LineLidar Python class

-

Introduction

Table of Contents

Purpose

This document describes the essential concepts needed to use the LineLidar Python class (henceforth referred to as "the class") to communicate with a Noptel LineLidar LL60 device.

The class implements the protocol described in the LineLidar ICD document and abstracts the low-level communication between the application and the device. However, the reader should be familiar with the way the device communicates as described in the ICD.

This document also assumes the reader has some familiarity with the Python language.

Note: the class is written in Python3. It will not work in Python2

Installing the class

Strictly-speaking, the class does not need to be installed provided the `linelidarclass` directory is in the Python search path – e.g. if the `linelidarclass` directory is present at the same directory level as the application's main module, or if the application adds its location to `sys.path` as in the examples provided in the class' archive.

However, it is recommended to install the class, either as a regular user or as root/administrator, to avoid making the application dependent on its exact location. See `INSTALL.txt` in the class' archive.

Basic classes

LLsrv

Description: Enum-type LineLidar service

Import:
```
from linelidarclass import LLsrv
```
-or-
```
from linelidarclass.linelidar import LLsrv
```

Help:
```
>>> help(LLsrv)
```

Enum values:
LLsrv.DEVICE_INFO
LLsrv.DEVICE_CONFIG
LLsrv.HW_CONFIG
LLsrv.RESULTS
LLsrv.DEBUG

Enum members:
LLsrv.<enum_value>.name
LLsrv.<enum_value>.value #Service ID

Example:

```
>>> print(LLsrv.DEVICE_CONFIG.name)
DEVICE_CONFIG

>>> print(LLsrv.DEVICE_CONFIG.value)
2
```

Enum methods:

- __eq__

  Example:

  ```
  >>> LLsrv.DEVICE_CONFIG == LLsrv.DEBUG
  False
  ```

LLchr

Description: Enum-type LineLidar characteristic

Import:
```
from linelidarclass import LLchr
```
-or-
```
from linelidarclass.linelidar import LLchr
```

Help:
```
>>> help(LLchr)
```

Enum values:
LLchr.SERIAL_NUMBER
LLchr.FW_VERSION
LLchr.NETWORK
LLchr.TIME
LLchr.TEMPERATURE
LLchr.MAC
LLchr.DEFAULT_NETWORK
LLchr.MIN_DISTANCE
LLchr.MAX_DISTANCE
LLchr.AMPLITUDE_THRESHOLD
LLchr.MIN_ANGLE
LLchr.MAX_ANGLE
LLchr.TRIGGER_SOURCE
LLchr.NB_PEAKS
LLchr.REPORT_ZERO_RESULTS
LLchr.TRIGGER_TYPE
LLchr.TRIGGER_DIVISION
LLchr.CALIBRATED_ANGLES
LLchr.RANGE
LLchr.MEASURING_RATE
LLchr.RESET_DEVICE

Enum members:
LLchr.<enum_value>.name
LLchr.<enum_value>.value #Tuple (associated LLsrv, characteristic ID)

Example:

```
>>> print(LLchr.DEFAULT_NETWORK.name)
DEFAULT_NETWORK

>>> print(LLchr.DEFAULT_NETWORK.value)
(<linelidarclass._LLsrv object at 0x7f4f055d9850>, 19)

>>> print(LLchr.DEFAULT_NETWORK.value[0].name)
DEVICE_INFO
```

Enum methods:

- __eq__

  Example:

  ```
  >>> LLchr.DEFAULT_NETWORK == LLchr.TIME
  False
  ```

LLcmd

Description: Enum-type LineLidar command or command response

Note: LLcmd doesn't normally need to be explicitely imported as it is used internally by all of the LineLidar class' methods.

Import:
```
from linelidarclass import LLcmd
```
-or-
```
from linelidarclass.linelidar import LLcmd
```

Help:
```
>>> help(LLcmd)
```

Enum values:

LLcmd.ERROR	#Response
LLcmd.READ	#Command
LLcmd.WRITE	#Command
LLcmd.NOTIFICATION	#Response
LLcmd.SET_NOTIFICATION	#Command
LLcmd.SAVE_SERVICE	#Command
LLcmd.RESTORE_SERVICE	#Command

Enum members:
LLcmd.<enum_value>.name
LLcmd.<enum_value>.value #Command code

Example:

```
>>> print(LLcmd.WRITE.name)
WRITE

>>> print(LLcmd.WRITE.value)
2
```

