


This page is called [SerialCommunication](#)

Pages below this page:

Contents of this page:

Page Tree

 Brief Overview of Serial Communication

## Brief Overview of Serial Communication

Serial Communication refers to a class of communications protocols which transmit data one bit at a time. The most common of these protocols in a laboratory setting is RS-232. While most electronic measurement devices will feature RS-232 connectivity, there are many different ways in which this is implemented. The following document should act as a guide to RS-232 interfacing in general, with specific information on the IGC-3 Pressure Gauge used in the Omicron pod.

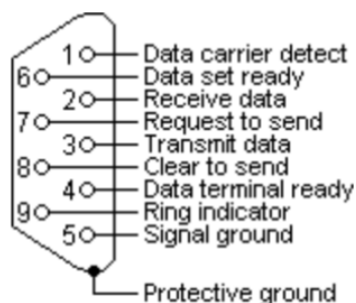
### Physical Connection

Often one of the most difficult aspects of setting up an interfacing system is identifying the correct cable to use. A device with serial connectivity may use one of three port types: DB-9, DB-25, or RJ45 (ethernet). DB-25 is relatively uncommon, and so only DB-9 and RJ45 will be covered in detail.

#### DB-9

DB-9 is the standard 9 pin RS-232 connection type. In most laboratory applications, only pins 2 (receive, Rx), 3 (transmit, Tx) and 5 (ground, GND) are used, and commonly in what is referred to as a “null-modem” or “crossover” connection. In a null-modem connection the transmit pin of one device is connected to the receive pin of the other, and vice-versa. This allows a two way communication between devices. The standard DB-9 pinout is as shown:

**RS232 DB9 pinout**



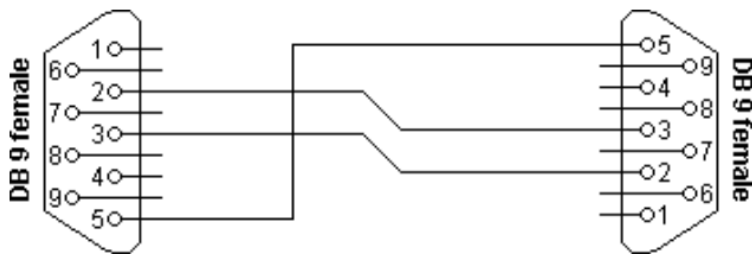
If the device in question utilizes a null-modem connection, you can either purchase a readymade null-modem cable or build one yourself. As there are different null-modem configurations (3 wire, 4 wire, etc.), it may be simpler (and cheaper) to construct a cable specific to your application.

To build such a cable you will need:

- (1) 2 RJ45-to-DB9 adaptors. These are inexpensive and widely available online. As DB9 has both male and female connectors, make sure to order the appropriate gender.
- (2) A CAT-5E standard ethernet cable.
- (3) A RS232-to-USB adaptor. An inexpensive option which has worked well is the TRENDNet TU-S9, available on Amazon and other online retailers.

Note that the RS232-to-USB adaptor is essential; a typical ethernet-to-USB adaptor will not allow you to bypass this. Such adaptors will try to interpret the data as an ethernet signal, whereas RS232-to-USB adaptors come with drivers to convert the signal back to RS232.

Constructing a 3-wire null-modem cable is quite simple, and other configurations require only small alterations. Take the two RJ45/DB9 adaptors and choose three colours of wire to work with (assuming the adaptors use the same colour scheme). We will use red, green and blue here but any other three could be used. Attach the red wire to pin 2 on one adaptor and pin 3 on the other, the blue wire to pin 3 on one and pin 2 on the other, and the blue wire to pin 5 on both (as shown in diagram below). After attaching them together with the ethernet cable and testing to ensure things are working properly, remove or isolate the unused wires.



## RJ-45

In the case that your device uses RJ45 ports for serial connections, you will need to reference the manual of the device to find the pinout configuration used. An ethernet cord in this application can be thought of simply as a collection of eight wires encased together. The term pinout refers to how these wires are utilized by the device. As there is no standard configuration, I will use the IGC-3's pinout and cable construction as an example.

