Serial Communication

CONTRIBUTORS: [*[https://static.sparkfun.com/avatar/f2ed505f510e392f58811e4aa03e6fe0?d=retro&s=20&r=pg](https://www.sparkfun.com/users/69916) JIMB0*](https://www.sparkfun.com/users/69916)

[**FAVORITE**](https://learn.sparkfun.com/tutorials/serial-communication)**35**

[**SHARE**](https://learn.sparkfun.com/tutorials/serial-communication)

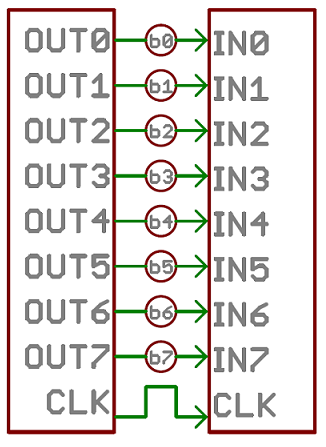
[Share on Twitter](https://twitter.com/intent/tweet?text=Serial+Communication&url=http%3A%2F%2Fsfe.io%2Ft8&via=sparkfun)[Share on Facebook](http://www.facebook.com/sharer.php?u=http%3A%2F%2Fsfe.io%2Ft8&t=Serial+Communication)[Pin It](https://www.pinterest.com/pin/create/button/?url=http%3A%2F%2Fsfe.io%2Ft8&media=https%3A%2F%2Fcdn.sparkfun.com%2F%2Fr%2F500-1000%2Fassets%2F7%2Fd%2Ff%2F9%2F9%2F50d24be7ce395f1f6c000000.jpg&description=Serial+Communication)

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.

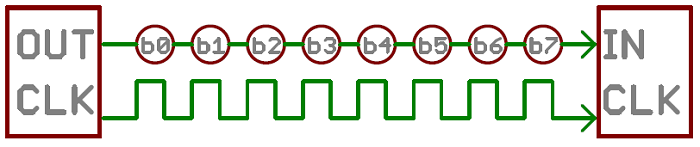
Parallel vs. Serial

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.

[](https://cdn.sparkfun.com/assets/c/a/c/3/a/50e1cca6ce395fbc27000000.png)

*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.

[](https://cdn.sparkfun.com/assets/e/5/4/2/a/50e1ccf1ce395f962b000000.png)

*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](https://www.sparkfun.com/products/11021) to a [Mega](https://www.sparkfun.com/products/11061), 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, I2C, 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 I2C.

*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](https://learn.sparkfun.com/tutorials/how-to-read-a-schematic)
* [Analog vs Digital](https://learn.sparkfun.com/tutorials/analog-vs-digital)
* [Logic Levels](https://learn.sparkfun.com/tutorials/logic-levels)
* [Binary](https://learn.sparkfun.com/tutorials/binary)
* [Hexadecimal](https://learn.sparkfun.com/tutorials/hexadecimal)
* [ASCII](https://learn.sparkfun.com/tutorials/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.

[](https://cdn.sparkfun.com/assets/f/9/c/0/2/50d2066fce395fc43b000000.png)

*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](http://www.asciitable.com/) 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 is01001011. 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.

[](https://cdn.sparkfun.com/assets/c/a/0/b/7/50d2201ece395f0a15000001.png)

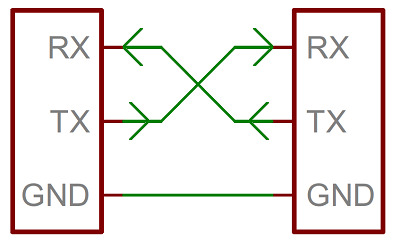
Since we’re transferring at 9600 bps, the time spent holding each of those bits high or low is 1/(9600 bps) or 104 µs per bit.

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**.

[](https://cdn.sparkfun.com/assets/2/5/c/4/5/50e1ce8bce395fb62b000000.png)

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.

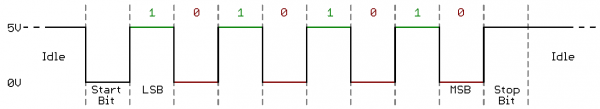
A serial interface where both devices may send and receive data is either **full-duplex** or **half-duplex**. Full-duplex means both devices can send and receive simultaneously. Half-duplex communication means serial devices must take turns sending and receiving.