Enum methods:

- __eq__

  Example:

  ```
  >>> LLcmd.READ == LLcmd.WRITE
  False
  ```

LLsta

Description: Enum-type LineLidar status

Import:
```
from linelidarclass import LLsta
  -or-
from linelidarclass.linelidar import LLsta
```

Help:
```
>>> help(LLsta)
```

Enum values:
LLsta.OK
LLsta.CHAR_DECODING_FAILED
LLsta.CHAR_OUT_OF_RANGE
LLsta.PROCEDURE_IN_PROGRESS
LLsta.PROCEDURE_FAILED
LLsta.NOT_ALLOWED

Enum members:
LLsta.<enum_value>.name
LLsta.<enum_value>.value #Status code

Example:

```
>>> print(LLsta.OK.name)
OK

>>> print(LLsta.OK.value)
0
```

Enum methods:

- __eq__

  Example:

  ```
  >>> LLsta.OK == LLsta.NOT_ALLOWED
  False
  ```

ipaddress

Description: Simple IPv4 address class

Import:
```
from linelidarclass import ipaddress
```
-or-
```
from linelidarclass.linelidar import ipaddress
```

Help:
```
>>> help(ipaddress)
```

Instantiation:
```
ipaddress(str)
```
-or-
```
ipaddress(bytes)
```

Example:

```
>>> addr = ipaddress(“192.168.0.2”)
>>> addr = ipaddress(b"\xc0\xa8\x00\x02")
```

Methods:

- __eq__

  Example:

  ```
  >>> ipaddress(b"\xc0\xa8\x00\x02") == ipaddress(“192.168.0.2”)
  True
  ```

- __repr__

  Example:

  ```
  >>> print(ipaddress(b"\xc0\xa8\x00\x02"))
  192.168.0.2
  ```

macaddress

Description: Simple MAC address class

Import:
```
from linelidarclass import macaddress
```
-or-
```
from linelidarclass.linelidar import macaddress
```

Help:
```
>>> help(macaddress)
```

Instantiation:
```
macaddress(str)
```
-or-
```
macaddress(bytes)
```

Example:

```
>>> mac = macaddress("8c:1f:64:93:10:14")
>>> mac = macaddress(b"\x8c\x1fd\x93\x10\x14")
```

Methods:

- __eq__

  Example:

  ```
  >>> macaddress("8c:1f:64:93:10:14") ==
  macaddress(b"\x8c\x1fd\x93\x10\x14")
  True
  ```

- __repr__

  Example:

  ```
  >>> print(macaddress(b"\x8c\x1fd\x93\x10\x14"))
  8c:1f:64:93:10:14
  ```

LLresponse

Description: LineLidar response

Import:
```
from linelidarclass import LLresponse
```
-or-
```
from linelidarclass.linelidar import LLresponse
```

Note: LLresponse doesn't normally need to be explicitely imported, as it is returned by the relevant LineLidar class' methods.

Help:
```
>>> help(LLresponse)
```

Members:

LLresponse.recv_local_timestamp	#Always present
LLresponse.cmd	#Not present in notifications
LLresponse.msgid	#Always present
LLresponse.status	#Not present in notifications
LLresponse.char	#Only present in notifications and #responses to read commands
LLresponse.value	#Depending on the response
LLresponse.major	#Depending on the response
LLresponse.minor	#Depending on the response
LLresponse.bugfix	#Depending on the response
LLresponse.addr	#Depending on the response
LLresponse.defaultgw	#Depending on the response
LLresponse.netmask	#Depending on the response
LLresponse.port	#Depending on the response
LLresponse.time	#Depending on the response
LLresponse.temperature	#Depending on the response
LLresponse.mac	#Depending on the response
LLresponse.distance	#Depending on the response
LLresponse.angle	#Depending on the response
LLresponse.source	#Depending on the response
LLresponse.type	#Depending on the response
LLresponse.div	#Depending on the response
LLresponse.frequency	#Depending on the response
LLresponse.threshold	#Depending on the response
LLresponse.angles	#Depending on the response
LLresponse.peaks	#Depending on the response
LLresponse.timestamp	#Depending on the response
LLresponse.measurementid	#Depending on the response
LLresponse.nbresults	#Depending on the response
LLresponse.targets	#Depending on the response
LLresponse.reset	#Depending on the response

