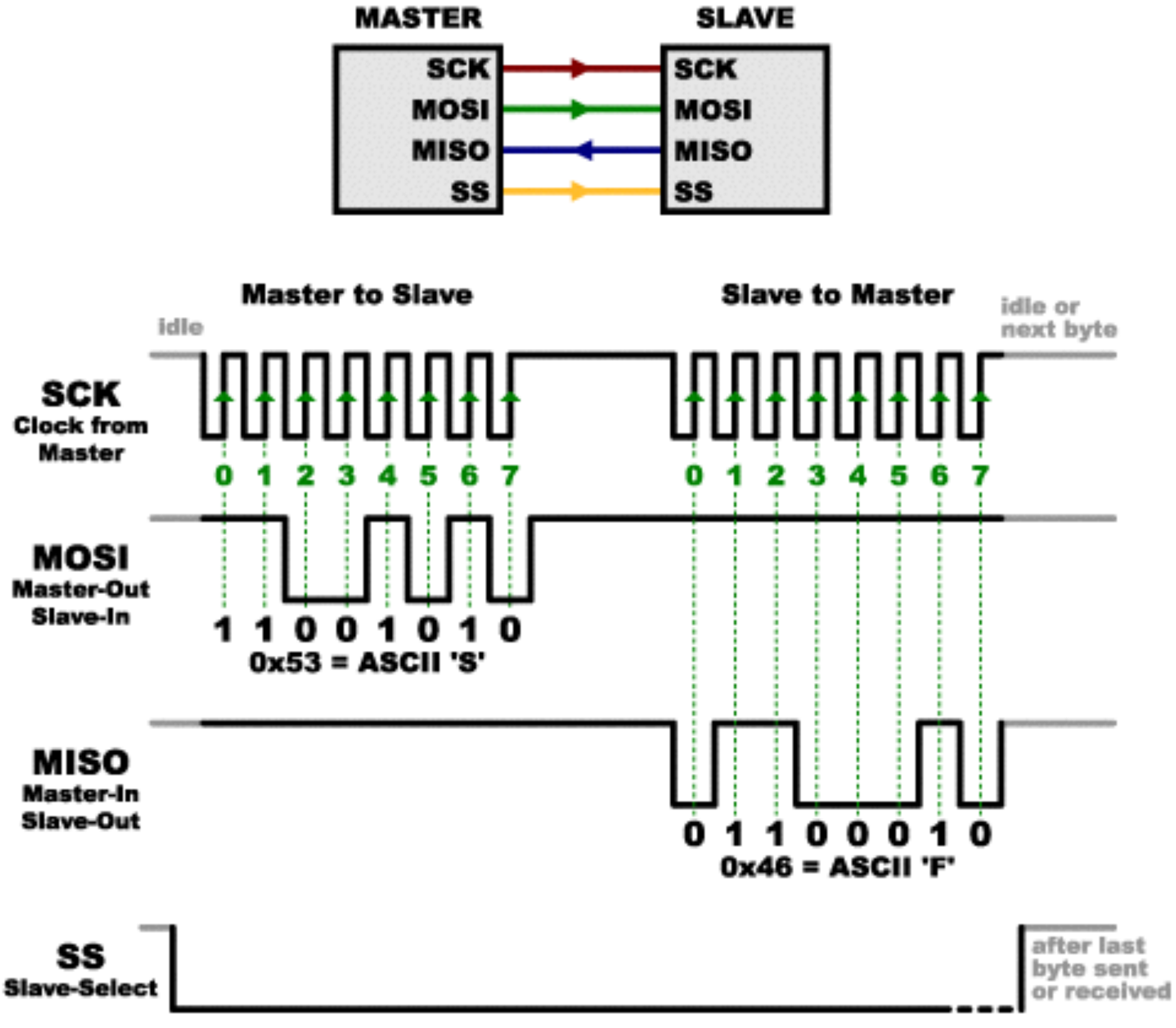


Image: sparkfun.com

Multiple slave devices can be added onto a bus

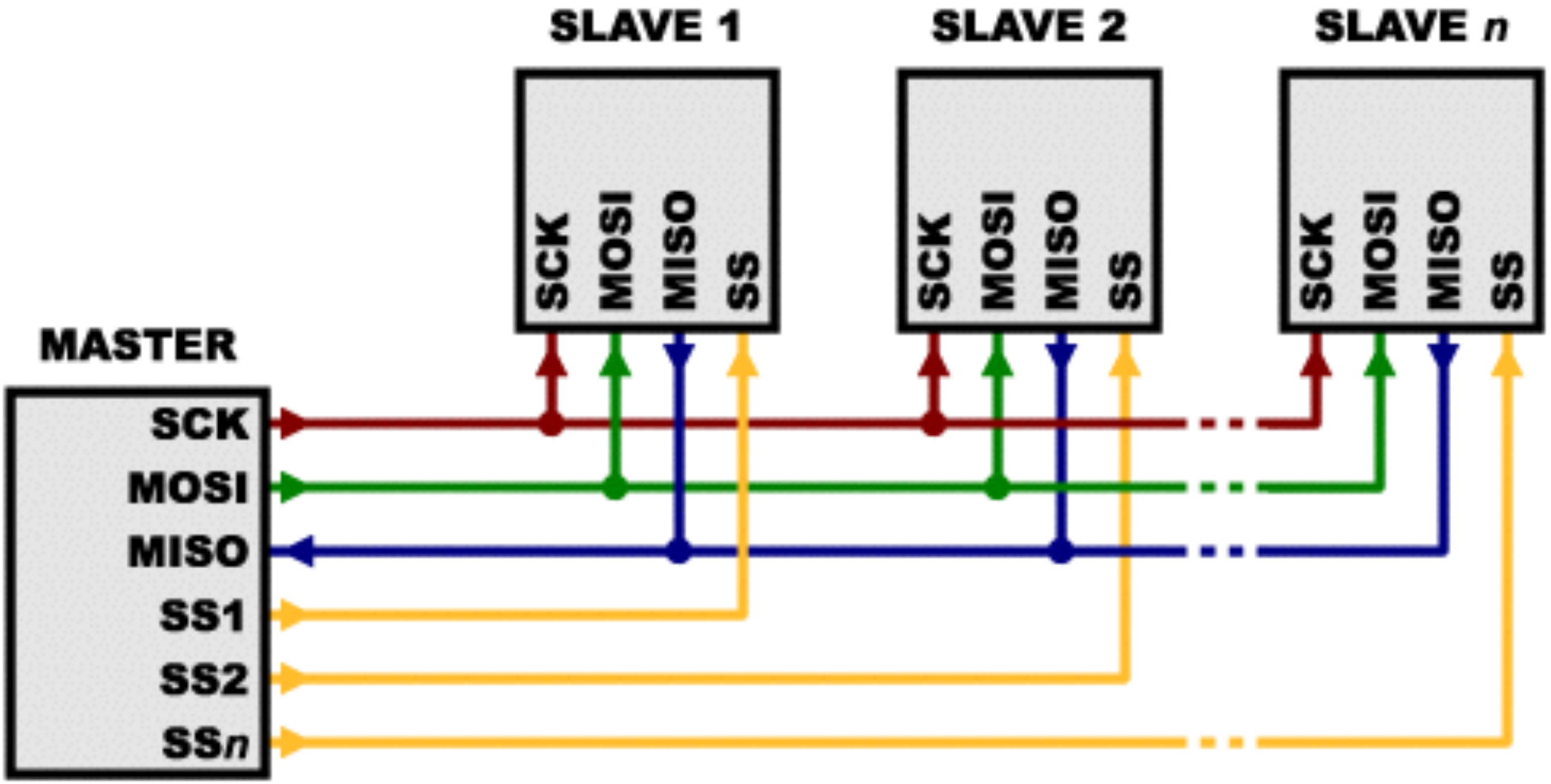


Image: sparkfun.com

Some devices utilize a “daisy-chain” configuration

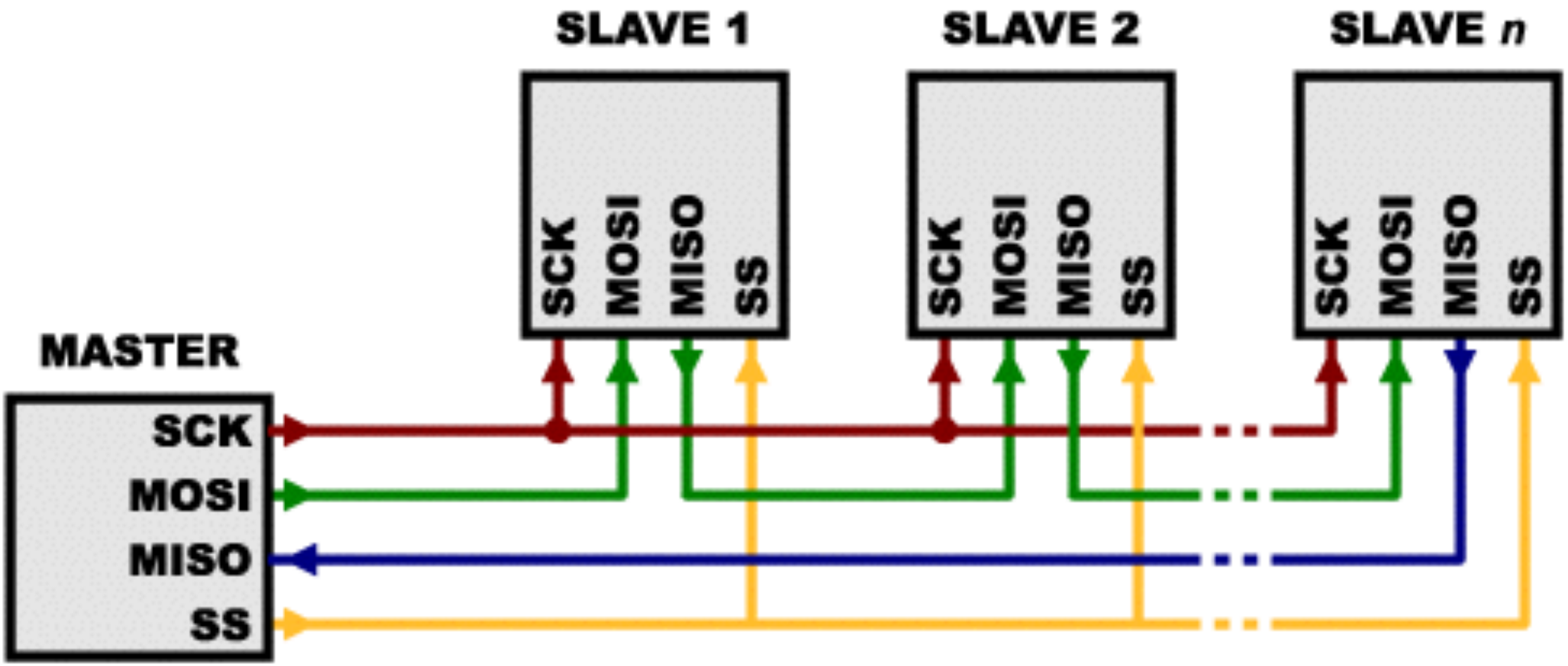


Image: sparkfun.com

SPI library

This library allows you to communicate with SPI devices, with the Arduino as the master device.

A Brief Introduction to the Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used by microcontrollers for communicating with one or more peripheral devices quickly over short distances. It can also be used for communication between two microcontrollers.

