OpenXC for Python Documentation

Release 0.12.0

Christopher Peplin

Oct 09, 2019

Contents

Version 0.12.0

Web http://openxcplatform.com

Download http://pypi.python.org/pypi/openxc/

Documentation http://python.openxcplatform.com

Source http://github.com/openxc/openxc-python/

The OpenXC Python library (for Python 2.6 or 2.7) provides an interface to vehicle data from the OpenXC Platform. The primary platform for OpenXC applications is Android, but for prototyping and testing, often it is preferrable to use a low-overhead environment like Python when developing.

In addition to a port of the Android library API, the package also contains a number of command-line tools for connecting to the vehicle interface and manipulating previously recorded vehicle data.

This Python package works with Python 2.6 and 2.7. Unfortunately we had to drop support for Python 3 when we added the protobuf library as a dependency.

For general documentation on the OpenXC platform, visit the main OpenXC site.

CHAPTER 1

Installation

1.1 Install Python and Pip

This library (obviously) requires a Python language runtime - the OpenXC library currently works with Python 2.6 and 2.7, but not Python 3.x.

- **Mac OS X and Linux**

  Mac OS X and most Linux distributions already have a compatible Python installed. Run `python --version` from a terminal to check - you need a 2.7.x version, such as 2.7.8.

- **Windows**

  1. Download and run the Python 2.7.x MSI installer. Make sure to select to option to `Add python.exe to Path`.
  2. Add the Python Scripts directory your PATH: `PATH=%PATH%;c:\Python27\Scripts`. If you aren't sure how to edit your `PATH`, see this guide for all versions of Windows. Log out and back in for the change to take effect.
  3. Install pip, a Python package manager by saving the `get-pip.py` script to a file and running it from a terminal.

- **Cygwin**

  From the `setup.exe` package list, select the `python` and `python-setuptools` packages. Then, inside Cygwin install `pip` using `easy_install`:

  ```
  $ easy_install pip
  ```

1.2 Install the openxc Package

You can install or upgrade the OpenXC library from the Python Package Index (PyPI) with `pip` at the command line:

```
$ [sudo] pip install -U openxc
```

1.3 USB Backend

If you intend to use the library to connect to a vehicle interface via USB, you must also install a native USB backend - `libusb-1.0` is the recommended library.

- **Mac OS X**

  First install Homebrew, then run:

  ```
  $ brew install libusb
  ```

- **Ubuntu**

  `libusb` is available in the main repository:

  ```
  $ sudo apt-get install libusb-1.0-0
  ```

- **Arch Linux**

  Install `libusb` using `pacman`:

  ```
  $ sudo pacman -S libusbx
  ```

- **Windows**

  Download and install the OpenXC VI USB driver. You must install the driver manually through the Device Manager while the VI is plugged in and on - either running the emulator firmware so it never turns off, or plugged into a real car.

- **Cygwin**

  Install the VI USB driver as in a regular Windows installation.

  If you get the error `Skipping USB device: [Errno 88] Operation not supported or unimplemented on this platform` when you run any of the OpenXC Python tools, make sure you **do not** have the `libusb` Cygwin package installed - that is explicitly not compatible.

1.4 Using the development version

You can clone the repository and install the development version like so:

```
$ git clone https://github.com/openxc/openxc-python
$ cd openxc-python
$ pip install -e .
```

Any time you update the clone of the Git repository, all of the Python tools will be updated too.

CHAPTER 2

Command Line Tools

With all tools, the library will attempt to autodetect the payload format being used by the VI. If it's not sending any messages this is not possible, so you may need to provide the current payload format explicitly with the `--format` flag. For example, here's a command to change the passthrough status of bus 1, but with the payload format for the request explicitly set the protocol buffers:

```
$ openxc-control set --bus 1 --passthrough --format protobuf
```

2.1 Vehicle Interface Firmware Configuration Tool

The OpenXC vehicle interface firmware uses a JSON-formatted configuration file to set up CAN messages, signals and buses. The configuration options and many examples are included with the VI firmware docs. The configuration file is used to generate C++ that is compiled with the open source firmware.

The OpenXC Python library contains a command line tool, `openxc-generate-firmware-code`, that can parse VI configuration files and generate a proper C++ implementation to compile the VI firmware.

Once you've created a VI configuration file, run the `openxc-generate-firmware-code` tool to create an implementation of the functions in the VI's `signals.h`. In this example, the configuration is in the file `mycar.json`.

```
$ openxc-generate-firmware-code --message-set mycar.json > signals.cpp
```

2.2 `openxc-control` - write messages to the VI and change its settings

`openxc-control` is a command-line tool that can send control messages to an attached vehicle interface.

2.2.1 Basic use

version

Print the current firmware version and vehicle platform of the attached CAN translator:

```
$ openxc-control version
```

id

Print the unique ID of the VI, if it has one. This is often the MAC address of the Bluetooth module.

```
$ openxc-control id
```

set

Modify the run-time configuration of the VI. Currently, you can change the acceptance filter (AF) bypass status, passthrough CAN message output, and the payload format used from the OpenXC message format.

