# Analog to Digital Converter

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.

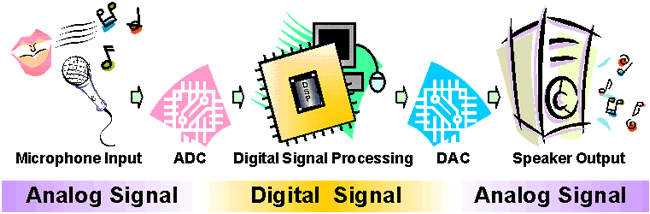


Figure 1: Audio Signal Processing

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.

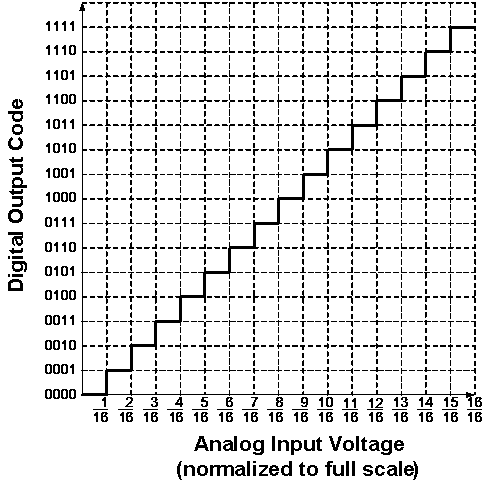


Figure 2: Ideal Transfer Curve of a 4-bit ADC

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.

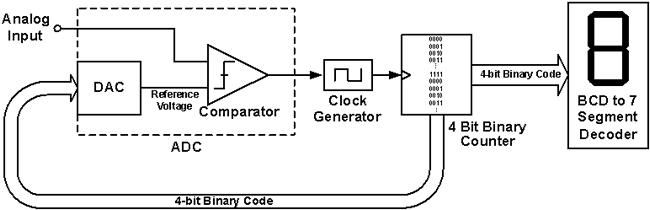


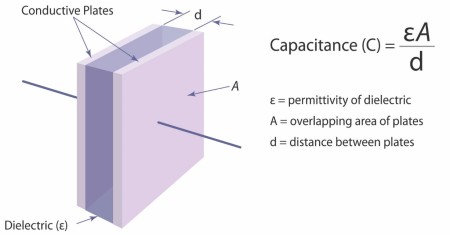
Figure 1: Block diagram of the 1-Digit Voltmeter

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.

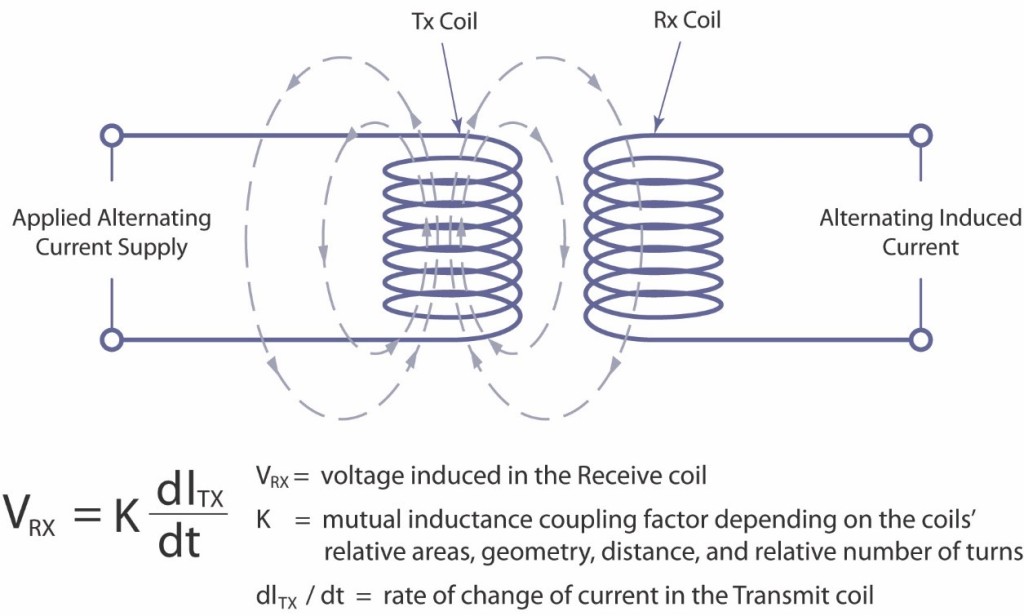
Unfortunately, capacitance is also sensitive to factors other than displacement. If the capacitor’s plates are surrounded by air then its permittivity will also vary with temperature and humidity, because water has a different dielectric constant to air. A nearby object which varies the permittivity 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**