Methods:

- __repr__

  Example:

```
>>> print(LineLidar("192.168.0.2").read_chr(LLchr.MAC))
cmd: READ, msgid: 3, status: OK, srv: DEVICE_INFO, chr: MAC,
recv_local_timestamp: datetime.datetime(2024, 1, 23, 11, 15,
54, 570080), mac: 8c:1f:64:93:10:11
```

Main class

LineLidar

Description: Main LineLidar class

Import:
```
from linelidarclass.linelidar import LineLidar
```

Help:
```
>>> help(LineLidar)
```

Instantiation:
```
LineLidar()
```
-or-
```
LineLidar(<open method arguments>)
```

Example:

```
>>> ll = LineLidar()
>>> ll = LineLidar("192.168.0.2")
```

Core methods: (Strictly sufficient to fully interact with a LineLidar)

- __enter__
- __exit___

Optional context manager.

Example:

```
>>> with LineLidar("192.168.0.2") as ll:
...
```

- open
- close

Open and close communication with a LineLidar.
open returns a socket object.

Note: if open method arguments are passed to __init__ upon creating the LineLidar object, the communication is opened immediately and open does not need to be called.

Example:

```
>>> ll.open("192.168.0.2")
<socket.socket fd=3, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_DGRAM, proto=0, laddr=('0.0.0.0', 45501)>

>>> ll.close()
```

- read_chr
- write_chr

Read and write a characteristic.
Return a LLresponse object.

Note: all characteristics being unique, the class automatically infers the service IDs to which they belong, so the service ID does not need to be passed

Example:

```
>>> ll.read_chr(LLchr.SERIAL_NUMBER)
cmd: READ, msgid: 3, status: OK, srv: DEVICE_INFO, chr:
SERIAL_NUMBER, recv_local_timestamp: datetime.datetime(2024, 1,
22, 13, 5, 5, 755284), value: 20

>>> ll.read_chr(LLchr.SERIAL_NUMBER).value
20

>>> ll.write_chr(LLchr.MAX_DISTANCE, distance = 12.0)
cmd: WRITE, msgid: 3, status: OK, recv_local_timestamp:
datetime.datetime(2024, 1, 22, 13, 8, 17, 774958)

>>> ll.write_chr(LLchr.NB_PEAKS, peaks = 2).status == LLsta.OK
True
```

- save_srv
- restore_srv

Save and restore a service to/from the non-volatile memory.
Return a LLresponse object.

Example:

```
>>> ll.save_srv(LLsrv.DEVICE_CONFIG)
cmd: SAVE_SERVICE, msgid: 3, status: OK, recv_local_timestamp:
datetime.datetime(2024, 1, 22, 13, 19, 50, 952422)


>>> ll.restore_srv(LLsrv.DEVICE_CONFIG)
cmd: RESTORE_SERVICE, msgid: 3, status: OK,
recv_local_timestamp: datetime.datetime(2024, 1, 22, 13, 20,
12, 392725)

>>> ll.restore_srv(LLsrv.DEVICE_CONFIG).status == LLsta.OK
True
```

- set_notification

Enable or disable notification on a characteristic.
Return a LLresponse object.

Note: all characteristics being unique, the class automatically infers the service IDs to which they belong, so the service ID does not need to be passed

Example:

```
>>> ll.set_notification(LLchr.RANGE, enabled = True)
cmd: SET_NOTIFICATION, msgid: 3, status: OK,
recv_local_timestamp: datetime.datetime(2024, 1, 22, 13, 30,
53, 420367)
```

- get_notification

  Get one notification – either from any characteristic, or from the list of characteristics supplied in the characteristics mask.

  Returns a LLresponse object.

  get_notification returns the oldest notification in the notification stack if the stack isn't empty, otherwise it waits for the device to send a notification.

