Serial Communication

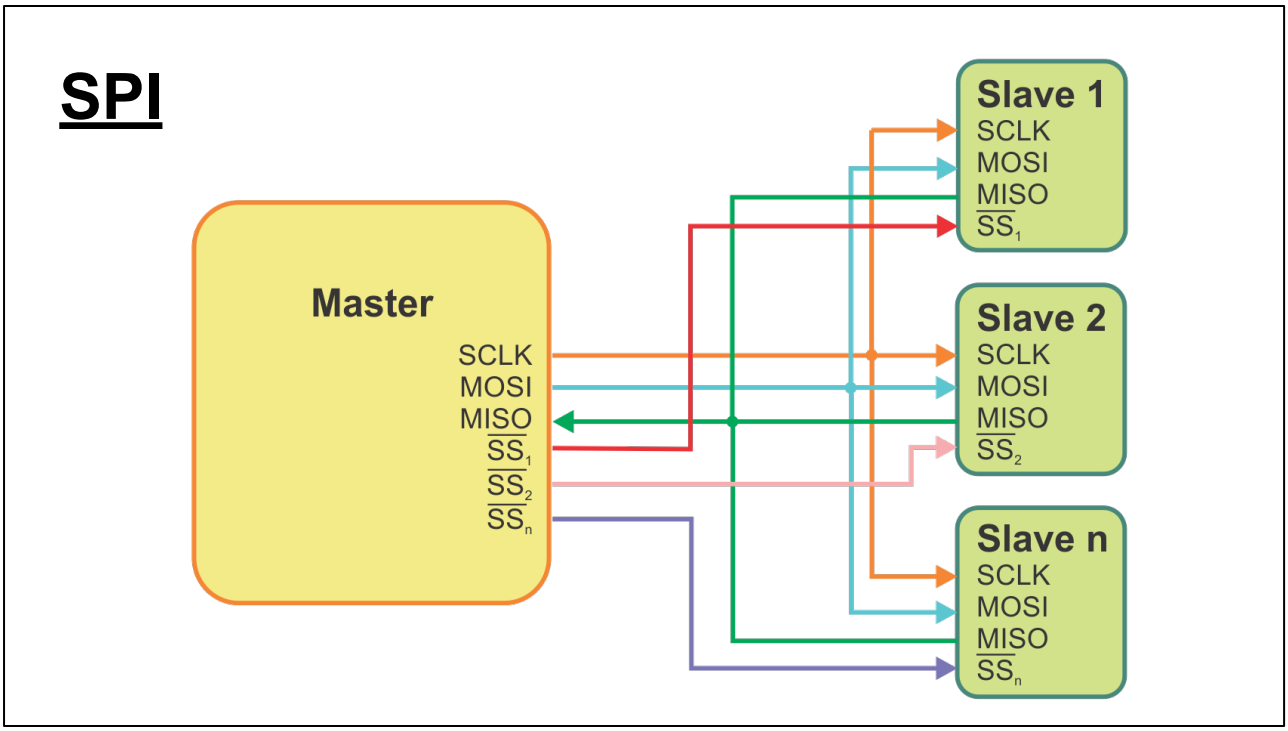
As we've discussed before, the Arduino has a lot of limitations. No ethernet, no display, limited storage, and only so many pins -- one microcontroller can only do so much! Fortunately, there are well-established technologies by which uC's -- including Arduino -- can talk to other chips and devices.

Your computer (or a Raspberry Pi) is going to be the most powerful such device you'll want to consider using, so we'll focus on it. But surprisingly, the protocol we use to speak to it is among the most archaic of the communications available to the Arduino. Let's briefly review some other ones first so you know what they are as you get further into electronics.

There are actually a variety of serial communication schemes, though people often use "serial" as shorthand for the more specific "UART". Serial just means that each bit comes after the next one -- we transmit information one bit at a time, instead of in *parallel*, with several bits transmitted at once.

One more word of preface: a lot of the terminology in this lesson discusses "masters" (devices that set the terms of the communication interchange) and "slaves" (devices that have a more passive role). This terminology is wince-inducing at best, and some people are very upset by it. Various engineering communities are in the process of changing this terminology. But this process is likely to be slow. And the master/slave language is baked into some otherwise-impenetrable acronyms that we will be using. So apologies, but we will be employing it here, too, for the sake of clarity. Feel free to do a search/replace for "manager" and "secretary" if you prefer.