Some serial busses might get away with just a single connection between a sending and receiving device. For example, our [Serial Enabled LCDs](https://www.sparkfun.com/products/10097) 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.

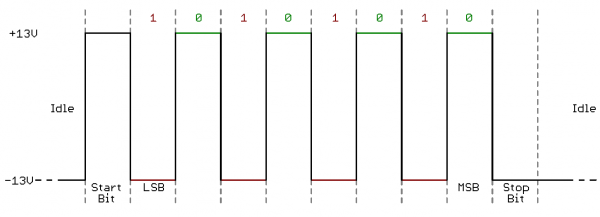
Hardware Implementation

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.

[](https://cdn.sparkfun.com/assets/1/8/d/c/1/51142c09ce395f0e7e000002.png)

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.

[](https://cdn.sparkfun.com/assets/b/d/a/1/3/51142cacce395f877e000006.png)

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.

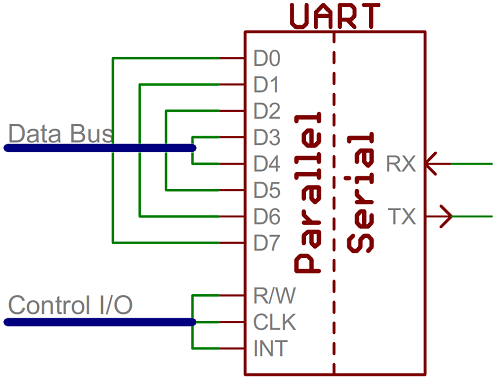
When you’re connecting two serial devices together, it’s important to make sure their signal voltages match up. You can’t directly interface a TTL serial device with an RS-232 bus. You’ll have to [shift those signals](http://www.sparkfun.com/tutorials/215)!

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.

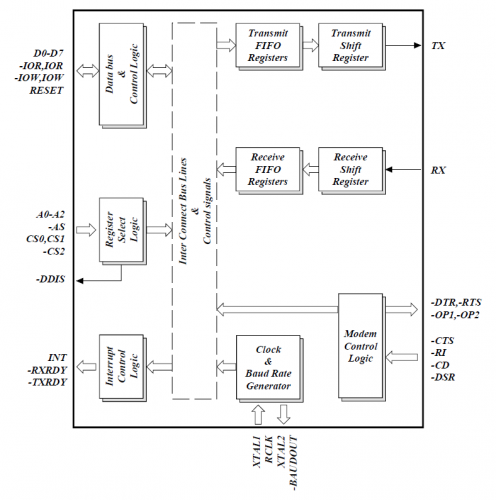
A universal asynchronous receiver/transmitter (UART) is a block of circuitry responsible for implementing serial communication. Essentially, the UART acts as an intermediary between parallel and serial interfaces. On one end of the UART is a bus of eight-or-so data lines (plus some control pins), on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

[](https://cdn.sparkfun.com/assets/d/1/f/5/b/50e1cf30ce395fb227000000.png)

*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.

[](https://cdn.sparkfun.com/assets/e/9/7/5/4/50d24680ce395f7172000000.png)

*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](http://arduino.cc/en/Reference/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.

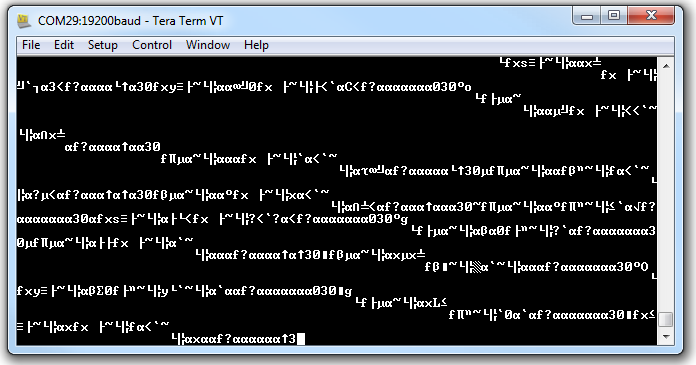
[](https://cdn.sparkfun.com/assets/7/d/f/9/9/50d24be7ce395f1f6c000000.jpg)

[*FTDI Basic*](https://www.sparkfun.com/products/9716)*programming a*[*Pro Mini*](https://www.sparkfun.com/products/11113)*. Note RX and TX’s crossed!*

Contrary to what the esteemed Dr. Egon Spengler would [warn](http://www.youtube.com/watch?v=jyaLZHiJJnE), **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.