  Example:

```
>>> ll.get_notification()
cmd: NOTIFICATION, msgid: 892, srv: RESULTS, chr: RANGE,
recv_local_timestamp: datetime.datetime(2024, 1, 22, 13, 39, 8,
823435), timestamp: datetime.timedelta(seconds=893,
microseconds=343190), measurementid: 1, nbresults: 187,
triggercounter: 0, targets: [(2.8647, -30.363, 47280), (2.8586,
-30.048000000000002, 45765), (2.8547000000000002, -29.738,
39765), ...

>>> ll.get_notification([LLchr.TEMPERATURE])
cmd: NOTIFICATION, msgid: 1322, srv: DEVICE_INFO, chr:
TEMPERATURE, recv_local_timestamp: datetime.datetime(2024, 1,
22, 13, 46, 11, 372554), temperature: 34.04295349121094

>>> ll.get_notification([LLchr.TEMPERATURE]).temperature
34.796531677246094
```

- flush_notifications

  Flush the notification stack.

  Notifications are pushed into the stack when they are received while waiting for the response of a command while notification is enabled on one or more characteristics and ranging is enabled.

Convenience methods:

- enable_notification

  Enable notification on a characteristic

  Equivalent of `set_notification(<characteristic>, True)`

- disable_notification

  Disable notification on a characteristic

  Equivalent of `set_notification(<characteristic>, False)`

- disable_all_notifications

  Disable notification on all characteristics

  Equivalent of `set_notification(<characteristic>, False)` applied to all relevant characteristics

- report_zero_result

  Enable or disable the reporting of zero results in range notifications.

  Equivalent of `write_chr(LLchr.REPORT_ZERO_RESULTS, on = <on>)`

- set_sampling_rate

  Set the sampling rate at the desired frequency. Starts sampling if it was stopped.

  Equivalent of `write_chr(LLchr.MEASURING_RATE, frequency = <frequency>)`

- stop_sampling

  Stop sampling by setting the sampling rate to 0 Hz

  Equivalent of `write_chr(LLchr.MEASURING_RATE, frequency = 0)`

- wait_device_quiet

  Wait for the device to stay silence for at least ¼ of the default communication timeout, i.e. 1.5 / 4 = 0.25 s when communicating directly with UDP, 5 / 4 = 1.25 s when using a SSH tunnel.

- set_clean_state

  Put the device in a known state: ranging stopped and all notifications disabled.

  Unless otherwise specified, the open method assumes the device is in an unknown state and calls set_clean_state after opening the communication.

- reset

  Soft-reset the device

  Equivalent of `write_chr(LLchr.RESET_DEVICE)`, followed by default by an attempt to reestablishing communication with the device by doing a `read_chr(LLchr.RESET_DEVICE)`

Routines

discover

Description: Discover LineLidar devices on a network

Returns a list of discovered IPs as strings.

By default, try to discover LineLidar devices on all non-loopback network interface that are up with an IPv4 address. If a network address is specified, limit the discovery to the corresponding network interface.

Import:
```
from linelidarclass.linelidar import discover
```

Help:
```
>>> help(discover)
```

Example:
```
>>> discover()
['192.168.0.2', '192.168.52.220']

>>> discover("192.168.0.0/24")
['192.168.0.2']
```

Basic operations

Opening / closing communication with the device

The device may be opened and closed the following ways:

1. Passing the open method arguments while instantiating the LineLidar object: the returned LineLidar object has opened communication with the LineLidar device. The close method of the object should be called when operations with the device are over.

   Example:

   ```
   ll = LineLidar("192.168.0.2")
   ...
   ll.close()
   ```

2. Instantiating the LineLidar object without arguments, then explicitely calling the open method: this is useful if you want to recover the socket object returned by open to override the class and communicate with the LineLidar device yourself at the lowest level. The close method of the object should be called when operations with the device are over.

   Example:

   ```
   ll = LineLidar()
   sock = ll.open("192.168.0.2")
   ...
   ll.close()
   ```

3. In a context manager: like 1. but without having to explicitely call close when operations with the device are over.

   Example:

   ```
   with LineLidar("192.168.0.2") as ll:
     ...
   ```

4. In a context manager: like 2. but without having to explicitely call close when operations with the device are over.

   Example:

   ```
   with LineLidar() as ll:
     sock = ll.open("192.168.0.2")
     ...
   ```

The open method arguments offer a variety of options to open communication in different ways, such as specifying a different port, additional optional operations to perform on the device after opening or using a SSH server as a jump host to tunnel the communication over SSH. For details, do:

```
>>> help(LineLidar.open)
```

Synchronous operations

Reading/writing characteristics, saving/restoring services and enabling/disabling notification on a characteristic are synchronous operations: the class sends the command and the LineLidar replies to that command immediately.