Enable and disable CAN AF bypass for a bus:

```
$ openxc-control set --bus 1 --af-bypass
$ openxc-control set --bus 1 --no-af-bypass
```

Enable and disable passthrough of CAN messages to the output interface (e.g. USB or Bluetooth):

```
$ openxc-control set --bus 1 --passthrough
$ openxc-control set --bus 1 --no-passthrough
```

Change the payload format to Protocol Buffers, then back to JSON:

```
$ openxc-control set --new-payload-format json
$ openxc-control set --new-payload-format protobuf
```

write

Send a write request to the VI, either for a simple vehicle message write (to be translated by the VI to a CAN message), or a raw CAN message.

To write a simple vehicle message, the `--name` and `--value` parameters are required. The `--event` parameter is optional.

```
$ openxc-control write --name turn_signal_status --value left
```

To write a CAN messages, the `--bus`, `--id` and `--data` parameters are required. `data` should be a hex string.

```
$ openxc-control write --bus 1 --id 0x124 --data 0x0234567812345678
```

A CAN message with an ID greater than can be represented with 11 bits will automatically be sent using the extended frame format. If you want to send a message with a lower ID using the extended frame format, you can use the `--frame-format` flag:

```
$ openxc-control write --bus 1 --id 0x124 --data 0x0234567812345678 --frame-format␣
↪extended
```

Note: The vehicle interface must be running firmware that supports CAN writes, and must allow writing the specific message that you request with `openxc-control`.

2.2.2 Command-line options

An overview of all possible command line options can be found via `--help`.

2.3 `openxc-dashboard` - an ncurses UI to view data received from a VI

`openxc-dashboard` is a command-line tool that displays the current values of all OpenXC messages simultaneously. The dashboard uses `curses` to draw a basic GUI to the terminal.

Only OpenXC messages in the official public set will be displayed. Unofficial messages may be received, but will not appear on the dashboard.

For each message type, the dashboard displays:

- Message name
- Last received value
- A simple graph of the current value and the range seen
- Total number received since the program started
- A rough calculation of the frequency the message is sent in Hz

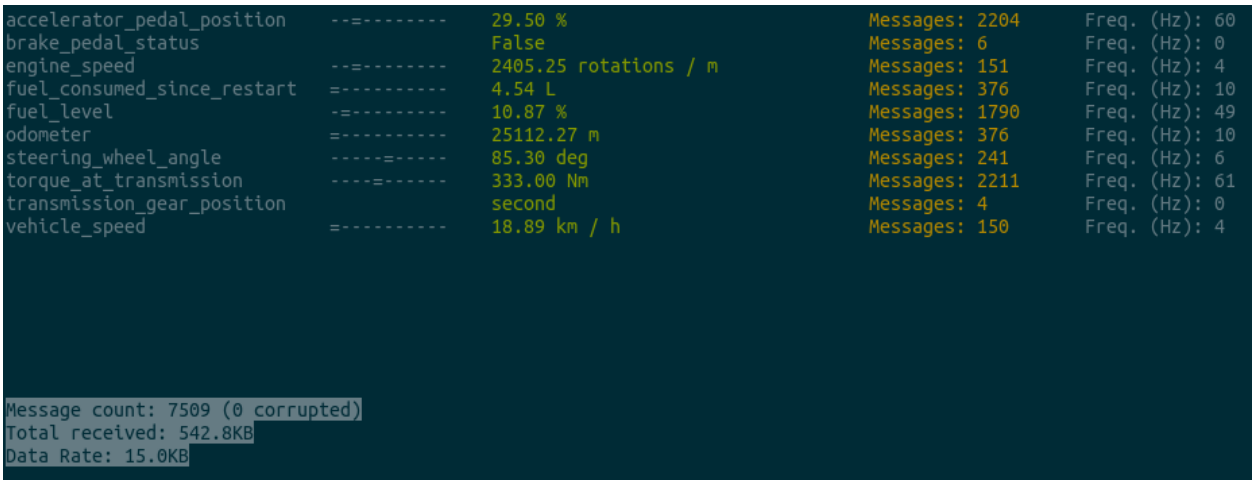
If the terminal window is not wide enough, only a subset of this data will be displayed. The wider you make the window, the more you'll see. The same goes for the list of messages - if the window is not tall enough, the message list will be truncated.

The dashboard also displays some overall summary data:

