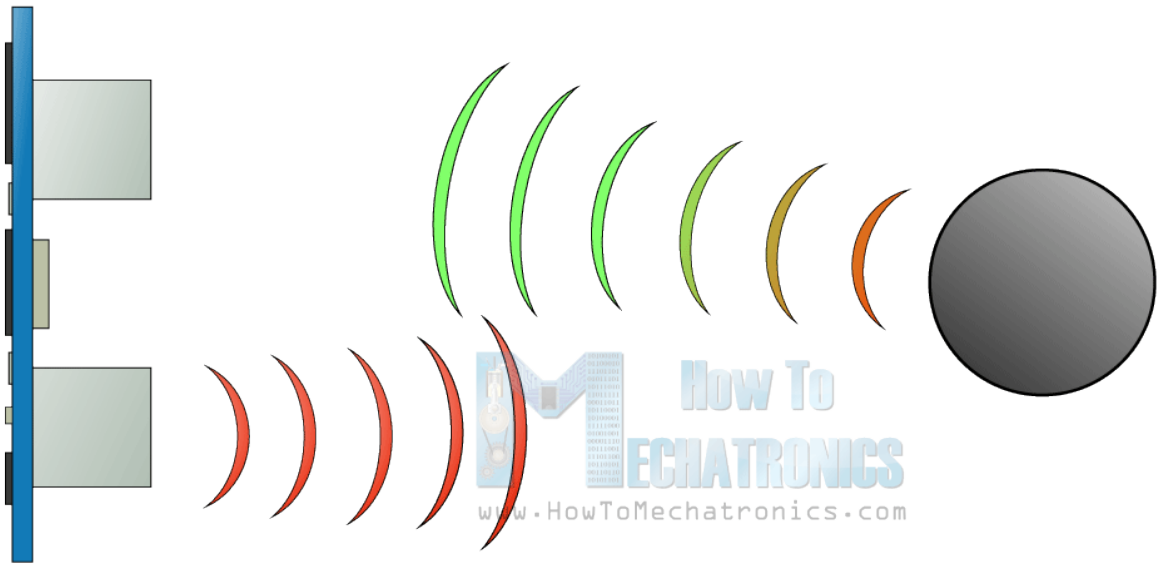
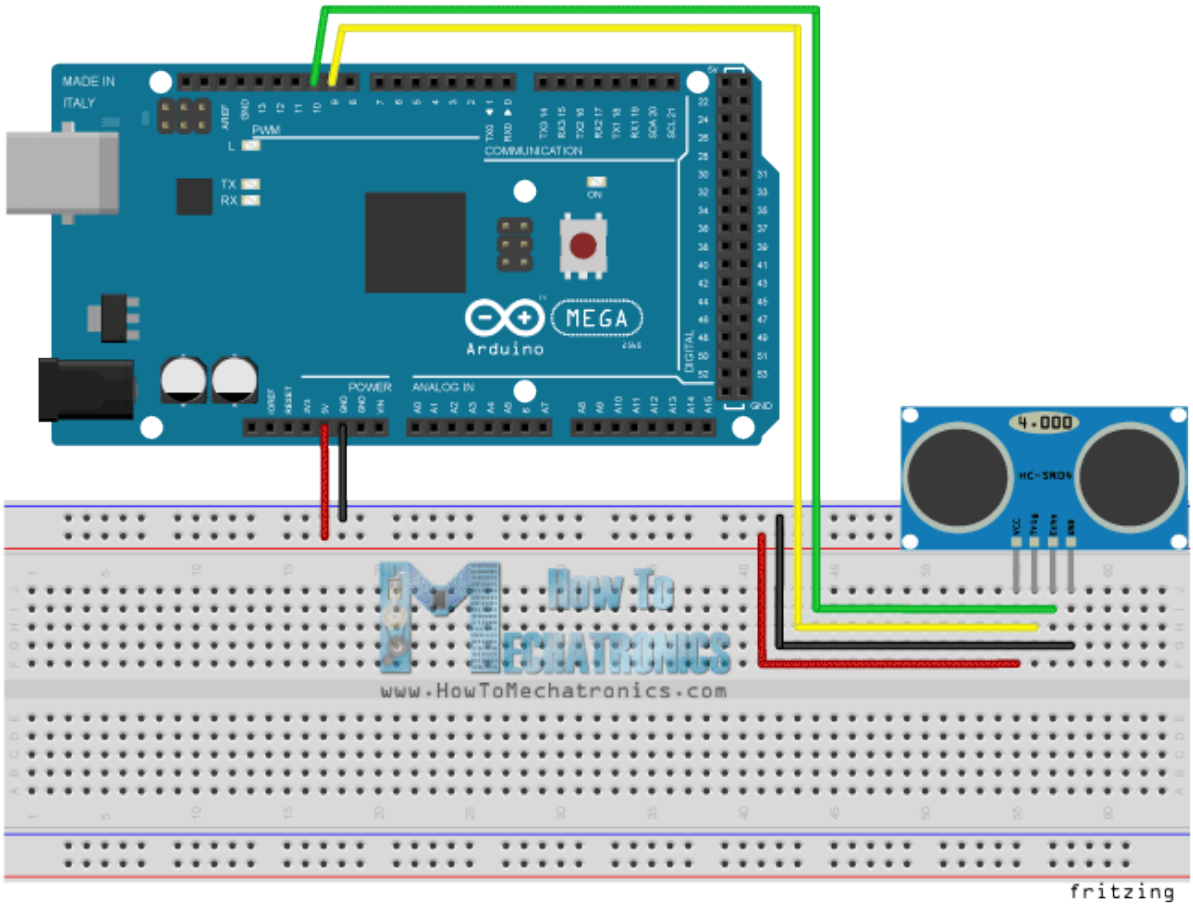
INTERFACING AN ULTRASONIC SENSOR (HCSR04)

