**Serial Communication**

Serial communication works on 1s and 0s. Also known as binary, the Arduino sends these 1s and 0s (bits) one by one, or Serially. These bits are sent in the form of Highs(1) and Lows(0). These bits form together and turn into bytes. A byte is composed of 8 bits. Here's a little tutorial on binary:  
Each bit represents a certain number to add. the first bit (Far right) represents the ones place, similar to the common decimal system.  
1 = 1 and 0 = 0.  
the next bit's value is multiplied by two, so:  
10 = 2, 11 = 3, 00 = 0 and 01 = 1.  
  
similarly:  
100 = 4,  
1000 = 8,  
10000 = 16,  
100000 = 32,  
1000000 = 64  
10000000 = 128.  
  
Now these bits can form a value up to 255 (11111111).  
  
This value can be turned into ASCII encoded symbols and letters.

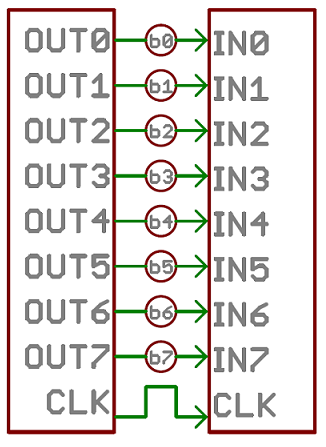
Serial Print is a function that prints whatever you specify on the Serial Monitor  
In the void setup, we begin the serial monitor at a baud rate of 9600.  
  
The baud rate is the rate at which information is transferred in a communication channel. In the serial port context, "9600 baud" means that the serial port is capable of transferring a maximum of 9600 bits per second.

At higher bad rates, above 76,800, the cable length will need to be reduced.  The higher the baud rate, the more sensitive the cable become to quality of installation, such as how much of the wire is untwisted around each device.

Serial communication can be of two types i.e Serial or Parallel Communication

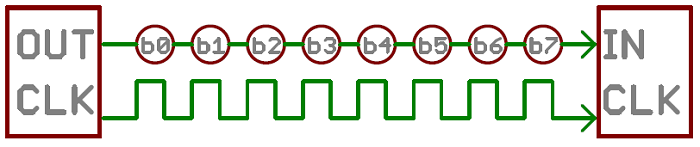
Parallel Serial Communication:

Parallel interfaces transfer multiple bits at the same time. They usually require **buses** of data - transmitting across eight, sixteen, or more wires. Data is transferred in huge, crashing waves of 1’s and 0’s.



An 8-bit data bus, controlled by a clock, transmitting a byte every clock pulse. 9 wires are used.

**Serial Communication**

Serial interfaces stream their data, one single bit at a time. These interfaces can operate on as little as one wire, usually never more than four.  


Example of a serial interface, transmitting one bit every clock pulse. Just 2 wires required!

Think of the two interfaces as a stream of cars: a parallel interface would be the 8+ lane mega-highway, while a serial interface is more like a two-lane rural country road. Over a set amount of time, the mega-highway potentially gets more people to their destinations, but that rural two-laner serves its purpose and costs a fraction of the funds to build.

Parallel communication certainly has its benefits. It’s fast, straightforward, and relatively easy to implement. But it requires many more input/output (I/O) lines. If you’ve ever had to move a project from a basic Arduino Uno to a Mega, you know that the I/O lines on a microprocessor can be precious and few. So, we often opt for serial communication, sacrificing potential speed for pin real estate.

Hence we can use a serial communication for simplicity and transmitting data where less speeds can do, but if we need high speed we can opt for parallel communication for such purpose and speed data transfer.

**More into Baud Rate**

The baud rate specifies **how fast** data is sent over a serial line. It’s usually expressed in units of bits-per-second (bps). If you invert the baud rate, you can find out just how long it takes to transmit a single bit. This value determines how long the transmitter holds a serial line high/low or at what period the receiving device samples its line.

Baud rates can be just about any value within reason. The only requirement is that both devices operate at the same rate. One of the more common baud rates, especially for simple stuff where speed isn’t critical, is **9600 bps**. Other “standard” baud are 1200, 2400, 4800, 19200, 38400, 57600, and 115200.