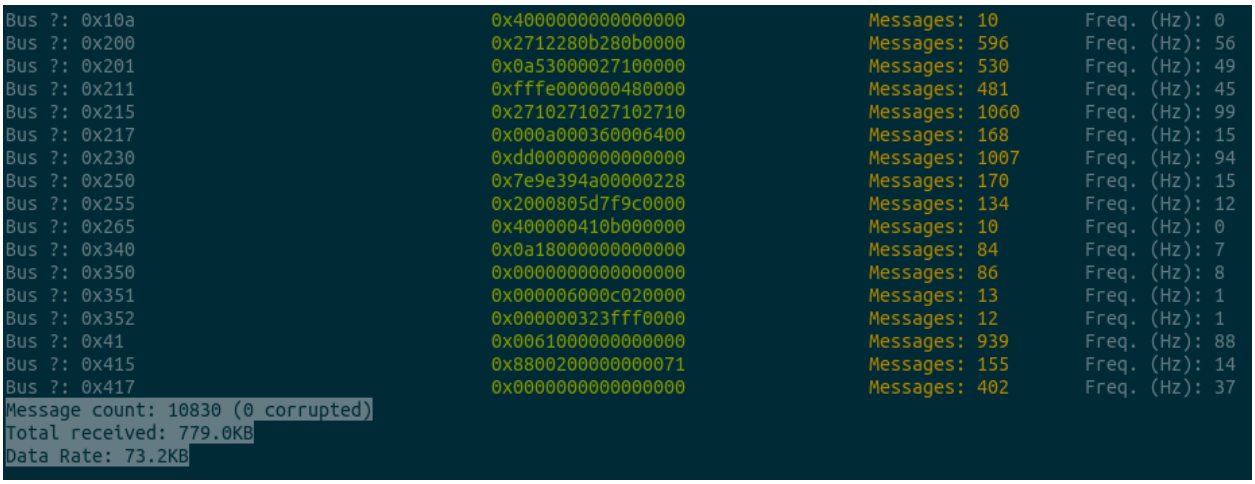
- Total messages received of any type
- Total amount of data received over the source interface
- Average data rate since the program started

If the number of message types is large, you can scroll up and down the list with the arrow keys or Page Up / Page Down keys.

This is a screenshot of the dashboard showing all possible columns of data.



This screenshot shows the dashboard displaying raw CAN messages (the vehicle interface must have CAN passthrough enabled).



2.3.1 Basic use

Open the dashboard:

```
$ openxc-dashboard
```

Use a custom USB device:

```
$ openxc-dashboard --usb-vendor 4424
```

Use a a vehicle interface connected via serial instead of USB:

```
$ openxc-dashboard --serial --serial-device /dev/ttyUSB1
```

The `serial-device` option is only required if the virtual COM port is different than the default `/dev/ttyUSB0`.

Play back a trace file in real-time:

```
$ openxc-dashboard --trace monday-trace.json
```

2.3.2 Command-line options

An overview of all possible command line options can be found via `--help`.

2.4 `openxc-diag` - Send and receive OBD-II diagnostic messages

`openxc-diag` is a command-line tool for adding new recurring or one-time diagnostic message requests through a vehicle interface.

2.4.1 Perform a single diagnostic request

This example will add a new one-time diagnostic request - it will be sent once, and any respones will be printed to the terminal via stdout. The `--message-id` and `--mode` options are required. This particular request sends a functional broadcast request (ID `0x7df`) for the mode 3 service, to store a "freeze frame". See the Unified Diagnostics Service and On-Board Diagnostics standards for more information on valid modes.

The `bus` option is not required; the VI wlil use its default configured CAN bus if one is not specified.

```
$ openxc-diag add --message-id 0x7df --mode 0x3
```

Note: The vehicle interface must be running firmware that supports diagnostic requests.

2.4.2 Add a recurring diagnostic request

This example will register a new recurring diagnostic request with the vehicle interface. It will request the OBD-II engine speed parameter at 1Hz, so if you subseqeuntly run the `openxc-dump` command you will be able to read the responses.

```
$ openxc-diag add --message-id 0x7df --mode 0x1 --pid 0xc --frequency 1
```

2.4.3 Cancel an existing recurring diagnostic request

This example will cancel the recurring diagnostic request we added in the previous example. Deleting requests also uses the combination of bus, ID, mode and PID to identify a request.

```
$ openxc-diag cancel --message-id 0x7df --mode 0x1 --pid 0xc
```

2.4.4 Cancelling a non-recurring diagnostic request

If you're wondering why there are no examples of canceling an existing request that is not recurring, it's because they either complete or timeout withing 1 second, so there's no reason to try and modify them.

2.4.5 Command-line options

A description overview of all possible command line options can be found via `--help`.

2.5 `openxc-dump` - view the unfiltered stream of data from a VI

`openxc-dump` is a command-line tool to view the raw data stream from an attached vehicle interface or trace file. It attempts to read OpenXC messages from the interface specified at the command line (USB, Bluetooth (Linux), serial a trace file) and prints each message received to `stdout`.

2.5.1 Basic use

View everything:

```
$ openxc-dump
```

View only a particular message:

```
$ openxc-dump | grep steering_wheel_angle
```

Use a custom USB device:

```
$ openxc-dump --usb-vendor 4424
```

Use a vehicle interface connected via serial instead of USB:

```
$ openxc-dump --serial --serial-device /dev/ttyUSB1
```

The `serial-device` option is only required if the virtual COM port is different than the default `/dev/ttyUSB0`.

Use a VI with a Bluetooth adapter (this is only supported when connecting from Linux at the moment):

```
$ openxc-dump --bluetooth
```

This will scan and discover for an OpenXC VI, connect and start streaming the data. If you know the MAC address, you can also provide that explicitly with the `--bluetooth-address` flag.

Play back a trace file in real-time:

```
$ openxc-dump --trace monday-trace.json
```

2.5.2 Command-line options

An overview of all possible command line options can be found via `--help`.

