An analog-to-digital converter, or ADC as it is more commonly called, is a [device](https://www.webopedia.com/TERM/D/device.html) that converts [analog](https://www.webopedia.com/TERM/A/analog.html) signals into [digital](https://www.webopedia.com/TERM/D/digital.html) signals. Analog information is transmitted by modulating a continuous transmission signal by amplifying a signal's strength or varying its frequency to add or take away data. Digital information describes any system based on discontinuous data or events. [Computers](https://www.webopedia.com/TERM/C/computer.html), which handle data in digital form, require analog-to-digital converters to turn signals from analog to digital before it can be read. One example is a [modem](https://www.webopedia.com/TERM/M/modem.html) which turns signals from digital to analog before transmitting those signals over communication lines such as telephone lines that carry only analog signals. The signals are turned back into digital form ([demodulated](https://www.webopedia.com/TERM/M/modulate.html)) at the receiving end so that the computer can process the [data](https://www.webopedia.com/TERM/D/data.html) in its digital format.

**Analog to Digital Converter** (ADC) and **Digital** to**Analog Converter** (DAC) are very important components in electronic equipment. Since most real world signals are **analog**, these two converting interfaces are necessary to allow **digital** electronic equipments to process the **analog** signals.

**Analog To Digital Converter**

In electronics, an Analog to Digital Converter (ADC) is a device for converting an analog signal (current, voltage etc.) to a digital code, usually binary. In the real world, most of the signals sensed and processed by humans are analog signals. Analog-to-Digital conversion is the primary means by which analog signal are converted into digital data that can be processed by computers for various purposes, Figure 3.

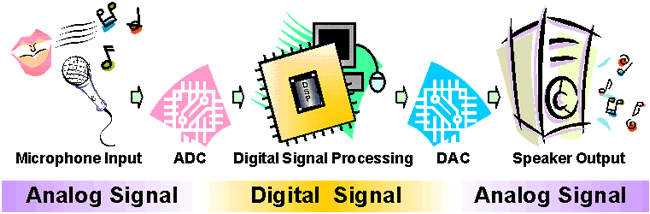


Figure 1: Audio Signal Processing

There are many types of ADC for different applications. The most inexpensive type of ADC is a Successive-Approximation ADC. Figure 4 shows the transfer curve of a 4-bit ADC. Inside a Successive-Approximation ADC, a series of digital codes, each corresponds to a fix analog level, are generated successively by an internal counter to compare with the analog signal under conversion. The generation is stopped when the analog level becomes just larger than the analog signal. The digital code corresponds to the analog level is the desired digital representation of the analog signal.

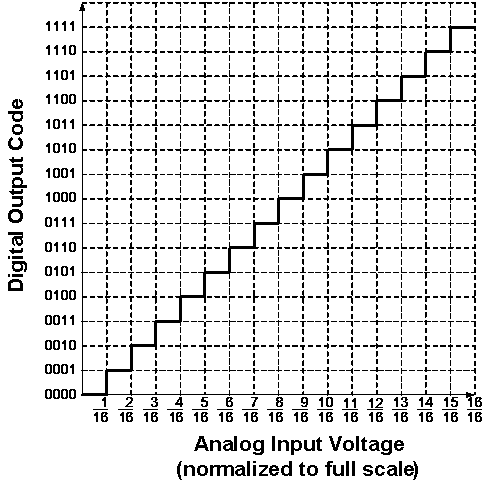


Figure 2: Ideal Transfer Curve of a 4-bit ADC

The performance of ADCs and DACs mainly depends on their Resolution and Speed.

The Resolution of a converter is expressed in the number of Bit. For an ADC, the Resolution states the number of intervals or levels which can be divided from a certain analog input range. An n-bit ADC has the resolution of 1 / 2n. For example, the Resolution of a 16-bit ADC is 1 / 65536, since 216 = 65536. If the measuring voltage range is 10 V, then this input range can be resolved into 10 V / 65536 = 0.153 mV precision.

The Speed of a converter is expressed by the Sampling Frequency. It is the number of times that the converter samples the analog signal, its unit is Hertz (Hz). In audio signal processing, Sampling Frequencies of 44 kHz, 22 kHz and 11 kHz are mostly used. Using 44 kHz Sampling Frequency means the converter is sampling the analog audio signal and doing analog to digital conversion at 44000 times per second. The higher the Sampling Frequency, the lower the distortion and the better the sound quality.

ADCs are used virtually everywhere, whenever an analog signal has to be transported, it is processed and stored in digital form. They are always used together with different transducers to convert physical sense and measurement such as temperature, pressure, humidity, speed, vibration, sound, picture etc. in digital signal for further processing by microprocessor.

**1-Digit Voltmeter**

A Voltmeter is a measuring instrument for measuring the Voltage between two points in an electric circuit.

This Exhibit is a simple application of the ADC for measuring the value of an analog input. Figure 4 is its block diagram. The 1-Digit Voltmeter consists of an ADC, a Clock Generator, a 4-bit Binary Counter, a BCD to 7-Segment Decoder and a 7-Segment LED Display.

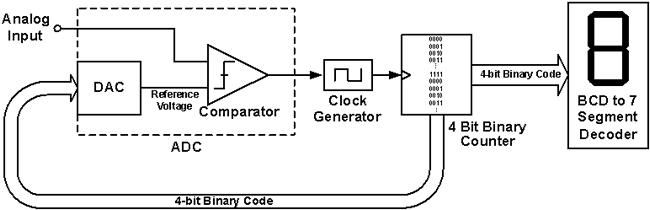


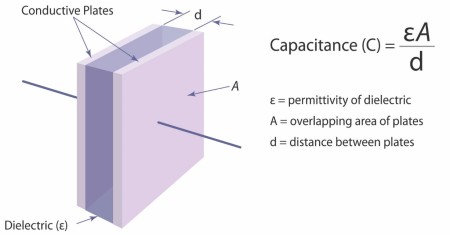
Figure 1: Block diagram of the 1-Digit Voltmeter

The operation flow is as follow:

1. Initially, the DAC Reference Voltage is set to zero, which is smaller than the Analog Input, therefore the Comparator outputs a signal to enable the Clock Generator.
2. The Binary Counter receives the Clock Signal and increases the Binary Code, from (0000)2 towards (1111)2.
3. The DAC converts this 4-bit Binary Code into a new Reference Voltage.
4. This Reference Voltage is compared to the Analog Input again. If it is still smaller than the Analog Input, the above cycle continues. If it is larger than the Analog Input, the Comparator will change its output and disable the Clock Generator, the Binary Counter stops increasing, and the conversion of the Analog Input to a Binary Code is done.
5. This resulting Binary Code is the one that causes the DAC to produce an analog voltage that is as close as to the analog input as possible without exceeding it.
6. This Binary Code is fed to the BCD to 7-Segment Decoder to drive the 7-Segment LED to display the 1-digit decimal value.