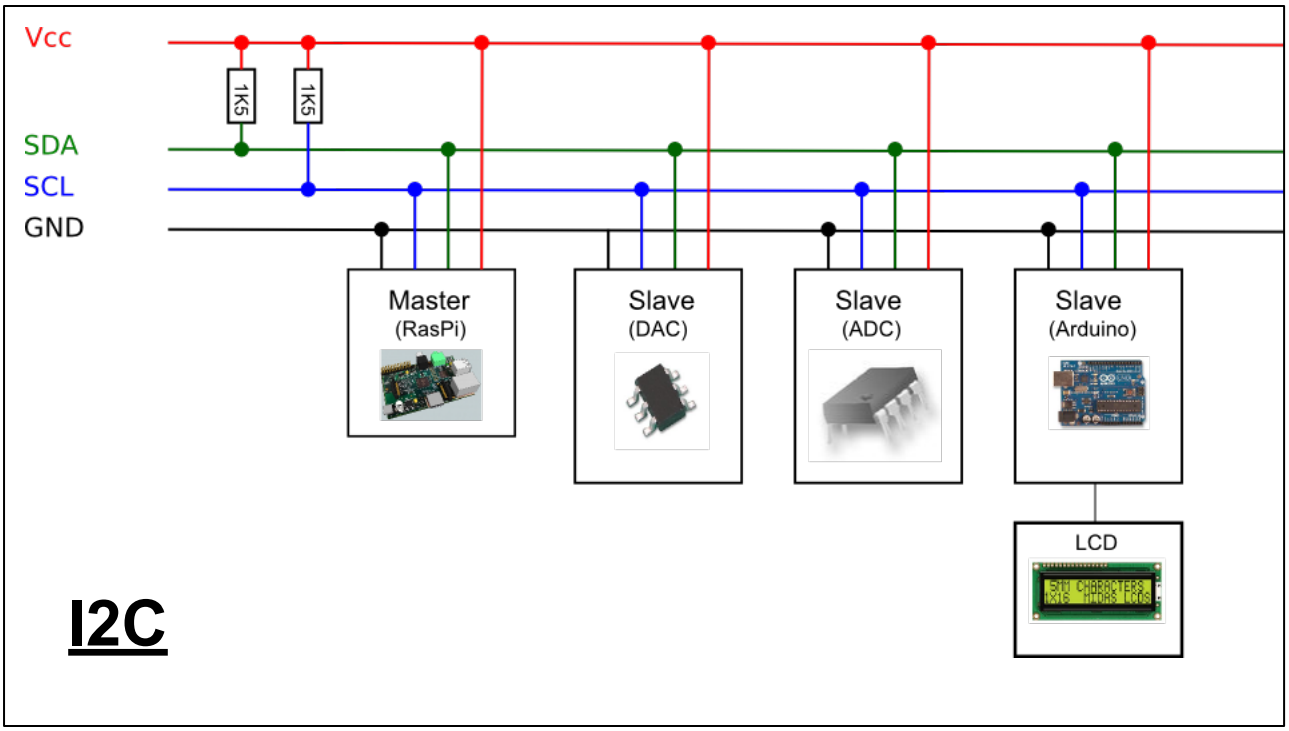
Let’s talk about a couple of serial schemes that are more advanced than the UART system we’ll be working with. You don’t need to understand these, I just want the acronyms to rattle around your brain for the day when you run into them.



SPI is very, very fast and fairly simple. It can achieve speeds in the megabits per second, which is pretty good. There are three primary pins: MOSI (Master Out/Slave In); MISO (Master In/Slave Out) and SCLK (Serial Clock). For each clock cycle, the Master emits one bit on MOSI and the Slave emits one bit on MISO. Each device collects the incoming bit and the cycle repeats.