2.5.3 Traces

You can record a trace of JSON messages from the CAN reader with `openxc-dump`. Simply redirect the output to a file, and you've got your trace. This can be used directly by the openxc-android library, for example.

```
$ openxc-dump > vehicle-data.trace
```

2.6 `openxc-gps` - convert a vehicle data stream to GPX format

`openxc-gps` is a command-line tool to convert a raw OpenXC data stream that includes GPS information (namely latitude and longitude) into one of a few popular formats for GPS traces. The output file is printed to *stdout*, so the output must be redirected to save it to a file.

The only format currently supported is *.gpx*, which can be imported by Google Earth, the Google Maps API and many other popular tools.

2.6.1 Basic use

Convert a previously recorded OpenXC JSON trace file to GPX:

```
$ openxc-gps --trace trace.json > trace.gpx
```

Convert a real-time stream from a USB vehicle interface to GPX in real-time (using all defaults, and printing to *stdout*):

```
$ openxc-gps
```

2.6.2 Command-line options

An overview of all possible command line options can be found via `--help`.

2.7 `openxc-obd2scanner` - detect OBD-II PIDs supported by a vehicle

`openxc-obd2scanner` is a simple and quick tool to check what OBD-II PIDs a vehicle actually supports. It sequentially scans all valid PIDs and prints the responses to stdout.

2.7.1 Basic use

```
$ openxc-obd2scanner
```

2.7.2 Command-line options

A description overview of all possible command line options can be found via `--help`.

2.8 `openxc-scanner` - scanner for determining support diagnostic requests

`openxc-scanner` is a is a rudimentary diagnostic scanner that can give you a high level view of the what message IDs are used by modules on a vehicle network and to which diagnostics services they (potentially) respond.

When you run `openxc-scanner`, it will send a Tester Present diagnostic request to all possible 11-bit CAN message IDs (or arbitration IDs). For each module that responds, it then sends a blank request for each possible diagnostic

service to the module's arbitration ID. Finally, for each service that responded, it fuzzes the payload field to see if anything interesting can happen.

Make sure you do not run this tool while operating your car. The Tester Present message can put modules into diagnostic modes that aren't safe for driving, or other unexpected behaviors may occur (e.g. your powered driver's seat may reset the position, or the powered trunk may open up).

2.8.1 Basic use

There's not much to it, just run it and view the results. It may take a number of minutes to complete the scan if there are many active modules.

```
$ openxc-scanner
```

2.8.2 Scanning a specific message ID

If you wish to scan only a single message ID, you can skip right to it:

```
$ openxc-scanner --message-id 0x7e0
```

2.8.3 Command-line options

A description overview of all possible command line options can be found via `--help`.

2.9 `openxc-trace-split` - split merged OpenXC trace files into separate trips

`openxc-trace-split` is a command-line tool to re-split a collection of previously recorded OpenXC trace files by different units of time.

Often, trace files are recorded into arbitrarily sized chunks, e.g. a new trace file every hour. The trace files are often most useful if grouped into more logical chunks e.g. one "trip" in the vehicle.

This tool accepts a list of JSON trace files as arguments, reads them into memory and sorts by time, then re-splits the file into new output files based on the requested split unit. The unit is "trips" by default, which looks for gaps of 5 minutes or more in the trace files to demarcate the trips.

The output files are named based on the timestamp of the first record recorded in the segment.

2.9.1 Basic use

Re-combine two trace files and re-split by trip (the default split unit) instead of the original day splits:

```
$ openxc-trace-split monday.json tuesday.json
```

Re-combine two trace files and re-split by hour instead of the original day splits:

```
$ openxc-trace-split --split hour monday.json tuesday.json
```

Re-split an entire directory of JSON files by trip

```
$ openxc-trace-split *.json
```

2.9.2 Command-line options

A quick overview of all possible command line options can be found via `--help`.

`-s, --split` `<unit>`
Change the time unit used to split trace files - choices are `day`, `hour` and `trip`. The default unit is `trip`, which looks for large gaps of time in the trace files where no data was recorded.

CHAPTER 3

Example Code

Read an unfiltered stream of OpenXC messages from a USB vehicle interface:

```
from openxc.interface import UsbVehicleInterface

def receive(message, **kwargs):
    # this callback will receive each message received as a dict
    print(message['name'])

vi = UsbVehicleInterface(callback=receive)
vi.start()
# This will block until the connection dies or you exit the program
vi.join()
```

If you want to connect to a Bluetooth interface (currently only supported in Linux), just replace `UsbVehicleInterface` with `BluetoothVehicleInterface`.

The base `VehicleInterface` classes all implement the `Controller` API, which also supports writing CAN messages, creating diagnostic requests and sending configuration commands to the VI.

For example, to create a diagnostic request and wait for responses:

```
message_id = 42
mode = 1
bus = 1
pid = 3

responses = vi.create_diagnostic_request(message, mode,
        bus=bus, pid=pid, wait_for_first_response=True)
```

To write a low-level CAN message (the VI must be configured to allow this):

```
vi.write(bus=1, id=42, data="0x1234567812345678")
```

To put the CAN acceptance filter in bypass mode for bus 1:

```
vi.set_passthrough(1, true)
```

There are many more commands and options, and most have documented APIs in the code base. You are encouraged you to dig around as the library is fairly small and should be easy to grok. More examples and documentation would be a most welcome contribution!

