sparkfun

Serial Communication

CONTRIBUTORS: *JIMB0*

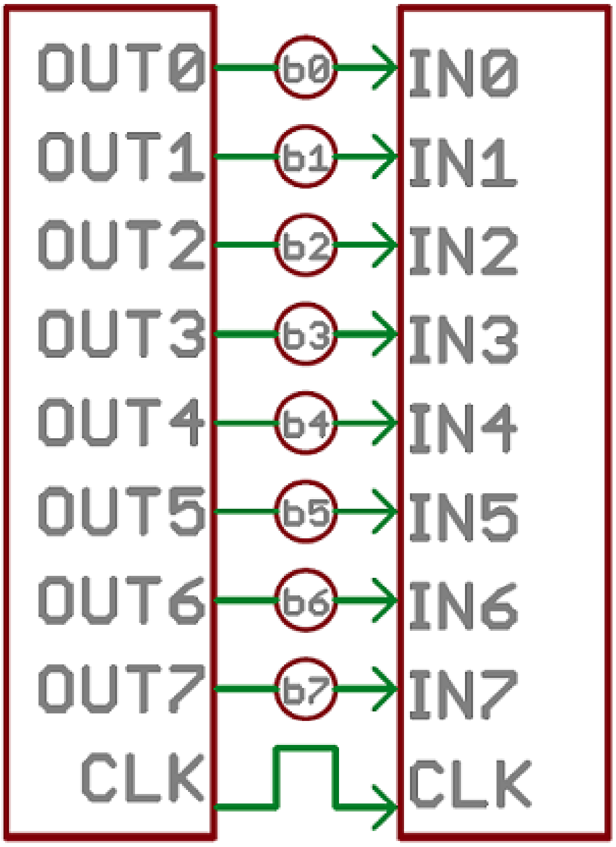
♡ FAVORITE 11

Introduction

Embedded electronics is all about interlinking circuits (processors or other integrated circuits) to create a symbiotic system. In order for those individual circuits to swap their information, they must share a common communication protocol. Hundreds of communication protocols have been defined to achieve this data exchange, and, in general, each can be separated into one of two categories: parallel or serial.

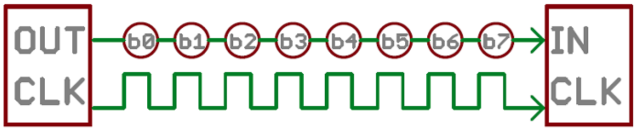
Parallel vs. Serial

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 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.

Asynchronous Serial

Over the years, dozens of serial protocols have been crafted to meet particular needs of embedded systems. USB (universal *serial* bus), and Ethernet, are a couple of the more well-known computing serial interfaces. Other very common serial interfaces include SPI, I^{2}C, and the serial standard we're here to talk about today. Each of these serial interfaces can be sorted into one of two groups: synchronous or asynchronous.

A synchronous serial interface always pairs its data line(s) with a clock signal, so all devices on a synchronous serial bus share a common clock. This makes for a more straightforward, often faster serial transfer, but it also requires at least one extra wire between communicating devices. Examples of synchronous interfaces include SPI, and I^{2}C.

Asynchronous means that data is transferred **without support from an external clock signal**. This transmission method is perfect for minimizing the required wires and I/O pins, but it does mean we need to put some extra effort into reliably transferring and receiving data. The serial protocol we'll be discussing in this tutorial is the most common form of asynchronous transfers. It is so common, in fact, that when most folks say "serial" they're talking about this protocol (something you'll probably notice throughout this tutorial).

The clock-less serial protocol we'll be discussing in this tutorial is widely used in embedded electronics. If you're looking to add a GPS module, Bluetooth, XBee's, serial LCDs, or many other external devices to your project, you'll probably need to whip out some serial-fu.

Suggested Reading

This tutorial builds on a few lower-level electronics concepts, including:

- How to read a schematic
- Analog vs Digital
- Logic Levels
- Binary
- Hexadecimal
- ASCII

If you’re not super familiar with any of those concepts, consider checking those links out.

Now then, let’s go on a serial journey…

Rules of Serial

The asynchronous serial protocol has a number of built-in rules - mechanisms that help ensure robust and error-free data transfers. These mechanisms, which we get for eschewing the external clock signal, are:

- Data bits,
- Synchronization bits,
- Parity bits,
- and Baud rate.

