a fake serial device to practice on

In [1]:
```
# fakeSerial.py
# D. Thiebaut

# a Serial class emulator
class Serial:

    ## init(): the constructor.  Many of the arguments have default values
    # and can be skipped when calling the constructor.
    def __init__( self, port='/dev/ttyS1', baudrate = 4800, timeout=1,
                  bytesize = 8, parity = 'N', stopbits = 1, xonxoff=0,
                  rtscts = 0):
        self.name     = port
        self.port     = port
        self.timeout  = timeout
        self.parity   = parity
        self.baudrate = baudrate
        self.bytesize = bytesize
        self.stopbits = stopbits
        self.xonxoff  = xonxoff
        self.rtscts   = rtscts
        self._isOpen  = True
        self._receivedData = ""
#         self._data = "It was the best of times.\nIt was the worst of times.\n"
        self._data = bytearray([0x01, 0x04, 0x49, 0x31, 0x6C, 0x00]) # typical 6 byte pulser response

    ## isOpen()
    # returns True if the port to the Arduino is open.  False otherwise
    def isOpen( self ):
        return self._isOpen

    ## open()
    # opens the port
    def open( self ):
        self._isOpen = True

    ## close()
    # closes the port
    def close( self ):
        self._isOpen = False

    ## write()
    # writes a string of characters to the Arduino
```

```
    def write( self, string ):
        print( 'Arduino got: "' + string + '"' )
        self._receivedData += string

    ## read()
    # reads n characters from the fake Arduino. Actually n characters
    # are read from the string _data and returned to the caller.
    def read( self, n=1 ):
        s = self._data[0:n]
        self._data = self._data[n:]
        #print( "read: now self._data = ", self._data )
        return s

    ## readline()
    # reads characters from the fake Arduino until a \n is found.
    def readline( self ):
        returnIndex = self._data.index( "\n" )
        if returnIndex != -1:
            s = self._data[0:returnIndex+1]
            self._data = self._data[returnIndex+1:]
            return s
        else:
            return ""

    ## __str__()
    # returns a string representation of the serial class
    def __str__( self ):
        return  "Serial<id=0xa81c10, open=%s>( port='%s', baudrate=%d," \
              % ( str(self.isOpen), self.port, self.baudrate ) \
              + " bytesize=%d, parity='%s', stopbits=%d, xonxoff=%d, rtscts=%d)"\
              % ( self.bytesize, self.parity, self.stopbits, self.xonxoff,
                  self.rtscts )
```

In [2]:
```
# testSerialSimulator.py
# D. Thiebaut
# This program energizes the fakeSerial simulator using example code taken
# from http://pyserial.sourceforge.net/shortintro.html
#

# import the simulator module (it should be in the same directory as this program)
# import fakeSerial as serial


# Example 1  from http://pyserial.sourceforge.net/shortintro.html
def Example1():
#     ser = serial.Serial(0)  # open first serial port
    ser = Serial(0)  # open first serial port
    print( ser.name )       # check which port was really used
    ser.write("hello")      # write a string
    ser.close()             # close port

# Example 2  from http://pyserial.sourceforge.net/shortintro.html
def Example2():
#     ser = serial.Serial('/dev/ttyS1', 19200, timeout=1)
    ser = Serial('/dev/ttyS1', 19200, timeout=1)
    x = ser.read()          # read one byte
    print( "x = ", x )
    s = ser.read(10)        # read up to ten bytes (timeout)
    print( "s = ", s )
    line = ser.readline()   # read a '\n' terminated line
    ser.close()
    print( "line = ", line )

# Example 3  from http://pyserial.sourceforge.net/shortintro.html
def Example3():
#     ser = serial.Serial()
    ser = Serial()
    ser.baudrate = 19200
    ser.port = 0
    print( ser )

    ser.open()
    print( str( ser.isOpen() ) )

    ser.close()
```

```
    print( ser.isOpen() )


# Example1()
# Example2()
# Example3()
```