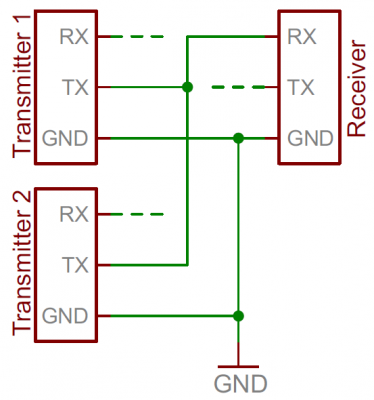
[](https://cdn.sparkfun.com/assets/c/e/2/d/a/50d247c5ce395fdc6b000000.png)

*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….

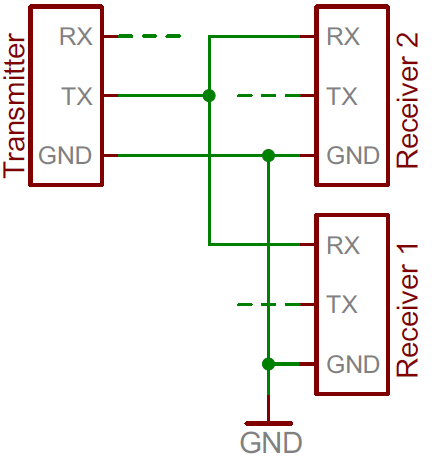
For example, if you’re connecting a GPS module up to your Arduino, you may just wire that module’s TX line up the Arduino’s RX line. But that Arduino RX pin is already wired up to the TX pin of the USB-to-serial converter, which is used whenever you program the Arduino or use the *Serial Monitor*. This 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.

[](https://cdn.sparkfun.com/assets/0/7/4/f/b/50d249a8ce395faa6f000000.png)

*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.

[](https://cdn.sparkfun.com/assets/d/8/8/c/8/5114296cce395f8f7d000004.png)

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!

# Universal asynchronous receiver/transmitter

From Wikipedia, the free encyclopedia

|  |  |
| --- | --- |
| [https://upload.wikimedia.org/wikipedia/en/thumb/9/99/Question_book-new.svg/50px-Question_book-new.svg.png](https://en.wikipedia.org/wiki/File:Question_book-new.svg) | This article **needs additional citations for**[**verification**](https://en.wikipedia.org/wiki/Wikipedia:Verifiability). Please help [improve this article](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit) by [adding citations to reliable sources](https://en.wikipedia.org/wiki/Help:Introduction_to_referencing_with_Wiki_Markup/1). Unsourced material may be challenged and removed.*(November 2010)* *(*[*Learn how and when to remove this template message*](https://en.wikipedia.org/wiki/Help:Maintenance_template_removal)*)* |

A **universal asynchronous receiver/transmitter** (**UART** [/ˈjuːɑːrt/](https://en.wikipedia.org/wiki/Help:IPA_for_English)), is a [computer hardware](https://en.wikipedia.org/wiki/Computer_hardware) device for [asynchronous serial communication](https://en.wikipedia.org/wiki/Asynchronous_serial_communication) in which the data format and transmission speeds are configurable. The electric signaling levels and methods (such as [differential signaling](https://en.wikipedia.org/wiki/Differential_signaling), etc.) are handled by a driver circuit external to the UART.