1.HC- SR04 ULTRASONIC SENSOR

2.JUMPER WIRES(4 male-male)

3.BREADBOARD

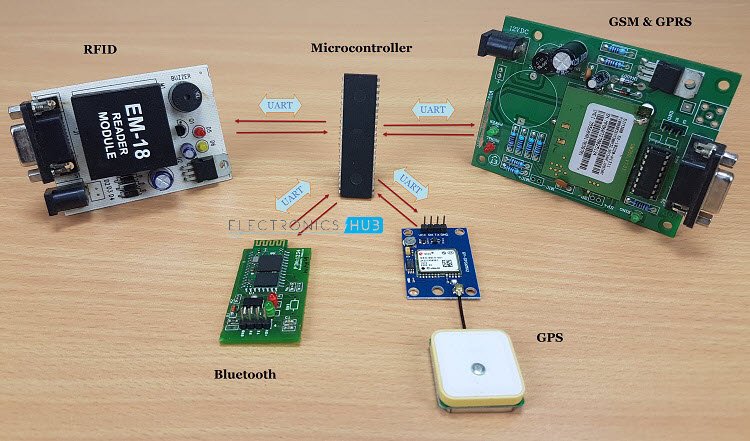
4.ARDUINO WITH USB CABLE(ANY MODEL)

# 

# Basics of UART Communication

UART or Universal Asynchronous Receiver Transmitter is a dedicated hardware associated with serial communication. The hardware for UART can be a circuit integrated on the microcontroller or a dedicated IC. This is contrast to SPI or I2C, which are just communication protocols.

UART is one of the most simple and most commonly used Serial Communication techniques. Today, UART is being used in many applications like GPS Receivers, Bluetooth Modules, GSM and GPRS Modems, Wireless Communication Systems, RFID based applications etc.



If you remember older computer systems, devices like Mouse, Printer and Modem are connected using a heavy connectors at the back. All these devices communicated using UART.

Even though USB has replaced all types of communication standards on computers and other devices, UART is still being used in the above mentioned applications.

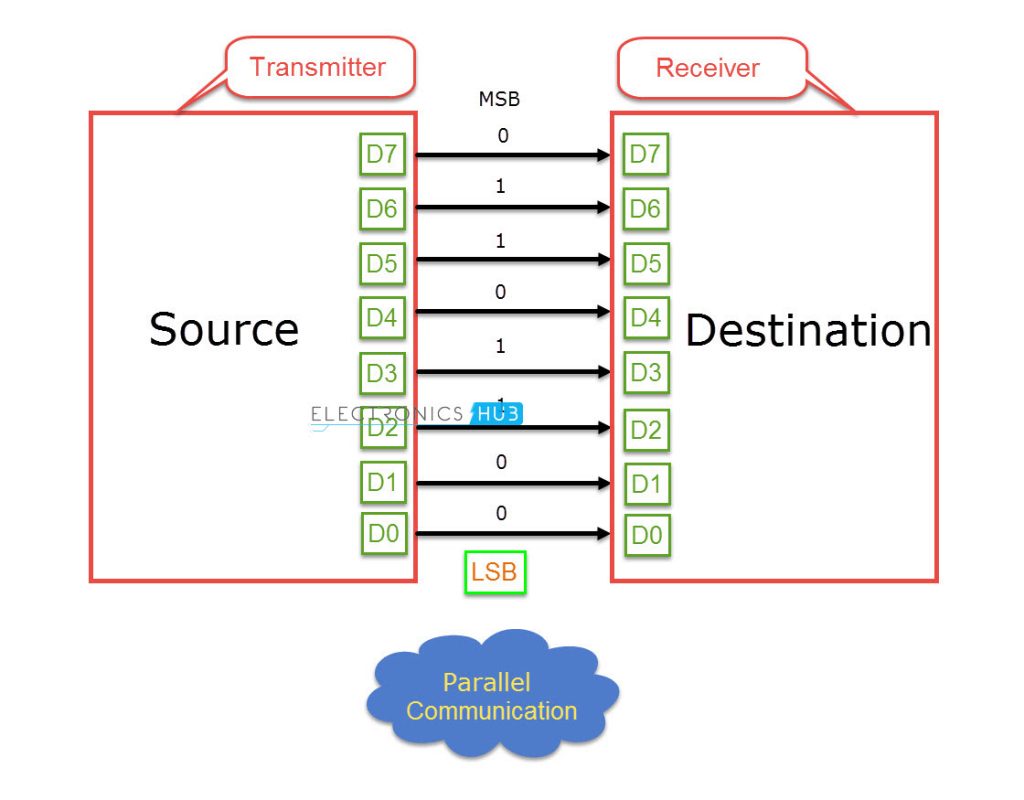
Almost all microcontrollers have dedicated UART hardware built in to their architecture. The main reason for integrating the UART hardware in to microcontrollers is that it is a serial communication and requires only two wires for communication.