How It Works – Ultrasonic Sensor

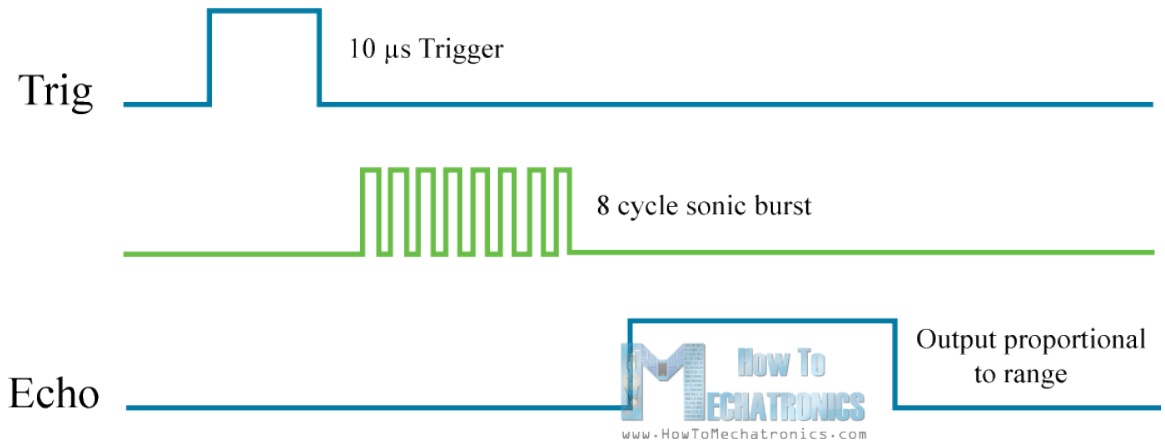
It emits an ultrasound at 40 000 Hz which travels through the air and if there is an object or obstacle on its path It will bounce back to the module. Considering the travel time and the speed of the sound you can calculate the distance.



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.

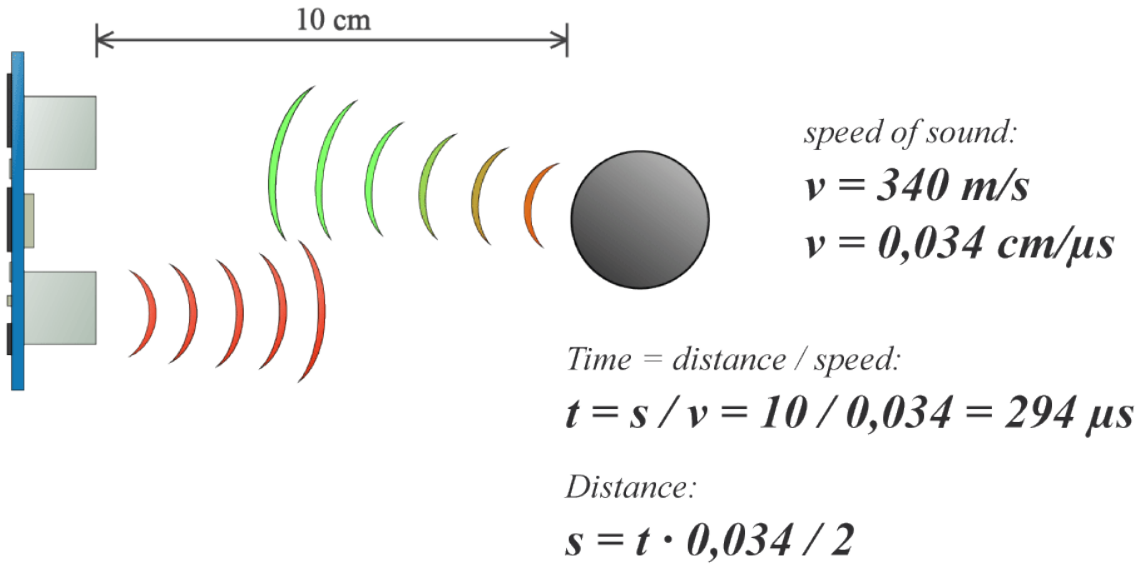


In order to generate the ultrasound you need to set the Trig on a High State for 10 µs. That will send out an 8 cycle sonic burst which will travel at the speed sound and it will be received in the Echo pin. The Echo pin will output the time in microseconds the sound wave traveled.



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 2.



Source Codes

First you have to define the Trig and Echo pins. In this case they are the pins number 9 and 10 on the Arduino Board and they are named trigPin and echoPin. Then you need a Long variable, named “duration” for the travel time that you will get from the sensor and an integer variable for the distance.

In the setup you have to define the trigPin as an output and the echoPin as an Input and also start the serial communication for showing the results on the serial monitor.

In the loop first you have to make sure that the trigPin is clear so you have to set that pin on a LOW State for just 2 µs. Now for generating the Ultra sound wave we have to set the trigPin on HIGH State for 10 µs. Using the ***pulseIn()*** function you have to read the travel time and put that value into the variable “duration”. This function has 2 parameters, the first one is the name of the echo pin and for the second one you can write either HIGH or LOW. In this case, HIGH means that the ***pulsIn()*** function will wait for the pin to go HIGH caused by the bounced sound wave and it will start timing, then it will wait for the pin to go LOW when the sound wave will end which will stop the timing. At the end the function will return the length of the pulse in microseconds. For getting the distance we will multiply the duration by 0.034 and divide it by 2 as we explained this equation previously. At the end we will print the value of the distance on the Serial Monitor.

```
// defines pins numbers

const int trigPin = 9;

const int echoPin = 10;

// defines variables

long duration;
```

```
int distance;


void setup() {
pinMode(trigPin, OUTPUT); // Sets the trigPin as an Output
pinMode(echoPin, INPUT); // Sets the echoPin as an Input
Serial.begin(9600); // Starts the serial communication
}


void loop() {
// Clears the trigPin
digitalWrite(trigPin, LOW);
delayMicroseconds(2);

// Sets the trigPin on HIGH state for 10 micro seconds
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);
digitalWrite(trigPin, LOW);

// Reads the echoPin, returns the sound wave travel time in microseconds
duration = pulseIn(echoPin, HIGH);
// Calculating the distance
distance= duration*0.034/2;
// Prints the distance on the Serial Monitor
Serial.print("Distance: ");
Serial.println(distance);
```

