National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL

Logging Data From Serial Ports

What is a serial port?

> In computing, a serial port is a serial communication physical interface through which information transfers in or out one bit at a time (in contrast to a parallel port). Throughout most of the history of personal computers, data was transferred through serial ports to devices such as modems, terminals and various peripherals.

(ref: Wikipedia)

Usually, this means the RS-232 standard, or, less frequently, RS-485 .

They are a very basic method of communication with devices, still in common use for scientific equipment. However, most modern desktop and laptop computers do not come with serial ports, and instead we use them via a USB converter.

Physically, the serial port is (usually) a 9- or 25- pin "D-Sub" port, although other connectors are possible, including a 3.5mm jack of the sort used for headphones.

Today we'll be using a USB serial 9-pin male D-sub converter and a thermometer with a 9-pin female connector.

Serial port access in python

Access to serial ports in Python is done via the pyserial module. Your laptops should already have it installed, so it can be imported in the usual way with:

```
import serial
```

As we will usually only need the Serial class, you could also use

```
from serial import Serial
```

if you prefer.

```
#!/usr/bin/env python

import serial
```

```
ser = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate=9600,
    bytesize=serial.EIGHTBITS,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE
)

# "8" here is specific to the Papouch thermometer device
print ser.read(size=8)
ser.close()
```

I've added the most common parameters that describe how to connect to a particular device. "baudrate" defines the speed of the communication; bytesize, parity and stopbits define the format and error-checking of the incoming data. Eight bits, no parity checking and one stopbit is the most common, but others do exist - you will need to check the datasheet for the particular device.

The Papouch thermometer we're using today communicates at 9600 baud, Eight bits, no parity checking and one stopbit, (you may see a shorthand like "9600 8N1" for this) which are pyserial's defaults. We can therefore reduce the above snippet to:

```
#!/usr/bin/env python

import serial

ser = serial.Serial(
    port='/dev/ttyUSB0',
)

print ser.read(size=8)

ser.close()
```

in this particular case. You may need to specify other parameters for other devices(see pyserial's API documentation for more details).

If you run this code in the usual way:

```
$ python readserial_basic.py
+025.1C
```

A note on port names

"`/dev/ttyUSB0`" is Linux's way of referring to the first USB serial port. (if you had another, it would be `/dev/ttyUSB1`, etc. Built-in serial ports would be `/dev/ttyS0` etc.) On Windows machines, the portname will be of the form `COM3` (or `COM4`, `COM1`, etc.) You may need to experiment to determine which the USB converter has attached to.

Why `ser.read(size=8)` ?

If you refer to the Papouch thermometer datasheet 's "Communication Protocol" section, you will see:

```
<sign><3 characters – integer °C>
<decimal point><1 character – tenths of °C>
<C><Enter>
```

as a description of the output. In ASCII coding, each character is one byte so each temperature from the thermometer is eight bytes. We can't use the `readline()` function at this stage as the End of Line character is a carriage return rather than a newline. More on this later.

Date and time

`+025.1C` is a valid temperature, but if you're logging an instrument, you will also need the time (for a static instrument; obviously you may also need to log position, etc.)

Assuming your computer has an accurate clock, you can use that. We'll need Python's datetime module.

```
#!/usr/bin/env python

from datetime import datetime
import serial

ser = serial.Serial(
   port='/dev/ttyUSB0',
   baudrate=9600,
)

print datetime.utcnow().isoformat(), ser.read(size=8)

ser.close()
```

`datetime.utcnow().isoformat()` is, as you might expect, a command to return the current UTC is ISO format, e.g.:

```
2014-03-06T11:55:43.852953 +025.3C
```

Obviously, this is much more precise than required here, but you may well have instruments that are capable of sub-second accuracy at some stage.