background info gleaned from the manual

In [3]:
```
# D Command - enter assign address mode
# 0x00 0x00 0x44 0x00 0x00

# I Command - information
# 0x00 0x00 0x49 Value 0x00
# value =
# 0x00 - returns 9 bytes - instrument type
# 0x01 - returns 9 - instrument serial number
# 0x02 - returns 5 - firmware and hardware revisions
# 0x03 - returns 9 - hardware serial number
# 0x04 - returns 5 - bandwidth
# 0x05 - returns 6 - max excitation pulse amplitude
# 0x06 - returns 19 - highpass filter cutoff frequencies
# 0x07 - returns 19 - lowpass filter cutoff frequencies
# 0x08 - returns 20 - pulser energy capacitor values
# 0x09 - returns 5 - front panel hardware and firmware revisions
# 0x0A - returns 10 - gain range

# A Command - assign instrument address - 0x01 is OK
# 0x00 0x00 0x41 Address 0x00

# E Command - exit assign address mode
# 0x00 0x00 0x45 Address 0x00

# Function Control
# Address 0x00 Command Value 0x00

# [b] Blink Command - sets the blink rate of the power LED
# Byte 3 = 0x62
# Byte 4 = 100-255 100 = Blink Slowly, ... 254 = Blink Rapidly, 255 = LED fully on.

# [c] Configure Command
# Byte 3 = 0x63
# Byte 4 = 0x00 to 0x03
# If Byte 4 bit 0 is set to 0, a 5kHz limit will be applied to ext. triggering. Trigger
# pulses that exceed this rate will be ignored. Set bit 0 to 1 to disable this feature.
# If Byte 4 bit 1 is set to 0, automatic update messages will be sent to the remote PC to
# confirm front panel control changes. Set bit 1 to 1 to disable this feature.

# [d] Damping Command - sets the damping impedance value
# Byte 3 = 0x64
# Byte 4 = 0-15
```

```
# The damping impedance, in Ohms, for Byte 4 values from 0 - 15 are,
# 1000, 333, 200, 143, 111, 91, 77, 67, 58, 52, 47, 43, 40, 37, 34, and 32 respectively.

# [e] Energy Command - sets the energy value for the pulser.
# Byte 3 = 0x65
# Byte 4 = 0-3
# Sets pulse energy level (0 - 3) => (min, .. , max)
# The energy contained in a pulse is a function of both the 0 - 3 pulse energy control value
# and the 0 - 15 pulser voltage control value. Expressed as a function, the pulse energy is:
# Pulse Energy (Joules) = L*(100 + 25*V) 2 for 475V pulsers
# Pulse Energy (Joules) = L*(100 + 53.3*V) 2 for 900V pulsers
# where the value of L for the corresponding pulse energy control values (0,1,2,3) is,
# respectively, (155 x10 -12 , 310 x10 -12 , 675 x10 -12 , 1350x10 -12 ), and V is the pulse voltage
# function value, (0,..,15). For example, for a 475V pulser with the maximum energy control
# value of 3 and V = 15, the pulse energy equals .3046 mJ
# See also the PRF command regarding allowable limits to
# the pulser firing rate for DPR300 units with the 900V pulser voltage option installed.

# [g] Gain Command - sets the receiver gain
# Byte 3 = 0x67
# Byte 4 = 0-79
# Sets gain in 1 dB steps (0 - 79) => (-13, .. , 66) dB

# [h] High Pass Filter Command
# Byte 3 = 0x68
# Byte 4 = 0-5
# Sets high pass filter
# (0 - 5) => (DC, 1, 2.5, 5, 7.5, 12.5) MHz.

# [l] Low Pass Filter Command
# Byte 3 = 0x6C
# Byte 4 = 0-5
# Sets low pass filter
# (0 - 5) => (3, 7.5, 10, 15, 22.5, 35) for 35 MHz
# (0 - 5) => (5, 10, 15, 22.5, 35, 50) for 50 MHz.

# [m] Mode Command
# This command selects which functions should respond to front panel control. In other
# words, manual controls can either be totally or selectively disabled in DPR300 units that
# possess both a manual front panel and remote PC control options. Two data bytes are
# required for the mode command and thus the mode command consists of the following byte
# sequence:
# Byte1 Byte2
```

```