}

BASICS OF THE SPI COMMUNICATION PROTOCOL

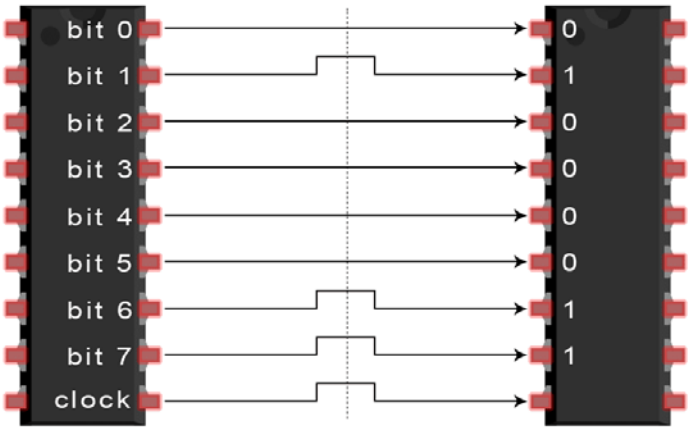
Communication between electronic devices is like communication between humans. Both sides need to speak the same language. In electronics, these languages are called *communication protocols*. Luckily for us, there are only a few communication protocols we need to know when building most DIY electronics projects. In this series of articles, we will discuss the basics of the three most common protocols: Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), and Universal Asynchronous Receiver/Transmitter (UART) driven communication.

SPI, I2C, and UART are quite a bit slower than protocols like USB, Ethernet, Bluetooth, and Wi-Fi, but they're a lot simpler and use less hardware and system resources. SPI, I2C, and UART are ideal for communication between microcontrollers and between microcontrollers and sensors where large amounts of high speed data don't need to be transferred.

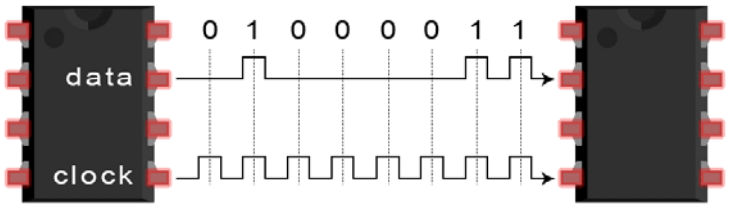
SERIAL VS. PARALLEL COMMUNICATION

Electronic devices talk to each other by sending *bits* of data through wires physically connected between devices. A bit is like a letter in a word, except instead of the 26 letters (in the English alphabet), a bit is binary and can only be a 1 or 0. Bits are transferred from one device to another by quick changes in voltage. In a system operating at 5 V, a 0 bit is communicated as a short pulse of 0 V, and a 1 bit is communicated by a short pulse of 5 V.

The bits of data can be transmitted either in parallel or serial form. In parallel communication, the bits of data are sent all at the same time, each through a separate wire. The following diagram shows the parallel transmission of the letter "C" in binary (01000011):



In serial communication, the bits are sent one by one through a single wire. The following diagram shows the serial transmission of the letter “C” in binary (01000011):

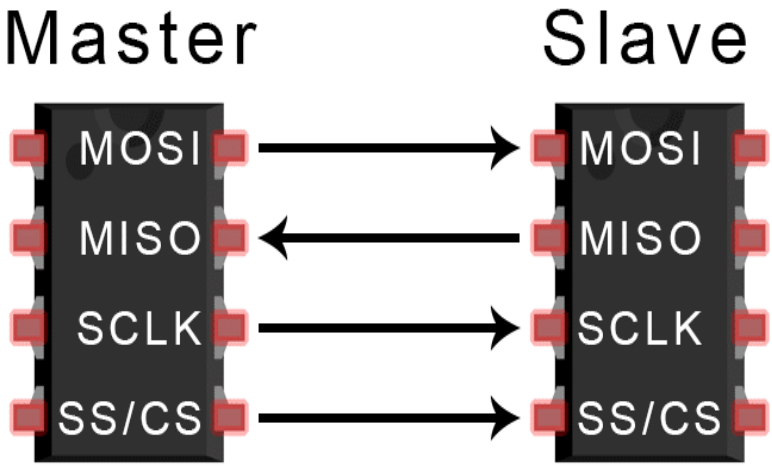


INTRODUCTION TO SPI COMMUNICATION

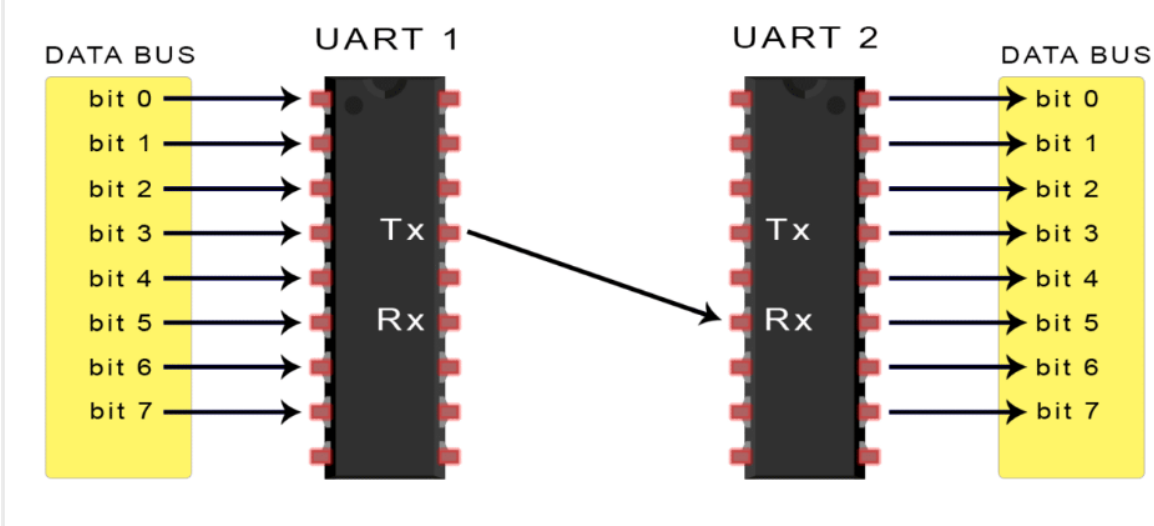
SPI is a common communication protocol used by many different devices. For example, SD card modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream. With I2C and UART, data is sent in packets, limited to a specific number of bits. Start and stop conditions define the beginning and end of each packet, so the data is interrupted during transmission.

