

OSC communication between Arduino and PC via serial connection

OSC (open sound control)

Features:

- Open-ended, dynamic, URL-style symbolic naming scheme
- Symbolic and high-resolution numeric argument data
- Pattern matching language to specify multiple recipients of a single message
- High resolution time tags
- "Bundles" of messages whose effects must occur simultaneously

Data types:

- 32-bit two's complement signed integers
- 32-bit IEEE floating point numbers
- Null-terminated arrays of 8 bit encoded data (C-style strings)
- arbitrary sized blob (e.g. audio data, or a video frame)
-

OSC messages consist of an address pattern, a type tag string, arguments and an optional time tag.

Arguments are represented in binary form with 4-byte alignment.

OSC addresses are URL style (e.g. /foo/bar) and therefore always begin with the character '/'.

Looking at the raw byte stream, however, it is not clear where an OSC message actually begins because there is no singular character signifying the start or end of a message. This means that OSC messages have to be send/received using a protocol.

For transmission via internet/local network, it is common to use UDP.

For serial communication (e.g. arduino with USB), the preferred protocol is SLIP.

https://en.wikipedia.org/wiki/Open_Sound_Control

<https://opensoundcontrol.org>

Serial communication

There are two basic types of serial communication:

Synchronous: SPI (serial peripheral interface), I²C (inter-integrated circuit), ...

Asynchronous: 1-Wire, RS-232, TTL-Serial, ...

RS-232 and TTL-Serial are commonly referred to as simply "Serial". They only differ in voltage levels and logic. The Arduino Uno has TTL-Serial (5 volt).

TTL-Serial is bidirectional, using one wire for each direction (marked as Tx and Rx). It is generally meant for communicating only between two devices.

Because there's usually no external clock signal, both devices have to operate at the same baud rate (here: bits per second). They must also use the same packet configuration and endianness (the order in which data bits are send).

Each packet consists of:

1 stop bit | 5-9 data bits | 0-1 parity bits | 1-2 end bits

The most typical configuration is *8N1*: 8 data bits, no parity, and 1 stop bit (the start bit is implied).

More info here: <https://learn.sparkfun.com/tutorials/serial-communication>

SLIP (serial line internet protocol)

SLIP is a very simple protocol which modifies a stream of bytes by

- appending a special "END" byte (ASCII character 192) to mark the boundary between datagrams (sequence of bytes)
- if the END byte occurs in the data to be sent, it is "escaped" by sending the two byte sequence ESC (219), ESC_END (220) is sent instead,
- if the ESC byte occurs in the data, the two byte sequence ESC (219), ESC_ESC (221) is sent.
- variants of the protocol may also begin packets with END.

SLIP requires a serial port configuration of 8 data bits, no parity. It doesn't provide any error checking.

https://en.wikipedia.org/wiki/Serial_Line_Internet_Protocol

OSC between Arduino and PC via USB

There is an OSC library for Arduino developed at CNMAT:

<https://github.com/CNMAT/OSC>

Many example sketches seem to be broken, but the library itself works. You can find documentation on the git hub site.

The example sketch *testOSC.ino* shows how to use the library for simple bidirectional OSC communication.

Processing

In Processing you can use the "Serial" library for sending/reading bytes via a serial connection.

<https://processing.org/reference/libraries/serial/>

There is no built in SLIP encoding/decoding function, so you have to implement it.
(See the *ArduinoSLIPSerialToUDP.pde* sketch for hints how to do it.)

Pure Data

In Pure Data there is an external called [comport] for serial communication:

<https://puredata.info/downloads/comport>

The "mrpeach" library has two objects called [slipenc] and [slipdec] which perform SLIP encoding/decoding. It also has some useful objects for dealing with OSC messages, like [packOSC], [unpackOSC] and [routeOSC].)

Both are included in Pd extended, but you can also get them in Pd vanilla using the deken plug-in (or copying the externals from a Pd extended folder).

See the *ArduinoSLIPSerialToUDP.pd* patch for how to work with these objects.

Other programs

If your target program cannot deal with raw serial input (e.g. Ableton, Reaper) but you still want to communicate with an Arduino via a serial connection, you can use the processing sketch *ArduinoSLIPSerialToUDP.pde* or the Pure Data patch *ArduinoSLIPSerialToUDP.pd* as a convenient interface between Arduino and your program. You can test these two quickly with *ArduinoTestOSC.pde* or *ArduinoTestOSC.pd* together with the Arduino sketch *testOSC.ino*.

Note: Make sure that the baudrates always match!

Alternatives to OSC

If you want to send many, many packets from and to the Arduino in very quick intervals, there's a point where OSC via serial could get too slow. This is because the usual baud rates (e.g. 9600, 19200, 38400, 57600, 115200) for serial devices is quite low compared to, for example, Gigabit Ethernet. The Arduino serial monitor is limited to 115200 baud, but apparently this is not the maximum rate an Arduino can handle. Some people claim to have achieved baud rates up to 2 Mbit/s:

<http://arduino.stackexchange.com/questions/296/how-high-of-a-baud-rate-can-i-go-without-errors>

But there's another way to speed up the transfer rate:

If you can limit yourself to 7 bit (0-127) and don't really need symbolic addresses, you can send plain SLIP packets over the serial connection (all the "special" bytes of the SLIP protocol are above 127, so their won't be any conflicts). Consider a message like this:

ID, value1, value2, ...

You could easily route such a message by its ID (0-127). If you want to send integers with more than 7 bit – like 10 bit integers from `analogRead()` –, you can store them into adjacent bytes, e.g. the first 7 bits into value1 and the next 3 bits into value2.

Short comparison:

The OSC Message `"/analog 1000"` needs 18 bytes (after SLIP encoding):

`(192 47 97 110 97 108 111 103 0 44 105 0 0 0 3 232 192)`

while a plain SLIP packet might only need 5 bytes:

`(192 0 104 7 192)`

[ID = 0, value = 104 + 7 << 7 = 1000]