The higher a baud rate goes, the faster data is sent/received, but there are limits to how fast data can be transferred. You usually won’t see speeds exceeding 115200 - that’s fast for most microcontrollers. Get too high, and you’ll begin to see errors on the receiving end, as clocks and sampling periods just can’t keep up.

**WE GENERALLY USE SERIAL COMMUNICATION PROTOCOL TO CONNECT TWO MICROCONTROLLERS WITH EACH OTHER**

In this we will use one of the Microcontroller as a primary and one of them a s a secondary one. Just like we do programming using classes here in the same way we will use the two microcontrollers for the processing and other part. Eg fitting of two Arduino’s in a automatic car: Primary is used for sensing the obstacles and the secondary is used to run the motors and other systems as required etc.

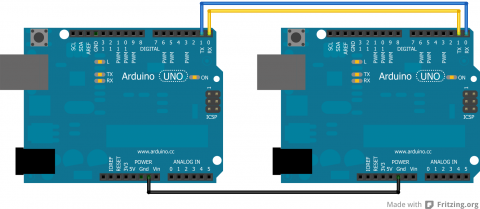
The Arduinos are interfaced using serial communication ( The baud rate of both of them must be same else it will not work, the data will be loss or the data will be manipulated if the baud rate does not match.

For doing the connections we can connect the rx and tx of the one to the respective tx and rx of the other Arduino as shown in the schematic.

**TX->RX**

**RX->TX**

**(Primary->Secondary)**



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.

Why do we need a common ground connection?

A common ground connection is required, so both devices have a common reference point to measure voltage. Since all the signals are relative hence we need to give a common reference point to both of them so that the data is transmitted and received by the other has the same meaning.

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);  
 }  
}

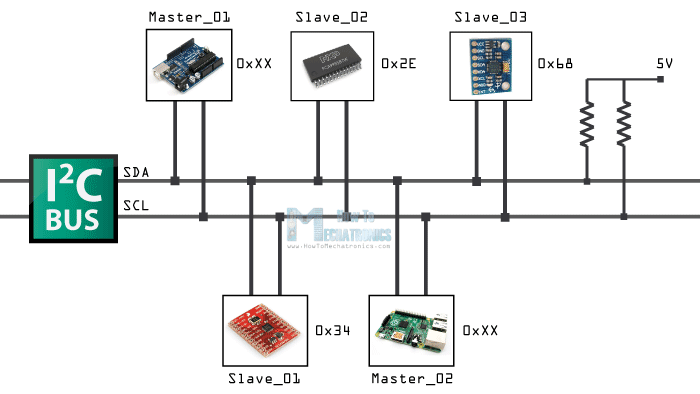
There is one flaw with this program as it is now. It results in only a character array, so if the integers were desired, they are lost without further work on the data.

**I2C COMMUNICATION:**

The Inter-integrated Circuit (I2C) Protocol is a protocol intended to allow multiple “slave” digital integrated circuits (“chips”) to communicate with one or more “master” chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.

The two wires, or lines are called Serial Clock (or SCL) and Serial Data (or SDA).  The SCL line is the clock signal which synchronize the data transfer between the devices on the I2C bus and it’s generated by the master device. The other line is the SDA line which carries the data.

The two lines are “open-drain” which means that pull up resistors needs to be attached to them so that the lines are high because the devices on the I2C bus are active low. Commonly used values for the resistors are from 2K for higher speeds at about 400 kbps, to 10K for lower speed at about 100 kbps.



### **Why do we prefer I2C over Serial?**

Because serial ports are **asynchronous** (no clock data is transmitted), devices using them must agree ahead of time on a data rate. The two devices must also have clocks that are close to the same rate, and will remain so–excessive differences between clock rates on either end will cause garbled data.

Asynchronous serial ports require hardware overhead–the UART at either end is relatively complex and difficult to accurately implement in software if necessary. At least one start and stop bit is a part of each frame of data, meaning that 10 bits of transmission time are required for each 8 bits of data sent, which eats into the data rate.

Another core fault in asynchronous serial ports is that they are inherently suited to communications between two, and only two, devices. While it is *possible* to connect multiple devices to a single serial port, **bus contention** (where two devices attempt to drive the same line at the same time) is always an issue and must be dealt with carefully to prevent damage to the devices in question, usually through external hardware.

