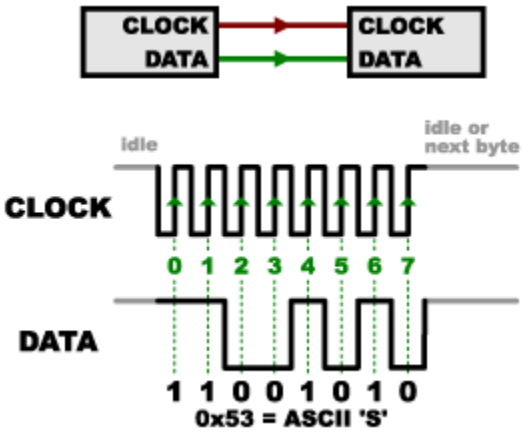
A SYNCHRONOUS SOLUTION

SPI works in a slightly different manner. It's a "synchronous" data bus, which means that it uses separate lines for data and a "clock" that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. This could be the rising (low to high) or falling (high to low) edge of the clock signal; the datasheet will specify which one to use. When the receiver detects that edge, it will immediately look at the data line to read the next bit (see the arrows in the below diagram). Because the clock is sent along with the data, specifying the speed isn't important, although devices will have a top speed at which they can operate (We'll discuss choosing the proper clock edge and speed in a bit).



One reason that SPI is so popular is that the receiving hardware can be a simple shift register. This is a much simpler (and cheaper!) piece of hardware than the full-up UART (Universal Asynchronous Receiver / Transmitter) that asynchronous serial requires.

Programming for SPI

Many microcontrollers have built-in SPI peripherals that handle all the details of sending and receiving data, and can do so at very high speeds. The SPI protocol is also simple enough that you (yes, you!) can write your own routines to manipulate the I/O lines in the proper sequence to transfer data. (A good example is on the Wikipedia SPI page.)

If you're using an Arduino, there are two ways you can communicate with SPI devices:

1. You can use the shiftIn() and shiftOut() commands. These are software-based commands that will work on any group of pins, but will be somewhat slow.
2. Or you can use the SPI Library, which takes advantage of the SPI hardware built into the microcontroller. This is vastly faster than the above commands, but it will only work on certain pins.

You will need to select some options when setting up your interface. These options must match those of the device you're talking to; check the device's datasheet to see what it requires.

- The interface can send data with the most-significant bit (MSB) first, or least-significant bit (LSB) first. In the Arduino SPI library, this is controlled by the setBitOrder() function.
- The slave will read the data on either the rising edge or the falling edge of the clock pulse. Additionally, the clock can be considered "idle" when it is high or low. In the Arduino SPI library, both of these options are controlled by the setDataMode() function.
- SPI can operate at extremely high speeds (millions of bytes per second), which may be too fast for some devices. To accommodate such devices, you can adjust the data rate. In the Arduino SPI library, the speed is set by the setClockDivider() function, which divides the master clock (16MHz on most Arduinos) down to a frequency between 8MHz (/2) and 125kHz (/128).

- If you’re using the SPI Library, you must use the provided SCK, MOSI and MISO pins, as the hardware is hardwired to those pins. There is also a dedicated SS pin that you can use (which must, at least, be set to an output in order for the SPI hardware to function), but note that you can use any other available output pin(s) for SS to your slave device(s) as well.
- On older Arduinos, you’ll need to control the SS pin(s) yourself, making one of them low before your data transfer and high afterward. Newer Arduinos such as the Due can control each SS pin automatically as part of the data transfer; see the Due SPI documentation page for more information.

Source: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi