# python-uds Documentation

Richard Clubb

May 12, 2020

# Contents:

Python-uds is a communication protocol agnostic UDS tool.

It was designed to provide a high-level uds interface which can utilise any communication protocol (e.g. LIN, FlexRay, DoIP). It has a parser tool which can parse an ODX file and produce an easy-to-use interface based on the ODX definition.

```python
import uds
from uds import Uds

if __name__ == "__main__":

    # This creates an Uds object from the Bootloader.odx file
    odxEcu = uds.createUdsConnection("Bootloader.odx", "", inteface="peak")

    # This sends a request for Ecu Serial number and stores the result
    esn = odxEcu.readDataByIdentifier("ECU Serial Number")

    # This will be the printed ASCII string
    print(esn)

    # This creates a Uds object manually
    rawEcu = Uds(reqId=0x7E0, resId=0x7E8, interface="peak")

    # This sends the request for Ecu Serial Number
    esnRaw = rawEcu.send([0x22, 0xF1, 0x8C])

    # This prints the raw payload returned from the ECU
    print(esnRaw)
```

This is an example of how to interface with an ECU using both the raw object and the methods created from the Bootloader.odx file.

Currently it supports diagnostics on CAN using a CAN Transport Protocol defined in ISO-15765 and uses the python-can package to utilise the can device interface.

The final report is available in the repository.

# CHAPTER 1

# Installation

## 1.1 Pip Module Install

Installation of the package is done via pip

```
pip install python-uds
```

Currently python-uds works on Python 3.7 and with Python-can 3.0. Python-can will be installed as a dependency if it is not already installed.

## 1.2 Source Install

To install from source you can checkout the latest github code.

Make sure to add the path to your PYTHONPATH variable to use the source in your code.

# CHAPTER 2

# Configuration

Currently the objects have the following hierarchy.

```
Uds -> Tp -> Device
```

For CAN this would be

```
Uds -> CanTp -> CanInterface
```

Each class has its own config file which includes parameters relevant to that scope, however these can be passed "down the chain" from an upper level object to initialise the lower level classes.

E.g.

- Uds() will use the default values as defined in the local config.ini
- Uds(reqId=0x600, resId=0x650, interface="peak", baudrate=500000) would initialse a connection using Request Id of 0x600, Response Id of 0x650 using the peak interface and a baudrate of 500 kbps

## 2.1 Keyword Arguments

To configure a UDS connection instance the kwargs are passed down to each called object down the chain. The configuration can be passed down either by a config file, or by manually typing the keyword into the function call. The precidence is as follows:

- Default Config
- Config File
- Keyword argument

The following sections detail the different keywords for each class.

## 2.2 Uds

These keywords are used to configure the UDS instance:

- P2_CAN_Server (DEFAULT: 1)
- P2_CAN_Client (DEFAULT: 1)
- transportProtocol (DEFAULT: CAN) Currently CAN is the only supported transport protocol

## 2.3 CanTp

These keywords are used to configure the CAN Transport Protocol Instance (ISO 14229):

- addressingType (DEFAULT: NORMAL)
- reqId (DEFAULT: 0x600) This is just a default ID used by the author
- resId (DEFAULT: 0x650) This is just a default ID used by the author
- N_SA (DEFAULT: 0xFF) This is currently NOT SUPPORTED
- N_TA (DEFAULT: 0xFF) This is currently NOT SUPPORTED
- N_AE (DEFAULT: 0xFF) This is currently NOT SUPPORTED
- Mtype (DEFAULT: DIAGNOSTICS)

## 2.4 LinTp

These keywords are used to configure the CAN Transport Protocol Instance:

- nodeAddress (DEFAULT: 1)
- STMin (DEFAULT: 0.001)

## 2.5 Can Interface

These keywords are used to configure the CAN interface:

- interface (DEFAULT: virtual)
- baudrate (DEFAULT: 500000)

## 2.6 Peak

These keywords are specific to PEAK devices and bus configuration:

- device (DEFAULT: PCAN_USBBUS1)

## 2.7 Vector

These keywords are specific to Vector devices and bus configuration:

- appName (DEFAULT: testApp) This is done so that a vector software licence is not required, but this does require some setup in the vector hardware interface
- channel (DEFAULT: 0) The channel configured for communications

## 2.8 Virtual

These keywords are specific to the python-can virtual loopback interface:

- interfaceName (DEFAULT: virtualInterface) This needs to be the same attempting to interface using the loop-back interface with Python-CAN.