After opening communication with the LineLidar, the device sits idle and is ready to reply to commands. By default, it will NOT send anything without being prompted.

Example:

1. Reading the device's MAC address:

```
...
mac_address = mac = ll.read_chr(LLchr.MAC).mac
```

2. Setting the scanning range between 3 m and 10 m:

```
...
ll.write_chr(LLchr.MIN_DISTANCE, distance = 3)
ll.write_chr(LLchr.MAX_DISTANCE, distance = 10)
```

3. Enabling notification on the RANGE characteristic, to get ranging results when sampling:

```
...
ll.enable_notification(LLchr.RANGE)
```

Asynchronous operations

Getting notifications is an asynchronous operation: once ranging is enabled, the LineLidar will send notifications at the specified sampling rate WITHOUT PROMPTING. Your code must get and process notifications fast enough to cope with the rate at which the LineLidar sends them.

Notifications are read from the socket when doing any synchronous operation or when calling get_notification explicitely. In the former case, notifications that arrive before the response to the synchronous command are put into the notification stack to be retrieved by the next get_notification call. In the latter case, if the notification stack is empty, get_notification blocks until a new notification is received.

Notifications will NOT be read in the background without performing either a synchronous operation or calling get_notification: if your application needs to do other things while waiting for notifications from the LineLidar, it's up to you to implement multi-processing or multi-threading and run the LineLidar communication code in a separate process or thread.

To enable ranging at a given frequency, write the frequency to the RANGE characteristic (or use the set_sampling_rate method), then start getting the notifications.

To stop ranging and stop the notifications, write a frequency of 0 to the range characteristic (or use the stop_sampling method).

Example:

1. Starting sampling at 10 Hz (10 measurements per second, may generate more than 10 notifications per second! See below):

```
...
ll.set_sampling_rate(10)
```

2. Getting 50 notifications and displaying their associated measurement ID:

```
...
for _ in range(50):
  print(ll.get_notification().measurementid)
```

3. Stopping sampling:

```
...
ll.stop_sampling()
```

Extracting measurements from the RANGE notifications

One RANGE notification contains one target per angle.

However, if the NB_PEAKS is set to more than 1 and more than 1 target is detected at the same angle (for instance, a small object at 5 m and a wall behind it at 7 m), the LineLidar will send several notifications with the same measurement ID to list all the targets it has detected in this round of measurement.

Therefore, if any one of the SPAD cells has detected 2 targets, the LineLidar will send 2 notifications with the same measurement ID. If a SPAD cell has detected 3 targets, the LineLidar will send 3 notifications.

Putting it all together: complete sample program

The following sample program opens communication with a LineLidar, sets the scanning range between 3 m and 10 m, allows up to 3 targets per SPAD cell, starts sampling, gets 20 measurements, displays the number of targets per measurement then stops sampling:

```
# Import the main class and the characteristics enum class
from linelidarclass.full.linelidar import LineLidar, LLchr

# Open communication with LineLidar at IP address 192.168.0.2
with LineLidar("192.168.0.2") as ll:

  # Set the scanning range between 3 m and 10 m
  ll.write_chr(LLchr.MIN_DISTANCE, distance = 3)
  ll.write_chr(LLchr.MAX_DISTANCE, distance = 10)

  # Enable up to 3 targets per SPAD cell
  ll.write_chr(LLchr.NB_PEAKS, peaks = 3)

  # Disable the reporting of zero targets in the RANGE notifications
  ll.report_zero_results(False)

  # Enable notification on RANGE
  ll.enable_notification(LLchr.RANGE)

  # Start sampling at 10 Hz
  ll.set_sampling_rate(10)

  measurement_no = 0
  last_measurementid = None
  nb_targets = 0

  # Get 20 measurements
  while measurement_no < 20:

    # Get one RANGE notification
    notif = ll.get_notification()

    # Has the measurement ID changed?
    if last_measurementid is not None and \
        notif.measurementid != last_measurementid:

      # Display the number of targets for this measurement
      print("Measurement #{}: {} targets".format(measurement_no, nb_targets))

      # Increment the number of measurements received and reset the number of
      # targets for the new measurement
      measurement_no += 1
      nb_targets = 0

    # Tally the number of targets in this notification
```

```
    nb_targets += len(notif.targets)

    last_measurementid = notif.measurementid

# Stop sampling
ll.stop_sampling()
```