CHAPTER 4

Vehicle Data API Reference

4.1 Controllers

Contains the abstract interface for sending commands back to a vehicle interface.

class openxc.controllers.base.**CommandResponseReceiver**(*queue*, *request*, *quit_after_first=True*)

A receiver that matches the 'command' field in responses to the original request.

Construct a new ResponseReceiver.

queue - A multithreading queue that this receiver will pull potential responses from. request - The request we are trying to match up with a response.

class openxc.controllers.base.**Controller**

A Controller is a physical vehicle interface that accepts commands to be send back to the vehicle. This class is abstract, and implementations of the interface must define at least the `write_bytes` method.

complex_request(*request*, *wait_for_first_response=True*)

Send a compound command request to the interface over the normal data channel.

request - A dict storing the request to send to the VI. It will be serialized to the currently selected output format.

wait_for_first_response - If true, this function will block waiting for a response from the VI and return it to the caller. Otherwise, it will send the command and return immediately and any response will be lost.

create_diagnostic_request(*message_id*, *mode*, *bus=None*, *pid=None*, *frequency=None*, *payload=None*, *wait_for_ack=True*, *wait_for_first_response=False*, *decoded_type=None*)

Send a new diagnostic message request to the VI

Required:

message_id - The message ID (arbitration ID) for the request. mode - the diagnostic mode (or service).

Optional:

bus - The address of the CAN bus controller to send the request, either 1 or 2 for current VI hardware.

pid - The parameter ID, or PID, for the request (e.g. for a mode 1 request).

frequency - The frequency in hertz to add this as a recurring diagnostic requests. Must be greater than 0, or None if it is a one-time request.

payload - A bytearray to send as the request's optional payload. Only single frame diagnostic requests are supported by the VI firmware in the current version, so the payload has a maximum length of 6.

wait_for_ack - If True, will wait for an ACK of the command message. wait_for_first_response - If True, this function will block waiting for

a diagnostic response to be received for the request. It will return either after timing out or after 1 matching response is received - there may be more responses to functional broadcast requests that arrive after returning.

Returns a tuple of

([list of ACK responses to create request], [list of diagnostic responses received])

delete_diagnostic_request(*message_id*, *mode*, *bus=None*, *pid=None*)

device_id()
Request the unique device ID of the attached VI.

set_acceptance_filter_bypass(*bus*, *bypass*)
Control the status of CAN acceptance filter for a bus.

Returns True if the command was successful.

set_passthrough(*bus*, *enabled*)
Control the status of CAN message passthrough for a bus.

Returns True if the command was successful.

set_payload_format(*payload_format*)
Set the payload format for messages sent to and from the VI.

Returns True if the command was successful.

set_predefined_obd2_requests(*enabled*)
Control if pre-defined OBD2 requests should be sent.

Returns True if the command was successful.

stop()

version()
Request a firmware version identifier from the VI.

write(***kwargs*)
Serialize a raw or translated write request and send it to the VI, following the OpenXC message format.

write_bytes(*data*)
Write the bytes in `data` to the controller interface.

write_raw(*message_id*, *data*, *bus=None*, *frame_format=None*)
Send a raw write request to the VI.

write_translated(*name*, *value*, *event=None*)
Send a translated write request to the VI.

exception openxc.controllers.base.**ControllerError**

class openxc.controllers.base.**DiagnosticResponseReceiver**(*queue*, *request*)
A receiver that matches the bus, ID, mode and PID from a diagnostic request to an incoming response.

class openxc.controllers.base.**ResponseReceiver**(*queue*, *request*, *quit_after_first=True*)
All commands to a vehicle interface are asynchronous. This class is used to wait for the response for a particular request in a thread. Before making a request, a ResponseReceiver is created to wait for the response. All responses received from the VI (which may or may not be in response to this particular command) are passed to the ResponseReceiver, until it either times out waiting or finds a matching response.

The synchronization mechanism is a multiprocessing Queue. The ResponseReceiver blocks waiting on a new response to be added to the queue, and the vehicle interface class puts newly received responses in the queues of ResponseReceivers as they arrive.

Construct a new ResponseReceiver.

queue - A multithreading queue that this receiver will pull potential responses from. request - The request we are trying to match up with a response.

COMMAND_RESPONSE_TIMEOUT_S = 0.5

handle_responses()
Block and wait for responses to this object's original request, or until a timeout (self.COMMAND_RESPONSE_TIMEOUT_S).

This function is handy to use as the target function for a thread.

The responses received (or None if none was received before the timeout) is stored in a list at self.responses.

start()

wait_for_responses()
Block the thread and wait for the response to the given request to arrive from the VI. If no matching response is received in COMMAND_RESPONSE_TIMEOUT_S seconds, returns anyway.

Controller implementation for a virtual serial device.

class openxc.controllers.serial.**SerialControllerMixin**
An implementation of a Controller type that connects to a virtual serial device.

This class acts as a mixin, and expects `self.device` to be an instance of `serial.Serial`.