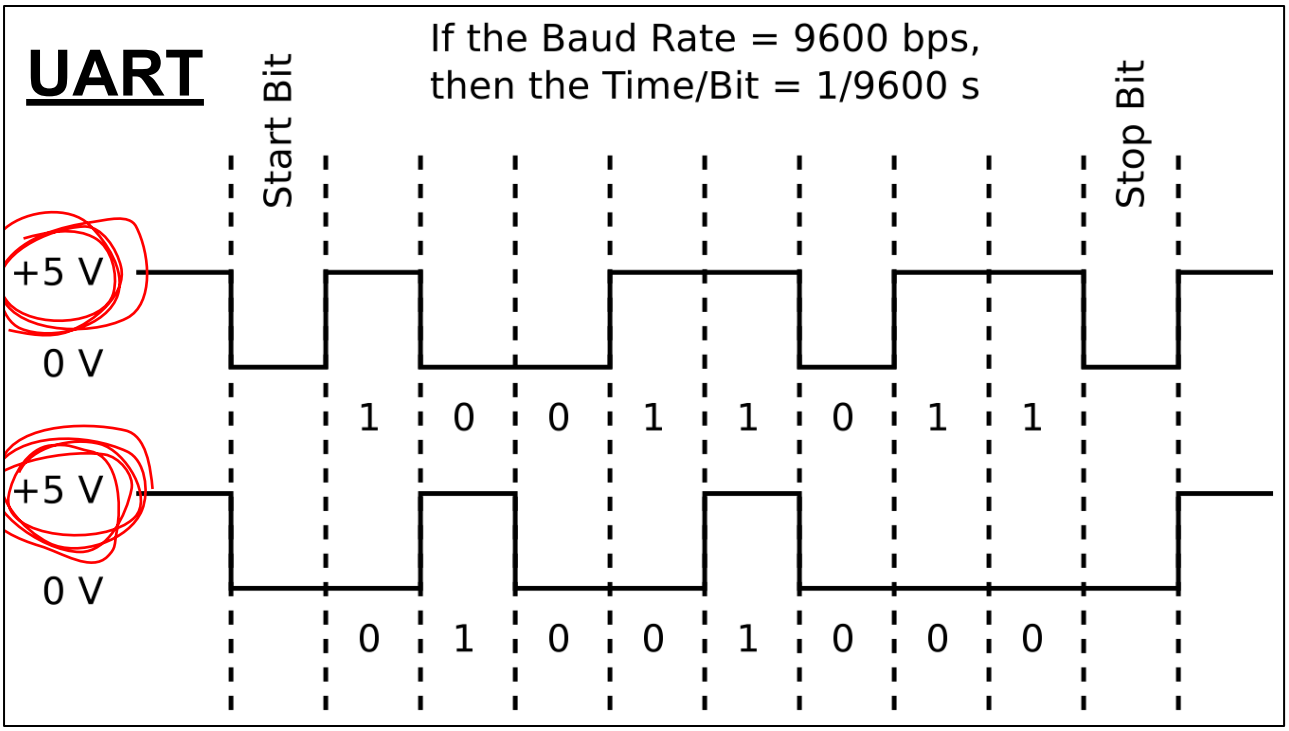
Note that this exchange is *always* bidirectional. If one device has nothing to say -- for instance, a Master that's taking a reading from a temperature sensor that cannot accept any configuration instructions -- it still has to send bits. It just doesn't matter what they are.

The other pins are for "Slave Select" and are used when you're talking to multiple SPI devices. It is simply set to HIGH or LOW. Only one SS pin in the system can be active at a time; this lets all the other devices know to ignore whatever messages they see. This allows all devices to share MISO, MOSI and SCLK, which cuts down on wiring complexity. But you still need an SS pin for each device, which introduces wiring complexity of its own.



SPI is very fast; I2C is not. Speeds vary, and some devices support high-speed modes, but 10 or 100 kilobits per second is normal. It lets you cut down on wiring, though, by having all of your devices share a few common wires, without the need for SS connections. This is accomplished by assigning addresses to each slave device in the I2C network. It's not too dissimilar from how we use IP addresses to send messages over the internet.