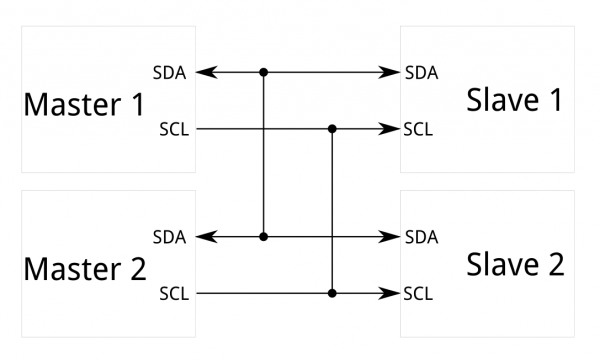
Finally, data rate is an issue. While there is no *theoretical* limit to asynchronous serial communications, most UART devices only support a certain set of fixed baud rates, and the highest of these is usually around 230400 bits per second.

**I2C requires a mere two wires**, like asynchronous serial, but those two wires can support up to 1008 slave devices. Also, unlike SPI, I2C can support a multi-master system, allowing more than one master to communicate with all devices on the bus (although the master devices can’t talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI; most I2C devices can communicate at 100kHz or 400kHz. There is some overhead with I2C; for every 8 bits of data to be sent, one extra bit of meta data (the “ACK/NACK” bit, which we’ll discuss later) must be transmitted.

The hardware required to implement I2C is more complex than SPI, but less than asynchronous serial. It can be fairly trivially implemented in software.

Each I2C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus master; some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out). This is called “clock stretching” and is described on the protocol page.

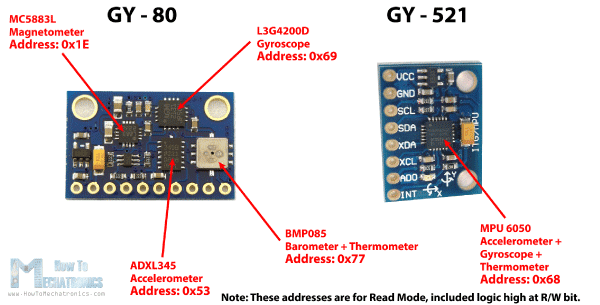


**Where do we prefer to use I2C?**

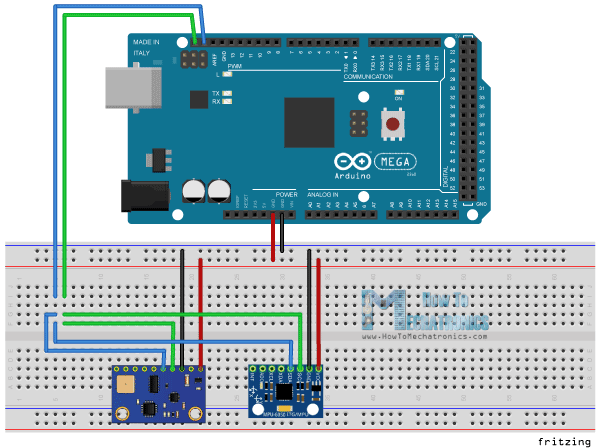
I2C provides good support for communication with various slow, on-board peripheral devices that are accessed intermittently, while being extremely modest in its hardware resource needs. It is a simple, low-bandwidth, short-distance protocol. Most available I2C devices operate at speeds up to 400Kbps, with some venturing up into the low megahertz range. I2C is easy to use to link multiple devices together since it has a built-in addressing scheme.

## I2C and Arduino

As an example I will use the GY-80 breakout board which consists 5 different sensors and the GY-521 breakout board which consists 3 different sensors. So we can get data from 8 different sensors with just two wires with the I2C bus.



Here’s how we will connect the boards. The Serial Clock pin of the Arduino Board will be connected to the Serial Clock pins of the two breakout boards, the same goes for the Serial Data pins and we will power the boards with the Gnd and the 5V pin from the Arduino Board. Note here we are not using pull-up resistors because the breakout boards already have.