Before going in to further explanation of UART, how it works and steps involved in transmission and reception of data, we will try to recollect some information about Serial Communication and a small comparison between serial and parallel communications.

### Brief Note on Parallel and Serial Communication

Transfer of Digital Data from one device to another can be achieved in two ways: Parallel Data Transfer and Serial Data Transfer. In parallel data transfer, all the bits are transferred from the source to destination at once.

This is possible because parallel data transfer uses multiple lanes or wires between the transmitter and receiver in order to transfer the data.

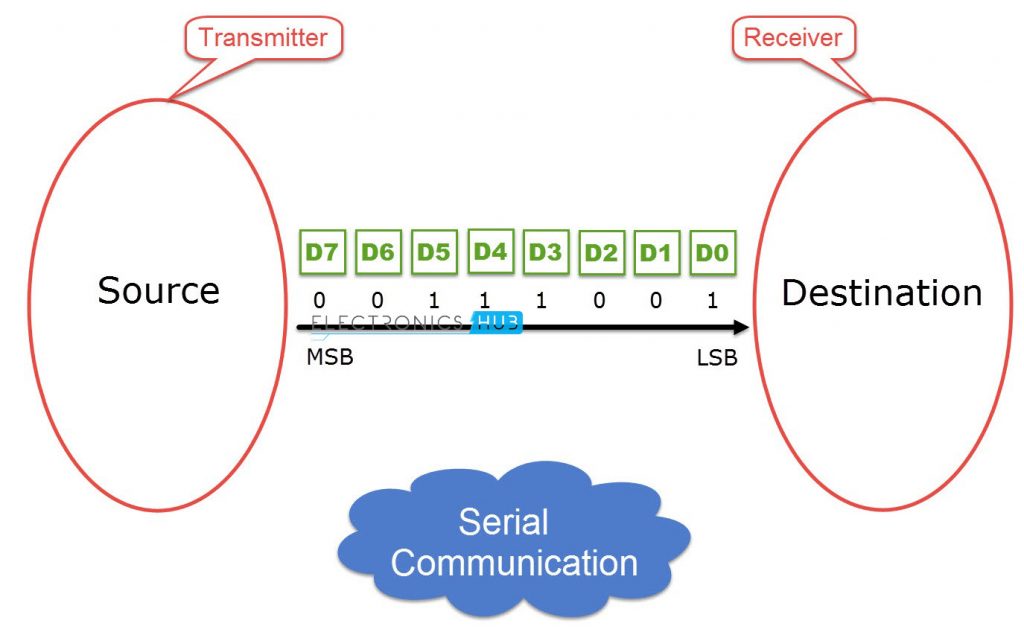
[](https://www.electronicshub.org/wp-content/uploads/2017/06/Parallel-Communication.jpg)

Parallel Data Transfer methods are faster and expensive as they needs more hardware and a lot of wires. Olden day’s printers are the best example for external parallel communication. Other examples are RAM, PCI, etc.

With the progress in integrated circuit technology, the digital IC’s are becoming smaller and faster and as a result the transfer rates in Parallel Communication with multiple lanes have reached a bottle neck.

Serial Communication on the hand, transfers data bit by bit using a single line or wire. For two way communication between the transmitter and receiver, we need just two wires for successful serial data transfer.

Since serial communication needs less circuitry and wires, the cost of implementing is less. As a result, using serial communication in complex circuitry might be more practical than parallel communication.