# Address 0x01
# Byte3
# 0x6D
# Byte4
# Value
# Byte5
# Value
# Byte6
# 0x00
# The Byte 4 and Byte 5 bit values correspond to the following DPR300 functions:
# Byte 4 bits0 - 5 Unassigned
# bit6 - Pulser impedance
# bit7 - Pulser voltage
# Byte 5 bit0 - Echo/Through receiver source
# bit1 - Int/Ext trigger source for pulser
# bit2 - PRF (Internal trigger pulse repetition frequency)
# bit3 - Pulser energy
# bit4 - Receiver low pass filter
# bit5 - Receiver high pass filter
# bit6 - Receiver gain
# bit7 - Pulser damping
# When a mode bit is set to value 1, the corresponding function will respond to the front
# panel controls. When the bit is set to value 0, the function will not respond to the front
# panel. Any combination of front panel controls may be enabled or disabled. To enable
# all front panel controls, set the mode byte values to 0xFF.

# [o] Pulser On/Off Command
# The pulser on/off command enables/disables the pulser from firing and enables/disables the
# pulser voltage supply. When the pulser is disabled and then enabled, a short recovery
# period of 500 mSec is necessary to allow the pulser voltage supply to stabilize. For fastest
# power supply recovery, the pulser should not be fired during this recovery period.
# Byte 3 = 0x6F
# Byte 4 = 0,1
# Disable/enable pulser
# (0,1) => (Disable, Enable)
# There is no Pulser on/off switch on the front panel. Thus, the ‘o’ command is unaffected by
# the mode command described above. The ‘o’ command can be used to disable the pulser
# even if the mode command has configured all instrument functions to be controlled by the
# front panel.

# [p] PRF Command - selects the pulse repetition frequency (PRF) of the pulser when internal
# triggering is selected.
# Byte 3 = 0x70
```

```
# Byte 4 = 0-15
# Program Control Values in Hz. corresponding to the PRF function values (index) from 0 -
# 15 are, respectively, 100, 200, 400, 600, 800, 1000, 1250, 1500, 1750,
# 2000, 2500, 3000, 3500, 4000, 4500, and 5000.
# DPR300 instruments with the 900V pulser option automatically limit the PRF when the
# instrument is operating in internal-trigger mode so as to protect the pulser against excess
# power dissipation. The applied PRF limit depends upon the pulser voltage and energy
# settings. The following table expresses the recommended maximum PRF index as a
# function of the pulser voltage index and the pulser energy index.

# [r] Receiver Command - selects between Pulse-Echo and Through mode operation
# Byte 3 = 0x72
# Byte 4 = 0,1
# Sets Receiver to Echo/Through (0,1) => (Echo, Through)

# [t] Trigger Command - selects between internal and external triggering
# Byte 3 = 0x74
# Byte 4 = 0,1
# Sets Int/Ext trigger source
# (0,1) => (Internal, External)

# [v] Voltage Command - selects the pulser voltage.
# See also the PRF command regarding allowable limits to the pulser
# firing rate for DPR300 units with the 900V pulser voltage option installed.
# Byte 3 = 0x76
# Byte 4 = 0-15
# The pulser voltage supply values in Volts corresponding to the pulser
# voltage function values from 0 - 15 are, respectively 100, 125, 150, 175,
# 200, 225, 250, 275, 300, 325, 350, 375, 400, 425, 450, and 475. Each
# step increments the pulser supply voltage by 25 volts. For DPR300 units
# with the 900V pulser option, the voltages corresponding to the function
# values 0 - 15 are, respectively, 100, 153, 207, 260, 313, 367, 420, 473,
# 527, 580, 633, 687, 740, 793, 847, and 900.

# [z] Impedance Command - selects between low and high pulser impedance
# Byte 3 = 0x7A
# Byte 4 = 0,1
# Sets pulser impedance (0,1) => (max, min) Ohms.


# Response to Commands - returns 6 bytes
# Following the issuance of a command, the addressed instrument will confirm that the command
# has been implemented by responding with the byte sequence shown below:
```

```
# Byte 1 Byte 2 Byte 3 Byte 4 Byte 5 Byte 6
# Address 0x04 Command Value 'FP Value" Indicator

# where ‘Address’ is the instrument address. Byte 2 describes the number of bytes that follow
# starting with Byte 3 (four bytes in the above example). ‘Command’ is the byte value of the
# received command, Byte 4 is the data byte received with the command and Byte 5 is the current
# value of the associated front panel control. Indicator Byte 6 has value 0x00 or 0x01 if the
# instrument function was last set to the Byte 4 or Byte 5 value respectively.
```