Through the variety of these signaling mechanisms, you’ll find that there’s no one way to send data serially. The protocol is highly configurable. The critical part is making sure that **both devices on a serial bus are configured to use the exact same protocols**.

Baud Rate

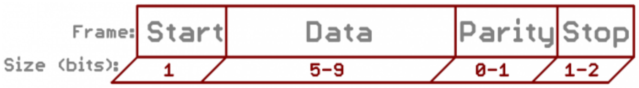
The baud rate specifies **how fast** data is sent over a serial line. It’s usually expressed in units of bits-per-second (bps). If you invert the baud rate, you can find out just how long it takes to transmit a single bit. This value determines how long the transmitter holds a serial line high/low or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

Framing the data

Each block (usually a byte) of data transmitted is actually sent in a *packet* or *frame* of bits. Frames are created by appending synchronization and parity bits to our data.



A serial frame. Some symbols in the frame have configurable bit sizes.

Let’s get into the details of each of these frame pieces.

Data chunk

The real meat of every serial packet is the data it carries. We ambiguously call this block of data a *chunk*, because its size isn’t specifically stated. The amount of data in each packet can be set to anything from 5 to 9 bits. Certainly, the standard data size is your basic 8-bit byte, but other sizes have their uses. A 7-bit data chunk can be more efficient than 8, especially if you’re just transferring 7-bit ASCII characters.

After agreeing on a character-length, both serial devices also have to agree on the **endianness** of their data. Is data sent most-significant bit (msb) to least, or vice-versa? If it's not otherwise stated, you can usually assume that data is transferred **least-significant bit (lsb) first**.

Synchronization bits

The synchronization bits are two or three special bits transferred with each chunk of data. They are the **start bit** and the **stop bit(s)**. True to their name, these bits mark the beginning and end of a packet. There's always only one start bit, but the number of stop bits is configurable to either one or two (though it's commonly left at one).

The start bit is always indicated by an idle data line going from 1 to 0, while the stop bit(s) will transition back to the idle state by holding the line at 1.

Parity bits

Parity is a form of very simple, low-level error checking. It comes in two flavors: odd or even. To produce the parity bit, all 5-9 bits of the data byte are added up, and the evenness of the sum decides whether the bit is set or not. For example, assuming parity is set to even and was being added to a data byte like `0b01011101` , which has an odd number of `1` 's (5), the parity bit would be set to `1` . Conversely, if the parity mode was set to odd, the parity bit would be `0` .

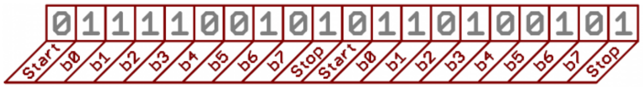
Parity is *optional*, and not very widely used. It can be helpful for transmitting across noisy mediums, but it'll also slow down your data transfer a bit and requires both sender and receiver to implement error-handling (usually, received data that fails must be re-sent).

9600 8N1 (an example)

9600 8N1 - 9600 baud, 8 data bits, no parity, and 1 stop bit - is one of the more commonly used serial protocols. So, what would a packet or two of 9600 8N1 data look like? Let's have an example!

A device transmitting the ASCII characters 'O' and 'K' would have to create two packets of data. The ASCII value of *O* (that's uppercase) is 79, which breaks down into an 8-bit binary value of `01001111` , while *K*'s binary value is `01001011` . All that's left is appending sync bits.

It isn't specifically stated, but it's assumed that data is transferred least-significant bit first. Notice how each of the two bytes is sent as it reads from right-to-left.



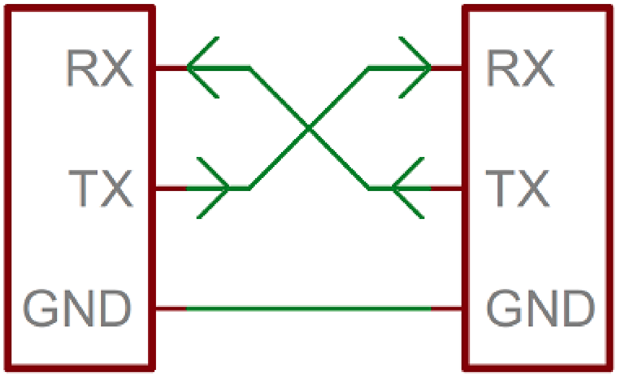
Since we're transferring at 9600 bps, the time spent holding each of those bits high or low is 1/(9600 bps) or 104 µs per bit.

