PyVESC Documentation

Release 1.0

Liam Bindle

Dec 13, 2017

Contents

PyVESC is aimed at being a easy to use and robust python implementation of the communication protocol used by the VESC - Open Source ESC project by Benjamin Vedder. Its a great project with a really great community and I'd urge anyone interested in motor controllers to take a look at it.

That being said, if you're here you probably already know about it. This small project was written by Liam Bindle for the University of Saskatchewan Space Design Team as our primary language for non-embedded system is Python. You might wonder why you might need a library to handling packing and parsing VESC messages since Pythons standard struct module is great for almost exaclty this. PyVESC's usefulness comes from the fact that PyVESC is:

- Well tested
- Robust in handling corrupt packets in a buffer
- Messages are easily extensible so that PyVESC can be used as a generic message/codec protocol (ie. at the USST we use PyVESC for sending messages to all of our embedded systems by *Implementing Additional Messages*)
- Implements a number of common-error catching exceptions to speed up your development

CHAPTER 1

Quick Example

This is just a quick example of how PyVESC can be used. First lets see how PyVESC can be used to go from a message (VESCMessage) to a packet (bytes).

```
# make a SetDutyCycle message
my_msg = pyvesc.SetDutyCycle(1e5)
print(my_msg.duty_cycle) # prints value of my_msg.duty_cycle
my_packet = pyvesc.encode(my_msg)
# my_packet (type: bytes) can now be sent over your UART connection
```

Now lets look at how we can parse messages from a buffer (assuming the buffer is being filled by your UART connection)

```
# buff is bytes filled from your UART connection
my_msg, consumed = pyvesc.decode(buff)
buff = buff[consumed:]  # remove consumed bytes from the buffer
if my_msg:
  print(my_msg.duty_cycle)     # prints value of my_msg.duty_cycle
```

CHAPTER 2

Installation

PyVESC is available on the Python Package Index.

```
pip install pyvesc
```

CHAPTER 3

Usage

PyVESC serves two purposes:

1. Allows messages to be created and manipulated easily
2. Performs message encoding (to packet) and robust message decoding (to message object)

CHAPTER 4

Messages

Here is a list of the messages currently supported in PyVESC. Note that not all of VESC's messages are implemented. This is because we have only implemented the messages we use as we don't want to distribute anything that hasn't been tested. If you need additional VESC messages, they are very easy to implement (and we'd greatly appreciate a pull request after). For more infromation see *Implementing Additional Messages*.

It should be noted that all message objects can be created in 3 ways:

1. No constructor arguments
2. Variadic arguments (field values)
3. From decoding the next packet in a buffer

CHAPTER 5

Setter Messages

These are the setter messages which are currently implemented.

class pyvesc.**SetDutyCycle**
Set the duty cycle.

Variables **duty_cycle** – Value of duty cycle to be set (range [-1e5, 1e5]).

class pyvesc.**SetRPM**
Set the RPM.

Variables **rpm** – Value to set the RPM to.

class pyvesc.**SetCurrent**
Set the current (in milliamps) to the motor.

Variables **current** – Value to set the current to (in milliamps).

class pyvesc.**SetCurrentBrake**
Set the current brake (in milliamps).

Variables **current_brake** – Value to set the current brake to (in milliamps).

class pyvesc.**SetPosition**
Set the rotor angle based off of an encoder or sensor

Variables **pos** – Value to set the current position or angle to.

class pyvesc.**SetRotorPositionMode**
Sets the rotor position feedback mode.

It is reccomended to use the defined modes as below:

- DISP_POS_OFF
- DISP_POS_MODE_ENCODER
- DISP_POS_MODE_PID_POS
- DISP_POS_MODE_PID_POS_ERROR

Variables `pos_mode` – Value of the mode

CHAPTER 6

Getter Messages

These are the getters that are currently implemented.

class pyvesc.**GetValues**
Gets internal sensor data

class pyvesc.**GetRotorPosition**
Gets rotor position data

Must be set to DISP_POS_MODE_ENCODER or DISP_POS_MODE_PID_POS (Mode 3 or Mode 4). This is set by SetRotorPositionMode (id=21).

CHAPTER 7

Implementing Additional Messages

Here we'll take a look at how to implement your own messages. You're message class must have the metaclass *pyvesc.VESCMessage*. In addition to this you must define two static attributes:

1. *id* (uint8): The ID for your message
2. *fields* (list of tuples): Declaration of the fields your message has. Each element in the list declares a field. The first element of the field tuple is the name of the field (type: str), and the second element is the field type format characters.

This is probably easiest explained with an example. Here is the declaration of the SetDutyCycle message.

```
class SetDutyCycle(metaclass=pyvesc.VESCMessage):
    id = 5
    fields = [
        ('duty_cycle', 'i')
    ]
```

That's it! Taking a look at the declaration we see:

- The message's ID is 5
- The message has a single field with a name *duty_cycle* and type int (this is what the format characters *'i'* is)

If you are interested in the details of how this works, the *pyvesc.VESCMessage* metaclass has a registry of all its children this registry is a dictionary with key's being the messages ID and values being the messages class. This metaclass also ensures that you define both fields and check that the ID is unique.

CHAPTER 8

Encoding

The following is the function call you should use to get a packet for your message.

pyvesc.**encode**(*msg*)

Encodes a PyVESC message to a packet. This packet is a valid VESC packet and can be sent to a VESC via your serial port.

Parameters **msg** (*PyVESC message*) – Message to be encoded. All fields must be initialized.

Returns The packet.

Return type bytes

Encoding is done by first serializing the message object and then framing it in a VESC packet.

CHAPTER 9

Decoding

The following is the function you should call to decode messages from the buffer.

pyvesc.**decode**(*buffer*)

Decodes the next valid VESC message in a buffer.

Parameters **buffer** (*bytes*) – The buffer to attempt to parse from.

Returns PyVESC message, number of bytes consumed in the buffer. If nothing was parsed returns (None, 0).

Return type *tuple*: (PyVESC message, int)

Decoding is done by checking if the buffer has a full VESC packet which can be parsed. If it does then we begin parsing it, else we return having consumed 0 bytes from the buffer. To parse a message we must parse the packet payload, construct a message object from the payload's ID and then fill the message's fields values. In addition to this we must ensure that if the integrity of the packet has been comprimised (by checking CRC or packet framing), we properly handle recoving our location in the buffer so that subsequent packets are retained.

CHAPTER 10

Contributing

Pull request are always welcome! If you have implemented any additional messages and tested that they are working properly feel free to make a pull request so we can have it availible to everyone.

In addition to this, if you discover any bugs please create an issue so that we can resolve it for everyone.

Find our repository here: https://github.com/LiamBindle/PyVESC.

CHAPTER 11

License

PyVESC is distributed under a Creative Commons ShareALike 4.0 International License.

Index