With an SPI connection there is always one master device (usually a microcontroller) which controls the peripheral devices. Typically there are three lines common to all the devices:

- **MISO** (Master In Slave Out) - The Slave line for sending data to the master,
- **MOSI** (Master Out Slave In) - The Master line for sending data to the peripherals,
- **SCK** (Serial Clock) - The clock pulses which synchronize data transmission generated by the master

and one line specific for every device:

- **SS** (Slave Select) - the pin on each device that the master can use to enable and disable specific devices.

Functions

See also

- shiftOut()
- shiftIn()

I2C uses only two wires and allows multiple master devices

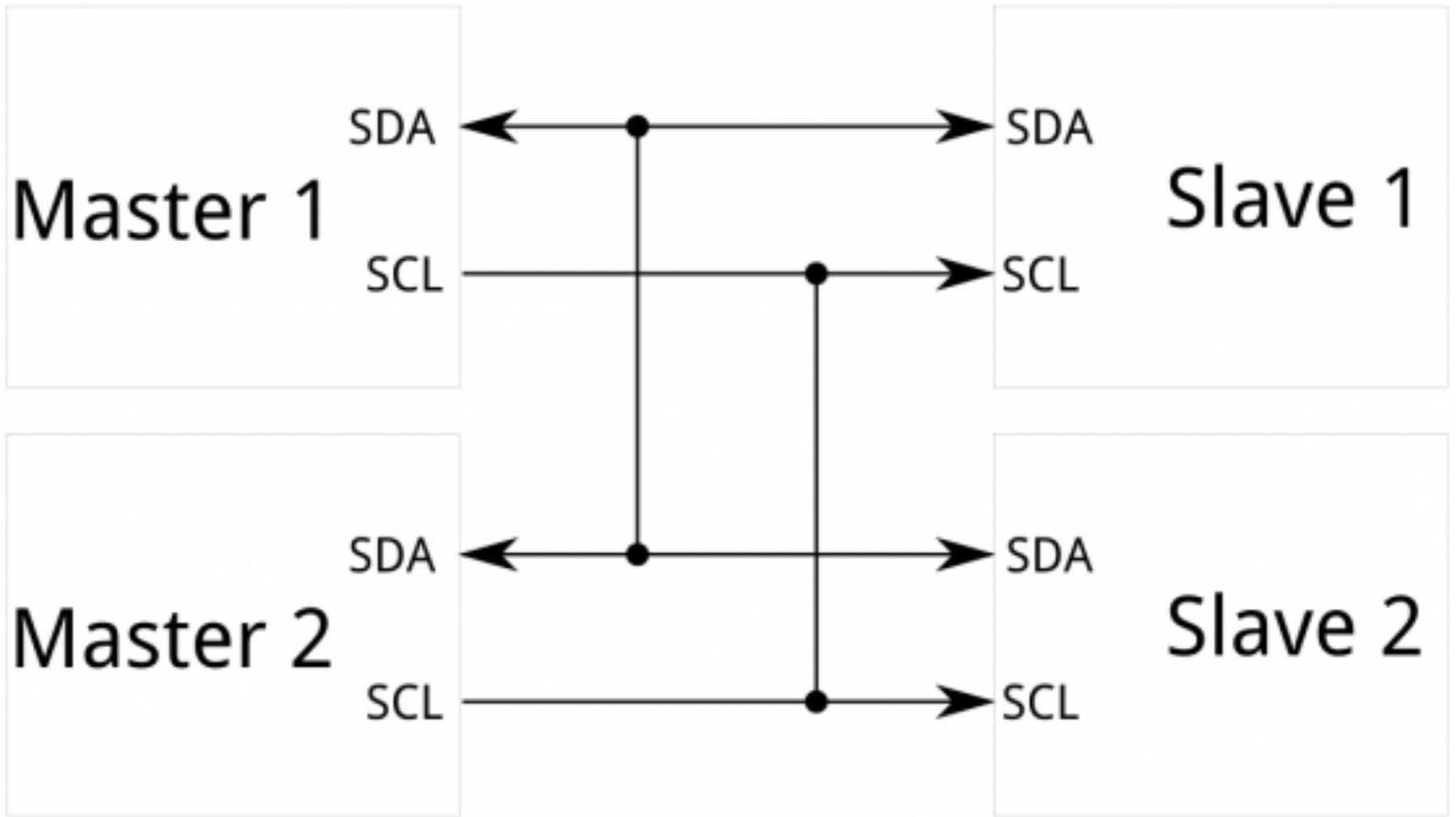


Image: sparkfun.com

The protocol is significantly more complicated than SPI though

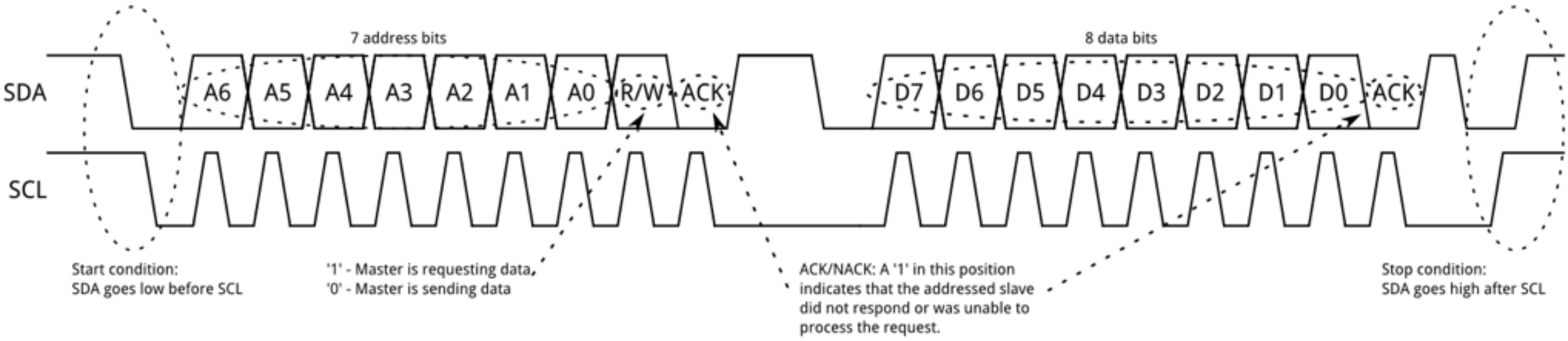


Image: sparkfun.com

Wire Library

This library allows you to communicate with I2C / TWI devices. On the Arduino boards with the R3 layout (1.0 pinout), the SDA (data line) and SCL (clock line) are on the pin headers close to the AREF pin. The Arduino Due has two I2C / TWI interfaces SDA1 and SCL1 are near to the AREF pin and the additional one is on pins 20 and 21.