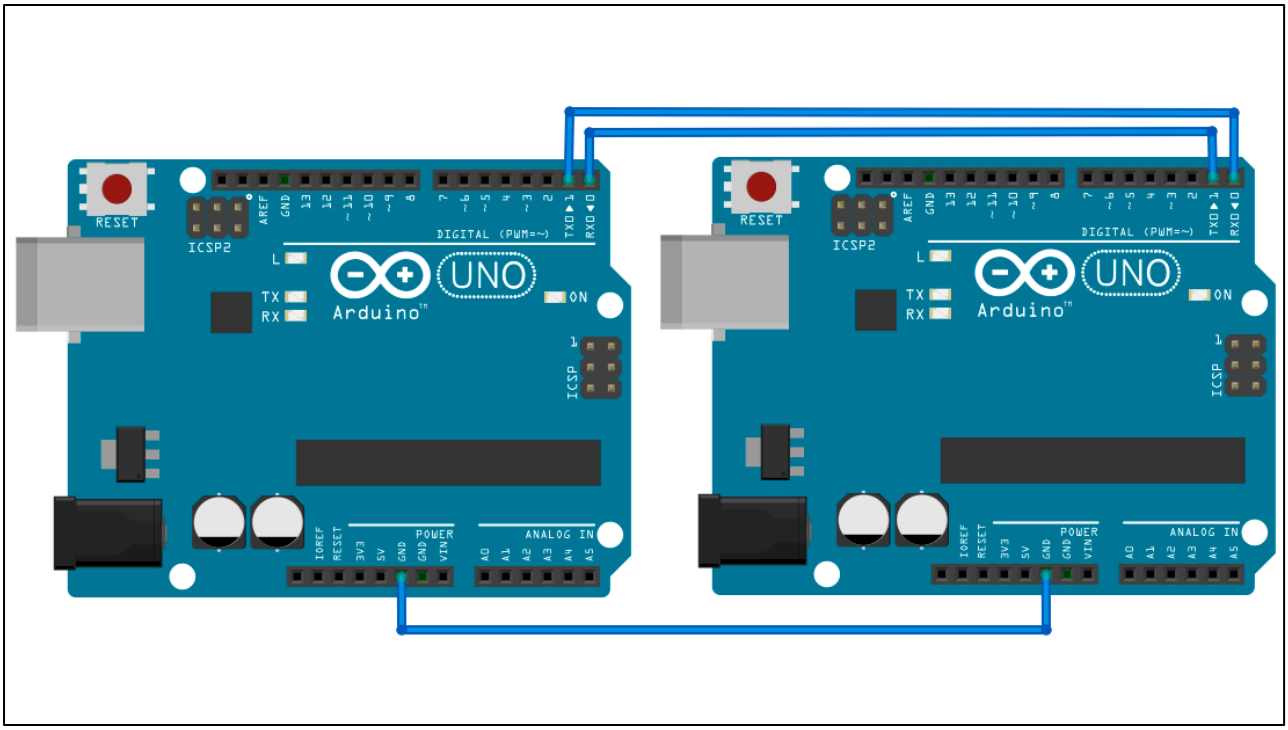
A related protocol is 1-Wire, which is even slower but combines Vcc, SDA and SCL into a single wire(!). Ever used the weird little electronic button locker key things at Spa World? They are probably 1-Wire.



Finally, we have UART - universal asynchronous receiver/transmitter. The "*asynchronous*" is the key part: there is no coordinating clock signal. The two ends both just have to agree to an expected speed (otherwise, how do you tell how many 1' s are in a block that goes 1111111111?) and do some complicated engineering tricks to try to stay synced up. But tricks can only go so far: UART limits top speeds quite severely (115200 bits per second is the usual maximum) and introduces other complexity, like parity and stop bit settings. It's just not a very good approach. But it is an old one, and if someone says "serial" they probably mean "UART".

It's also an easy one to use! On Arduino, moving bytes (not bits) with UART is quite simple.