Now in order to communicate with these chips or sensors we need to know their unique addresses. We can find them from the datasheets of the sensors. For the GY-80 breakout board we have the following 4 addresses: a hexadecimal 0x53 for the 3 Axis Accelerometer sensor, a hexadecimal 0x69 for the 3 Axis Gyro, a hexadecimal 0x1E for the 3 Axis Magnetometer and a hexadecimal 0x77 for the Barometer and Thermometer sensor. For the GY-521 breakout board we have only one address and that’s a hexadecimal 0x68. We can also get or check the addresses using the I2C Scanner sketch which can be found from the Arduino official website. So here if we upload and run that sketch, we will get the addresses of the connected devices on the I2C bus.

After we have found the addresses of the devices we also need to find the addresses of their internal registers in order to read the data from them. For example if we want to read the data for the X axis from the 3 Axis Accelerometer sensor of the GY-80 breakout board, we need to find the internal register address where the data of the X axis is stored. From the datasheet of the sensor, we can see that data for the X axis is actually stored in two registers, DATAX0 with a hexadecimal address 0x32 and DATAX1 with a hexadecimal address 0x33.

## Source Code

Now let’s make the code that will get the data for the X axis. So we will use the Arduino Wire Library which has to be include in the sketch.  Here first we have to define the sensor address and the two internal registers addresses that we previously found. The Wire.begin() function will initiate the Wire library and also we need to initiate the serial communication because we will use the Serial Monitor to show the data from the sensor.

In the loop() we will start with the Wire.beginTransmission() function which will begin the transmission to the particular sensor, the 3 Axis Accelerometer in our case. Then with the Wire.write() function we will ask for the particular data from the two registers of the X axis. The Wire.endTransmission() will end the transmission and transmit the data from the registers. Now with the Wire.requestFrom() function we will request the transmitted data or the two bytes from the two registers. The Wire.available() function will return the number of bytes available for retrieval and if that number match with our requested bytes, in our case 2 bytes, using the Wire.read() function we will read the bytes from the two registers of the X axis. At the end we will print the data into the serial monitor. Here’s that data but keep in mind that this is raw data and some math is needed to be done in order to get the right values of the X axis. You can find more details for that in my next tutorial for using accelerometers with the Arduino Board because I don’t want to overload this tutorial because its main goal was to explain how the I2C communication works.

1. /\*
2. \* How I2C Communication Protocol Works - Example01
3. \*
4. \* by Dejan Nedelkovski, www.HowToMechatronics.com
5. \*
6. \*/
7. #include <Wire.h>
8. **int** ADXLAddress = 0x53; // Device address in which is also included the 8th bit for selecting the mode, read in this case.
9. #define X\_Axis\_Register\_DATAX0 0x32 // Hexadecima address for the DATAX0 internal register.
10. #define X\_Axis\_Register\_DATAX1 0x33 // Hexadecima address for the DATAX1 internal register.
11. #define Power\_Register 0x2D // Power Control Register
12. **int** X0,X1,X\_out;
13. **void** setup() {
14. Wire.begin(); // Initiate the Wire library
15. Serial.begin(9600);
16. delay(100);
17. // Enable measurement
18. Wire.beginTransmission(ADXLAddress);
19. Wire.write(Power\_Register);
20. // Bit D3 High for measuring enable (0000 1000)
21. Wire.write(8);
22. Wire.endTransmission();
23. }
24. **void** loop() {
25. Wire.beginTransmission(ADXLAddress); // Begin transmission to the Sensor
26. //Ask the particular registers for data
27. Wire.write(X\_Axis\_Register\_DATAX0);
28. Wire.write(X\_Axis\_Register\_DATAX1);
29. Wire.endTransmission(); // Ends the transmission and transmits the data from the two registers
30. Wire.requestFrom(ADXLAddress,2); // Request the transmitted two bytes from the two registers
31. if(Wire.available()<=2) { //
32. X0 = Wire.read(); // Reads the data from the register
33. X1 = Wire.read();
34. }
35. Serial.print("X0= ");
36. Serial.print(X0);
37. Serial.print(" X1= ");
38. Serial.println(X1);
39. }

References and sites with good information:

<http://www.instructables.com/id/Serial-Communications-with-Arduino/>

<https://learn.sparkfun.com/tutorials/serial-communication>

<https://learn.sparkfun.com/tutorials/i2c>

<http://www.embedded.com/electronics-blogs/beginner-s-corner/4023816/Introduction-to-I2C>

<http://howtomechatronics.com/tutorials/arduino/how-i2c-communication-works-and-how-to-use-it-with-arduino/>