Devices communicating via SPI are in **a master-slave relationship**. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave.



BASICS OF UART COMMUNICATION



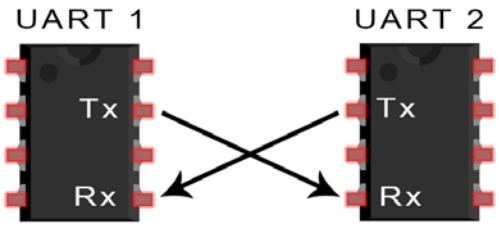
While USB has almost completely replaced those old cables and connectors, UARTs are definitely not a thing of the past. You'll find UARTs being used in many DIY electronics projects to connect GPS modules, Bluetooth modules, and RFID card reader modules to your Raspberry Pi, Arduino, or other microcontrollers.

UART stands for **Universal Asynchronous Receiver/Transmitter**. It's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART's main purpose is to transmit and receive serial data. One of the best things about UART is that it only uses two wires to transmit data between devices.

INTRODUCTION TO UART COMMUNICATION

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only

two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:



UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits. When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate.* Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.

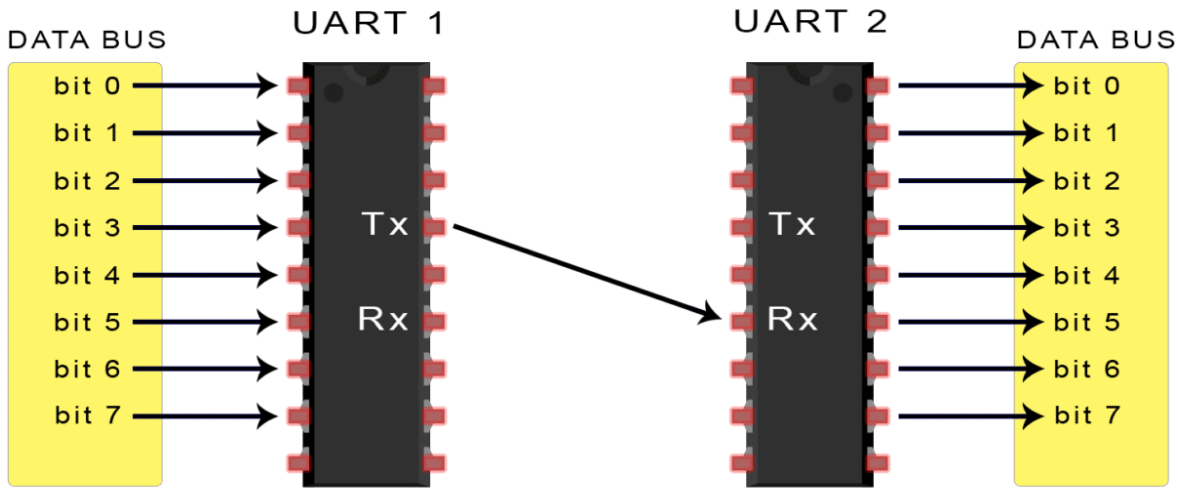
Both UARTs must also be configured to transmit and receive the same data packet structure.

Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous?	Asynchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	1

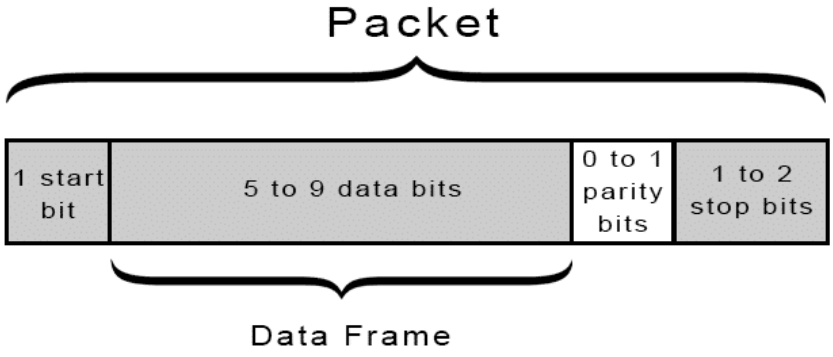
HOW UART WORKS

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data

from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end:



UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:



START BIT

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

DATA FRAME

The data frame contains the actual data being transferred. It can be 5 bits up to 8 bits long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long. In most cases, the data is sent with the least significant bit first.

PARITY

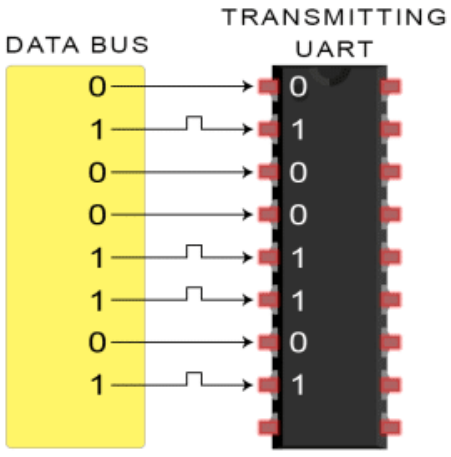
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