**Some inductive and capacitive position sensors can look quite similar and so it is no surprise that design engineers can find the differences between them confusing. Both use a non-contact technique to measure position and both can be built using printed circuit boards. Nevertheless, the basic physics, on which each type of sensor relies, is quite different. Ultimately, what this means in practice is that each type is suited to particular applications. This article explains the physics behind each technology and compares the consequent strengths and weaknesses of each approach.**

Capacitive Position Sensors – Operating Principles



The capacitance effect is used in lots of sensors, notably in the touch sensors of devices such as mobile phones and tablet computers. These capacitive sensors detect the absence or presence of a person’s finger, acting as an alternative to a push button switch. The presence of a person’s finger – or rather the water in it – is to change the relative static permittivity causing a shift in capacitance.

Another type of capacitive sensor is the capacitive displacement sensor, which works by measuring change in capacitance from the change in dimensions of the capacitor. As can be seen by the mathematical formula in ***Fig 1***, capacitance varies in proportion to the distance between the plates (*d*) as well as their overlap (*A*). Displacement can be measured axially (variation in *d*) or in the planar direction of plate overlap (variation in *A*). Advantageously, capacitor plates can be generated using printed circuit boards.

In order to store any significant amount of charge, the separation dimension d must be small compared to the area of the plates. Dimension *d* is usually <<1mm. Hence, such a technique is well-suited to load or strain measurement which might cause relatively large changes in this small dimension. Similarly, capacitive linear or rotary sensors can be arranged so that displacement causes a variation in *A*, the effective overlap of the plates. In other words one set of plates is on the moving element of the sensor while the other set is on the stationary element.  As the two elements displace relative to each other, *A* varies.

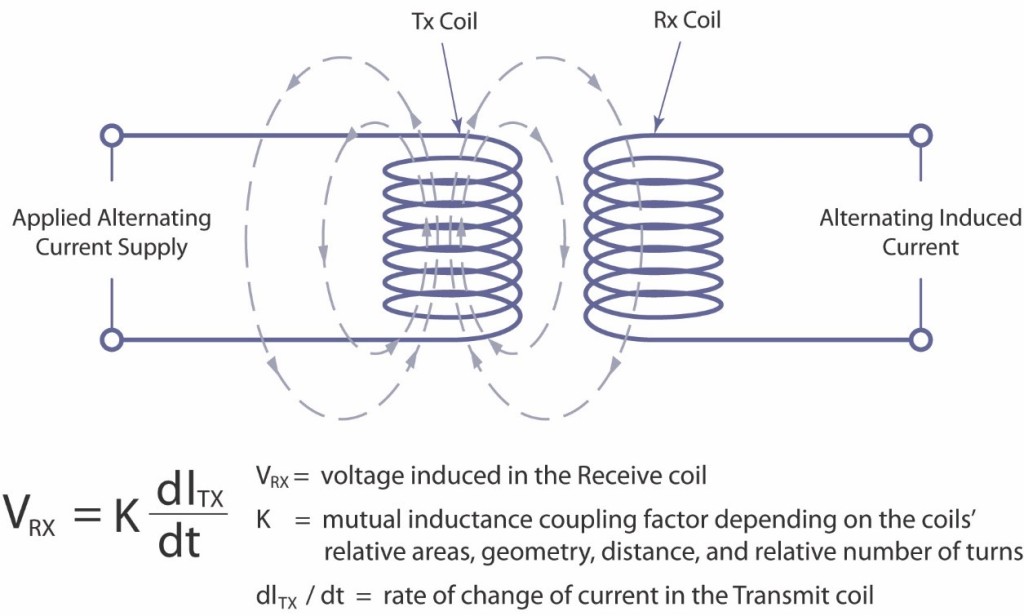
Unfortunately, capacitance is also sensitive to factors other than displacement. If the capacitor’s plates are surrounded by air then its permittivity will also vary with temperature and humidity, because water has a different dielectric constant to air. A nearby object which varies the permittivity of the surrounding area will also vary the capacitance. With a touch sensor, it is the water in the finger that causes a change in local permittivity, changing the capacitance and thus triggering a switch. This is why the operation of unresponsive touch sensors can be improved by wetting the end of the operator’s finger.

Unless the surrounding environment can be sealed or tightly controlled, capacitive sensors are not suited to harsh environments where there is the possibility of ingress of foreign matter or large temperature swings. Unsurprisingly, capacitive sensors are not suitable for environments where condensation may occur at lower temperatures.

Given the inherent physics, the distance between the sensor’s plates must be kept small relative to the size of the capacitor plates and set within tight limits. This can require extremely precise mechanical installation of the sensor and this may not be practical or economical, as differential thermal expansion, vibration or mechanical tolerances of the host system will cause the separation distance to vary and hence distort measurement.

Inductive Position Sensors – Operating Principles

Inductive principles have been widely used as a basis for position and speed measurement in devices such as resolvers, synchros and[linearly variable differential transformers (LVDTs)](https://en.wikipedia.org/wiki/Linear_variable_differential_transformer). The basic theory can be explained by considering two coils – a transmit coil (Tx), with an alternating current applied to it, and a receive (Rx) coil, in which a current will be induced:

[](http://www.zettlex.com/wp-content/uploads/Faradays-Induction-Law.jpg)

***Fig 2****– Faraday’s Induction Law*

The voltage signal in the receive coil is proportional to the relative areas, geometry and displacement of the two coils. However, as with capacitive techniques, other factors can also affect the behaviour of the coils. One such factor is temperature but this effect can be negated by the use of multiple receive coils and by calculating position from the ratio of the received signals (as in a differential transformer). Accordingly, if temperature changes, the effect is cancelled out since the ratio of the signals is unaltered for any given position.