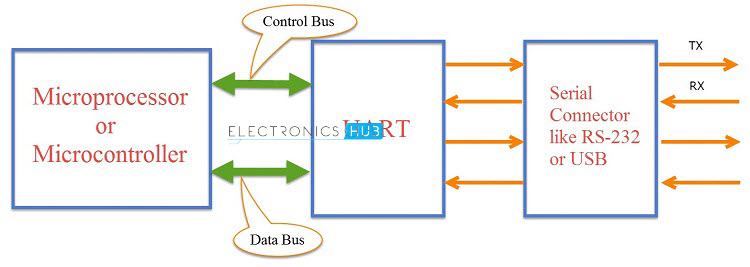
[](https://www.electronicshub.org/wp-content/uploads/2017/06/Serial-Communication.jpg)

But the only concern with serial data transfers is speed. Since the data transfer occurs over a single line, the speed of transfer in serial communication is less than that of parallel communication. Now – a – days, the speed of serial data transfer isn’t a concern as advancements in technology have led to faster transfer speeds.

### Introduction to UART Communication

UART or Universal Asynchronous Receiver Transmitter is a serial communication device that performs parallel – to – serial data conversion at the transmitter side and serial – to – parallel data conversion at the receiver side. It is universal because the parameters like transfer speed, data speed, etc. are configurable.

As mentioned in the introduction section, UART is a piece of hardware that acts as a bridge between the processor and the serial communication protocol or port. The following image shows this interface briefly. The serial communication can be anything like USB, RS – 232, etc.

[](https://www.electronicshub.org/wp-content/uploads/2017/06/UART-Interface.jpg)  
The letter ‘A’ in UART stands for Asynchronous i.e. there is no clock signal to synchronize or validate the data transmitted from transmitter and received by the receiver (Asynchronous Serial Communication).

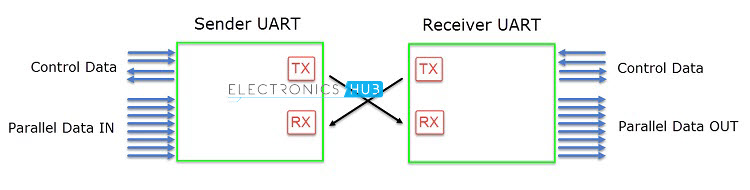
This is in contrast to Synchronous Serial Communication, which uses a clock signal that is shared between the transmitter and receiver in order to “Synchronize” the data between them.

If there is no clock (or any other timing signal) between the transmitter and receiver, then how does the receiver know when to read the data?

In UART, the transmitter and receiver must agree on timing parameters beforehand. Also, UART uses special bits at the beginning and ending of each data word to synchronize the transmitter and receiver. More about these special bits in the later sections.

In UART based Serial Communication, the transmitter and receiver communicate in the following manner. The UART on the sender device i.e. the transmitting UART receives parallel data from the CPU (microprocessor or microcontroller) and converts it in to serial data.

This serial data is transmitted to the UART on the receiver device i.e. receiving UART. The receiving UART, upon receiving the serial data, converts it back to parallel data and gives it to the CPU.

[](https://www.electronicshub.org/wp-content/uploads/2017/06/UART-Basic-Connection-1.jpg)

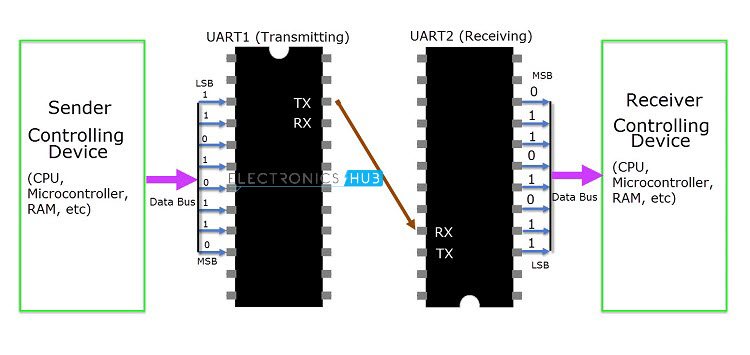
The pin on the transmitting UART, which transmits the serial data is called TX and the pin on the receiving UART, which receives the serial data is called RX.

Since the UART involves parallel – to – serial and serial – to – parallel data conversion, shift registers are an essential part of the UART hardware (two shift registers to be specific: Transmitter Shift Register and Receiver Shift Register).