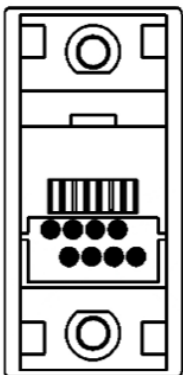
A RJ45-DB9 adaptor contains 8 wires, each of which corresponding to a particular wire in the ethernet cord. Unfortunately the colour coding of these wires is also not standardized, but a commonly configuration is the following:

Blue	1
Orange	2
Black	3
Red	4
Green	5
Yellow	6
Brown	7
White	8

Colour coding of wires in ethernet cables is standardized, and wires will commonly be referred to by colour rather than by number. This standard is as follows:

Orange/White	1
Orange	2
Green/White	3
Blue	4
Blue/White	5
Green	6
Brown/White	7
Brown	8

If your adaptor comes with a similar chart, disregard the one above and refer to this. If it does not and the chart above does not correspond to the wire colours used in your adaptor, numbering can be determined by looking into the adaptor: in the diagram below, the top four dots correspond to connections for wires 1,3,5,7 and the bottom four correspond to 2,4,6,8.



Once wire colours are determined, it is simply a matter of connecting the wires as your device's documentation

specifies. With the IGC-3, the following chart is given:

Host RS232 pin	Wire colour	PVC pin & function
0V or "Ground Return"	Orange/White	Pin 1 – 0V
Transmit	Orange	Pin 2 – Receive
Receive	Green/White	Pin 3 – Transmit

Referencing above charts and DB-9 pinout, the following connections are made:

- Blue wire of adaptor (corresponding to orange/white ethernet wire) is connected to pin 5 of adaptor (ground).
- Orange wire of adaptor (corresponding to orange ethernet wire) connected to pin 3 of adaptor (transmit).
- Black wire of adaptor (corresponding to green/white ethernet wire) is connected to pin 2 of adaptor (receive).

Note: With the IGC-3 in particular, it is possible to daisy chain units together via ethernet cables. To do this, simply connect the RJ45 port of one unit to the next (each has two ports). This allows a computer to interface with multiple units through a single connection. Additional information on this provided in the section on command generation below.

## Interfacing Software

The necessary software to interface with a given device is highly dependent upon that device's implementation of serial communication. There are, however, general guidelines that should apply to all devices.

Any common programming language should have packages/modules for serial communication available, and a wealth of online resources to help you through the process. [PySerial](#), a Python package, will be used here for demonstration purposes, but the parameters and functions used should have analogues in any similar package.

NOTE: All code that follows assumes that [PySerial](#) and the struct module have been imported.

## Port Configuration

The device manual should outline the necessary value for all of these parameters, or indicate how to configure them. It is essential that ports in your program are configured with exactly the same parameters as the device port you are connecting to.

In PySerial, the most commonly used parameters are:

- port - This specifies which USB port to communicate through
- baudrate - specifies the number of bits per second
- bytesize - specifies the number of data bits per "word", or message sent
- parity - enable/disable parity checking, a simple form of error detection
- stopbits - specifies number of stop bits, indicating the end of a message to the device
- timeout - specifies how long your program will wait for a response from the device

The name of the port you are using can be determined in multiple ways, but the easiest is using PySerial's built in

`list_ports` function from the command line by entering `python -m serial.tools.list_ports`. The rest of the parameters should be provided to you in the device manual, or if they are configurable there should be a recommended setting for each. A port is configured with the following syntax:

```
myPort = serial.Serial(  
    '/dev/cu.usbserial', #your port name here  
    baudrate = 9600,  
    bytesize = 8,  
    timeout = 3,  
    stopbits = serial.STOPBITS_ONE,  
    parity = serial.PARITY_NONE,  
)
```

See Pyserial's [documentation](#) for a more in-depth explanation of port configuration.

## Command Generation

Once ports are properly configured you can begin communicating with the device. In [PySerial?](#), this is done using the `read()` and `write()` functions. `write()` takes a command string as a parameter and sends it to the device. `read()` then receives the response string. The content of these strings is determined by the device you are working with.

Most modern devices will use an ASCII based communications protocol and supply full documentation on the construction of serial commands, as well as how to interpret responses. In some cases, as with the IGC-3, more esoteric protocols may be used.