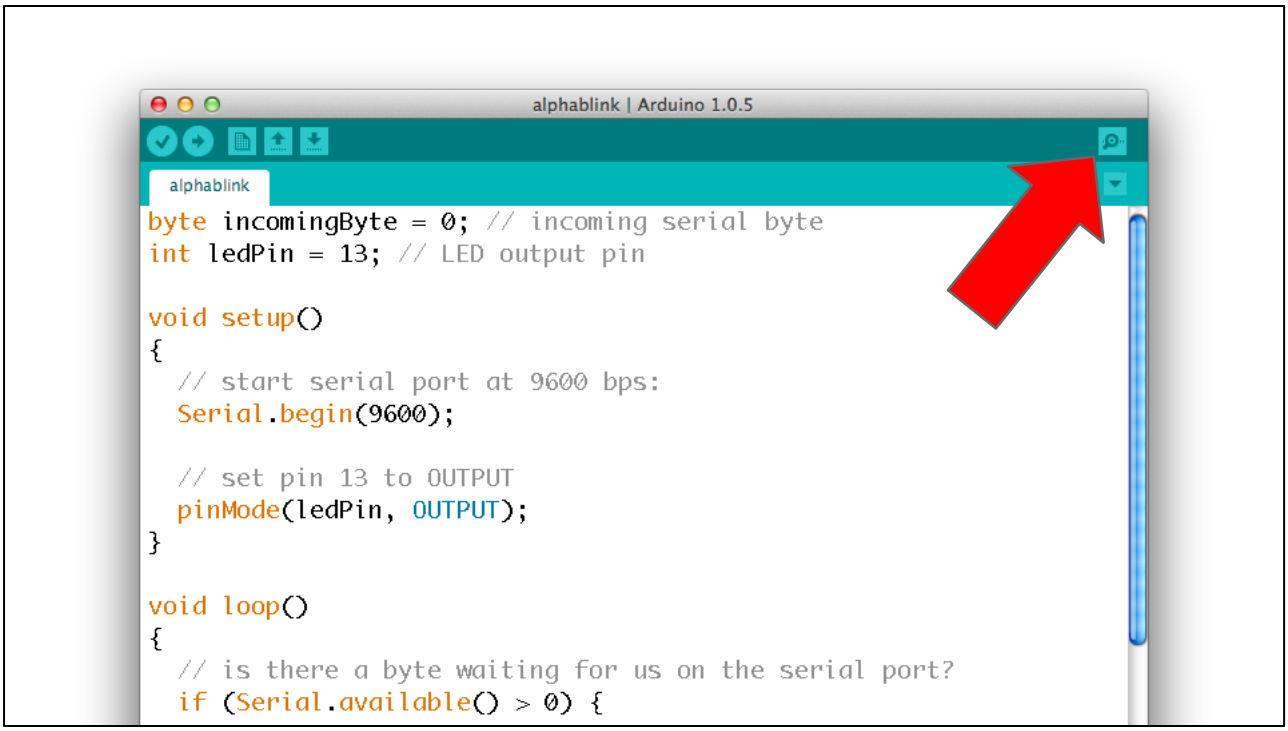
A word of warning, however: the Arduino is a five volt device. Many -- if not most -- serial devices are 3.3 volt devices. If you send 5V data from an Arduino to one of these, you can burn it out! You will need to use a "level shifter" device, the most famous of which is the MAX232.



Here is the simplest notion of how you would connect two serial devices -- in this case, two Arduinos. TX (transmit) on one goes to RX (receive) on the other. You also need to unify the grounds -- this is an important step for any electrical system, *all ground connections must be the same*. You cannot, for instance, have these two Arduinos running off different batteries (unless their grounds are connected).

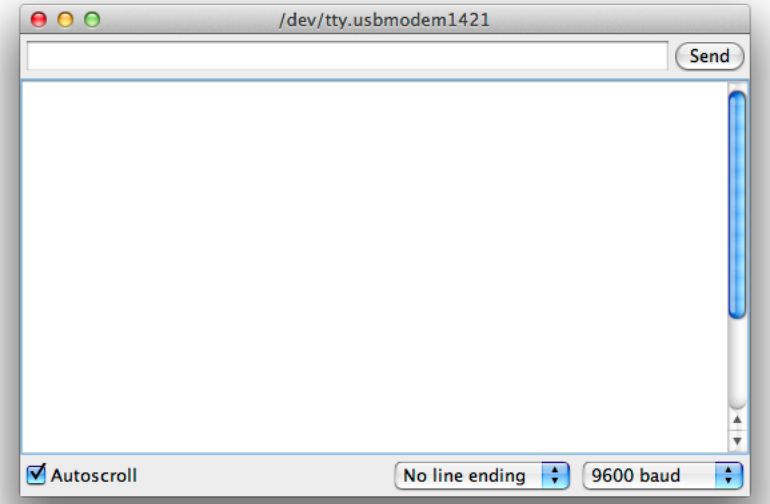
There are a few situations where ground doesn't have to be unified, but they are quite rare. Usually they involve dangerously high voltages or audio/video interference concerns, and use a device called an optoisolator to turn a signal into photons (via an LED) and then back into electricity (via a photodetector) all within a single plastic blob. You are unlikely to run into these situations any time soon.

Note: these serial pins are also connected to the USB serial system, which is used to program the Arduino! This makes keeping pins 0 and 1 unused (for anything besides serial communication) is a VERY good idea unless you absolutely have to. If you connect devices to them, serial communication can be affected, which means Arduino-program-uploading can be affected. If you must use them, it is a good idea to disconnect everything from pins 0 and 1 while reprogramming the device, then reattach for testing.



Go ahead and open the alphablink program from the Github repository (https://github.com/sbma44/arduino-class/tree/master/lesson-3) and load it onto your Arduino.

The easiest way to debug serial communication in Arduino is with the environment's built-in Serial Monitor. Click the little button!