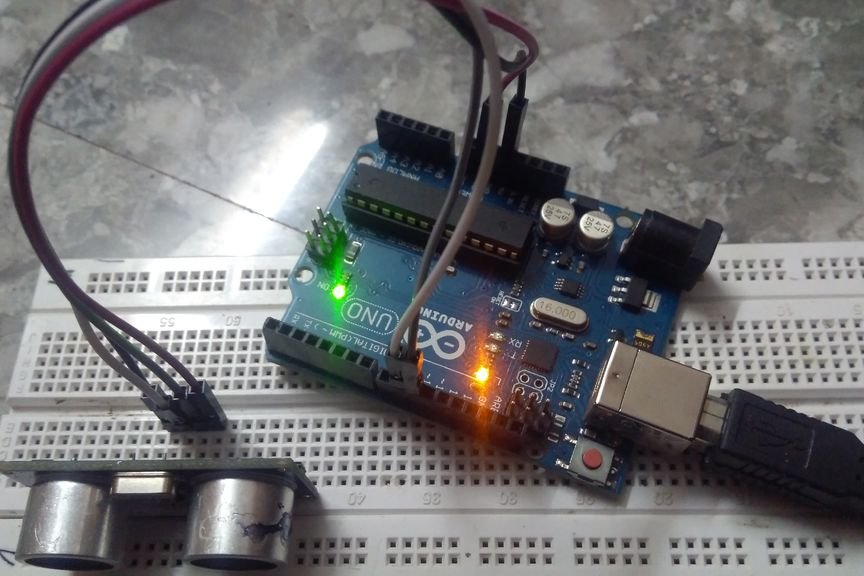
Unlike capacitive methods, inductive techniques are much less affected by foreign matter such as water or dirt. Since the coils can be a relatively large distance apart, precision of the installation is less of an issue, and the principal components of inductive position sensors can be installed with relatively relaxed tolerances. This not only helps to minimize costs of both sensor and host equipment, but also enables the components to be encapsulated, allowing the sensors to withstand environmental effects such as long-term immersion, extreme shock, vibration or the effects of explosive gaseous or dust-laden environments.

Inductive sensors provide a robust, reliable and stable approach to position sensing, and are thus the preferred choice in applications where harsh conditions are common, such as defence, aerospace, industrial and the oil & gas sectors.

Despite their robustness and reliability, however, traditional inductive sensors have some downsides which have prevented their uptake from becoming more widespread. Their construction uses a series of wound conductors or spools, which must be wound accurately in order to achieve accurate position measurement. A significant number of coils must be wound in order to achieve strong electrical signals. This wound spool construction makes traditional inductive position sensors bulky, heavy and expensive.

Electromagnetic noise susceptibility is often cited as a concern by engineers considering inductive position sensors. The concern is misplaced given that resolvers have been used for many years within the harsh electromagnetic environments of motor enclosures for commutation and speed control.  As with temperature stability, robustness in harsh electromagnetic environments can be achieved using a differential approach whereby the electromagnetic energy entering different parts of the sensor is effectively self-cancelling. This is why inductive sensors such as resolvers and LVDTs have been the preferred choice in notoriously safety conscious sectors such as civil aerospace applications for many years.

**INTERFACING HC- SR04 ULTRASONIC SENSOR WITH ARDUINO**

[](https://cdn.instructables.com/FYD/295C/J52QAHIZ/FYD295CJ52QAHIZ.LARGE.jpg)

1.The ultrasonic sensor also known as distance measuring sensor provides 2cm to 400cm of non-contact measurement functionality with a ranging accuracy that can reach up to 3mm.Each HC-SR04 module includes an ultrasonic transmitter, a receiver and a control circuit.There are only four pins that you need to worry about on the HC-SR04: VCC (Power), Trig (Trigger), Echo (Receive), and GND (Ground). You will find this sensor very easy to interface

**Step 1: COMPONENTS REQUIRED**

[](https://cdn.instructables.com/FWT/H38N/J52QAHCM/FWTH38NJ52QAHCM.LARGE.jpg)

[](https://cdn.instructables.com/FD0/FNN8/J52QAHCP/FD0FNN8J52QAHCP.LARGE.jpg)

[](https://cdn.instructables.com/FMS/H9J6/J52QAHDO/FMSH9J6J52QAHDO.LARGE.jpg)

[](https://cdn.instructables.com/FBK/YLAB/J52QAHCO/FBKYLABJ52QAHCO.LARGE.jpg)

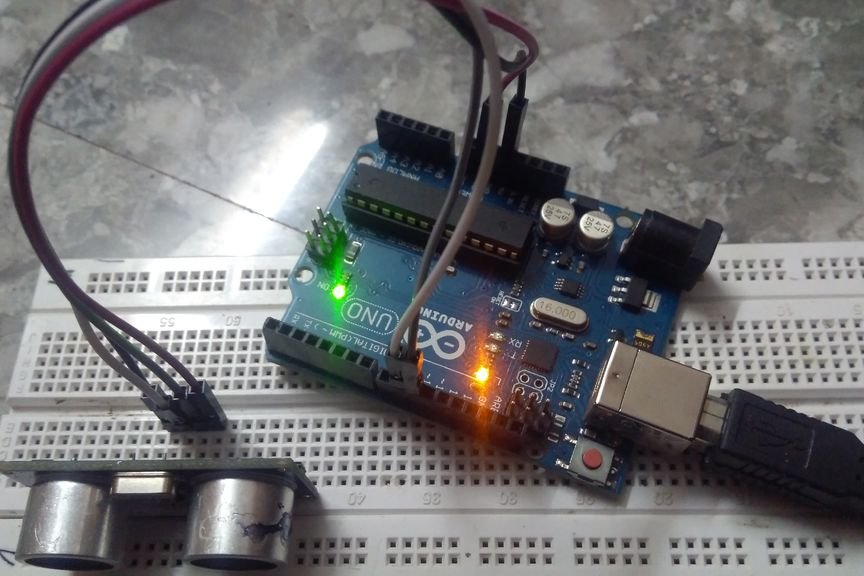
1.HC- SR04 ULTRASONIC SENSOR

2.JUMPER WIRES(4 male-male)

3.BREADBOARD

4.ARDUINO WITH USB CABLE(ANY MODEL)

**Step 2: WIRING THE COMPONENTS**

[](https://cdn.instructables.com/F87/G2BK/J52QAHDP/F87G2BKJ52QAHDP.LARGE.jpg)

Wire the components as given

1.VCC-5V ON ARDUINO

2.GND-GND ON ARDUINO

3.ECHO-PIN 10 ON ARDUINO

4.TRIG -PIN 9 ON ARDUINO

**CODE:**

****

**Step 4: WORKING**

The sensor emits an ultrasond of 40,000hz and travels in air when it hits an obstacle it is detected by the receiver and time taken is taken as the base for calculating the distance

The HC-SR04 Ultrasonic Module has 4 pins, Ground, VCC, Trig and Echo. The Ground and the VCC pins of the module needs to be connected to the Ground and the 5 volts pins on the Arduino Board respectively and the trig and echo pins to any Digital I/O pin on the Arduino Board.

