# Skill Demo 7: Communication Protocols

v1.5

**Goals:**

1. Learn the basics of RS-232 Serial, SPI, and I²C
2. Learn the advantages and disadvantages of each
3. Learn to read/write remote registers via I²C

**Tools/supplies:**

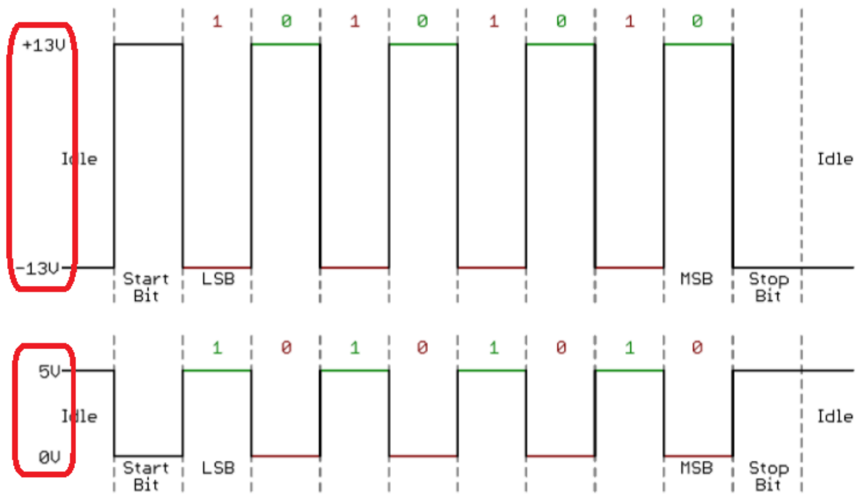
- Microcontroller
- A breadboard and breadboarding wire
- MFRC522-based RFID reader from Mega 2560 kit
- GY-521 accelerometer and gyroscope from Mega 2560 kit
- Libraries: MFRC522 for the RFID reader and Adafruit MPU 6050 for the GY 521

**Background**

- 3651: Communication Protocols Overview
- https://learn.sparkfun.com/tutorials/serial-communication/rules-of-serial

## Background

While simpler ICs often interface using digital or analog inputs and outputs, more complicated ICs implement digital communications protocols over a reduced number of wires. These protocols are usually serial in nature, sending multiple bits over the wires one after another.



This timing diagram shows both a TTL (bottom) and RS-232 signal sending 0b01010101

## RS-232

The first of these communications protocols is RS-232, which we have already been using with the Serial library in Arduino. This protocol allows two devices to communicate with one another using two dedicated lines, a transmit and a receive. Each device puts out digital 0's or 1's on its transmit line, and reads in 0's or 1's on its receive line. You have to ensure each Tx line is driven by only one device, as swapping the Tx and Rx line is an easy mistake to make.

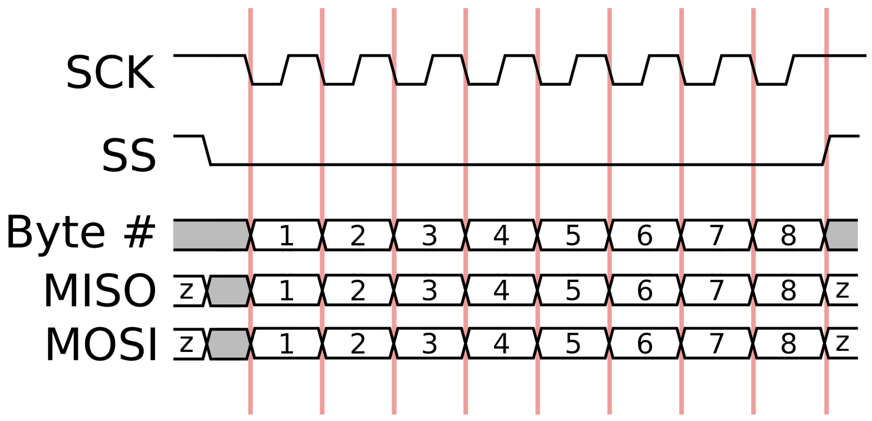
The voltages for a '0' or a '1' are defined by RS-232 to be ±3 to ±15V, but such voltages are typically difficult to find on modern logic ICs (like our microcontrollers), so a de facto standard, often called 'TTL serial' has become popular, using 0V for a '0' and 3.3V or 5V as a '1'. This 5V voltage is what our Arduinos use.