UARTs are commonly used in conjunction with communication standards such as [TIA](https://en.wikipedia.org/wiki/Telecommunications_Industry_Association) (formerly [EIA](https://en.wikipedia.org/wiki/Electronic_Industries_Alliance)) [RS-232](https://en.wikipedia.org/wiki/RS-232), [RS-422](https://en.wikipedia.org/wiki/RS-422) or [RS-485](https://en.wikipedia.org/wiki/RS-485). A UART is usually an individual (or part of an) [integrated circuit](https://en.wikipedia.org/wiki/Integrated_circuit) (IC) used for [serial communications](https://en.wikipedia.org/wiki/Serial_communications) over a computer or peripheral device [serial port](https://en.wikipedia.org/wiki/Serial_port). UARTs are now commonly included in microcontrollers. A dual UART, or *DUART*, combines two UARTs into a single chip. Similarly, a quadruple UART or *QUART*, combines four UARTs into one package, such as the NXP 28L194. An octal UART or *OCTART* combines eight UARTs into one package, such as the Exar XR16L788 or the NXP SCC2698. A related device, the [Universal Synchronous/Asynchronous Receiver/Transmitter](https://en.wikipedia.org/wiki/Universal_Synchronous/Asynchronous_Receiver/Transmitter) (USART) also supports synchronous operation.

Transmitting and receiving serial data[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=1)]

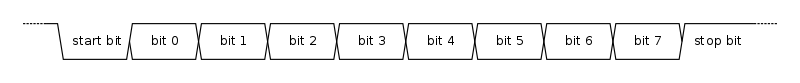
*See also:*[*Asynchronous serial communication*](https://en.wikipedia.org/wiki/Asynchronous_serial_communication)

The universal asynchronous receiver/transmitter (UART) takes bytes of data and transmits the individual bits in a sequential fashion.[[1]](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#cite_note-Osborne80-1) At the destination, a second UART re-assembles the bits into complete bytes. Each UART contains a [shift register](https://en.wikipedia.org/wiki/Shift_register), which is the fundamental method of conversion between serial and parallel forms. Serial transmission of digital information (bits) through a single wire or other medium is less costly than parallel transmission through multiple wires.

The UART usually does not directly generate or receive the external signals used between different items of equipment. Separate interface devices are used to convert the [logic level](https://en.wikipedia.org/wiki/Logic_level) signals of the UART to and from the external signalling levels. External signals may be of many different forms. Examples of standards for voltage signaling are [RS-232](https://en.wikipedia.org/wiki/RS-232), [RS-422](https://en.wikipedia.org/wiki/RS-422) and [RS-485](https://en.wikipedia.org/wiki/RS-485) from the [EIA](https://en.wikipedia.org/wiki/Electronic_Industries_Alliance). Historically, current (in [current loops](https://en.wikipedia.org/wiki/Current_loop)) was used in telegraph circuits. Some signaling schemes do not use electrical wires. Examples of such are [optical fiber](https://en.wikipedia.org/wiki/Fiber_optics), [IrDA](https://en.wikipedia.org/wiki/Infrared_Data_Association) ([infrared](https://en.wikipedia.org/wiki/Infrared)), and (wireless) [Bluetooth](https://en.wikipedia.org/wiki/Bluetooth) in its Serial Port Profile (SPP). Some signaling schemes use modulation of a carrier signal (with or without wires). Examples are modulation of audio signals with phone line[modems](https://en.wikipedia.org/wiki/Modem), RF modulation with data radios, and the [DC-LIN](http://www.yamar.com/DC-LIN.html) for [power line communication](https://en.wikipedia.org/wiki/Power_line_communication).

Communication may be *simplex* (in one direction only, with no provision for the receiving device to send information back to the transmitting device), *full duplex* (both devices send and receive at the same time) or *half duplex* (devices take turns transmitting and receiving).

**Data framing**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=2)]

[](https://en.wikipedia.org/wiki/File:UART_timing_diagram.svg)

The idle, no data state is high-voltage, or powered. This is a historic legacy from telegraphy, in which the line is held high to show that the line and transmitter are not damaged. Each character is sent as a logic low start bit, a configurable number of data bits (usually 8, but users can choose 5 to 8 or 9 bits depending on which UART is in use), an optional [parity bit](https://en.wikipedia.org/wiki/Parity_bit) if the number of bits per character chosen is not 9 bits, and one or more logic high stop bits. In most applications the least significant data bit (the one on the left in this diagram) is transmitted first, but there are exceptions (such as the [IBM 2741](https://en.wikipedia.org/wiki/IBM_2741) printing terminal).