a dictionary to hold all command structures

In [4]:
```
cmdDict = \
    {'D': [0x00, 0x00, 0x44, 0x00, 0x00],
     'I': [0x31, 0x32, 0x49, 'Value', 0x00],
     'E': [0x00, 0x00, 0x45, 'Address', 0x00],
     'A': [0x00, 0x00, 0x41, 'Address', 0x00],
     'F': ['Address', 0x00, 'Command', 0x00],
         'FC': {'b': [0x62, 'Value'],
                'c': [0x63, 'Value'],
                'd': [0x64, 'Value'],
                'e': [0x65, 'Value'],
                'g': [0x67, 'Value'],
                'h': [0x68, 'Value'],
                'l': [0x6C, 'Value'],
                'o': [0x6F, 'Value'],
                'p': [0x70, 'Value'],
                'r': [0x72, 'Value'],
                't': [0x74, 'Value'],
                'v': [0x76, 'Value'],
                'z': [0x7A, 'Value'],
               },
     'M': ['Address', 0x01, 0x6D, 'Value', 'Value2', 0x00],
     'Q': ['Address', 0x00, 'Query', 0x00, 0x00],
         'QC': {'b': [0xE2],
                'c': [0xE3],
                'd': [0xE4],
                'e': [0xE5],
                'g': [0xE6],
                'h': [0xE7],
                'i': [0xE9],
                'l': [0xE8],
                'o': [0xE9],
                'p': [0xEC],
                'r': [0xED],
                's': [0xF3],
                't': [0xF4],
                'v': [0xF6],
                'z': [0xFA],
               },
    }
```

a function to build a command byte array to send to serial port

In [5]:
```
def buildCommand(Command, subCommand=None, Address=None, Value=None, Value2=None):

    if Command == 'D':
        cmd = cmdDict['D'].copy()
        print (cmd)
        return bytearray(cmd)

    if Command == 'I':
        cmd = cmdDict['I'].copy()
        print (cmd)
        cmd[3] = eval(cmd[3])
        return bytearray(cmd)

    if Command == 'E':
        cmd = cmdDict['E'].copy()
        print (cmd)
        cmd[3] = eval(cmd[3])
        return bytearray(cmd)

    if Command == 'A':
        cmd = cmdDict['A'].copy()
        print (cmd)
        cmd[3] = eval(cmd[3])
        return bytearray(cmd)

    if Command == 'F':
        cmd = cmdDict['F'].copy()
        print (cmd)
        cmd[0] = eval(cmd[0])

        # subCommand: ['b','c','d','e','g','h','l','o','p','r','t','v','z']:
        subcmd = cmdDict['FC'][subCommand].copy()
        i = cmd.index('Command')
        cmd[i:i+1] = subcmd[0], eval(subcmd[1])
        return bytearray(cmd)

    if Command == 'M':
        cmd = cmdDict['M'].copy()
        print (cmd)
        cmd[0] = eval(cmd[0])
        cmd[3] = eval(cmd[3])
        cmd[4] = eval(cmd[4])
```

```
        return bytearray(cmd)

    if Command == 'Q':
        cmd = cmdDict['Q'].copy()
        print (cmd)
        cmd[0] = eval(cmd[0])

        # subCommand: ['b','c','d','e','g','h','i', l','o','p','r','s',t','v','z']:
        subcmd = cmdDict['QC'][subCommand].copy()
        i = cmd.index('Query')
        cmd[i] = subcmd[0]
        return bytearray(cmd)
```