As a reference the table below shows where TWI pins are located on various Arduino boards.

Board	I2C / TWI pins
Uno, Ethernet	A4 (SDA), A5 (SCL)
Mega2560	20 (SDA), 21 (SCL)
Leonardo	2 (SDA), 3 (SCL)
Due	20 (SDA), 21 (SCL), SDA1, SCL1

As of Arduino 1.0, the library inherits from the Stream functions, making it consistent with other read/write libraries. Because of this, send() and receive() have been replaced with read() and write().

Functions

- begin()
- requestFrom()
- beginTransmission()
- endTransmission()
- write()
- available()
- read()
- onReceive()
- onRequest()

Functions chunk up code into reusable blocks

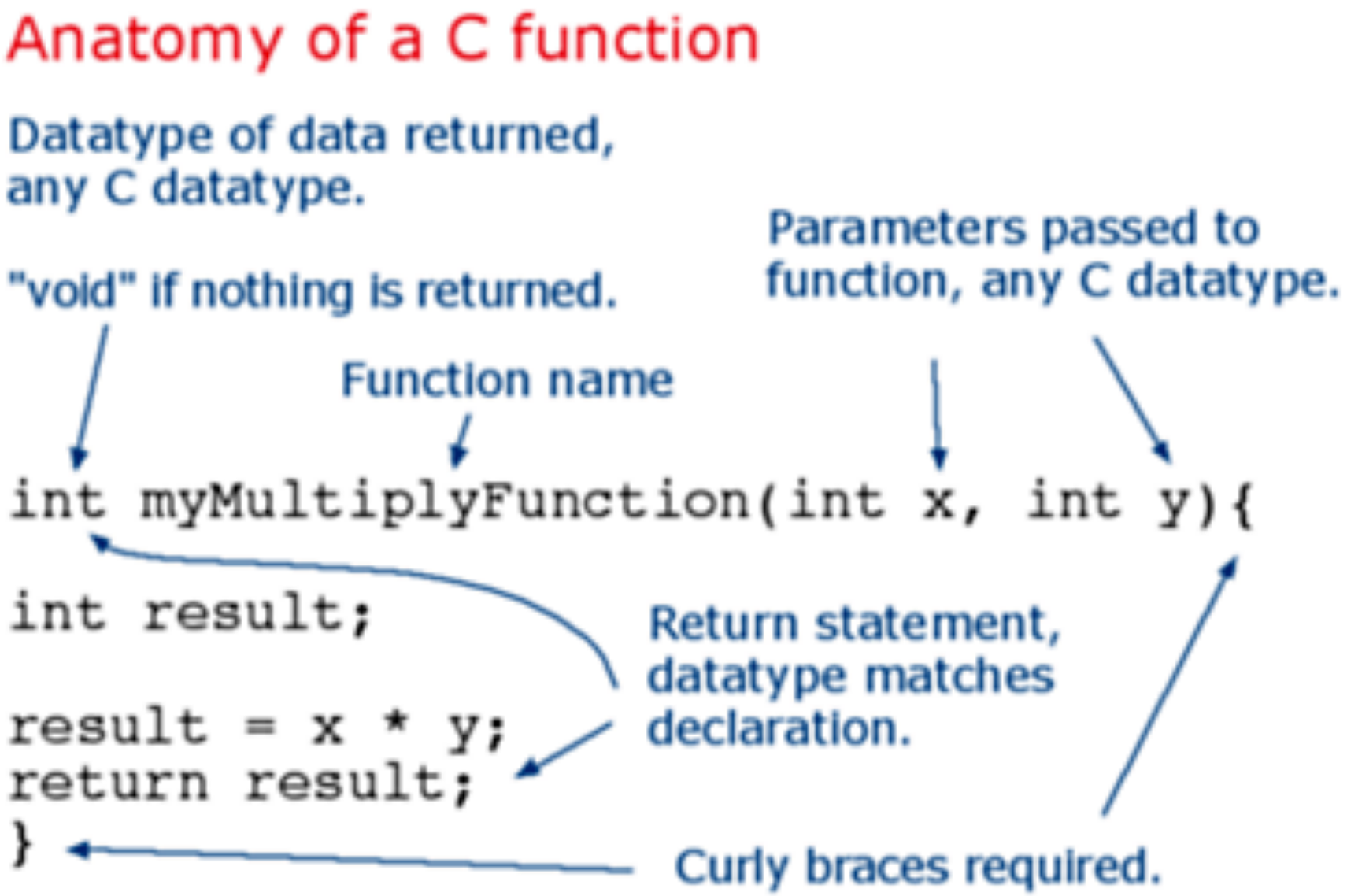


Image: arudino.com

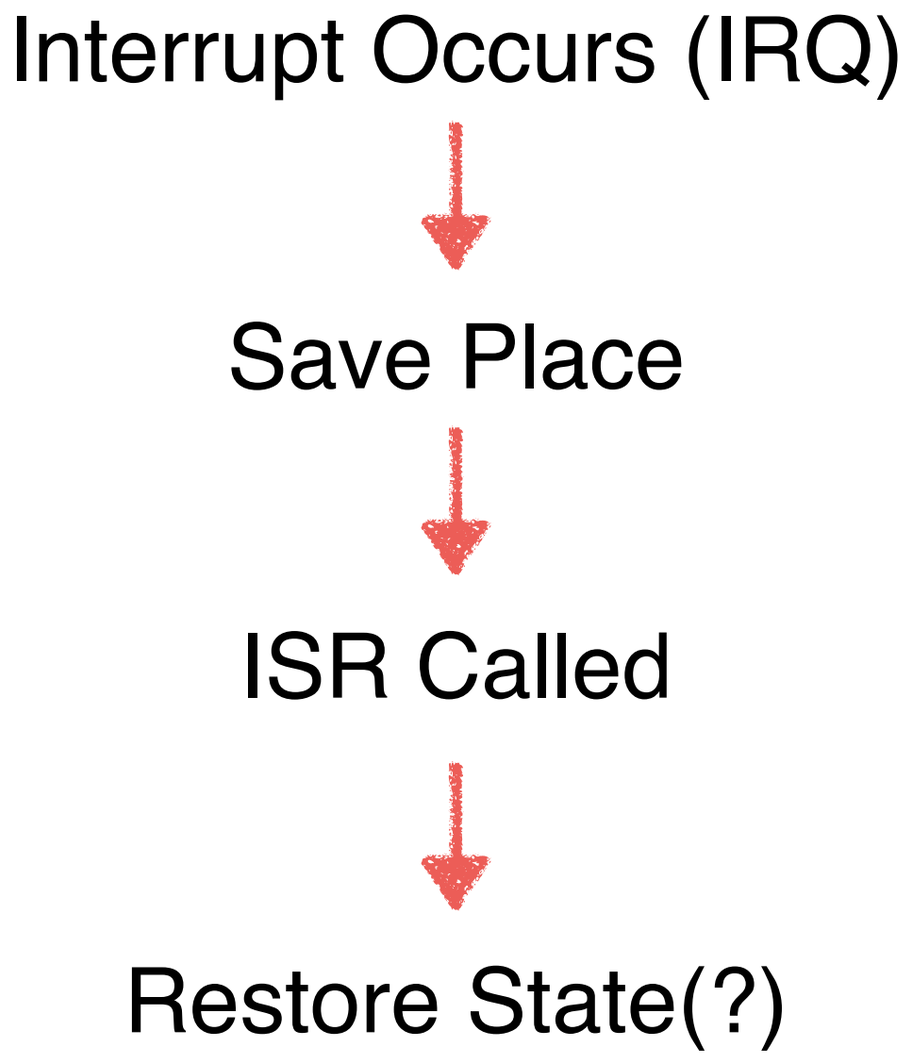
Let's create a multiply function

```
void setup(){
  Serial.begin(9600);
}

void loop() {
  int i = 2;
  int j = 3;
  int k;

  k = myMultiplyFunction(i, j); // k now contains 6
  Serial.println(k);
  delay(500);
}

int myMultiplyFunction(int x, int y){
  int result;
  result = x * y;
  return result;
}
```

Polling repeatedly checks for a condition or event

```
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);
  }
}
```

Interrupts allow event handling, but can be difficult to debug



Interrupts allow event handling, but can be difficult to debug

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

// variables will change:
volatile 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);
  // Attach an interrupt to the ISR vector
  attachInterrupt(0, pin_ISR, CHANGE);
}

void loop() {
  // Nothing here!
}

void pin_ISR() {
  buttonState = digitalRead(buttonPin);