For every byte of data transmitted, there are actually 10 bits being sent: a start bit, 8 data bits, and a stop bit. So, at 9600 bps, we're actually sending 9600 bits per second or 960 (9600/10) bytes per second.

Now that you know how to construct serial packets, we can move on to the hardware section. There we'll see how those 1's and 0's and the baud rate are implemented at a signal level!

Wiring and Hardware

A serial bus consists of just two wires - one for sending data and another for receiving. As such, serial devices should have two serial pins: the receiver, **RX**, and the transmitter, **TX**.



It's important to note that those *RX* and *TX* labels are with respect to the device itself. So the RX from one device should go to the TX of the other, and vice-versa. It's weird if you're used to hooking up VCC to VCC, GND to GND, MOSI to MOSI, etc., but it makes sense if you think about it. The transmitter should be talking to the receiver, not to another transmitter.

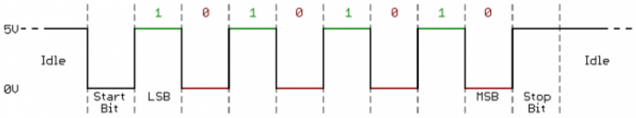
A serial interface where both devices may send and receive data is either **full-duplex** or **half-duplex**. Full-duplex means both devices can send and receive simultaneously. Half-duplex communication means serial devices must take turns sending and receiving.

Some serial busses might get away with just a single connection between a sending and receiving device. For example, our Serial Enabled LCDs are all ears and don't really have any data to relay back to the controlling device. This is what's known as **simplex** serial communication. All you need is a single wire from the master device's TX to the listener's RX line.

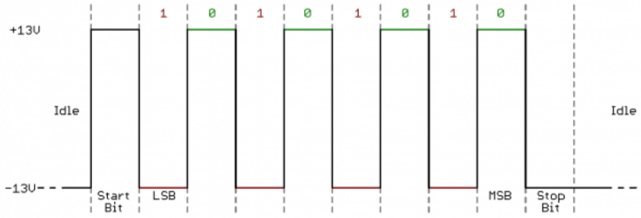
Hardware Implementation

We've covered asynchronous serial from a conceptual side. We know which wires we need. But how is serial communication actually implemented at a signal level? In a variety ways, actually. There are all sorts of standards for serial signaling. Let's look at a couple of the more popular hardware implementations of serial: logic-level (TTL) and RS-232.

When microcontrollers and other low-level ICs communicate serially they usually do so at a TTL (transistor-transistor logic) level. **TTL serial** signals exist between a microcontroller's voltage supply range - usually 0V to 3.3V or 5V. A signal at the VCC level (3.3V, 5V, etc.) indicates either an idle line, a bit of value 1, or a stop bit. A 0V (GND) signal represents either a start bit or a data bit of value 0.



RS-232, which can be found on some of the more ancient computers and peripherals, is like TTL serial flipped on its head. RS-232 signals usually range between -13V and 13V, though the spec allows for anything from +/- 3V to +/- 25V. On these signals a low voltage (-5V, -13V, etc.) indicates either the idle line, a stop bit, or a data bit of value 1. A high RS-232 signal means either a start bit, or a 0-value data bit. That's kind of the opposite of TTL serial.



Between the two serial signal standards, TTL is much easier to implement into embedded circuits. However the low voltage levels are more susceptible to losses across long transmission lines. RS-232, or more complex standards like RS-485, are better suited to long range serial transmissions.

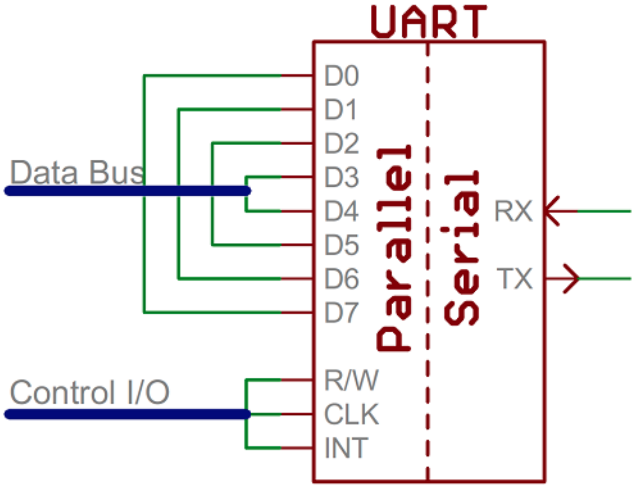
When you're connecting two serial devices together, it's important to make sure their signal voltages match up. You can't directly interface a TTL serial device with an RS-232 bus. You'll have to shift those signals!