In order to generate the ultrasound you need to set the Trig on a High State for 10 µs. That will send out an 8 cycle sonic burst which will travel at the speed sound and it will be received in the Echo pin. The Echo pin will output the time in microseconds the sound wave traveled.

For example, if the object is 10 cm away from the sensor, and the speed of the sound is 340 m/s or 0.034 cm/µs the sound wave will need to travel about 294 u seconds. But what you will get from the Echo pin will be double that number because the sound wave needs to travel forward and bounce backward. So in order to get the distance in cm we need to multiply the received travel time value from the echo pin by 0.034 and divide it by

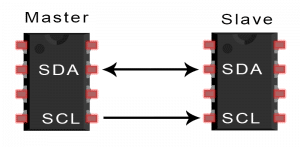
Different Communication Protocols:

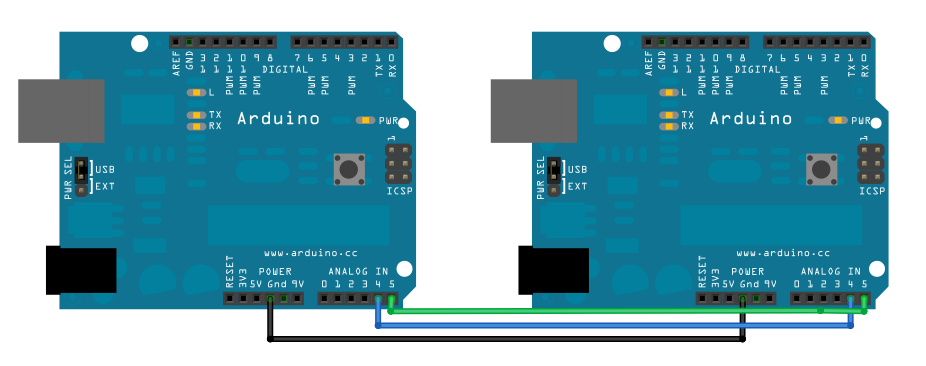
I2C communication Protocol:

#effective as Arduino has limited number of i/o pins

#Uses master/slave configuration wherein there is one master Arduino and there can be several Slave Configurations

WORKING AND CONNECTION:





CODE:

#First CODE:

//this demonstrates the I2C Master/Slave Communication Protocol

//by Kunal Keshav Singh Sahni

//including the library for communication

//this is first part for transmission Arduino

#include<Wire.h>

void setup() {

// put your setup code here, to run once:

Wire.begin();//no need to mention address

//for master arduino

//we'll mention for slave

}

byte x = 0;

void loop() {

// put your main code here, to run repeatedly:

Wire.beginTransmission(8);//address of slave arduino

//now after beginning transmission we can manipulate hell out of the other arduino

Wire.write("x is ");

Wire.write(x);

Wire.endTransmission();

x++;

delay(500);

}

/\*

//now we'll write code for Slave Arduino

#include<Wire.h>

void setup(){

Wire.begin(8);//need to mention address for slave

Wire.onReceive(receiveEvent);//register event using receive Event function that is defined later

Serial.begin(9600); //start serial for output

}

void loop(){

delay(100);

}

//now we need to define the receive Event function

void receiveEvent(){

while(1<Wire.available()){

char c = Wire.read();

Serial.print(c);

}

int x= Wire.read();

Serial.println(x);

}

\*/

I saw this in a video and tried to code for Arduino, basically this code is a two-Arduino based :

Turns the red LED on when you type R

Turns Green LED on when you type G // and switches off the other one

Turns Both LED on when you type 1

Switches both LED off when you type 0

Master Arduino’s Serial is the place where you type in and transmit

Slave Arduino receives signal and Controls The LED

//Code For Master

//By KunalKeshavSinghSahni

#include<Wire.h>

char incomingBytes;

void setup{

Serial.begin(9600);

Wire.begin();

}

void loop(){

Wire.beginTransmission();

If (Serial.available()){

while(Serial.available( )){

incomingByte = Serial.read();

Wire.write(incomingByte);

Serial.println(incomingByte);

}

Wire.endTransmission();

}

}

//Code For Slave

#include<Wire.h>

#define Red 8

#define Green 9

int x=0;

void setup(){

pinMode(Red,OUTPUT);

pinMode(Green,OUTPUT);

Wire.begin(9);

Wire.onReceive(receiveEvent);

}

void ReceiveEvent(int bytes){

x=Wire.read();

}

void loop(){

if(x==’R’){

digitalWrite(Red,HIGH);

digitalWrite(Green,LOW);

}

else if(x==’G’){

digitalWrite(Green,HIGH);

digitalWrite(Red,LOW);

}

else if(x==’1’){

digitalWrite(Red,HIGH);

digitalWrite(Green,HIGH);

}

else if(x==’0’){

digitalWrite(Red,LOW);

digitalWrite(Green,LOW);

)

}

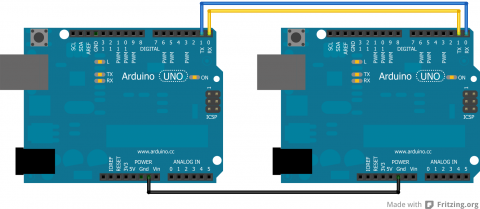
UART Communication

It is possible to chain Arduinos together in such a way as to get communication between the two. Having Arduino-Arduino communication can be useful for many projects, such as having one Arduino to run motors and having another sense the surroundings and then relay commands to the other Arduino. This can be done in several methods, using I2C and Serial, to list a few.

This tutorial will focus on Arduino-Arduino communication through the serial ports (RX and TX).

**Schematic**

The schematic below shows how to connect the two Arduinos together. This shows two Unos, but if a Mega is used, it can be connected to any of the Serial ports on the Mega as long as that is accounted for in the code.

[](http://robotic-controls.com/sites/default/files/learn/Arduino-ArduinoSerial.png)

There has to be a common ground between the two or else it will not function properly. Also, note that TX goes to RX and RX goes to TX.

**Coding**

When sending things through serial, everything is sent in bytes. These bytes are then read one byte at a time by the other Arduino. When it is just characters being sent through the serial, it is relatively easy to convert from characters to bytes. However, if there are both characters and numbers are going through, this can lead to messing up the data because a number and a character can have the same byte value, but that does not make them the same. Numbers are also tricky because they may not actually fit in the byte.