  digitalWrite(ledPin, buttonState);
}
```

Interrupts allow event handling, but can be difficult to debug

```
1  #include <avr/interrupt.h>
2                                       //
3  void setup(void)
4  {
5      pinMode(2, INPUT);
6      pinMode(13, OUTPUT);
7      digitalWrite(2, HIGH);     // Enable pullup resistor
8      sei();                     // Enable global interrupts
9      EIMSK |= (1 << INT0);      // Enable external interrupt INT0
10     EICRA |= (1 << ISC01);     // Trigger INT0 on falling edge
11 }
12                                      //
13 void loop(void)
14 {
15                                      //
16 }
17                                      //
18 // Interrupt Service Routine attached to INT0 vector
19 ISR(EXT_INT0_vect)
20 {
21     digitalWrite(13, !digitalRead(13));    // Toggle LED on pin 13
22 }
```

Timers can also generate interrupts

Read up on timers before using them - it can get complex

```
/* Arduino 101: timer and interrupts
   1: Timer1 compare match interrupt example
   more infos: http://www.letmakerobots.com/node/28278
   created by RobotFreak
*/

#define ledPin 13

void setup()
{
  pinMode(ledPin, OUTPUT);

  // initialize timer1
  noInterrupts();           // disable all interrupts
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1  = 0;

  OCR1A = 31250;            // compare match register 16MHz/256/2Hz
  TCCR1B |= (1 << WGM12);   // CTC mode
  TCCR1B |= (1 << CS12);    // 256 prescaler
  TIMSK1 |= (1 << OCIE1A);  // enable timer compare interrupt
  interrupts();             // enable all interrupts
}