examples of byte array commands

In [6]:
```
s = buildCommand(Command='F', subCommand='v', Address=0x31, Value=15)
s
```

```
['Address', 0, 'Command', 0]
```

Out[6]:
```
bytearray(b'1\x00v\x0f\x00')
```

In [7]:
```
s = buildCommand(Command='Q', subCommand='s', Address=0x31)
s
```

```
['Address', 0, 'Query', 0, 0]
```

Out[7]:
```
bytearray(b'1\x00\xf3\x00\x00')
```

In [8]:
```
s = buildCommand(Command='M', Address=0x31, Value=0x68, Value2=0x69)
s
```

```
['Address', 1, 109, 'Value', 'Value2', 0]
```

Out[8]:
```
bytearray(b'1\x01mhi\x00')
```

practice building and sending pulser commands to another unix terminal window

In [9]:

```
import serial
ser = serial.Serial('/dev/pts/1', 115200, timeout=1)
# for pulser: computer should be configured to 4800 baud with 1 start
# bit, 8 data bits, one stop bit, and no parity

# D Command - enter assign address mode
s = buildCommand(Command='D')
print (s)
ser.write(s)

# Information
# get instrument type
s = buildCommand(Command='I', Value=0x00)
ser.write(s)
r = ser.read(size=24) # read up to 24 bytes (then timeout)

# A Command - assign instrument address
PulserAddress = 0x01
s = buildCommand(Command='A', Address=PulserAddress)

# E Command - exit assign address mode
s = buildCommand(Command='E', Address=PulserAddress)

# Q Command
# get instrument status 's'
s = buildCommand(Command='Q', subCommand='s', Address=PulserAddress)
ser.write(s)
r = ser.read(size=24) # read up to 24 bytes (then timeout)

# we could now use function control and make queries

# we can look at the responses from the pulser
# print ('Address = ',   r[0])
# print ('nbytes = ',    r[1])
# print ('Query = ',     r[2])
# print ('Value = ',     r[3])
# print ('FP Value = ',  r[4])
# print ('Indicator = ', r[5])
# etc
```

```
[0, 0, 68, 0, 0]
bytearray(b'\x00\x00D\x00\x00')
```

```
[49, 50, 73, 'Value', 0]
[0, 0, 65, 'Address', 0]
[0, 0, 69, 'Address', 0]
['Address', 0, 'Query', 0, 0]
```

practice reading bytes from fake serial

In [10]:
```
ser = Serial('/dev/ttyS1', 19200, timeout=1)
r = ser.read(24)        # read up to 24 bytes (then timeout)
print( "r = ", r )
```

```
r =  bytearray(b'\x01\x04I1l\x00')
```

In [11]:
```
print (r[0], r[1], r[2], r[3], r[4], r[5])
```

```
1 4 73 49 108 0
```

In [12]:
```
r[3]
```

Out[12]:
```
49
```

In [13]:
```
hex(r[3])
```

Out[13]:
```
'0x31'
```

In [14]:
```
chr(r[3])
```

Out[14]:
```
'1'
```

now we could create a loop to set up instrument and change settings

based on some logic or external events

In [ ]:
```

```