**How UART Works?**

In UART Serial Communication, the data is transmitted asynchronously i.e. there is no clock or other timing signal involved between the sender and receiver. Instead of clock signal, UART uses some special bits called Start and Stop bits.

These bits are added to the actual data packet at the beginning and end respectively. These additional bits allows the receiving UART to identify the actual data.

[](https://www.electronicshub.org/wp-content/uploads/2017/06/UART-Working.jpg)

The image above shows a typical UART connection. The transmitting UART receives data from the controlling device through the data bus. The controlling device can be anything like a CPU of a microprocessor or a microcontroller, memory unit like a RAM or ROM, etc. The data received by the transmitting UART from the data bus is parallel data.

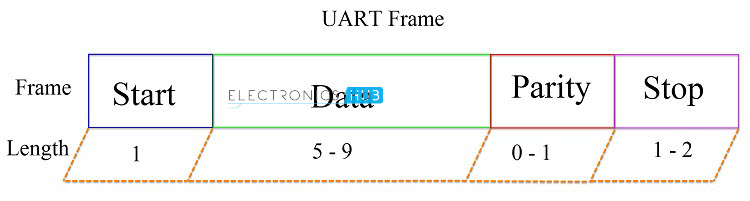
To this data, the UART adds Start, Parity and Stop bits in order to convert it into a data packet. The data packet is then converted from parallel to serial with the help of shift register and is transmitted bit – by – bit from the TX pin.

The receiving UART receives this serial data at the RX pin and detects the actual data by identifying the start and stop bits. Parity bit is used to check the integrity of the data.

Up on separating the start, parity and stop bits from the data packet, the data is converted to parallel data with the help of shift register. This parallel data is sent to the controller at the receiving end through a data bus.

**Structure of Data Packet or Frame**

The data in UART serial communication is organised in to blocks called Packets or Frames. The structure of a typical UART Data Packet or the standard framing of the data is shown in the following image.

[](https://www.electronicshub.org/wp-content/uploads/2017/06/UART-Frame.jpg)

Let us see about each piece of the frame.

**Start Bit:** Start bit is a synchronisation bit that is added before the actual data. Start bit marks the beginning of the data packet. Usually, an idle data line i.e. when the data transmission line is not transmitting any data, it is held at a high voltage level (1).

In order to start the data transfer, the transmitting UART pulls the data line from high voltage level to low voltage level (from 1 to 0). The receiving UART detects this change from high to low on the data line and begins reading the actual data. Usually, there is only one start bit.

**Stop Bit:** The Stop Bit, as the name suggests, marks the end of the data packet. It is usually two bits long but often only on bit is used. In order to end the transmission, the UART maintains the data line at high voltage (1).

**Parity Bit:** Parity allows the receiver to check whether the received data is correct or not. Parity is a low – level error checking system and comes in two varieties: Even Parity and Odd Parity. Parity bit is optional and it is actually not that widely used.

**Data Bits:** Data bits are the actual data being transmitted from sender to receiver. The length of the data frame can be anywhere between 5 and 9 (9 bits if parity is not used and only 8 bits if parity is used). Usually, the LSB is the first bit of data to be transmitted (unless otherwise specified).

**Rules of UART**

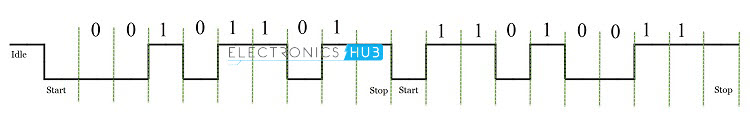
As mentioned earlier, there is no clock signal in UART and the transmitter and receiver must agree on some rules of serial communication for error free transfer of data. The rules include:

* Synchronisation Bits (Start and Stop bits)
* Parity Bit
* Data Bits and
* Baud Rate

We have seen about synchronisation bits, parity bit and data bits. Another important parameter is the Baud Rate.  
Baud Rate: The speed at which the data is transmitted is mentioned using Baud Rate. Both the transmitting UART and Receiving UART must agree on the Baud Rate for a successful data transmission.

Baud Rate is measured in bits per second. Some of the standard baud rates are 4800 bps, 9600 bps, 19200 bps, 115200 bps etc. Out of these 9600 bps baud rate is the most commonly used one.

Let us see an example data frame where two blocks of data i.e. 00101101 and 11010011 must be transmitted. The format of the frame is 9600 8N1 i.e. 9600 bps with 8 bits of data, no parity and 1 stop bit. In this example, we haven’t used the parity bit.