ISR(TIMER1_COMPA_vect)          // timer compare interrupt service routine
{
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1);   // toggle LED pin
}

void loop()
{
  // your program here...
}
```

State machines are used to control many systems, especially instruments and laboratory apparatus

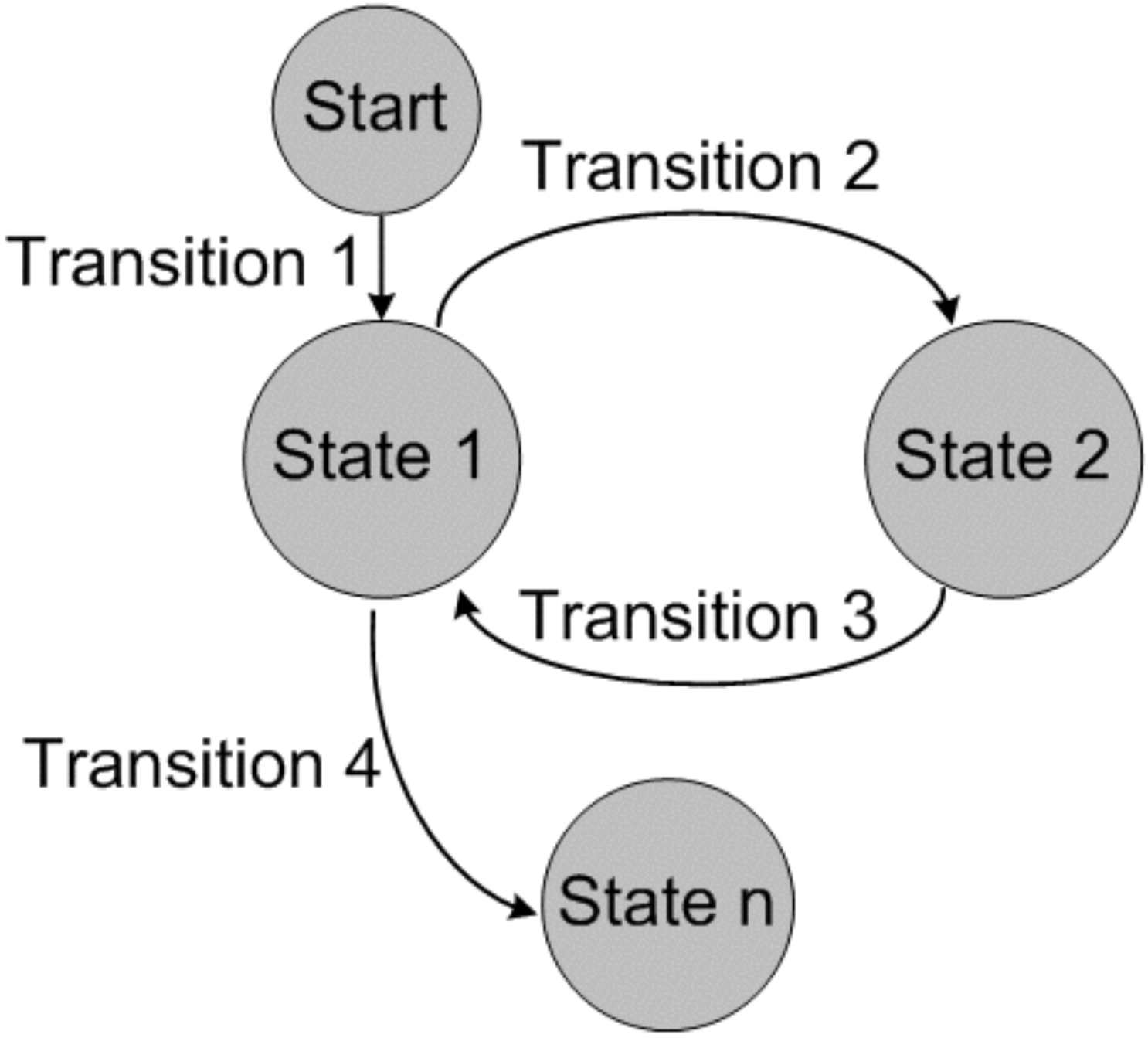
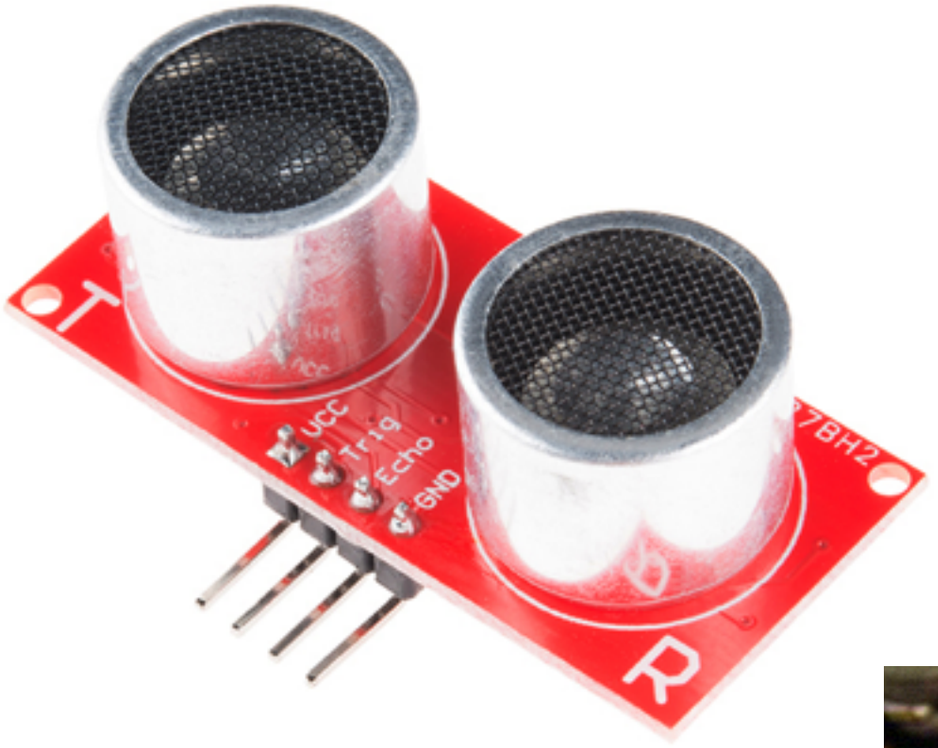


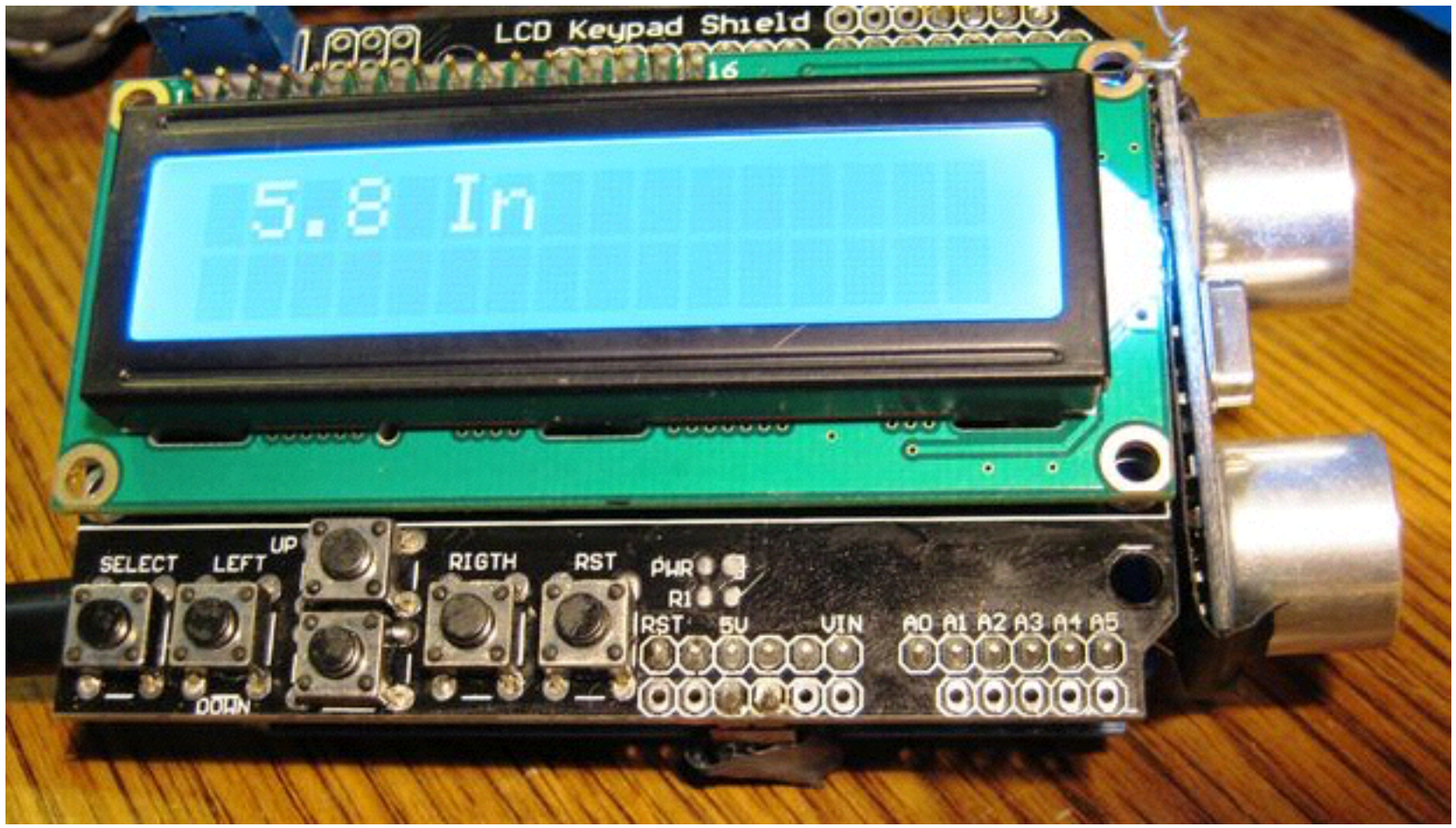
Image: oracle.com

Implementing a state machine

1. Outer While Loop
2. Inner Case Structure
3. Initial State
4. Next State Set
5. Shutdown State

Let’s create a state machine for an ultrasonic tape measure