A major problem with the above code is the line:

```
print datetime.utcnow().isoformat(), ser.read(size=8)
```

It is not immediately obvious that the `datetime.utcnow()` call can return in advance of the `ser.read()` call, which will stall until the thermometer returns data. To ensure that the timestamp and the temperature line up as closely as possible, we can store the data in a variable

and output the variable and the time at the same time, reducing the gap between them:

```
datastring = ser.read(size=8)
print datetime.utcnow().isoformat(), datastring
```

Date and Time formats

I have used the ISO format here. The `datetime` module is capable of outputting any format you require (left as an exercise for the reader) but I would urge you to consider very carefully if you are thinking of using another one. ISO format is unambiguous, well-defined, and easy to sort electronically, and these things will save you time when you come back to your data later.

Continuous logging

In most cases, you will need to log more than one data point. A basic modification is fairly simple, using a `while` loop:

```
#!/usr/bin/env python

from datetime import datetime
import serial

ser = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate=9600,
)

while ser.isOpen():
    datastring = ser.read(size=8)
    print datetime.utcnow().isoformat(), datastring

ser.close()
```

returns something like:

```
2014-03-06T14:20:28.147494 +023.9C
2014-03-06T14:20:28.849280 +024.0C
2014-03-06T14:20:38.769283 +024.0C
2014-03-06T14:20:48.688270 +024.1C
2014-03-06T14:20:58.608165 +024.1C
2014-03-06T14:21:08.528660 +024.2C
2014-03-06T14:21:18.447250 +024.3C
2014-03-06T14:21:28.367255 +024.3C
2014-03-06T14:21:38.288262 +024.3C
2014-03-06T14:21:48.208270 +024.2C
```

While you could redirect the output to a file in the shell:

```
$ python readserial_continuous.py >> ~/temperature_log
```

This has the drawback that the data are not always appended to the file until the output stops (i.e. the program is interrupted), or the buffer fills, so using Python's inbuilt file functions is preferable.

Here's the final version, also using `readline()`:

```
#!/usr/bin/env python
'''This version of the readserial program demonstrates using python to write
an output file'''

from datetime import datetime
import serial, io

outfile='/tmp/serial-temperature.tsv'

ser = serial.Serial(
   port='/dev/ttyUSB0',
   baudrate=9600,
)

sio = io.TextIOWrapper(
   io.BufferedRWPair(ser, ser, 1),
   encoding='ascii', newline='\r'
)

with open(outfile,'a') as f: #appends to existing file
   while ser.isOpen():
      datastring = sio.readline()
      #\t is tab; \n is line separator
      f.write(datetime.utcnow().isoformat() + '\t' + datastring + '\n')
      f.flush() #included to force the system to write to disk

ser.close()
```

A note on `readline()`

The example thermometer *always* returns exactly eight bytes, and so `ser.read(size=8)` is a good way of reading it. However, in the more general case, instruments do not always return fixed-length data, and instead separate the readings (or sets of readings) with a special character. Often, this is newline , but sometimes it is, as here, carriage return or a combination of the two. The pyserial module provides `readline()` to handle this case; it will read all data up to the separator (Usually called "end-of-line", or EOL).

In versions of Python prior to v2.6, the EOL character could be specified in a `readline()` call. However, due to change in the way the pyserial module functions, this is no longer directly possible in recent versions. The Serial class must be wrapped in the `io.TextIOWrapper` class:

```
sio = io.TextIOWrapper(io.BufferedRWPair(ser, ser, 1), encoding='ascii',
newline='\r')
```

and then the readline can be called on the wrapper:

```
datastring = sio.readline()
```

I have specified the EOL character as carriage return, although in fact Python's Universal Newlines function will automatically compensate once the `io.TextIOWrapper` is used.

Further exercises

Command-line options

It is possible to specify parameters such as the outputfile name and port on the command line using various python modules, such as optparse . If you have time it is good practice to make your programs as useful as possible by allowing options to be changed. (e.g. if you have two USB->serial converters and you need to listen to both)

`f.flush()` statement

What happens when this is removed? Why is this a problem?