STOP BITS

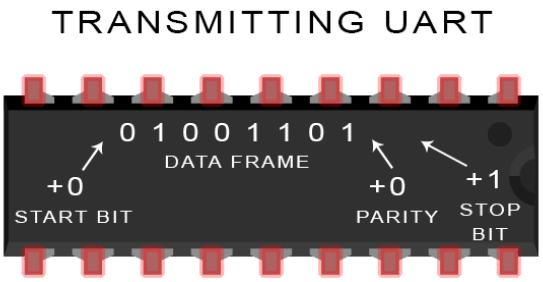
To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

STEPS OF UART TRANSMISSION

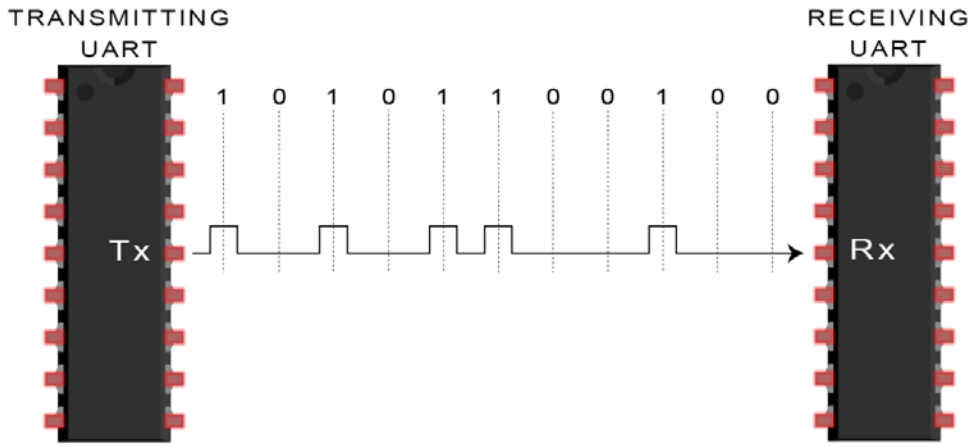
1. The transmitting UART receives data in parallel from the data bus:



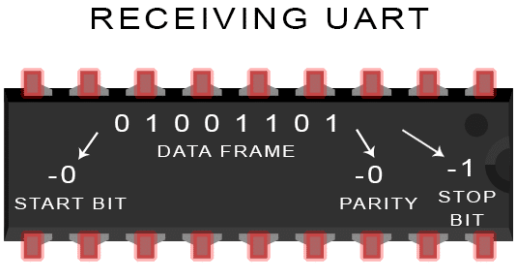
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



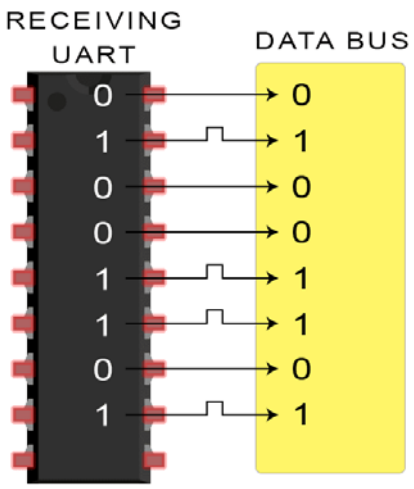
3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:



4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:



5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:



ADVANTAGES AND DISADVANTAGES OF UARTS

No communication protocol is perfect, but UARTs are pretty good at what they do. Here are some pros and cons to help you decide whether or not they fit the needs of your project:

ADVANTAGES

- Only uses two wires
- No clock signal is necessary
- Has a parity bit to allow for error checking
- The structure of the data packet can be changed as long as both sides are set up for it
- Well documented and widely used method

DISADVANTAGES

- The size of the data frame is limited to a maximum of 9 bits
- Doesn't support multiple slave or multiple master systems
- The baud rates of each UART must be within 10% of each other

I2C Communication protocol

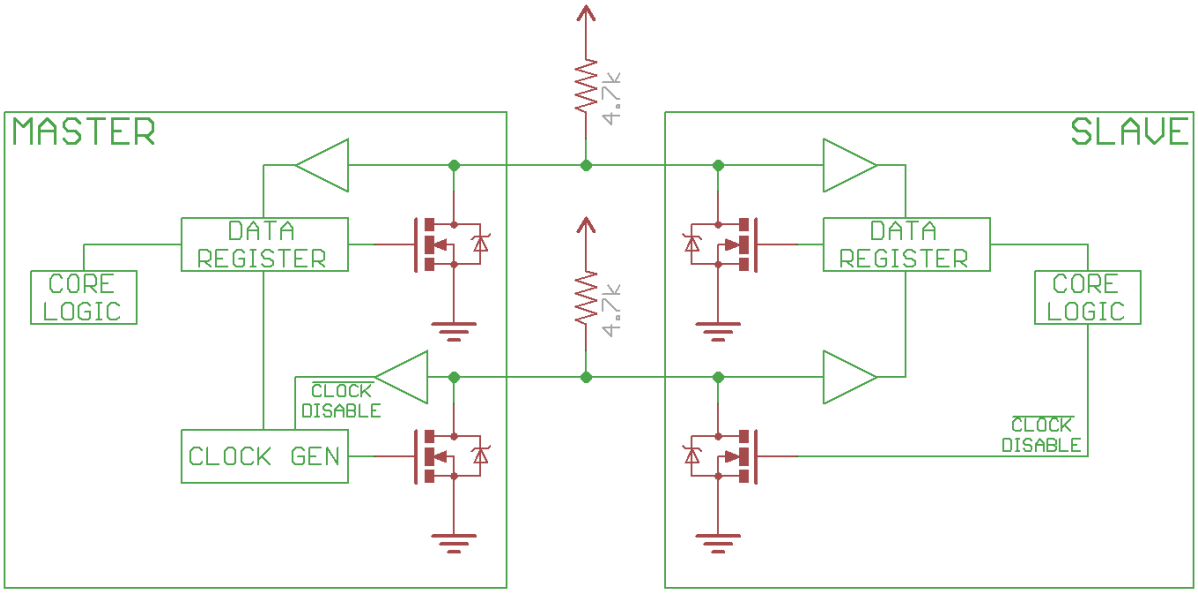
The Inter-integrated Circuit (I^2C) Protocol is a protocol intended to allow multiple "slave" digital integrated circuits ("chips") to communicate with one or more "master" chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.