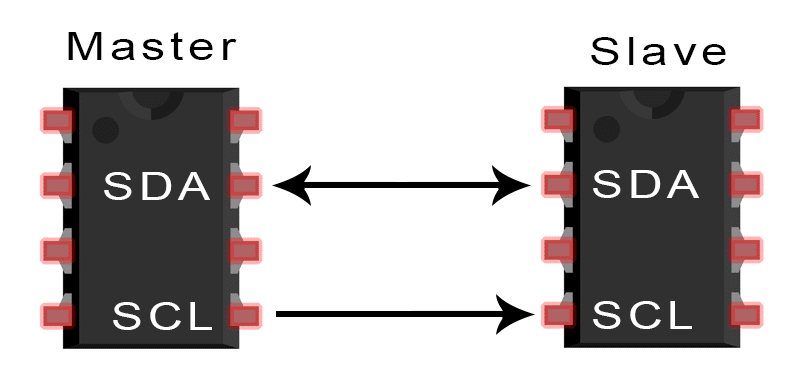
**[](https://www.electronicshub.org/wp-content/uploads/2017/06/UART-Example.jpg)  
Advantages of UART**

* Requires only two wires for full duplex data transmission (apart from the power lines).
* No need for clock or any other timing signal.
* Parity bit ensures basic error checking is integrated in to the data packet frame.

**Disadvantages of UART**

* Size of the data in the frame is limited.
* Speed for data transfer is less compared to parallel communication.
* Transmitter and receiver must agree to the rules of transmission and appropriate baud rate must be selected.

# BASICS OF THE I2C COMMUNICATION PROTOCOL

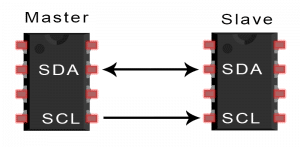


So far, we’ve talked about the basics of [SPI communication](http://www.circuitbasics.com/basics-of-the-spi-communication-protocol) and [UART communication](http://www.circuitbasics.com/basics-uart-communication/), so now let’s go into the final protocol of this series, the Inter-Integrated Circuit, or I2C.

## INTRODUCTION TO I2C COMMUNICATION

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.

Like UART communication, I2C only uses two wires to transmit data between devices:

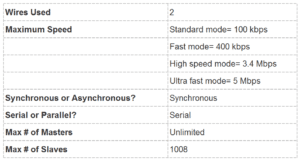


**SDA (Serial Data)** – The line for the master and slave to send and receive data.

**SCL (Serial Clock)** – The line that carries the clock signal.

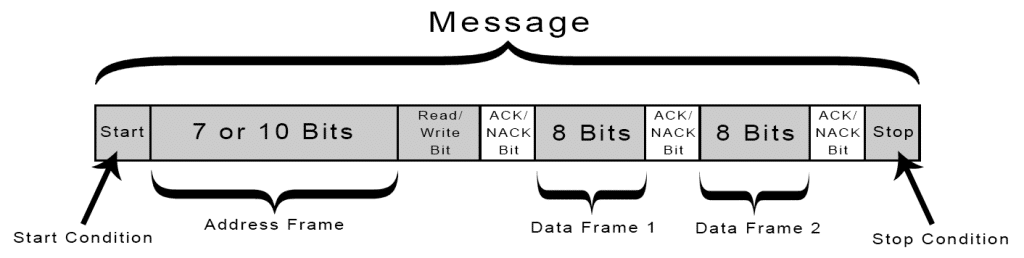
I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line).

Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

[](http://www.circuitbasics.com/wp-content/uploads/2016/02/Basics-of-the-I2C-Communication-Protocol-Specifications-Table.png)

## HOW I2C WORKS

With I2C, data is transferred in messages. Messages are broken up into frames of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:

[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Message-Frame-and-Bit-2.png)

**Start Condition:** The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

**Stop Condition:** The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

**Address Frame:** A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

**Read/Write Bit:**A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

**ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

### ADDRESSING

I2C doesn’t have slave select lines like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by addressing. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn’t match, the slave does nothing and the SDA line remains high.

### READ/WRITE BIT

The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

### THE DATA FRAME

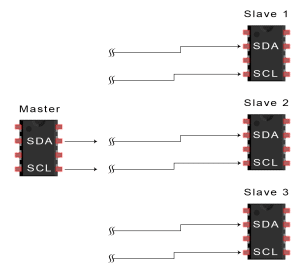
After the master detects the ACK bit from the slave, the first data frame is ready to be sent.