Note that the Arduino reboots whenever you open a serial connection to it, including by opening the Serial Monitor. This is necessary because of the way that the Arduino gets programmed. It's possible to override it, but it's usually easier to write your program in a way that expects this behavior.

Note a few things: baud rate and the send dialog. If the baud isn't set to what the Arduino is speaking, you'll get gibberish (or nothing)! The send box is good for sending individual strings.

The alphablink program reads one character at a time from the serial connection and blinks pin 13. A = 1 blink, B = 2 blinks, etc. Let's walk through it.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

One thing to remember: the Arduino doesn't know what English is. Serial connections just send bytes, one at a time. For convenience, we can refer to specific bytes as ASCII characters (according to this table). But this is arbitrary. A byte can mean anything!

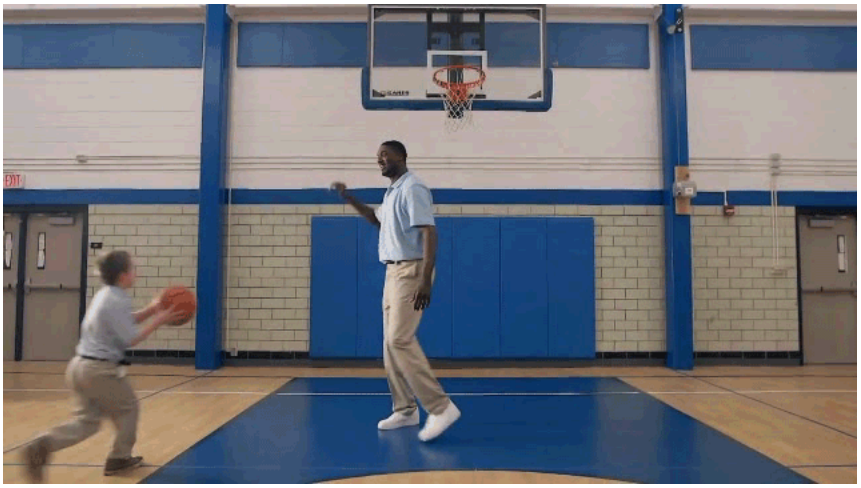
On your computer, several bytes are used for each character, so that extended characters can be represented. The Arduino software sticks to ASCII.

This is important to remember, and not just to make it easier to understand alphablink. You might want to send a bunch of bytes to an Arduino for things other than representing text! Say you want it to pause for some number of seconds. Sending the Arduino a byte that means 100 is MUCH easier than sending it the three bytes that spell out "100", splitting them up, figuring out how to map "1" to 1 and "0" to zero and then multiplying and adding everything back into 100. Sometimes the result of this is that what comes out of the Serial Monitor is ugly. That's okay!

Okay! Let's reverse what we've been doing. open up stutterer.ino and load it onto your Arduino. (Apologies to any stutterers out there -- I had trouble with it myself as a kid, but I know it's not funny if you're suffering from it.)

Let's walk through this code. It's pretty straightforward, with the exception of randomSeed(), which is just there to give us properly randomized numbers. Computers only generate pseudorandom numbers based on a starting "seed" number. Your computer does this based on the movements of your mouse, keys, the time -- all kinds of stuff. But the Arduino is so numb to the outside world that its random numbers are REALLY bad unless we take an analog voltage reading to start off with.

Obviously we don’t want to work in the serial monitor forever. Let’s add some Python! Open up stutterer_read_blocking.py. Change the serial port to the correct string for your computer (visible at the top of the Serial Monitor) and try running it. Make sure the Serial Monitor is closed -- only one serial connection at a time!



Alright, it's time to talk about blocking. A *blocking operation* is one that makes everything stop until it finishes. The pyserial readline() function is blocking. Once it's called, nothing else will happen in your Python program until it sees the newline character that tells it to return a value.

We don't have to live this way! Open up stutterer_read_nonblocking.py. Let's try running it, and walking through the differences in the code.

YOUR CHALLENGE: using the code skeleton in bikeshare/arduino_indicator.py, build an LED display that shows how many bikes are left. You can do it!

BONUS CHALLENGE: make your button code from last week initiate a web request from your computer.

What happens when you decide to make an Arduino project that relies on serial communication, but don't want to dedicate your laptop to it? Consider a Raspberry Pi! They cost $35 ($25 without ethernet) and can speak to the Arduino using the exact same Python code we used tonight.