Signals

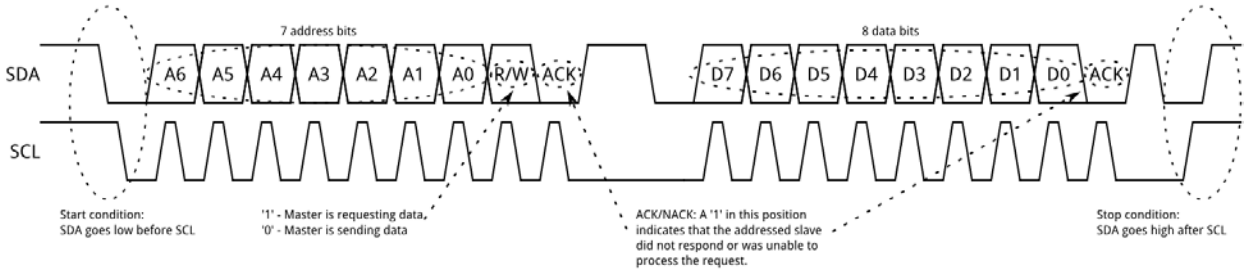
Each I^2C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus master; some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out). This is called "clock stretching".

Unlike UART or SPI connections, the I^2C bus drivers are "open drain", meaning that they can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus contention where one device is trying to drive the line high while another tries to pull it low, eliminating the potential for damage to the drivers or excessive power dissipation in the system. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low.

Resistor selection varies with devices on the bus, but a good rule of thumb is to start with 4.7k and adjust down if necessary. I^2C is a fairly robust protocol, and can be used with short runs of wire (2-3m). For long runs, or systems with lots of devices, smaller resistors are better.



Protocol



Messages are broken up into two types of frame: an address frame, where the master indicates the slave to which the message is being sent, and one or more data frames, which are 8-bit data messages passed from master to slave or vice versa. Data is placed on the SDA line after SCL goes low, and is sampled after the SCL line goes high. The time between clock edge and data read/write is defined by the devices on the bus and will vary from chip to chip.

Start Condition

To initiate the address frame, the master device leaves SCL high and pulls SDA low. This puts all slave devices on notice that a transmission is about to start. If two master devices wish to take ownership of the bus at one time, whichever device pulls SDA low first wins the race and gains control of the bus. It is possible to issue repeated starts, initiating a new communication sequence without relinquishing control of the bus to other masters; we'll talk about that later.

Address Frame

The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation.

The 9th bit of the frame is the NACK/ACK bit. This is the case for all frames (data or address). Once the first 8 bits of the frame are sent, the receiving device is given control over SDA. If the receiving device does not pull the SDA line low before the 9th clock pulse, it can be inferred that the receiving device either did not receive the data or did not know how to parse the message. In that case, the exchange halts, and it's up to the master of the system to decide how to proceed.

Data Frames

After the address frame has been sent, data can begin being transmitted. The master will simply continue generating clock pulses at a regular interval, and the data will be placed on SDA by either the master or the slave, depending on whether the R/W bit indicated a read or write operation. The number of data frames is arbitrary, and most slave devices will auto-increment the internal register, meaning that subsequent reads or writes will come from the next register in line.

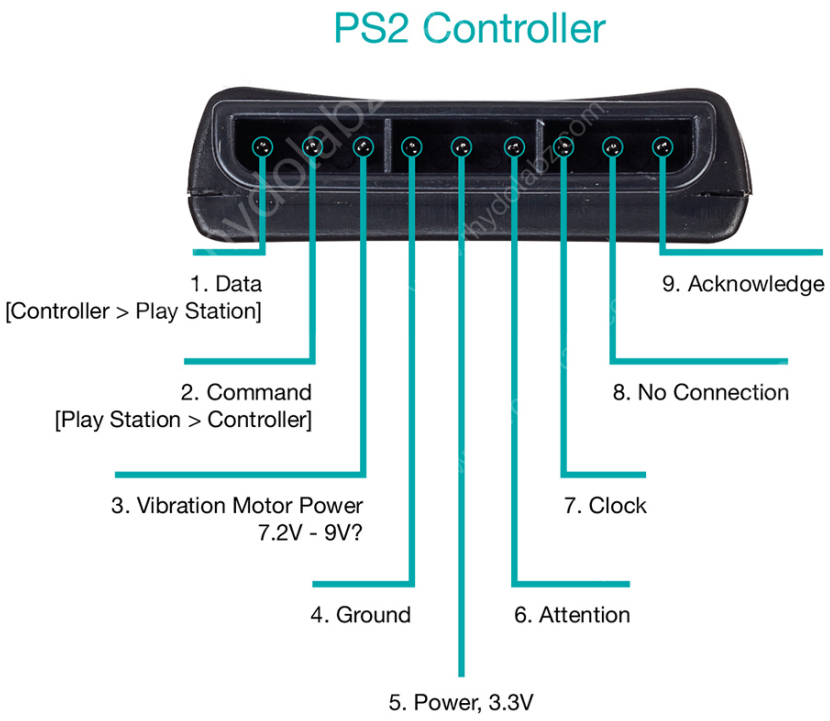
Stop condition

Once all the data frames have been sent, the master will generate a stop condition. Stop conditions are defined by a 0->1 (low to high) transition on SDA *after* a 0->1 transition on SCL, with SCL remaining high. During normal data writing operation, the value on SDA should **not** change when SCL is high, to avoid false stop conditions.

Interfacing PS2 Wireless Controller with Arduino

The PS2 wireless controller is a standard controller for the PlayStation 2 and is identical to the original DualShock controller for the PlayStation console. It features twelve analog (pressure-sensitive) buttons (X, O, Π, Δ, L1, R1, L2, R2, Up, Down, Left and Right), five digital button (L3, R3 Start, Select and the analog mode button) and two analog sticks. The controller also features two vibration motors, the left one being larger and more powerful than the one on the right. It is powered by two AAA batteries. It communicates with the console using 2.4 GHz RF protocol.