TODO Bah, this is kind of weird. refactor the relationship between sources/controllers.

WAITIED_FOR_CONNECTION = False

complex_request(*request*, *blocking=True*)
Send a compound command request to the interface over the normal data channel.

request - A dict storing the request to send to the VI. It will be serialized to the currently selected output format.

wait_for_first_response - If true, this function will block waiting for a response from the VI and return it to the caller. Otherwise, it will send the command and return immediately and any response will be lost.

write_bytes(*data*)
Write the bytes in `data` to the controller interface.

Controller implementation for an OpenXC USB device.

class openxc.controllers.usb.**UsbControllerMixin**
An implementation of a Controller type that connects to an OpenXC USB device.

This class acts as a mixin, and expects `self.device` to be an instance of `usb.Device`.

TODO bah, this is kind of weird. refactor the relationship between sources/controllers.

COMPLEX_CONTROL_COMMAND = 131

out_endpoint
Open a reference to the USB device's only OUT endpoint. This method assumes that the USB device configuration has already been set.

write_bytes(*data*)
Write the bytes in `data` to the controller interface.

4.2 Data Formats

JSON formatting utilities.

class openxc.formats.json.**JsonFormatter**

classmethod deserialize(*message*)

classmethod serialize(*data*)

class openxc.formats.json.**JsonStreamer**

SERIALIZED_COMMAND_TERMINATOR = b'\x00'

parse_next_message()

serialize_for_stream(*message*)

4.3 Measurements

Vehicle data measurement types pre-defined in OpenXC.

class openxc.measurements.**AcceleratorPedalPosition**(*value*, ***kwargs*)

name = 'accelerator_pedal_position'

class openxc.measurements.**BooleanMeasurement**(*value*, ***kwargs*)

DATA_TYPE
alias of `builtins.bool`

class openxc.measurements.**BrakePedalStatus**(*value*, ***kwargs*)

name = 'brake_pedal_status'

class openxc.measurements.**ButtonEvent**(*value*, ***kwargs*)

name = 'button_event'

states = ['up', 'down', 'left', 'right', 'ok']

class openxc.measurements.**CanMessage**(*name*, *value*, *event=None*, *override_unit=False*, ***kwargs*)

Construct a new Measurement with the given name and value.

Args: name (str): The Measurement's generic name in OpenXC. value (str, float, or bool): The Measurement's value.

Kwargs: event (str, bool): An optional event for compound Measurements. override_unit (bool): The value will be coerced to the correct units

if it is a plain number.

Raises: UnrecognizedMeasurementError if the value is not the correct units, e.g. if it's a string and we're expecting a numerical value

name = 'can_message'

class openxc.measurements.**DoorStatus**(*value*, ***kwargs*)

name = 'door_status'

states = ['driver', 'rear_left', 'rear_right', 'passenger']

class openxc.measurements.**EngineSpeed**(*value*, ***kwargs*)

name = 'engine_speed'

unit = ComposedUnit([LeafUnit('rotations', False)], [LeafUnit('m', True)], 1)

valid_range = <openxc.utils.Range object>

class openxc.measurements.**EventedMeasurement**(*value*, ***kwargs*)

DATA_TYPE
alias of builtins.str

class openxc.measurements.**FuelConsumed**(*value*, ***kwargs*)

name = 'fuel_consumed_since_restart'

unit = NamedComposedUnit('L', ComposedUnit([LeafUnit('m', True), LeafUnit('m', True),

valid_range = <openxc.utils.Range object>

class openxc.measurements.**FuelLevel**(*value*, ***kwargs*)

name = 'fuel_level'

class openxc.measurements.**HeadlampStatus**(*value*, ***kwargs*)

name = 'headlamp_status'

class openxc.measurements.**HighBeamStatus**(*value*, ***kwargs*)

name = 'high_beam_status'

class openxc.measurements.**IgnitionStatus**(*value*, ***kwargs*)

name = 'ignition_status'

states = ['off', 'accessory', 'run', 'start']

class openxc.measurements.**LateralAcceleration**(*value*, ***kwargs*)

name = 'lateral_acceleration'

unit = ComposedUnit([LeafUnit('m', True)], [LeafUnit('s', True), LeafUnit('s', True)],

valid_range = <openxc.utils.Range object>

class openxc.measurements.**Latitude**(*value*, ***kwargs*)

name = 'latitude'

unit = LeafUnit('deg', False)

valid_range = <openxc.utils.Range object>

class openxc.measurements.**Longitude**(*value*, ***kwargs*)

name = 'longitude'

unit = LeafUnit('deg', False)

valid_range = <openxc.utils.Range object>

class openxc.measurements.**LongitudinalAcceleration**(*value*, ***kwargs*)

name = 'longitudinal_acceleration'

unit = ComposedUnit([LeafUnit('m', True)], [LeafUnit('s', True), LeafUnit('s', True)],

valid_range = <openxc.utils.Range object>

class openxc.measurements.**Measurement**(*name*, *value*, *event=None*, *override_unit=False*, ***kwargs*)

The Measurement is the base type of all values read from an OpenXC vehicle interface. All values encapsulated in a Measurement have an associated scalar unit (e.g. meters, degrees, etc) to avoid crashing a rover into Mars.