**Simple Code**

The easiest way to get around this is to try to avoid using characters and numbers at the same time. This can be done by sending one character across, each with a different meaning. A good example of this comes from the Arduino [Physical Pixel](http://arduino.cc/en/Tutorial/PhysicalPixel) tutorial.

Upload the Physical Pixel code, which can be found in the Arduino IDE under: File >> Examples >> Communication, onto one Arduino.

On the other Arduino, upload:

void setup() {  
 Serial.begin(9600);  
}  
  
void loop() {  
 Serial.print('H');  
 delay(1000);  
 Serial.print('L');  
 delay(1000);  
}

When this is run, the LED attached to Pin 13 on the Arduino with the Physical Pixel code should flash on and off at a frequency of 0.5 Hz. To make sure this is actually the code doing that, the delays can always be changed in the above code.

In this code the job of 'H' was to turn an LED on and the job of 'L' was to turn the LED off. This can easily be applicable to getting various characters triggering more reactions.

However, depending on the application, this may not be enough and more drastic code may be required.

**Complex Code**

For sending data from one Arduino to another in a form that cannot be simplified, there are other options. One option is to turn everything sent from the Sender Arduino into characters and then have the Receiver Arduino read in the characters. The data is actually sent as bytes, but the Arduino can convert from characters to bytes and vice versa.

**Sender Code**

The sender code changes characters into bytes and, if necessary, it changes number values into characters before turning it into bytes. Below is a sample of the Sender code:

//Sender Code  
  
char str[4];  
  
void setup() {  
 Serial.begin(9600);  
}  
  
void loop() {  
 int value=1234; //this would be much more exciting if it was a sensor value  
   
 itoa(value, str, 10); //Turn value into a character array  
 Serial.write(str, 4);  
}

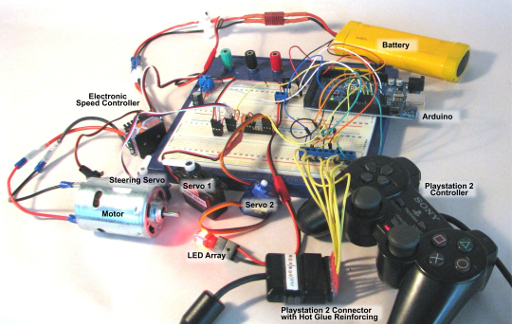
**Receiver Code**

The receiver will then receive the byte array from the other Arduino and interpret it there. Below is the code for the receiver. Note that this code is intended for a Mega since it will interpret the data received from the other Arduino and then print to the Serial Monitor what it received so that the user can check it. This debugging can be avoided by using an Uno and then printing what was found onto an LCD screen that uses I2C communication.

//Receiver Code  
  
char str[4];  
  
void setup() {  
 Serial.begin(9600);  
 Serial1.begin(9600);  
}  
  
void loop() {  
 int i=0;  
  
 if (Serial1.available()) {  
 delay(100); //allows all serial sent to be received together  
 while(Serial1.available() && i<4) {  
 str[i++] = Serial1.read();  
 }  
 str[i++]='\0';  
 }  
  
 if(i>0) {  
 Serial.println(str,4);  
 }  
}

Here’s an excellent resource to PS2 Interfacing

Introduction

[](http://www.techmonkeybusiness.com/images/MicroControllers/PS2_Demo_Circuit_Rig_Annotated-600dpi.jpg)

There has been millions of dollars poured into the development of the Playstation Controllers, and so they are very sophisticated, well designed, and robust pieces of equipment that are ideal for using as an interface for mechatronics projects. Even though the Playstation 2 game console is now a rather outdated piece of equipment, the Playstation 2 Controller Clones are still made and can be purchased through Trademe extraordinarily cheaply.

Not only that, but they are easy to connect to an Arduino. In addition to the two high quality joysticks, all the buttons are pressure sensitive, which adds even more functionality to the device.

This article demonstrates how you would connect a Playstation 2 Controller to an Arduino, and how easily it can be used to control various devices thanks to the PS2Xlib Arduino Library.

Applications for PS2 Controllers

So where could you use a Playstation controller aside from on a Playstation console?

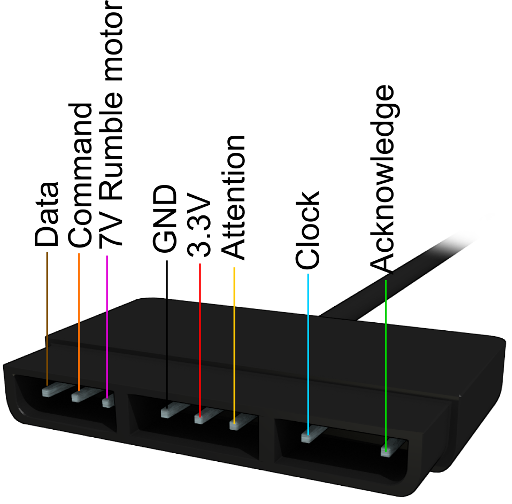
* Controlling a wheeled or tracked vehicle's motion.
* Controlling a robotic arm.
* Interacting with a computer.
* Controlling a pan and tilt camera mount.
* DIY radio control system by using a bluetooth module, or radio module to transmit the commands from the Playstation 2 controller to another Arduino which is controlling the vehicle.
* Then there is my own pet project which uses the Playstation 2 Controller as the pilot interface for an [ROV (aka. underwater drone)](http://www.techmonkeybusiness.com/opensource-rov.html).

Connecting the Playstation 2 Controller to the Arduino

There are two options for connecting the controller to the Arduino. The method for masochists, is to cut the connector off and connect the wires into a row of terminal headers or directly solder them to some other plug. The wire colours in the diagram below may help you identify which wire is which (The 7V – 9V rumble motor supply wire is often gray in colour). A better option is to retain the plug and find a suitable socket. In stark contrast to the ease of purchasing a controller, finding sockets is much harder. It is possible to purchase sockets and breakout boards from the [Robotshop](http://www.robotshop.com/en/ps2-connector.html) in the US. Unfortunately there are no local suppliers. The cheaper alternative if you are into a bit of hacking, is to extract the sockets from a Playstation 2 Controller to USB adapter (about $5 - $10 or so through [Trademe](http://www.trademe.co.nz/)). I have found the tabs on the back of the sockets to be quite fragile and it is worth embedding them in hot glue once you have soldered some wires to it.