PS2 Receiver Pin Out:



1. **DATA:** This is the data line from Controller to PS2. This is an open collector output and requires a pull-up resistor (1 to 10k, maybe more). (A pull-up resistor is needed because the controller can only connect this line to ground; it can't actually put voltage on the line).
2. **COMMAND:** This is the data line from PS2 to Controller.
3. **VIBRATION MOTOR POWER**
4. **GND:** Ground
5. **VCC:** VCC can vary from 5V down to 3V.
6. **ATT:** ATT is used to get the attention of the controller. This line must be pulled low before each group of bytes is sent / received, and then set high again afterwards. This pin consider as "Chip Select" or "Slave Select" line that is used to address different controllers on the same bus.
7. **CLK:** 500kH/z, normally high on. The communication appears to be SPI bus.
8. **Not Connected**

9. **ACK:** Acknowledge signal from Controller to PS2. This normally high line drops low about 12us after each byte for half a clock cycle, but not after the last bit in a set. This is a open collector output and requires a pull-up resistor (1 to 10k, maybe more).

PS2 Signals:

PS2 wireless controller communicates with Arduino using a protocol that is basically SPI. The play station sends a byte at the same time as it receives one (full duplex) via serial communication. There's a clock (SCK) to synchronize bits of data across two channels: DATA and CMD. Additionally, there's a "Attention" (ATT) channel which tells the slave whether or not it is "active" and should listen to data bits coming across the CMD channel, or send data bits across the DATA channel (Reasonably, only one slave device should be active at a time). The PlayStation 2 actually uses this plus an additional line that is not specifically part of the SPI protocol – an "Acknowledge" (ACK) line.
The clock is held high until a byte is to be sent. It then drops low (active low) to start 8 cycles during which data is simultaneously sent and received. The logic level on the data lines is changed by the transmitting device on the falling edge of clock. This is then read by the receiving device on the rising edge allowing time for the signal to settle. After each Command is received from the controller, that controller needs to pull ACK low for at least one clock cycle. If a selected controller does not ACK the PS2 will assume that there is no controller present. LSBs (least significant bits) are transmitting first.

Sample Project: Using a PS2 controller in a R/C vehicle

We will need:

1. Arduino Uno
2. L298N drive
3. Motor.
4. Connector wires.
5. SG90 servo.
6. PS2 joystick
7. Rechargeable batteries with charger

Upon each button press the Arduino receives the RF signal on the PS2 receiver. We followed the standard PS2 protocol for realizing the communication algorithm, identical to the SPI protocol. Our program on the Arduino detects and reads the button presses only, pressure values are not read.

Connection Details:

The PS2 receiver CLK line and ATT lines are held normally high. The ATT operates like the Slave Select line under SPI. You pull it low to tell the controller you are talking to it and then send it back high once a communications cycle is complete. CMD is the data line to the controller and DATA is the data coming from the controller. Here in our application we are not using the acknowledge pin.

Arduino Code:

```
#include <PS2X_lib.h>  //for v1.6
#include <Servo.h>
PS2X ps2x;
int PS2 = 0;
Servo LXservo;
const int in1 = 2;    // direction pin 1
const int in2 = 4;    // direction pin 2
const int ena = 3;    // PWM pin to change speed
int fspeed;          // forward speed

void setup(){
  LXservo.attach(6);
  PS2 = ps2x.config_gamepad(13,11,10,12, true, true);   // (clock, command, attention, data, true, true)
  pinMode(in1, OUTPUT);      // connection to L298n
  pinMode(in2, OUTPUT);      // connection to L298n
  pinMode(ena, OUTPUT);      // connection to L298n
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);   // Стопаки
  pinMode(5, OUTPUT);   // Стопаки от кнопки
  pinMode(1, OUTPUT);   // Поворот на лево
  pinMode(7, OUTPUT);   // Поворот на право
}

void loop(){
```

```
ps2x.read_gamepad();
if(ps2x.ButtonPressed(PSB_CIRCLE)) digitalWrite(8, HIGH);
if(ps2x.ButtonPressed(PSB_TRIANGLE)) digitalWrite(8, LOW);
 if(ps2x.ButtonPressed(PSB_SQUARE)) digitalWrite(5, HIGH);
 if(ps2x.ButtonPressed(PSB_CROSS)) digitalWrite(5, LOW);
 // if(ps2x.ButtonPressed(PSB_R1)) digitalWrite(7, HIGH);
 // if(ps2x.ButtonPressed(PSB_R2)) digitalWrite(7, LOW);
if (ps2x.Analog(PSS_LY) == 128) analogWrite(ena, 0);
if (ps2x.Analog(PSS_LY) > 128){
   fspeed = map(ps2x.Analog(PSS_LY), 129, 255, 0, 255);
   digitalWrite(in1, LOW);
   digitalWrite(in2, HIGH);
   analogWrite(ena, fspeed);
   digitalWrite(9, HIGH);
 }

 if (ps2x.Analog(PSS_LY) < 128){
   fspeed = map(ps2x.Analog(PSS_LY), 0, 127, 255, 0);
   digitalWrite(in1, HIGH);
   digitalWrite(in2, LOW);
   analogWrite(ena, fspeed);
   digitalWrite(9, LOW);
 }

 LXservo.write(map(ps2x.Analog(PSS_RX), 0, 255, 65, 105));
```

```
   if (ps2x.Analog(PSS_RX) < 120){
   digitalWrite(1, HIGH);
   digitalWrite(7, LOW);
  }

   if (ps2x.Analog(PSS_RX) > 150){
   digitalWrite(7, HIGH);
   digitalWrite(1, LOW);
  }
  if (ps2x.Analog(PSS_RX) > 120 && ps2x.Analog(PSS_RX) < 150)
{          digitalWrite(1, LOW);
           digitalWrite(7, LOW);
  }
delay(50);
}
```

LIST OF WIRELESS COMMUNICATION PROTOCOLS

Bluetooth

Bluetooth is a global 2.4 GHz personal area network for short-range wireless communication. Device-to-device file transfers, wireless speakers, and wireless headsets are often enabled with Bluetooth.