Continuing on, we'll explore the tool microcontrollers use to convert their data on a parallel bus to and from a serial interface. UARTs!

UARTs

The final piece to this serial puzzle is finding something to both create the serial packets and control those physical hardware lines. Enter the UART.

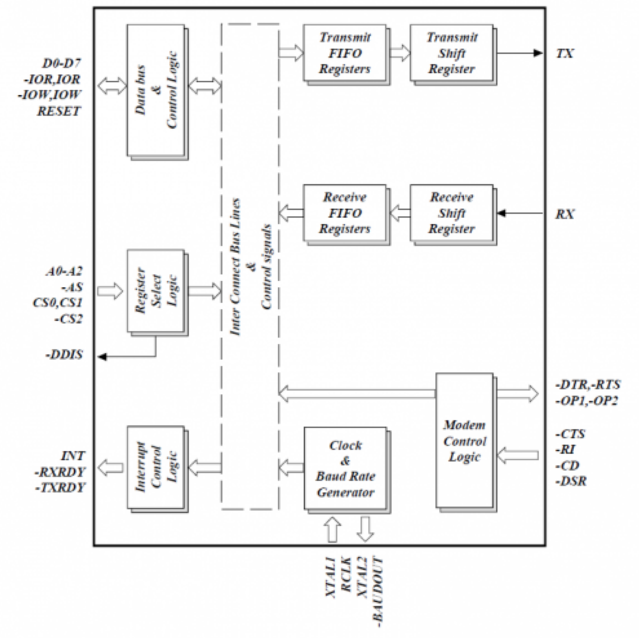
A universal asynchronous receiver/transmitter (UART) is a block of circuitry responsible for implementing serial communication. Essentially, the UART acts as an intermediary between parallel and serial interfaces. On one end of the UART is a bus of eight-or-so data lines (plus some control pins), on the other is the two serial wires - RX and TX.



Super-simplified UART interface. Parallel on one end, serial on the other.

UARTs do exist as stand-alone ICs, but they're more commonly found inside microcontrollers. You'll have to check your microcontroller's datasheet to see if it has any UARTs. Some have none, some have one, some have many. For example, the Arduino Uno - based on the "old faithful" ATmega328 - has just a single UART, while the Arduino Mega - built on an ATmega2560 - has a whopping four UARTs.

As the *R* and *T* in the acronym dictate, UARTs are responsible for both sending and receiving serial data. On the transmit side, a UART must create the data packet - appending sync and parity bits - and send that packet out the TX line with precise timing (according to the set baud rate). On the receive end, the UART has to sample the RX line at rates according to the expected baud rate, pick out the sync bits, and spit out the data.



Internal UART block diagram (courtesy of the Exar ST16C550 datasheet)

More advanced UARTs may throw their received data into a **buffer**, where it can stay until the microcontroller comes to get it. UARTs will usually release their buffered data on a first-in-first-out (FIFO) basis. Buffers can be as small as a few bits, or as large as thousands of bytes.

Software UARTs

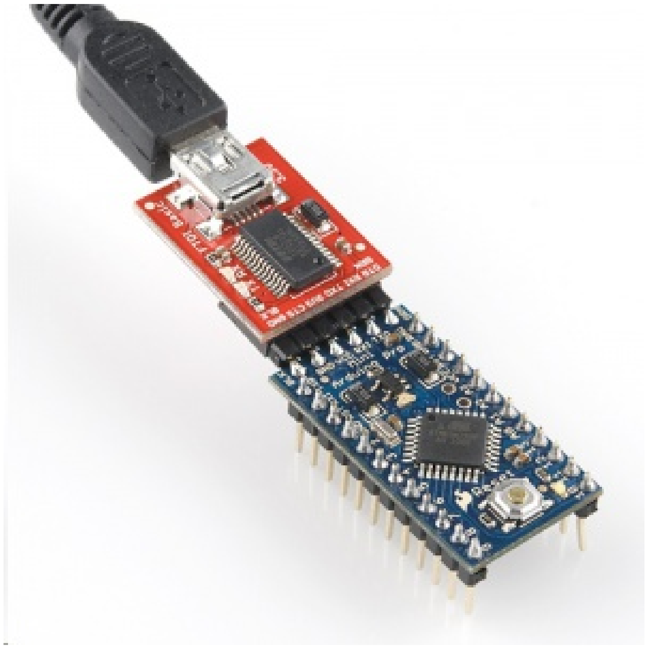
If a microcontroller doesn't have a UART (or doesn't have enough), the serial interface can be **bit-banged** - directly controlled by the processor. This is the approach Arduino libraries like SoftwareSerial take. Bit-banging is processor-intensive, and not usually as precise as a UART, but it works in a pinch!