Images: Sparkfun, Instructables

Multithreaded - producer/consumer can be useful for sensing and datalogging applications

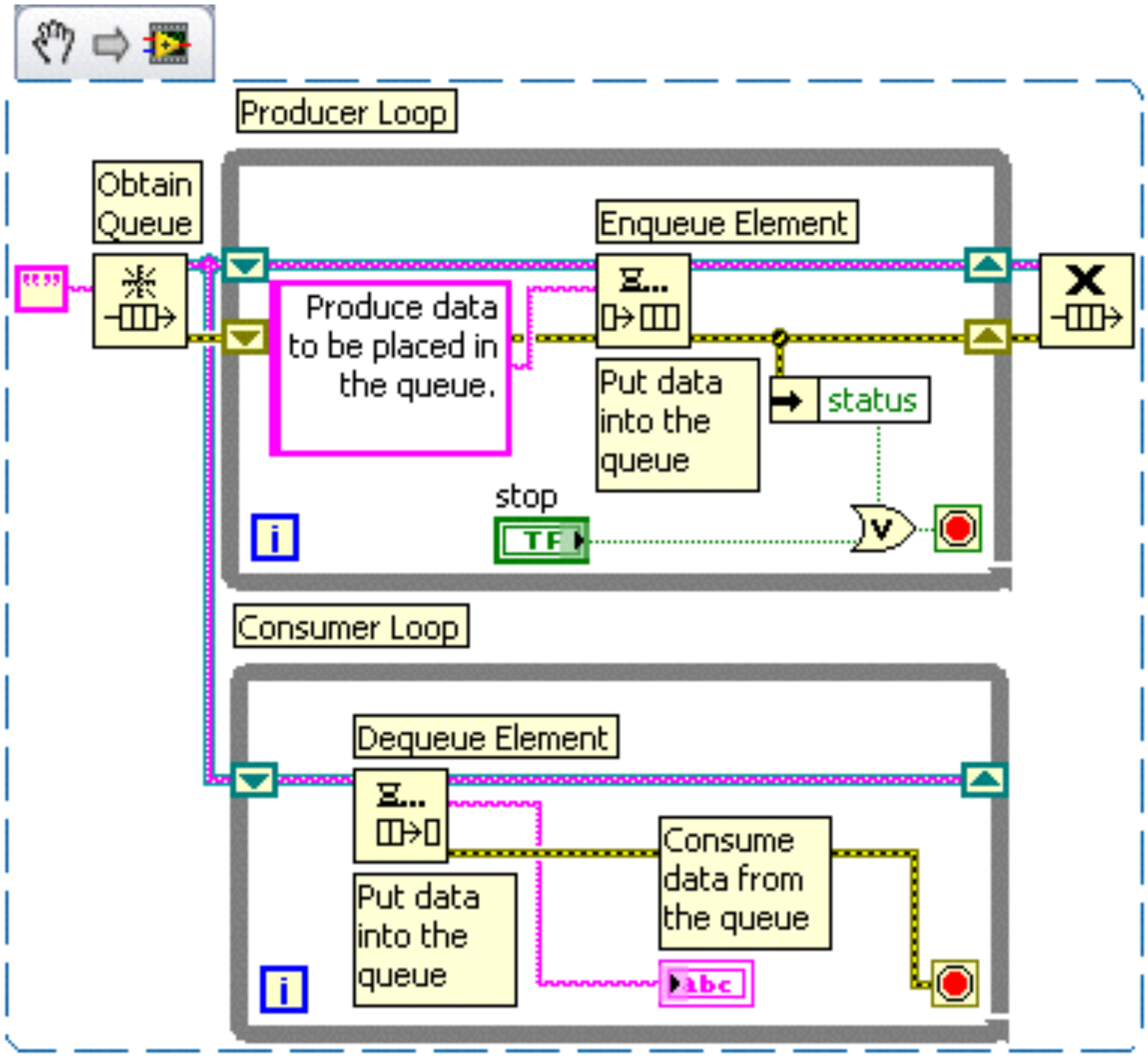
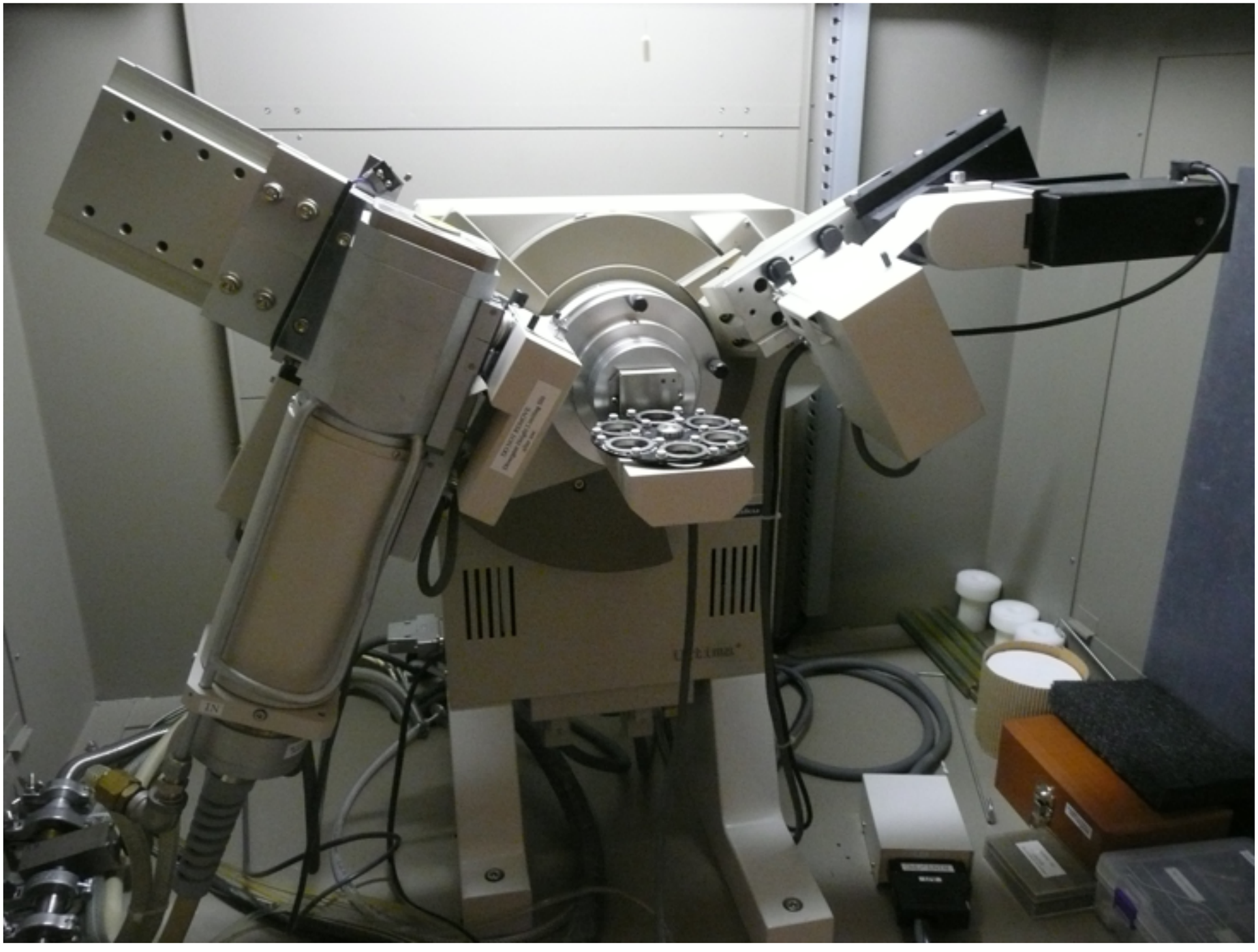


Image: ni.com

Assignment: Arduino Stop Light

DUE: 9/27/16

Activity: XRD State Machine



DUE: 9/20/16