Each byte is sent separately, with a 'start' bit and a 'stop' bit used to mark the ends of an 8-bit byte. Variations of this exist, with additional start bits, stop bits, and optional parity bits, but the most common is the '8-N-1' configuration with 8 data bits, no parity bits, and 1 stop bit. A constant bit rate has to be used so the receiver can figure out where the boundaries are between bits.

One issue with RS-232 is that the sender and receiver *have* to agree on a clock rate. This is usually described in 'baud', giving the number of 0's or 1's per second. The most common baud rate, and the default baud rate for Arduino sketches, is 9600 baud. Other common baud rates are  19200 and 115200 baud, which you might have seen used in Arduino sketches for sending serial data faster.

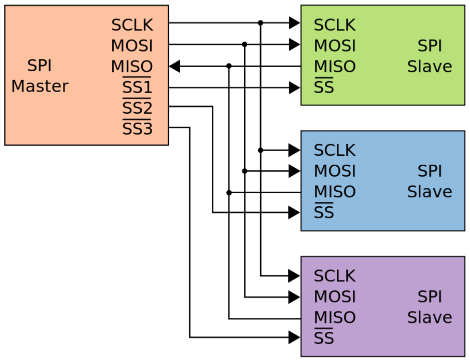
## Serial Peripheral Interface

SPI is a second common interface. It is similar to RS232, except that is uses extra pins for a separate clock line to explicitly define bit boundaries, and usually uses a dedicated line for a "chip select" or "slave select" to avoid needing to use start and stop bits.



SPI also has a 'master' and 'slave' device, with the master driving the dedicated clock line (SCK) and it's transmit pin (master out, slave in, MOSI). The master also drives the SS or CS pin if used. The ‘slave’ device only drives it’s transmit pin (master in, slave out, MISO), transmitting in lockstep with the SCK clock signal it receives. This means that for every 8 clock pulses, a byte is transferred from master to slave, while a second byte from slave to master.

While it uses more pins, SPI is easier to implement than RS-232, and the MISO, MOSI, and SCK pins can be reused across multiple slave devices, as long as only one slave is selected, and thus driving the shared MISO pin, at a time.

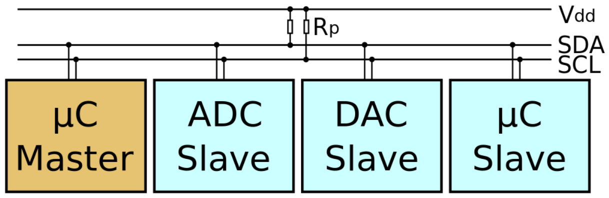


There are several variations in SPI devices as to the clock polarity and relative phase, leading to 4 different incompatible modes, and the maximum clock speed varies widely across devices, although it is usually above 1MHz.

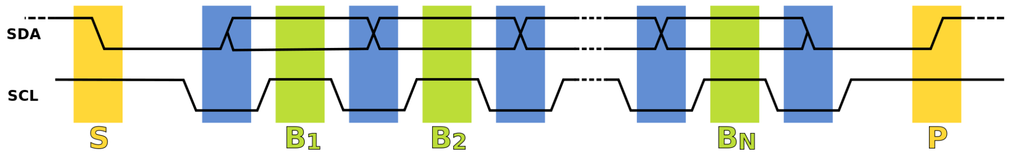
## I²C

The last common protocol is I²C, pronounced “I-two-C” or “I-squared-C”, is the strangest but most flexible of the three. It relies on two shared data lines, SCL for clock, and SDA for data. Each of these lines is pulled to VCC with a resistor (typically 2.2K), and one or more devices can pull each line to ground. Devices *never* drive these lines high, only pulling them low, allowing multiple devices to safely share the lines at all times, as they'll never attempt to drive the lines to opposite voltages simultaneously.

I²C can have up to 127 devices on a single pair of SDA/SCL lines, making efficient use of I/O pins and wires, but is limited in speed by the pull-up resistor configuration, usually running with a clock rate of 100 KHz or 400 KHz.



To transmit, a device temporarily becomes the ‘master’ using a special sequence on the SDA and SCL lines called a ‘start sequence’. It then sends an address byte by pulling SDA and SCL low in sequence, and then waits for an acknowledgement bit from the ‘slave’ that it’s sending to. The first byte sent after a start sequence is always an address, used to select a specific slave. Subsequent data bytes are then sent to that slave, with a special ‘stop sequence’ used to release control of the bus.