Common Pitfalls

That's about all there is to serial communication. I'd like to leave you with a few common mistakes that are easy for an engineer of any experience level to make:

RX-to-TX, TX-to-RX

Seems simple enough, but it's a mistake I know I've made more than a few times. As much as you want their labels to match up, always make sure to cross the RX and TX lines between serial devices.

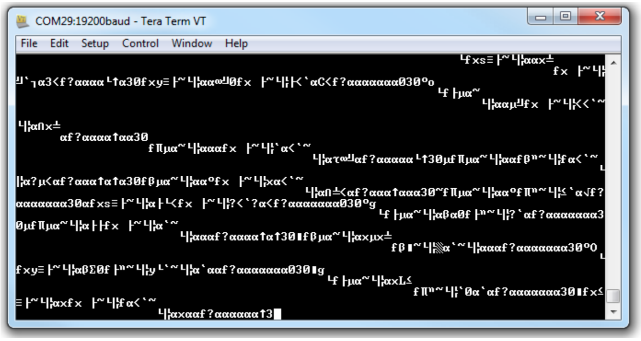


FTDI Basic programming a Pro Mini. Note RX and TX's crossed!

Contrary to what the esteemed Dr. Egon Spengler would warn, **cross the streams**.

Baud Rate Mismatch

Baud rates are like the languages of serial communication. If two devices aren't speaking at the same speed, data can be either misinterpreted, or completely missed. If all the receiving device sees on its receive line is garbage, check to make sure the baud rates match up.



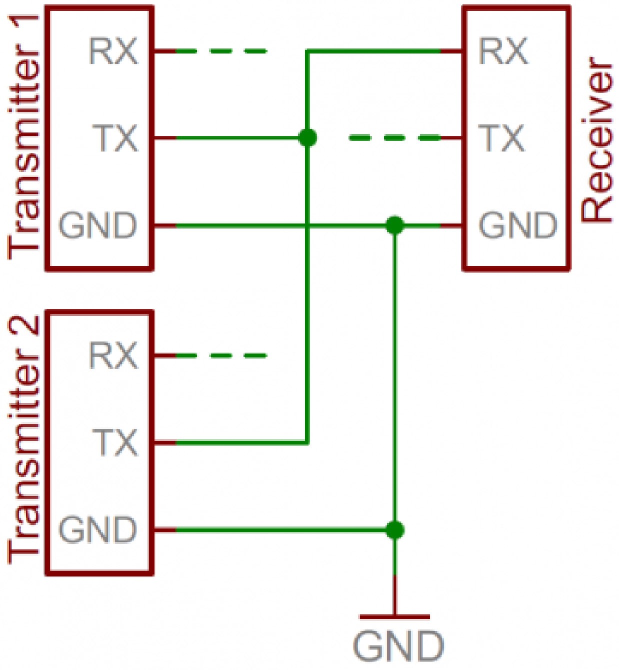
```
                                                                   ╘fxs≡├~╘╣╠ααx±
                                                                                fx ├~╘╣
╝`┐α3<f?αααα└↑α30fxy≡├~╘╣╠αα∞╝0fx ├~╘╣╠<`αC<f?ααααααα030ºo
                                                                ╘f├μα~
                                                                      ╘╣╠ααμ╝fx ├~╘╣<<`~

╘╣╠αΠx±
     αf?αααα↑αα30
                    fΠμα~╘╣╠αααfx ├~╘╣`α<`~
                                            ╘╣╠ατ∞╝αf?ααααα└↑30μfΠμα~╘╣╠ααfβⁿ~╘╣fα<`~
╠α?μ<αf?ααα↑α↑α30fβμα~╘╣╠ααºfx ├~╘╣xα<`~
                                           ╘╣╠αΠ±<αf?ααα↑ααα30~fΠμα~╘╣╠ααºfΠⁿ~╘╣≤`α√f?
ααααααα30αfxs≡├~╘╣╠α├╘<fx ├~╘╣?<`?α<f?ααααααα030ºg
                                                      ╘f├μα~╘╣╠αβα0f├ⁿ~╘╣?`αf?ααααααα3
0μfΠμα~╘╣╠α├├fx ├~╘╣╠α`~
                          ╘╣╠αααf?αααα↑α↑30▮fβμα~╘╣╠αxμx±
                                                         fβ▮~╘╣▒α`~╘╣╠αααf?ααααααα30ºO
fxy≡├~╘╣╠αβΣ0f├ⁿ~╘╣y└`~╘╣╠α`ααf?ααααααα030▮g
                                                ╘f├μα~╘╣╠αxL≤
                                                             fΠⁿ~╘╣`0α`αf?ααααααα30▮fx≤
≡├~╘╣╠αxfx ├~╘╣fα<`~
                        ╘╣╠αxααf?αααααα↑3█
```

Data transmitted at 9600 bps, but received at 19200 bps. Baud mismatch = garbage.

Bus Contention

Serial communication is designed to allow just two devices to communicate across one serial bus. If more than one device is trying to transmit on the same serial line you could run into bus-contention. Dun dun dun….

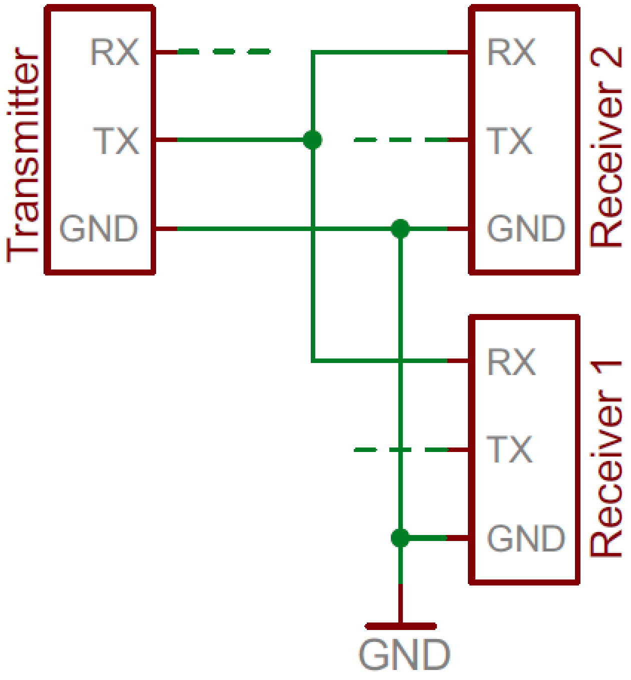
For example, if you're connecting a GPS module up to your Arduino, you may just wire that module's TX line up the Arduino's RX line. But that Arduino RX pin is already wired up to the TX pin of the USB-to-serial converter, which is used whenever you program the Arduino or use the *Serial Monitor*. This sets up the potential situation where both the GPS module and FTDI chip are trying to transmit on the same line at the same time.



Two transmitters sending to a single receiver sets up the possibility for bus contention.

Two devices trying to transmit data at the same time, on the same line, is bad! At "best" neither of the devices will get to send their data. At worst, both device's transmit lines go poof (though that's rare, and usually protected against).

It can be safe to connect multiple receiving devices to a single transmitting device. Not really up to spec and probably frowned upon by a hardened engineer, but it'll work. For example, if you're connecting a serial LCD up to an Arduino, the easiest approach may be to connect the LCD module's RX line to the Arduino's TX line. The Arduino's TX is already connected to the USB programmer's RX line, but that still leaves just one device in control of the transmission line.



Distributing a TX line like this can still be dangerous from a firmware perspective, because you can't pick and choose which device hears what transmission. The LCD will end up receiving data not meant for it, which could command it to go into an unknown state.

In general - one serial bus, two serial devices!

Resources and Going Further

With this shiny, new knowledge of serial communication, there are loads of new concepts, projects, and technologies to explore.

Would you like to learn more about other communication standards? Maybe something synchronous?

- Serial Peripheral Interface
- I^{2}C

Many technologies make heavy use of serial communication:

- GPS Basics
- Exploring XBees and XCTU

Or maybe you'd like to see serial in action?

- If you need to shift the logic levels of your serial communication, check out our Logic Level Converter tutorial.
- Using a Serial LCD
- Serial Terminal Basics
- WiFly Shield Hookup