The data frame is always 8 bits long, and sent with the most significant bit first. Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully. The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent.

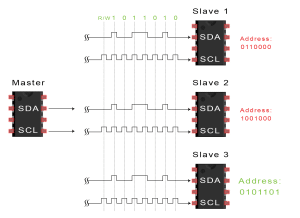
After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

## STEPS OF I2C DATA TRANSMISSION

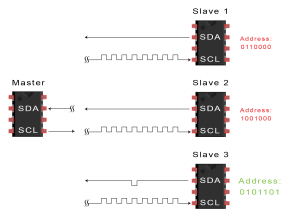
1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level before switching the SCL line from high to low:

[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Data-Transmission-Diagram-START-CONDITION-3.png)

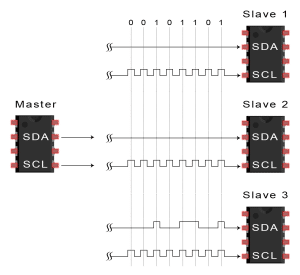
2. The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit:

[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Data-Transmission-Diagram-ADDRESS-FRAME-2.png)

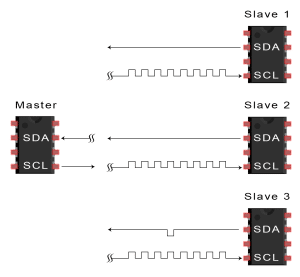
3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave’s own address, the slave leaves the SDA line high.



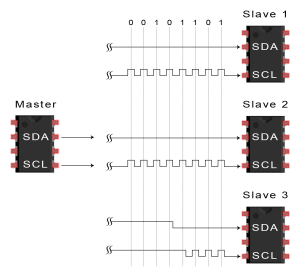
4. The master sends or receives the data frame:

[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Data-Transmission-Diagram-Data-Frame.png)

5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:

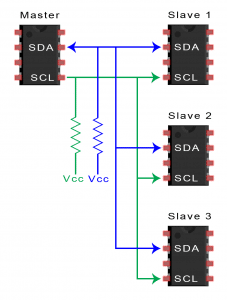
[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Data-Transmission-Diagram-ACK-bit-slave-to-master-2A.png)

6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:

[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Data-Transmission-Diagram-Stop-Condition.png)

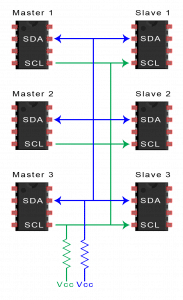
## SINGLE MASTER WITH MULTIPLE SLAVES

Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7 bit address, 128 (27) unique address are available. Using 10 bit addresses is uncommon, but provides 1,024 (210) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:

[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Single-Master-Multiple-Slaves-2.png)

## MULTIPLE MASTERS WITH MULTIPLE SLAVES

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line. To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it’s safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:

[](http://www.circuitbasics.com/wp-content/uploads/2016/01/Introduction-to-I2C-Multiple-Masters-Multiple-Slaves-2.png)

## ADVANTAGES AND DISADVANTAGES OF I2C

There is a lot to I2C that might make it sound complicated compared to other protocols, but there are some good reasons why you may or may not want to use I2C to connect to a particular device:

**ADVANTAGES**

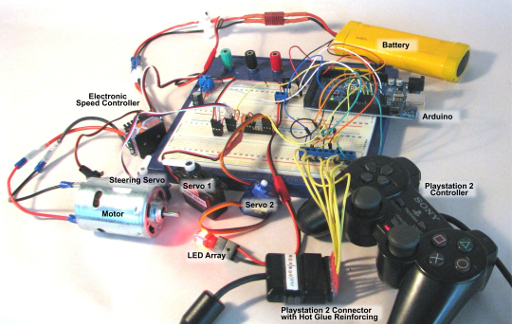
* Only uses two wires
* Supports multiple masters and multiple slaves
* ACK/NACK bit gives confirmation that each frame is transferred successfully
* Hardware is less complicated than with UARTs
* Well known and widely used protocol

**DISADVANTAGES**

* Slower data transfer rate than SPI
* The size of the data frame is limited to 8 bits
* More complicated hardware needed to implement than SPI