The IGC-3 uses a non-standard implementation of the MODBUS protocol, meaning that command strings are comprised of an array of hexadecimal values. The basic structure of an IGC-3 command string is as follows:

```
<device_address><function code><data package><crc>
```

- Device Address: A 2 byte hex value (01-99) which specifies the address of the IGC you are trying to communicate. This can be changed in the setup menu on the IGC.
- Function Code: A 2 byte hex value (17h). This must be included for the device to recognize the command.
- Data Package: When reading a single parameter, the data package is comprised of four bytes; the first two bytes specify the parameter to be read, the second two specify the number of bytes that will be read. Four zero bytes must be appended to this to indicate that no parameters are to be written. This will be outlined in more detail below.
- CRC: CRC error detection is provided by appending two hex bytes to the end of a command string. The content of these bytes is determined by the command string being sent. A method for generating CRC bytes is provided below.

Parameter codes for the IGC-3 can be found in the [EmComm Parameter List Handbook](#). For parameters which return a floating point value, 1 “word” (or 2 bytes) is read. The [EmComm MODBUS Communications Handbook](#)

contains a more indepth overview of command generation.

CRC bytes are generated using a standard algorithm. More information on this algorithm and CRC in general can be found [here](#), but understanding how these bytes are generated is not essential. [Here](#) is a lookup table and associated code in Python which will allow for easy generation of CRC bytes. This could should be easily portable to other languages if needed.

As an example, suppose we want to read the Ion Gauge Pressure parameter from the IGC with address 01. The command we write would be

"\x01\x17\x00\x9A\x00\x02\x00\x00\x00\x00\x00"; \x01 for the device address, \x17 for the function code, \x00\x9A to specify that we would

like to read from parameter 154, \x00\x02 to indicate that two bytes (1 word) are to be read, and \x00\x00\x00\x00 to indicate that no parameters are to be written. Using the linked code above, a full message including CRC can be generated as follows:

```
command = "\x01\x17\x00\x9A\x00\x02\x00\x00\x00\x00\x00"
crc = calcString(message1, INITIAL_MODBUS)
command += struct.pack('<H', crc)
```

Once the command string is generated, the following code performs the communication with the IGC:

```
response = []
myPort.write(command)
response += myPort.read(myPort.inWaiting())
```

## Response Parsing

For the IGC-3, response strings are structured as follows:

```
<device_address><function_code><number_of_data_bytes><data><crc>
```

After the code from the last section is run, response will contain a hexadecimal array of the form:

```
01h 17h 04h xxh xxh xxh xxh <crc1> <crc2>
```

where the =xxh='s are the data bytes. The following function takes an array of this form and returns a floating point value to three decimal places:

```
def dataFormat( raw ):
    hexString = ''.join(i for i in raw[3:7])
    return float( "%.3e" % struct.unpack('<f', hexString)[0])
```

See [this](#) file for a complete implementation of all the code described above.

If the device you are communicating with uses an ASCII based protocol, the response you get should be an ASCII valued array of hex bytes. To ensure that Python interprets these correctly, the following code can be used:

```
#assume the array named response already contains the received data

#create output string
output = ''

#interpret each character as 7-bit ASCII
for char in response:
    data += chr(ord(char) & 0x7F)
```

## Useful Links, etc.

- Lammert Bies' [website](#) is a great resource for anything to do with serial communication.
- A full manual of the IGC-3 can be found [here](#).
- [EmComm MODBUS Communications Handbook](#).
- [EmComm Parameter List Handbook](#).
- [Pyserial downloads/documentation](#).
- A full Python example script can be found [here](#).
- A printable version of this document is available [here](#).
- If further support is needed in interfacing with the IGC-3 specifically, Richard Kubiak from Epimax helped design the software system and is willing to provide support as needed. He can be contacted at [richard@epimax.com](mailto:richard@epimax.com).

--- This topic: [LairWiki](#) > [WebHome](#) > [ContentRoot](#) > [InformationTechnology](#) > [SerialCommunication](#)

Topic revision: 10 Sep 2016, DylanGreen

Copyright &© by the contributing authors. All material on this collaboration platform is the property of the contributing authors.

Ideas, requests, problems regarding Foswiki? [Send feedback](#)