Construct a new Measurement with the given name and value.

Args: name (str): The Measurement's generic name in OpenXC. value (str, float, or bool): The Measurement's value.

Kwargs: event (str, bool): An optional event for compound Measurements. override_unit (bool): The value will be coerced to the correct units

if it is a plain number.

Raises: UnrecognizedMeasurementError if the value is not the correct units, e.g. if it's a string and we're expecting a numerical value

DATA_TYPE
alias of `numbers.Number`

classmethod from_dict(*data*)
Create a new Measurement subclass instance using the given dict.

If Measurement.name_from_class was previously called with this data's associated Measurement sub-class in Python, the returned object will be an instance of that sub-class. If the measurement name in `data` is unrecognized, the returned object will be of the generic `Measurement` type.

Args:

data (dict): the data for the new measurement, including at least a name and value.

name = 'generic'

classmethod name_from_class(*measurement_class*)
For a given measurement class, return its generic name.

The given class is expected to have a `name` attribute, otherwise this function will raise an execption. The point of using this method instead of just trying to grab that attribute in the application is to cache measurement name to class mappings for future use.

Returns: the generic OpenXC name for a measurement class.

Raise:

UnrecognizedMeasurementError: if the class does not have a valid generic name

unit = LeafUnit('undef', False)

value

class openxc.measurements.**NamedMeasurement**(*value*, ***kwargs*)
A NamedMeasurement has a class-level `name` variable and thus the `name` argument is not required in its constructor.

class openxc.measurements.**NumericMeasurement**(*value*, ***kwargs*)
A NumericMeasurement must have a numeric value and thus a valid range of acceptable values.

percentage_within_range()

valid_range = None

within_range()

class openxc.measurements.**Odometer**(*value*, ***kwargs*)

name = 'odometer'

unit = NamedComposedUnit('km', ComposedUnit([LeafUnit('m', True)], [], 1000), False)

valid_range = <openxc.utils.Range object>

class openxc.measurements.**ParkingBrakeStatus**(*value*, ***kwargs*)

name = 'parking_brake_status'

class openxc.measurements.**PercentageMeasurement**(*value*, ***kwargs*)

unit = LeafUnit('%', False)

valid_range = <openxc.utils.Range object>

class openxc.measurements.**StatefulMeasurement**(*value*, ***kwargs*)
Must have a class-level `states` member that defines a set of valid string states for this measurement's value.

DATA_TYPE
alias of `builtins.str`

states = None

valid_state()
Determine if the current state is valid, given the class' `state` member.

Returns: True if the value is a valid state.

class openxc.measurements.**SteeringWheelAngle**(*value*, ***kwargs*)

    name = 'steering_wheel_angle'

    unit = LeafUnit('deg', False)

    valid_range = <openxc.utils.Range object>

class openxc.measurements.**TorqueAtTransmission**(*value*, ***kwargs*)

    name = 'torque_at_transmission'

    unit = NamedComposedUnit('Nm', ComposedUnit([NamedComposedUnit('N', ComposedUnit([Leaf

    valid_range = <openxc.utils.Range object>

class openxc.measurements.**TransmissionGearPosition**(*value*, ***kwargs*)

    name = 'transmission_gear_position'

    states = ['first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh', 'eighth',

class openxc.measurements.**TurnSignalStatus**(*value*, ***kwargs*)

    name = 'turn_signal_status'

exception openxc.measurements.**UnrecognizedMeasurementError**

class openxc.measurements.**VehicleSpeed**(*value*, ***kwargs*)

    name = 'vehicle_speed'

    unit = ComposedUnit([NamedComposedUnit('km', ComposedUnit([LeafUnit('m', True)], [], 1

    valid_range = <openxc.utils.Range object>

class openxc.measurements.**WindshieldWiperStatus**(*value*, ***kwargs*)

    name = 'windshield_wiper_status'

openxc.measurements.**all_measurements**()

4.4 Data Sinks

Common operations for all vehicle data sinks.

class openxc.sinks.base.**DataSink**

A base interface for all data sinks. At the minimum, a data sink must have a *receive()* method.

receive(*message*, ***kwargs*)

Handle an incoming vehicle data message.

Args: message (dict) - a new OpenXC vehicle data message

Kwargs:

data_remaining (bool) - if the originating data source can peek ahead in the data stream, this argument will True if there is more data available.

A data sink implementation for the core listener notification service of *openxc.vehicle.Vehicle*.

class openxc.sinks.notifier.**MeasurementNotifierSink**
Notify previously registered callbacks whenever measurements of a certian type have been received.

This data sink is the core of the asynchronous interface of `openxc.vehicle.Vehicle`.

class Notifier(*queue*, *callback*)

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

register(*measurement_class*, *callback*)
Call the `callback` with any new values of `measurement_class` received.

unregister(*measurement_class*, *callback*)
Stop notifying `callback` of new values of `measurement_class`.

If the callback wasn't previously registered, this method will have no effect.

Common functinality for data sinks that work on a queue of incoming messages.

class openxc.sinks.queued.**QueuedSink**
Store every message received and any kwargs from the originating data source as a tuple in a queue.