The start bit signals the receiver that a new character is coming. The next five to nine bits, depending on the code set employed, represent the character. If a parity bit is used, it would be placed after all of the data bits. The next one or two bits are always in the **mark** (logic high, i.e., '1') condition and called the stop bit(s). They signal the receiver that the character is completed. Since the start bit is logic low (0) and the stop bit is logic high (1) there are always at least two guaranteed signal changes between characters.

If the line is held in the logic low condition for longer than a character time, this is a [break condition](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#Break_condition) that can be detected by the UART.

**Receiver**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=3)]

|  |  |  |
| --- | --- | --- |
| **Some common speeds** | | |
| [**Bit Rate (Baud Rate)**](https://en.wikipedia.org/wiki/Symbol_rate) | **Time Per Bit** | [**Windows**](https://en.wikipedia.org/wiki/Microsoft_Windows) **Support**[[2]](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#cite_note-2) |
| 50 bit/s | 20000 µS | No |
| 75 bit/s | 13333.3 µS | Yes |
| 110 bit/s | 9090.9 µS | Yes |
| 134.5 bit/s | 7434.9 µS | Yes |
| 150 bit/s | 6666.6 µS | Yes |
| 300 bit/s | 3333.3 µS | Yes |
| 600 bit/s | 1666.7 µS | Yes |
| 1,200 bit/s | 833.3 µS | Yes |
| 1,800 bit/s | 555.6 µS | Yes |
| 2,400 bit/s | 416.7 µS | Yes |
| 4,800 bit/s | 208.3 µS | Yes |
| 7,200 bit/s | 138.9 µS | Yes |
| 9,600 bit/s | 104.2 µS | Yes |
| 14,400 bit/s | 69.4 µS | Yes |
| 19,200 bit/s | 52.1 µS | Yes |
| 38,400 bit/s | 26.0 µS | Yes |
| 56,000 bit/s | 17.9 µS | Yes |
| 57,600 bit/s | 17.4 µS | Yes |
| 76,800 bit/s | 13.0 µS | No |
| 115,200 bit/s | 8.68 µS | Yes |
| 128,000 bit/s | 7.81 µS | Yes |
| 230,400 bit/s | 4.34 µS | No |
| 256,000 bit/s | 3.91 µS | No |
| 460,800 bit/s | 2.17 µS | No |

All operations of the UART hardware are controlled by a clock signal which runs at a multiple of the data rate, typically 8 times the bit rate. The receiver tests the state of the incoming signal on each clock pulse, looking for the beginning of the start bit. If the apparent start bit lasts at least one-half of the bit time, it is valid and signals the start of a new character. If not, it is considered a spurious pulse and is ignored. After waiting a further bit time, the state of the line is again sampled and the resulting level clocked into a shift register. After the required number of bit periods for the character length (5 to 8 bits, typically) have elapsed, the contents of the shift register are made available (in parallel fashion) to the receiving system. The UART will set a flag indicating new data is available, and may also generate a processor [interrupt](https://en.wikipedia.org/wiki/Interrupt) to request that the host processor transfers the received data.

Communicating UARTs usually have no shared timing system apart from the communication signal. Typically, UARTs resynchronize their internal clocks on each change of the data line that is not considered a spurious pulse. Obtaining timing information in this manner, they reliably receive when the transmitter is sending at a slightly different speed than it should. Simplistic UARTs do not do this, instead they resynchronize on the falling edge of the start bit only, and then read the center of each expected data bit, and this system works if the broadcast data rate is accurate enough to allow the stop bits to be sampled reliably.

It is a standard feature for a UART to store the most recent character while receiving the next. This "double buffering" gives a receiving computer an entire character transmission time to fetch a received character. Many UARTs have a small first-in, first-out [FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)) buffer memory between the receiver shift register and the host system interface. This allows the host processor even more time to handle an interrupt from the UART and prevents loss of received data at high rates.

**Transmitter**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=4)]

Transmission operation is simpler as the timing does not have to be determined from the line state, nor is it bound to any fixed timing intervals. As soon as the sending system deposits a character in the shift register (after completion of the previous character), the UART generates a start bit, shifts the required number of data bits out to the line, generates and sends the parity bit (if used), and sends the stop bits. Since full-duplex operation requires characters to be sent and received at the same time, UARTs use two different shift registers for transmitted and received characters. High performance UARTs could contain a transmit FIFO (first in first out) buffer to allow a CPU or DMA controller to deposit multiple characters in a burst into the FIFO rather than have to deposit one character at a time into the FIFO. Since transmission of a single or multiple characters may take a long time relative to CPU speeds, a UART maintains a flag showing busy status so that the host system knows if there is at least one character in the transmit buffer or shift register; "ready for next character(s)" may also be signaled with an interrupt.