# CHAPTER 3

# Interface

The Uds object allows for raw sending of Uds Payloads with no checking on the validity of the returned value.

The createUdsConnection method creates a Uds object but also creates a series of methods providing access to the services defined in the associated ODX file.

## 3.1 Uds Raw Communication

Using the raw send command is very simple:

```
PCM = Uds(transportProtocol="can", reqId=0x7E0, resId=0x7E8)
a = PCM.send([0x22, 0xF1, 0x8C])
```

This will set up a connection to PCM (Typically Powertrain Control Module, also called an ECM in some companies) with the given parameters, and request the ECU Serial Number (As defined in the ISO-14229 standard, 0x22 is read data by identifier service, and the did 0xF18C is the ECU Serial Number DID)

For a correct response, "a" will be of the format:

```
[0x62, 0xf1, 0x8C, 0xXX, 0xXX ..... ]
```

depending on how long the serial number is (this is not defined in the standard)

For a negative response "a" will be of the format:

```
[0x7F, 0x22, 0xXX]
```

Where 0xXX will be the response code.

## 3.2 Using the createUdsConnection method

UDS is a simple enough protocol, but it relies on using a lot of magic numbers and these can change from manufacturer to manufacturer. There are some standardised DIDs, but most are specific to both the manufacturer and the model, and even the model year.

the createUdsConnection method provides the ability to parse an ODX file which defines the diagnostic interface into a more human readable format.

```
bootloader = createUdsConnection("Bootloader.odx", "", transportProtocol="can", reqId=0x600, resId=0x650)
a = bootloader.readDataByIdentifier("ECU Serial Number")
```

For a positive response, a will be of the format “0000000000000000” which is an encoded form of the Serial number rather than a raw array of bytes. This is because the ODX definition for ECU serial number is coded as an ASCII string of 16 bytes.

It is still possible to interface with the bootloader instance using the .send method to send raw packets, but it loses the benefits of encoding the request and response

# CHAPTER 4

# Services

This section contains a list of all the currently supported services and their interfaces.

An example ODX file is available which supports at least one instance of each of these services. This has been tested against an Embed E400 using their embedded UDS stack.

## 4.1 Tester Present (0x3E Service)

This service is used to send a heartbeat message to the ECU to keep it in a particular mode of operation. Some ECUs have timeouts on the lengh of time a session or security level is valid for after a request.

## 4.2 Diagnostic Session Control (0x10 Service)

This service is used to switch between the defined sessions. Usually most ECUs support Default, Programming, Extended and Service.

## 4.3 ECU Reset (0x11 Service)

This service is used to perform software resets. Usually most ECUs support Soft, Hard, and Reset to Bootloader.

## 4.4 Security Access (0x27 Service)

This service is used to pass security seed / key messages between the tester and ECU.

## 4.5 Read Data By Identifier (0x22 Service)

This service is used to read data from the ECU for detailed diagnostics. E.g. Engine Speed, actuator position, coolant temperature. It varies significantly from manufacturer to manufacturer and also from the supplier of the ECU.

## 4.6 Write Data By Identifier (0x2E Service)

This service is used to configure ECU parameters.

## 4.7 I/O Control (0x2F Service)

This service is used to perform temporary overrides for values in the ECU.

## 4.8 Routine Control (0x31 Service)

This service is used to perform set operations or sequences.

## 4.9 Request Download ()

This service is used to set up a transfer of data to the ECU, usually for re-flashing or calibration sets.

## 4.10 Request Upload()

NOTE: Currently this is not tested

This service is used to transfer data from the ECU, sometimes to download calibration sets or ROM images.