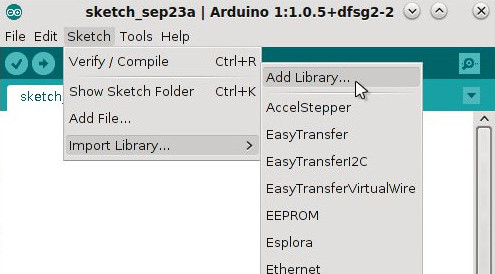
  
*Sources of Playstation 2 Sockets*

The illustration below shows the Playstation 2 plug, and the pin labels. In order to connect to the Arduino we only need to connect the Data, Command, Ground, 3.3V, Attention, and Clock pins. The rumble motor pin only needs to be connected to a 7-9V supply if rumble feedback is desired.

  
*Playstation 2 Plug Pins*

The Software Side

Thanks to one Bill Porter an easy to use library is available which will allow users to use a Playstation 2 controller (or even a Guitar Hero controller) with an Arduino. You can find it through Bill Porter's website: [“The Mind of Bill Porter”](http://www.billporter.info/). More specifically here is the link to his page with [links to his Library](http://www.billporter.info/2010/06/05/playstation-2-controller-arduino-library-v1-0/) and the [Library Source Code can be obtained from Github](https://github.com/madsci1016/Arduino-PS2X), just click on the “**Download ZIP**” button on the right side of the page. You can also download the PS2X library from the repository of libraries I have use in my various projects: [**A Collection of Arduino Libraries Used in This Project**](http://www.techmonkeybusiness.com/arduino-library-collection.html). Once you have the ***zip*** file downloaded, start your *Arduino Interface*, and navigate through the menu ***Sketch>Import Library>Add Library***. This will open a dialogue where you can navigate to your downloaded the **PS2X Library zip** file.



The example sketch included with the library is a great demonstration of the capabilities of the library and makes use of all of the features of the Playstation 2 Controller. The demonstration that is outlined in this article is not quite so sophisticated but covers the essentials needed to use a Playstation 2 Controller to actuate motors, servos, and switch things on and off.

To use the library in an Arduino Sketch, it needs to be called using the following commands;

#include <PS2X\_lib.h>

PS2X ps2x;

In the ***setup*** part of the sketch the command to let the Arduino know how the controller is connected to it, looks like;

ps2x.config\_gamepad(5,4,3,2, false, false);

where the numbers are the Arduino's digital pins where the Playstation 2 Controller pins are connected as below;

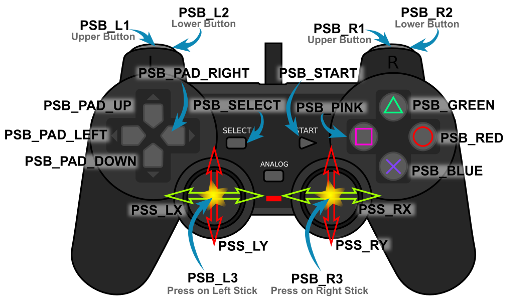
*gamepad(clock, command, attention, data, pressure sensitivity enabled, rumble enabled)*

If you wanted to use the button pressure sensitivity feature, in the command line above you would set this to **“true”** and if you wanted the rumble motor available to provide feedback then you would set this to **“true”** too. For use of the rumble feature please look at the example sketch that comes with the PS2X library.

Once the controller is set up, the Arduino loops through the sketch continuously. Once per loop through, the Arduino needs to communicate with the controller to gather all input data. This is done with the following command.

ps2x.read\_gamepad();

Now we can read which control has been used. The labels used in the PS2X library are very logical. The diagram below shows each of the names for the buttons and sticks.

[](http://www.techmonkeybusiness.com/images/MicroControllers/PS2Lib_Control_Labels-600dpi.jpg)  
*Control names used in the PS2X Library.*

The buttons with the coloured shapes can also be referred to by the names PSB\_TRIANGLE, PSB\_CIRCLE, PSB\_CROSS, and PSB\_SQUARE.

To use the analogue pressure sensitivity on the keys the names are the same except for substituting “PSAB” for “PSB”. So to allow a pressure reading from the Green Triangle button the name would be PSAB\_GREEN or PSAB\_TRIANGLE.

The buttons can be pressed, pressed and held, or pressed with a varying pressure, so there are a number of *methods* that can be applied to the buttons and joystick. The *methods* are; Button Pressed, Button, and Analog. Here are some examples of how these are used.

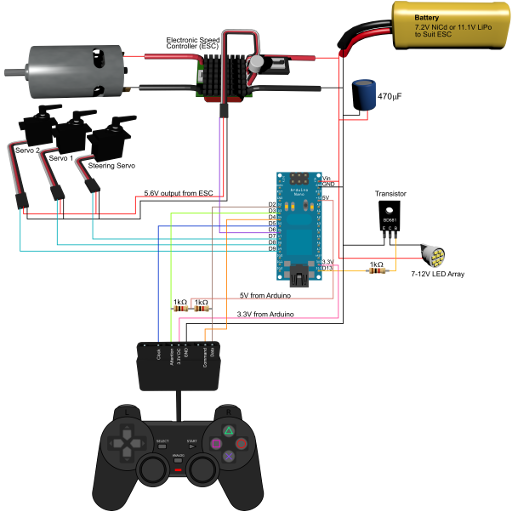
* ps2x.ButtonPressed(PSB\_RED) is for a simple press of the **red circle** button.
* ps2x.Button(PSB\_PAD\_DOWN) is for the **down button on the pad** being pressed and held.
* ps2x.Analog(PSAB\_CROSS) is the command for measuring the pressure applied to the “**X**” button, if pressure sensitivity has been enabled. As you can see the “PSAB” form of the name has been used for the button.
* ps2x.Analog(PSS\_RY) is the command to obtaining readings off the **right analogue stick** in the vertical direction.

Bringing it Together

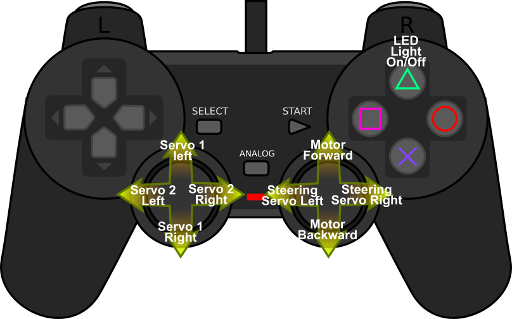
To illustrate how this works, here is an example project. It makes use of a DC motor, an electronic speed controller (in this case an RC car ESC), a series of servos, and an LED array designed to run on 7-12V. If this were to be applied to a real project, it could be a car with steering, and two extra servos to control a pan and tilt camera mount. A single LED would not need the transistor, but I have included a 12V LED array and transistor to illustrate that other high voltage and high current devices can be triggered by such a system.