**Application**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=5)]

Transmitting and receiving UARTs must be set for the same bit speed, character length, parity, and stop bits for proper operation. The receiving UART may detect some mismatched settings and set a "framing error" flag bit for the host system; in exceptional cases the receiving UART will produce an erratic stream of mutilated characters and transfer them to the host system.

Typical serial ports used with personal computers connected to modems use eight data bits, no parity, and one stop bit; for this configuration the number of ASCII characters per second equals the bit rate divided by 10.

Some very low-cost [home computers](https://en.wikipedia.org/wiki/Home_computers) or [embedded systems](https://en.wikipedia.org/wiki/Embedded_systems) dispense with a UART and use the [CPU](https://en.wikipedia.org/wiki/Central_processing_unit) to sample the state of an input port or directly manipulate an output port for data transmission. While very CPU-intensive (since the CPU timing is critical), the UART chip can thus be omitted, saving money and space. The technique is known as [bit-banging](https://en.wikipedia.org/wiki/Bit-banging).

History[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=6)]

Some early [telegraph](https://en.wikipedia.org/wiki/Electric_telegraph) schemes used variable-length pulses (as in [Morse code](https://en.wikipedia.org/wiki/Morse_code)) and [rotating clockwork mechanisms](http://www.railroad-signaling.com/tty/m19/M19_8w.jpg) to transmit alphabetic characters. The first serial communication devices (with fixed-length pulses) were rotating mechanical switches (*commutators*). Various [character codes](https://en.wikipedia.org/wiki/Character_encoding) using 5, 6, 7, or 8 data bits became common in teleprinters and later as computer peripherals. The teletypewriter made an excellent general-purpose I/O device for a small computer.

[Gordon Bell](https://en.wikipedia.org/wiki/Gordon_Bell) of [DEC](https://en.wikipedia.org/wiki/Digital_Equipment_Corporation) designed the first UART, occupying an entire circuit board called a *line unit*, for the [PDP](https://en.wikipedia.org/wiki/Programmed_Data_Processor) series of computers beginning with the [PDP-1](https://en.wikipedia.org/wiki/PDP-1).[[3]](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#cite_note-comp-eng-3)[[4]](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#cite_note-4) According to Bell, the main innovation of the UART was its use of [sampling](https://en.wikipedia.org/wiki/Sampling_(signal_processing)) to convert the signal into the digital domain, allowing more reliable timing than previous circuits that used analog timing devices with manually adjusted [potentiometers](https://en.wikipedia.org/wiki/Potentiometer).[[5]](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#cite_note-5) To reduce the cost of wiring, backplane and other components, these computers also pioneered [flow control using XON and XOFF characters](https://en.wikipedia.org/wiki/Software_flow_control) rather than hardware wires.

DEC condensed the line unit design into an early single-chip UART for their own use.[[3]](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#cite_note-comp-eng-3) [Western Digital](https://en.wikipedia.org/wiki/Western_Digital) developed this into the first widely available single-chip UART, the WD1402A, around 1971. This was an early example of a [medium scale integrated circuit](https://en.wikipedia.org/wiki/Integrated_circuit). Another popular chip was the SCN2651 from the [Signetics 2650](https://en.wikipedia.org/wiki/Signetics_2650) family.

An example of an early 1980s UART was the [National Semiconductor](https://en.wikipedia.org/wiki/National_Semiconductor) [8250](https://en.wikipedia.org/wiki/8250_UART). In the 1990s, newer UARTs were developed with on-chip buffers. This allowed higher transmission speed without data loss and without requiring such frequent attention from the computer. For example, the popular National Semiconductor [16550](https://en.wikipedia.org/wiki/16550_UART) has a 16 byte [FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)), and spawned many variants, including the *16C550, 16C650, 16C750, and 16C850*.