BLE

BLE is a version of Bluetooth designed for lower-powered devices that use less data. To conserve power, BLE remains in sleep mode except when a connection is initiated. This makes it ideal for wearable fitness trackers and health monitors.

ZigBee

ZigBee is a 2.4 GHz mesh local area network (LAN) protocol. It was originally designed for building automation and control—so things like wireless thermostats and lighting systems often use ZigBee.

Z-Wave

Z-Wave is a sub-GHz mesh network protocol, and is a proprietary stack. It's often used for security systems, home automation, and lighting controls.

6LoWPAN

6LoWPAN uses a lightweight IP-based communication to travel over lower data rate networks. It is an open IoT network protocol like ZigBee, and it is primarily used for home and building automation.

Thread

Thread is an open standard, built on IPv6 and 6LoWPAN protocols. You could think of it as Google's version of ZigBee. You can actually use some of the same chips for Thread and ZigBee, because they're both based on 802.15.4.

WiFi-ah (HaLow)

Designed specifically for low data rate, long-range sensors and controllers, 802.11ah is far more IoT-centric than many other WiFi counterparts.

2G (GSM)

2G is the "old-school" TDMA (usually) cellular protocol. ATMs and old alarm systems used this— and in most parts of the world it is phased out or in the process of being phased out.

3G & 4G

3G was the first "high speed" cellular network, and is a name that refers to a number of technologies that meet IMT-2000 standards. 4G is the generation of cellular standards that followed 3G, and is what most people use today for mobile cellular data. You can use 3G

and 4G for IoT devices, but the application needs a constant power source or must be able to be recharged regularly.

LTE Cat 0, 1, & 3

With LTE classes, the lower the speed, the lower the amount of power they use. LTE Cat 1 and 0 are typically more suitable for IoT devices. (You can learn more about them in this Radio-Electronics article.)

LTE-M1

This is the first cellular wireless protocol that was build from the ground up for IoT devices. That being said, it isn't available yet, so how it performs remains to be seen.

With LTE, it's worth understanding that carriers typically don't have to modify hardware for their base stations; upgrades can be done entirely through software. This really helps with infrastructure costs, because companies won't necessarily need new cellular base stations, just new endpoint hardware.

NB-IoT

NB-IoT, or Narrowband IoT, is another way to tackle cellular M2M for low power devices. It is based on a DSSS modulation similar to the old Neul version of Weightless-W. Huawei, Ericsson, and Qualcomm are active proponents of this protocol and are involved in putting it together.

5G

Though it likely won't be released for another five years, 5G is set to be the next generation of cellular network protocol. It's designed for high throughput, and it will probably face the same issues as 3G and 4G in regards to IoT.

NFC

Near field communication is precisely as it sounds—IoT network protocols used for very close communication. When you wave your phone over a card reader to pay for groceries, you're likely using NFC.

RFID

There are two types of radio frequency identification: active and passive. This protocol was designed specifically so devices without batteries could send a signal. In most systems, one side of an RFID system is powered, creating a magnetic field, which induces an electric current in the chip. This creates a system with enough power to send data wirelessly over and over again. Because of this, RFID tags are used for shipping and tracking purposes.

SigFox

SigFox is a global IoT network operator. It uses differential binary phase-shift keying (DBPSK) in one direction and Gaussian frequency shift keying (GFSK) in the other direction. SigFox and their partners set up antennas on towers (like a cell phone company) and receive data transmissions from devices such as parking sensors or water meters.

LoRaWAN

LoRaWAN is a media access control (MAC) layer protocol designed for large-scale public networks with a single operator. It is built using Semtech's LoRa modulation as the underlying PHY, but it is important to note that LoRa and LoRaWAN are two separate things that are often (mistakenly) conflated.

Ingenu

Ingenu has created something called random phase multiple access (RPMA), which uses Direct Sequence Spread Spectrum (DSSS) and is similar to code division multiple access (CDMA) cellular protocols. Before IoT was a thing, Ingenu (then OnRamp) was selling metering infrastructure that collected low power information from electricity meters. Now, it's rebranded and is trying to become a broader player in the field (like SigFox).

Weightless-N

Weightless-N is an ultra narrowband system that is very similar to SigFox. Instead of being a complete end-to-end enclosed system, it's made up of a network of partners. It uses differential binary phase shift keying (BPSK) in narrow frequency channels and is intended for uplink sensor data.

Weightless-P

Weightless-P is the latest Weightless technology. It offers two-way features and quality of service tiers, which we think is very important.

Weightless-W

Weightless-W is an open standard designed to operate in TV white space (TVWS) spectrum. Using TVWS is attractive in theory, because it takes advantage of good ultra high frequency (UHF) spectrum that's not otherwise in use—but it can be quite difficult in practice.

ANT & ANT+

If you have a Samsung device, you probably have a radio with their protocol in it. ANT & ANT+ seem somewhat like another type of BLE system, designed to create networks that piggyback off of existing hardware. A lot of devices have ANT or ANT+ compatible chips in them, and the idea is that if you get enough of these radios added to the world, you can use them together as a mesh.

DigiMesh

DigiMesh is one of a number of proprietary mesh systems. You can learn about the differences between it and ZigBee in this white paper.

MiWi

MiWi is Microchip's proprietary network protocol. It was created for short-range networks and designed to help customers reduce their products' time to market.

EnOcean

EnOcean is a protocol designed specifically for energy harvesting applications that are extremely low power. Thus, its applications are centered on building automation, smart homes, and wireless lighting control.

Dash7

Dash7 is an open-source wireless network protocol with a huge RFID contract with the U.S. Department of Defense.

WirelessHART

WirelessHART is built on the HART Communication Protocol, and is what the company considers “the industry’s first international open wireless communication standard.”