## 4.11 Transfer Data()

This service is used to set up a transfer data to the ECU, usually for re-flashing or calibration sets.

## 4.12 Transfer Exit()

This service is used to signify the end of a transfer data block.

## 4.13 Clear DTC ()

This service is used to clear DTC codes from the ECU memory.

## 4.14 Read DTC ()

This service is used to Read DTC information from the ECU.

To-do:

- Decode the DTC information
- snapshots

CHAPTER 5

# Examples

## 5.1 Example 1 - Simple Peak

This example sets up the connection using CAN with the Peak-USB Interface. This is using an E400 with the standard Embed bootloader which supports ISO-14229 UDS. The serial number of the ECU is an ASCII encoded string, in this case “0000000000000001”.

:: from uds import Uds

E400 = Uds(resId=0x600, reqId=0x650, transportProtocol=”CAN”, interface=”peak”, device=”PCAN_USBBUS1”) try:

response = E400.send([0x22, 0xF1, 0x8C]) # gets the entire response from the ECU

**except:** print(“Send did not complete”)

**if(serialNumberArary[0] = 0x7F):** print(“Negative response”)

**else:** serialNumberArray = response[3:] # cuts the response down to just the serial number serialNumberString = “” # initialises the string for i in serialNumberArray: serialNumberString += chr(i) # Iterates over each element and converts the array element into an ASCII string print(serialNumberString) # prints the ASCII string

As the send function can produce exceptions this needs to be checked before continuing. After this it needs to check for a negative response and then can begin decoding the response.

## 5.2 Example 2 - Simple Vector

This example sets up the connection using CAN with the Vector Interface. This is similar to example 1 so it only includes the initialisation.

It assumes that the Vector hardware has been set up with the application name “pythonUds”, this bypasses the licencing for a particular software suite (CANalyser, CANoe. etc)

```
E400 = Uds(resId=0x600, reqId=0x650, transportProtocol="can", interface="vector",␣
↪appName="pythonUds", channel=0)
```

Once this is initialised then the communication with the ECU is the same as Example 1

## 5.3 Example 3 - Simple LIN using Peak-USB Pro

This example sets up a connection over LIN

```
LightModule = Uds(nodeAddress=0x10, transportProtocol="LIN")
```

Once this is initialised then the communication with the ECU is the same as Example 1

## 5.4 Example 4 - Using the ODX Parser

CAN using Peak Interface with Bootloader Example ODX file.

```
bootloader = createUdsConnection("Bootloader.odx", "", reqId=0x600, resId=0x650,␣
↪interface="peak")
serialNumber = bootloader.readDataByIdentifier("ECU Serial Number")
print(serialNumber)
bootloader.writeDataByIdentifier("Engine Speed Cutoff", 5000)
```

This example uses the readDataByIdentifier and writeDataByIdentifier to get the Serial Number and set the Engine Speed Cutoff parameter used by the model. The string used to identify the instance of the service is defined in the ODX file as a human readable value to ease interfacing with the module.

The returned values are encoded into their physical datatype defined in the ODX file rather than the user having to know the encoding format.

## 5.5 Programming Sequence 1

This sequence is part of the standard programming sequence for an ECU over CAN. It includes a dummy seed-key exchange.

The file E400NewProgrammingSequence defines the programming sequence based on the Bootloader.odx file

CHAPTER 6

# Developer's Overview

**The code is separated into three main areas:**

- Uds Communications - This is the main communications interface for the system
- Uds Config Tool - This is the ODX parser and service configuration system
- Uds Configuration - This is to do with the config set for each of the components and passing configurations around the code

## 6.1 Uds Communications

This sub-module contains all the code related to the communications interface, it includes the Transport Protocol code for CanTp and LinTp

## 6.2 Uds Config Tool

This contains the ODX parser code to create the methods which attach to the UDS instance.

## 6.3 Uds Configuration

Primarily this is a utility for the rest of the code to unify how the system passes configuration items around.

The intention is also to extend this to provide logging functionality,