Depending on the manufacturer, different terms are used to identify devices that perform the UART functions. [Intel](https://en.wikipedia.org/wiki/Intel) called their [8251](https://en.wikipedia.org/wiki/Intel_8251) device a "Programmable Communication Interface". [MOS Technology](https://en.wikipedia.org/wiki/MOS_Technology) [6551](https://en.wikipedia.org/wiki/MOS_Technology_6551) was known under the name "Asynchronous Communications Interface Adapter" (ACIA). The term "Serial Communications Interface" (SCI) was first used at [Motorola](https://en.wikipedia.org/wiki/Motorola) around 1975 to refer to their start-stop asynchronous serial interface device, which others were calling a UART. Zilog manufactured a number of [Serial Communication Controllers](https://en.wikipedia.org/wiki/Zilog_SCC) or SCCs.

After the [RS-232](https://en.wikipedia.org/wiki/RS-232) [COM port](https://en.wikipedia.org/wiki/COM_(hardware_interface)) was removed from most [IBM PC compatible](https://en.wikipedia.org/wiki/IBM_PC_compatible) computers in the 2000s, an external USB-to-UART serial adapter cable was used to compensate for the loss. A major[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] supplier of these chips is [FTDI](https://en.wikipedia.org/wiki/FTDI).[[6]](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter#cite_note-6)

Structure[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=7)]

A UART usually contains the following components:

* a clock generator, usually a multiple of the bit rate to allow sampling in the middle of a bit period.
* input and output [shift registers](https://en.wikipedia.org/wiki/Shift_register)
* transmit/receive control
* read/write control logic
* transmit/receive buffers (optional)
* system data bus buffer (optional)
* First-in, first-out ([FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))) buffer memory (optional)
* Signals needed by a third party DMA controller (optional)
* Integrated bus mastering DMA controller (optional)

Special transceiver conditions[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=8)]

**Overrun error**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=9)]

An "overrun error" occurs when the receiver cannot process the character that just came in before the next one arrives. Various devices have different amounts of buffer space to hold received characters. The CPU or DMA controller must service the UART in order to remove characters from the input buffer. If the CPU or DMA controller does not service the UART quickly enough and the buffer becomes full, an Overrun Error will occur, and incoming characters will be lost.

**Underrun error**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=10)]

An "underrun error" occurs when the UART transmitter has completed sending a character and the transmit buffer is empty. In asynchronous modes this is treated as an indication that no data remains to be transmitted, rather than an error, since additional stop bits can be appended. This error indication is commonly found in USARTs, since an underrun is more serious in synchronous systems.

**Framing error**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=11)]

A "framing error" occurs when the designated "start" and "stop" bits are not found. As the "start" bit is used to identify the beginning of an incoming character, it acts as a reference for the remaining bits. If the data line is not in the expected state (hi/lo) when the "stop" bit is expected, a *Framing Error* will occur.

**Parity error**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=12)]

A [Parity Error](https://en.wikipedia.org/wiki/Parity_bit) occurs when the [parity](https://en.wikipedia.org/wiki/Parity_(mathematics)) of the number of 1 bits disagrees with that specified by the parity bit. Use of a parity bit is optional, so this error will only occur if parity-checking has been enabled.

**Break condition**[[edit](https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver/transmitter&action=edit&section=13)]

A "break condition" occurs when the receiver input is at the "space" (logic low, i.e., '0') level for longer than some duration of time, typically, for more than a character time. This is not necessarily an error, but appears to the receiver as a character of all zero bits with a framing error. The term "break" derives from [current loop](https://en.wikipedia.org/wiki/Current_loop) signaling, which was the traditional signaling used for [teletypewriters](https://en.wikipedia.org/wiki/Teletypewriter). The "spacing" condition of a current loop line is indicated by no current flowing, and a very long period of no current flowing is often caused by a break or other fault in the line.

Some equipment will deliberately transmit the "space" level for longer than a character as an attention signal. When signaling rates are mismatched, no meaningful characters can be sent, but a long "break" signal can be a useful way to get the attention of a mismatched receiver to do something (such as resetting itself). [Unix-like](https://en.wikipedia.org/wiki/Unix-like) systems can use the long "break" level as a request to change the signaling rate, to support dial-in access at multiple signaling rates.