The queue can be reference in subclasses via the *queue* attribute.

receive(*message*, ***kwargs*)
Add the *message* and *kwargs* to the queue.

Trace file recording operations.

class openxc.sinks.recorder.**FileRecorderSink**
A sink to record trace files based on the messages received from all data sources.

FILENAME_DATE_FORMAT = '%Y-%m-%d-%H'

FILENAME_FORMAT = '%s.json'

class Recorder(*queue*)

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class openxc.sinks.uploader.**UploaderSink**(*url*)
Uploads all incoming vehicle data to a remote web application via HTTP.

TODO document service side format

Args: url (str) - the URL to send an HTTP POST request with vehicle data

HTTP_TIMEOUT = 5000

UPLOAD_BATCH_SIZE = 25

class Uploader(*queue*, *url*)

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

4.5 Data Sources

4.6 Units

Define the scalar units used by vehicle measurements.

openxc.units.**Percentage**

openxc.units.**Meter**

openxc.units.**Kilometer**

openxc.units.**Hour**

openxc.units.**KilometersPerHour**

openxc.units.**RotationsPerMinute**

openxc.units.**Litre**

openxc.units.**Degree**

openxc.units.**NewtonMeter**

openxc.units.**MetersPerSecondSquared**

openxc.units.**Undefined**

4.7 Utils

Data containers and other utilities.

class openxc.utils.**AgingData**
Mixin to associate a class with a time of birth.

age
Return the age of the data in seconds.

class openxc.utils.**Range**(*minimum*, *maximum*)
Encapsulates a ranged defined by a min and max numerical value.

spread
Returns the spread between this Range's min and max.

within_range(*value*)
Returns True if the value is between this Range, inclusive.

openxc.utils.**fatal_error**(*message*)

openxc.utils.**find_file**(*filename*, *search_paths*)

openxc.utils.**load_json_from_search_path**(*filename*, *search_paths*)

openxc.utils.**merge**(*a*, *b*)

Merge two deep dicts non-destructively

Uses a stack to avoid maximum recursion depth exceptions

```
>>> a = {'a': 1, 'b': {1: 1, 2: 2}, 'd': 6}
>>> b = {'c': 3, 'b': {2: 7}, 'd': {'z': [1, 2, 3]}}
>>> c = merge(a, b)
>>> from pprint import pprint; pprint(c)
{'a': 1, 'b': {1: 1, 2: 7}, 'c': 3, 'd': {'z': [1, 2, 3]}}
```

openxc.utils.**quacks_like_dict**(*object*)

Check if object is dict-like

openxc.utils.**quacks_like_list**(*object*)

Check if object is list-like

4.8 Vehicle functions

This module is contains the Vehicle class, which is the main entry point for using the Python library to access vehicle data programatically. Most users will want to interact with an instance of Vehicle, and won't need to deal with other parts of the library directly (besides measurement types).

class openxc.vehicle.**Vehicle**(*interface=None*)

The Vehicle class is the main entry point for the OpenXC Python library. A Vehicle represents a connection to at least one vehicle data source and zero or 1 vehicle controllers, which can accept commands to send back to the vehicle. A Vehicle instance can have more than one data source (e.g. if the computer using this library has a secondary GPS data source).

Most applications will either request synchronous vehicle data measurements using the `get` method or or with a callback function passed to `listen`.

More advanced applications that want access to all raw vehicle data may want to register a `DataSink` with a Vehicle.

Construct a new Vehicle instance, optionally providing an vehicle interface from `openxc.interface` to user for I/O.

add_sink(*sink*)

Add a vehicle data sink to the instance. `sink` should be a sub-class of `DataSink` or at least have a `receive(message, **kwargs)` method.

The sink will be started if it is startable. (i.e. it has a `start()` method).

add_source(*source*)

Add a vehicle data source to the instance.

The Vehicle instance will be set as the callback of the source, and the source will be started if it is startable. (i.e. it has a `start()` method).

get(*measurement_class*)

Return the latest measurement for the given class or None if nothing has been received from the vehicle.

listen(*measurement_class*, *callback*)

Register the callback function to be called whenever a new measurement of the given class is received from the vehicle data sources.

If the callback is already registered for measurements of the given type, this method will have no effect.

unlisten(*measurement_class*, *callback*)
Stop notifying the given callback of new values of the measurement type.

If the callback was not previously registered as a listener, this method will have no effect.

CHAPTER 5

Contributing

Development of `openxc-python` happens at GitHub. Be sure to see our contribution document for details.

5.1 Test Suite

The `openxc-python` repository contains a test suite that can be run with the `tox` tool, which attemps to run the test suite in Python 2.7. If you wish to just run the test suite in your primary Python version, run

```
$ python setup.py test
```

To run it with tox:

```
$ tox
```

5.2 Mailing list

For discussions about the usage, development, and future of OpenXC, please join the OpenXC mailing list.

5.3 Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at http://github.com/openxc/openxc-python/issues/

5.4 Authors and Contributors

A complete list of all authors is stored in the repository - thanks to everyone for the great contributions.

Python Module Index

o

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W