The is nothing demanding about the sketch and it can run on any Arduino or Arduino clone.

Below is a diagram of the circuit and components. The ESC supplies 5.6V to the servos through its control connector. The ESC I have used has a switch on it that needs to be turned on before the motor will run and before any electricity is available to the servos. Other than that, the only components are the three resistors for the connection to the Playstation 2 controller and the LED's transistor. The capacitor is probably not necessary because the battery should be able to deliver a relatively smooth supply.

[](http://www.techmonkeybusiness.com/images/MicroControllers/PS2_Demo_Circuitv1-600dpi.jpg)

The controls we want to use are; a switch for the LED array, and the two analogue joysticks to actuate the servos and ESC. Because the ESC “speaks” Servo, we just treat it as a servo in the code.

  
*Playstation 2 controls used in this demonstration.*

Here is the sketch.

You can download the sketch from here: [**PS2Controlv0.ino**](http://www.techmonkeybusiness.com/Code/PS2Controlv0.ino)

*/\**

*PS2Controlv0.ino*

*21 September 2015*

*Hamish Trolove - www.techmonkeybusiness.com*

*This sketch illustrates the use of a Playstation 2 Controller to*

*actuate a series of servos and an Electronic Speed Controller (ESC)*

*as you might do with a vehicle. An LED light is used to illustrate*

*the use of the Playstation 2 Buttons.*

*Pin assignments are:*

*3.3V output to PS2 red pin*

*5V output to 1Kohm pull up resistors for PS2.*

*Pin D02 to PS2 brown pin (data)*

*Pin D03 to PS2 yellow pin (attention)*

*Pin D04 to PS2 orange pin (command)*

*Pin D05 to PS2 blue pin (clock)*

*Pin D06 to ESC Signal Pin*

*Pin D07 to Steering Servo Signal pin*

*Pin D08 to Servo 1 Signal pin*

*Pin D09 to Servo 2 Signal pin*

*Pin D13 to LED Transistor Base*

*The ESC servo connection supplies 5.6V to the servos.*

*The coding pulls on the PS2X library developed by Bill Porter.*

*See www.billporter.info for the latest from Bill Porter and to*

*download the library.*

*The controls used for this sketch are;*

*Right Stick - X-axis = Steering Servo left/right, Y-axis = ESC forward/backward*

*Left Stick - X-axis = Servo 2 left/right, Y-axis = Servo 1 left/right*

*Triangle = Toggle the LED*

*\*/*

#include <Servo.h> //For driving the ESCs and Servos

#include <PS2X\_lib.h> // Bill Porter's PS2 Library

PS2X ps2x; *//The PS2 Controller Class*

Servo SteeringServo; *//Create servo object representing SteeringServo*

Servo ServoN1; *//Create servo object representing Servo 1*

Servo ServoN2; *//Create servo object representing Servo 2*

Servo ESCcontrol; *//Create servo object representing ESC*

**const** int LEDpin = 13; *//green LED is on Digital pin 13*

**volatile** boolean LEDHdlts; *//LED headlights on/off toggle*

int PlyStnRStickUpDn = 0; *//Value read off the PS2 Right Stick up/down.*

int PlyStnRStickLtRt = 0; *//Value read off the PS2 Right Stick left/right*

int PlyStnLStickUpDn = 0; *//Value read off the PS2 Left Stick up/down*

int PlyStnLStickLtRt = 0; *// Value read off the PS2 Left Stick left/right*

int ESCSetting = 90; *//Setting for the ESC (degrees).*

int StrServoSetting = 90; *//Setting for the Steering Servo*

int ServoN1Setting = 90; *//Setting for the Servo 1*

int ServoN2Setting = 90; *//Setting for the Servo 2*

void setup()

{

ps2x.config\_gamepad(5,4,3,2, false, false);

*//setup pins and settings: GamePad(clock, command, attention, data, Pressures, Rumble)*

*//We have disabled the pressure sensitivity and rumble in this instance.*

pinMode(LEDpin, OUTPUT); *//Sets the LEDpin to output*

LEDHdlts = false; *//Sets the Headlights to off*

SteeringServo.attach(7);*// attaches the Steering Servo to pin 7*

ServoN1.attach(8);*// attaches the Servo 1 to pin 8*

ServoN2.attach(9);*// attaches the Servo 2 to pin 9*

ESCcontrol.attach(6,150,2250);*// attaches the ESC to pin 6*

*//The ESC attachment command above also includes the signal settings*

*//for the maximum and minimum that the ESC will recognise. This*

*//varies for different ESCs.*

*//Set all ESCs and Servos to a neutral 90 degree position*

*//this avoids the ESC trying to calibrate.*

ESCcontrol.write(90);

SteeringServo.write(90);

ServoN1.write(90);

ServoN2.write(90);

delay(5000); *//Five second delay to allow ESC and controller to*

*// fully initialise.*

}

void loop()

{

ps2x.read\_gamepad(); *//This needs to be called at least once a second*

*// to get data from the controller.*

**if**(ps2x.ButtonPressed(PSB\_GREEN)) *//Triangle pressed*

{

LEDHdlts = !LEDHdlts; *//Toggle the LED light flag*

}

*//Analogue Stick readings*

PlyStnRStickUpDn = ps2x.Analog(PSS\_RY); *//Right Stick Up and Down*

PlyStnRStickLtRt = ps2x.Analog(PSS\_RX); *//Right Stick Left and Right*

PlyStnLStickUpDn = ps2x.Analog(PSS\_LY); *//left Stick Up and Down*

PlyStnLStickLtRt = ps2x.Analog(PSS\_LX); *//Left Stick Left and Right*

*//Readings from PS2 Controller Sticks are from 0 to 255*

*//with the neutral being 128. The zero positions are to*

*//the left for X-axis movements and up for Y-axis movements.*

*//Variables to carry the settings for the ESCs and Servos*

*//The values from the PS2 Sticks are mapped to 0 to 180 degrees*

ESCSetting = map(PlyStnRStickUpDn,-256,256,0,179);

StrServoSetting = map(PlyStnRStickLtRt,-256,256,0,179);

ServoN1Setting = map(PlyStnLStickUpDn,-256,256,0,179);

ServoN2Setting = map(PlyStnLStickLtRt,-256,256,0,179);

*//Write it to the Servos or ESCs*

ESCcontrol.write(ESCSetting);

SteeringServo.write(StrServoSetting);

ServoN1.write(ServoN1Setting);

ServoN2.write(ServoN2Setting);

digitalWrite(LEDpin,LEDHdlts); *//Light the LED based on headlights status flag*

delay(15);

}

All things going well, when you connect this up and connect in the battery, you should be able to push the triangle button to light the LED and move the joysticks to move the servos and run the motor.