**Ps2 controller 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?

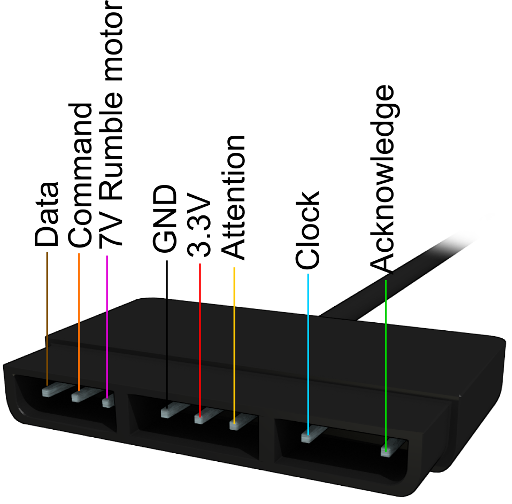
* Controlling a wheeled or tracked vehicle's motion.
* Controlling a robotic arm.
* Interacting with a computer.
* Controlling a pan and tilt camera mount.
* DIY radio control system by using a bluetooth module, or radio module to transmit the commands from the Playstation 2 controller to another Arduino 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.

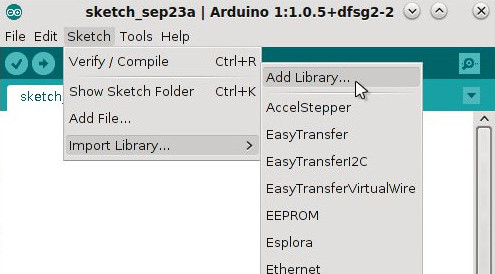
  
*Sources of Playstation 2 Sockets*

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;

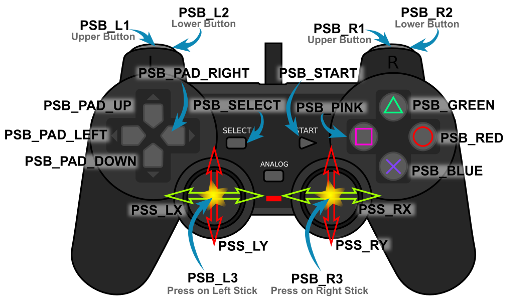
*gamepad(clock, command, attention, data, pressure sensitivity enabled, rumble enabled)*

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.

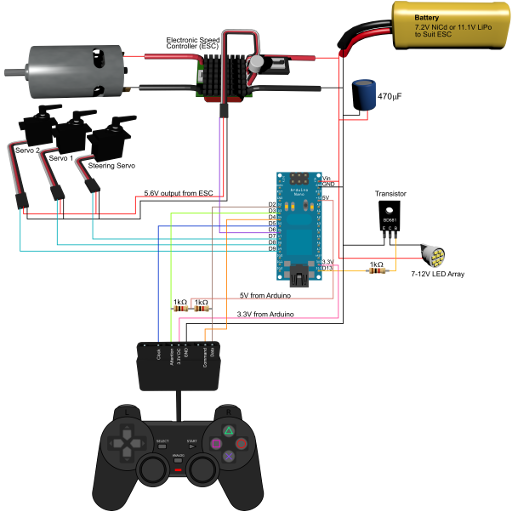
* ps2x.ButtonPressed(PSB\_RED) is for a simple press of the **red circle** button.
* ps2x.Button(PSB\_PAD\_DOWN) is for the **down button on the pad** being pressed and held.
* ps2x.Analog(PSAB\_CROSS) is the command for measuring the pressure applied to the “**X**” button, if pressure sensitivity has 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

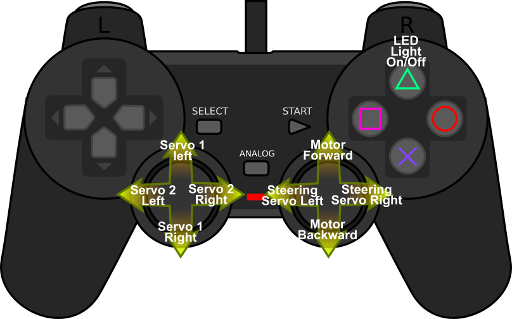
To illustrate how this works, here is an example project. It makes use of a DC motor, an electronic speed controller (in this case an RC car ESC), a series of servos, and an LED array designed to run on 7-12V. If this were to be applied to a real project, it could be a car with steering, and two extra 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.*

**Serial Monitor Communication.**

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.

}