Because of its complex nature, I²C is usually accessed using a library, allowing your code to select a slave device, and send and receive an arbitrary number of bytes to and from it. In Arduino, this is wrapped up in the Wire library.

I²C is also the communication used in Sparkfun's QWIIC connector system and Adafruit's STEMMA QT connector system, enabling hundreds of prototyping boards to be plug-and-play.

### Dedicated Hardware in Microcontrollers

While these protocols can be implemented using just digital reads and writes, that is usually too slow, so almost all microcontrollers have some dedicated hardware for speaking these protocols. Like the analog-to-digital converter or PWM, this dedicated hardware is usually connected to specific pins. Take a look at the documentation for your microcontroller for more information on which pins are used for these protocols. Keep in mind that the RS-232 interface is already used for communication with the PC, but the signals still exposed via the TX and RX lines on pins 0 and 1.

To make these protocols easier to use, there are already libraries for them: the Serial library we've already used, the SPI library for SPI, and the Wire library for I²C. To make things even simpler (and another advantage of open source) people have applied these libraries and made abstractions of the code specific to the hardware you might want to use.

For example if you want to read data from one of your sensors using the I²C bus, if you start from scratch you would have to look up the address of the device and how it transmits the various bits of data. Once you have this information you'd have to implement code, get the data and then clean it to make it parseable.

Libraries make this process easier. You might find a library for the sensor you are using, that takes care of the communication protocol for you and directly gives you the exact data you need from the sensor. In both the problems for the skill demo, we shall use such libraries - the MFRC522 for the RFID module and the Adafruit MPU6050 for the GY521 module. Note that these libraries also make use of the Arduino SPI and Wire libraries which are mentioned above, they just add a layer of abstraction on top of it which is hardware specific and makes our life easier.  A good place to start off with new libraries is to read and test out all of the examples (File ➜ Examples ➜ Library Name ➜ )

## Higher-level protocols

Higher level communication is usually implemented on top of these protocols. For example, you have probably used ASCII text terminated with newlines over RS-232 for debugging.

I2C and SPI are often used to read and write memory addresses on remote devices. This process is chip-dependent, but usually consists of writing a memory address byte before each read or write. Depending on the chip, the address might be two or more bytes, and the address might include a read/write flag to tell the slave if we are planning to do a read or a write.

I$^2$C is the core of many low-speed higher-level protocols, used for everything from reading computer temperature sensors to configuring HDMI displays. SPI, on the other hand, is more often used for older devices, or devices that need higher communication speeds. One common example is a backward-compatibility mode of SD and MMC cards, so you can read and write files on an MicroSD card using a few libraries.

# Skill Demo 7: Communication Protocols

NAME____________________ GTID_____________________
☐ CS3651A
☐ CS3651B

## 1. RS-232

Find the ASCII code for the first character of your first name.
Calculate the binary bit pattern of this character if it were sent over a 5V serial data connection, such as your Arduino's TX pin. Don't forget the start and stop bits. Assume 1 start bit, 8 data bits, and 1 stop bit.

What is the hex value of the byte? ___________

What are the binary values? _________

Draw the waveform assuming +/-3V is being used for the bits (be sure to label your axes!):

How long would it take to send 10 bytes like this if the clock rate were 1MHz? _________

Initials________________ Date_______________

## 2. SPI

You have been tasked with making a new security device with a RFID based security system. Your system should read the unique address from an RFID tag and produce some output which depends on the RFID tag that is read. Example outputs are

- turn on a green led if the ID matches your database and a red led if it does not
- have an rgb led that shows a unique color based on the rfid tag address
- rotate a servo motor (like opening a lock), etc.

Specifically, first write a program that can read the unique address on an RFID card. You can use the MFRC522 RFID reader and sample cards provided in your kit. Make sure you use the correct supply voltage, 3.3 V.

Your program should use the SPI protocol to communicate with the reader and be able to detect at least two states using the RFID cards. Based on the ID read you need to trigger some output, and your output should also demonstrate two states. Feel free to use any libraries available.

Initials________________ Date_______________

## 3. I2C

Use the I2C protocol to communicate with the GY-521 6-axis accelerometer/gyro board from your kit. Connect to the GT-521 using I²C, read values, and print those values via Serial. (optionally you can also plot the values in your serial plotter). You can use any libraries available.

Initials________________ Date_______________