If it does not automatically go into **Analogue Mode** (as indicated by the *red light* on the controller), press the **Analog button**. This will enable the joysticks on the controller and set things going.

The circuit and sketch are designed with a car ESC in mind. This means a servo command of *90 degrees* is the *neutral position*. The ESCcontrol.write(90); line in the *Setup* routine sets the throttle to neutral to allow the ESC to initialise. **If you are using a Radio Control Aircraft ESC you will need to initialise it at 0 degrees**.

Depending on the quality of the Arduino you are using you may need to have the USB connected to the computer to power the Arduino. Ordinarily the Arduino should be able to run on the 7.2v being supplied from the NiCd battery. If you have an 11.1V LiPo battery, and ESC designed to suit it then you should not have any problems.

Final Remark

With an input device as sophisticated as the Playstation 2 controller there are a wealth of different mechatronics projects that they can interface with. The PS2X library makes it very easy to access the full range of functions available from the Playstation 2 controller. Why reinvent the wheel when there are already ultra-high performance mags available for cheaper prices than a couple of components?

Serial.available()

Description

Get the number of bytes (characters) available for reading from the serial port. This is data that’s already arrived and stored in the serial receive buffer (which holds 64 bytes). available() inherits from the Stream utility class.

Syntax

Serial.available()

*Arduino Mega only:*

Serial1.available()  
Serial2.available()  
Serial3.available()

Parameters

None

Returns

The number of bytes available to read .

Example Code

The following code returns a character received through the serial port.

int incomingByte = 0; // for incoming serial data

void setup() {

Serial.begin(9600); // opens serial port, sets data rate to 9600 bps

}

void loop() {

// reply only when you receive data:

if (Serial.available() > 0) {

// read the incoming byte:

incomingByte = Serial.read();

// say what you got:

Serial.print("I received: ");

Serial.println(incomingByte, DEC);

}

}

# Serial.availableForWrite()

### Description

Get the number of bytes (characters) available for writing in the serial buffer without blocking the write operation.

### Syntax

Serial.availableForWrite()

Arduino Mega only:

Serial1.availableForWrite()  
Serial2.availableForWrite()  
Serial3.availableForWrite()

### Parameters

Nothing

### Returns

The number of bytes available to write.

# Serial.begin()

### Description

Sets the data rate in bits per second (baud) for serial data transmission. For communicating with the computer, use one of these rates: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. You can, however, specify other rates - for example, to communicate over pins 0 and 1 with a component that requires a particular baud rate.

An optional second argument configures the data, parity, and stop bits. The default is 8 data bits, no parity, one stop bit.

### Syntax

Serial.begin(speed) Serial.begin(speed, config)

Arduino Mega only:

Serial1.begin(speed)  
Serial2.begin(speed)  
Serial3.begin(speed)  
Serial1.begin(speed, config)  
Serial2.begin(speed, config)  
Serial3.begin(speed, config)

### Parameters

speed: in bits per second (baud) - long

config: sets data, parity, and stop bits. Valid values are

SERIAL\_5N1  
SERIAL\_6N1  
SERIAL\_7N1  
SERIAL\_8N1 (the default)  
SERIAL\_5N2  
SERIAL\_6N2  
SERIAL\_7N2  
SERIAL\_8N2  
SERIAL\_5E1  
SERIAL\_6E1  
SERIAL\_7E1  
SERIAL\_8E1  
SERIAL\_5E2  
SERIAL\_6E2  
SERIAL\_7E2  
SERIAL\_8E2  
SERIAL\_5O1  
SERIAL\_6O1  
SERIAL\_7O1  
SERIAL\_8O1  
SERIAL\_5O2  
SERIAL\_6O2  
SERIAL\_7O2  
SERIAL\_8O2

### Returns

Nothing

### Example Code

void setup() {

Serial.begin(9600); // opens serial port, sets data rate to 9600 bps

# }Serial.end()

### Description

Disables serial communication, allowing the RX and TX pins to be used for general input and output. To re-enable serial communication, call [Serial.begin()](https://www.arduino.cc/reference/en/language/functions/communication/serial/begin).

### Syntax

Serial.end()

Arduino Mega only:

Serial1.end()  
Serial2.end()  
Serial3.end()

### Parameters

Nothing

### Returns

Nothing

void loop() {}

# Serial.serialEvent()

### Description

Called when data is available. Use Serial.read() to capture this data.

NB : Currently, serialEvent() is not compatible with the Esplora, Leonardo, or Micro

### Syntax

void serialEvent(){

//statements

}

Arduino Mega only:

void serialEvent1(){

//statements

}

void serialEvent2(){

//statements

}

void serialEvent3(){

//statements

}

### Parameters

statements: any valid statements

### Returns

Nothing

# Serial.print()

### Description

Prints data to the serial port as human-readable ASCII text. This command can take many forms. Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is. For example-

* Serial.print(78) gives "78"
* Serial.print(1.23456) gives "1.23"
* Serial.print('N') gives "N"
* `Serial.print("Hello world.") gives "Hello world." `

# Serial.write()

### Description

Writes binary data to the serial port. This data is sent as a byte or series of bytes; to send the characters representing the digits of a number use the [print()](https://www.arduino.cc/reference/en/language/functions/communication/serial/print) function instead.

### Syntax

Serial.write(val)  
Serial.write(str)  
Serial.write(buf, len)

Arduino Mega also supports:

Serial1, Serial2, Serial3 (in place of Serial)

### Parameters

val: a value to send as a single byte

str: a string to send as a series of bytes

buf: an array to send as a series of bytes

### Returns

size\_t

write() will return the number of bytes written, though reading that number is optional

### Example Code

void setup(){

Serial.begin(9600);

}

void loop(){

Serial.write(45); // send a byte with the value 45

int bytesSent = Serial.write(“hello”); //send the string “hello” and return the length of the string.

}