1 CAN Communication with SAE J1939

1.1 Objectives

- Learn how to interface with Linux SocketCAN and can-utils
- Be able to look up a signal definition in the J1939 Digital Annex (spreadsheet)
- Use grep to search for specific strings from a candump
- Have a reliable CAN datalogger for use in future projects
- Plot data using matplotlib in Python.

1.1.1 Online Learning Resources

https://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction/

1.1.2 Suggested Hardware Materials

This exercise can be run with any device with CAN hardware. An example of a commercial product is the DG Technologies' Beacon device. An example of a hand-built project is the BeagleBone Black with a TruckCape.

1.1.3 Software Resources

You will need access to a database that containse the meaning for the CAN messages. The SAE J1939 digital annex is a spreadsheet version of the J1939-71 document, which contained application layer meaning for the parsed data. The SAE J1939-21 document contains information needed to decode and parse the J1939 Transport Protocol (TP) messages.

The exercises will use Python with access to information needed to decode a J1939 message. This information could be specific knowledge on the just a few messages, or the entire database. The source of the decoding information is usually the digital annex of SAE J1939.

1.2 SAE J1939 CAN Frame Format

J1939 message use the CAN 2.0 specification that enables 29-bit arbitration identifiers. These identifiers are used to explain the meaning of the data in the CAN frame. In SAE J1939-21, the J1939 message ID structure is defined and summarized in Table 1.1

on the following page. Messages in J1939 are categorized into four different types: 1) Broadcast, 2) Request, 3) Command, 4) Acknowledgement. By far, the most common message is a broadcast message. The network behavior is setup to work in two different ways: 1) point-to-point messages and 2) broadcast messages. Even though the copper wires and voltage signaling are seen by all the ECUs on the network, a point-to-point message in J1939 is intended for messages to be used between just 2 nodes, or a 1-to-1 messaging system. Broadcast messages are 1-to-many in nature with no specific destination in mind. An example of a broadcast message is the Cruise Control/Vehicle Speed message that reports the wheel-based vehicle speed. Usually the engine controller sends this message out to the entire network. An instrument cluster on the network would consume this message an display the speed on the speedometer. A telematics device could report the speed back to the main office, which means the same message was used by multiple modules. An example for a point-to-point message, on the other hand, would be a torque control command message from the brake controller to the engine controller instructing the engine to reduce torqe output when the brakes are trying to stop the vehicle. Even though every node sees the message, only the engine controller can act on it. Using functionality as a criteria for broadcast vs

The J1939 message formats are defined as protocol data units (PDUs). There are two formats that correspond to the point-to-point messaging or broadcast message. PDU1 message formats are used for point-to-point messaging and must contain a source address and a destination address. These two data elements are shown as the Destination Address (DA) and Source Address (SA) in the PDU1 format of Table 1.1 on the next page. The PDU2 messages are broadcast to the network without regard to the recipient. This means their destination address is always the global address, which is defined as 0xFF or 255 in decimal. Since this is always the case for a broadcast message, there is no need to use the destination address field. This enables the PDU2 format to include the PDU Specific (PS) field and use 18 bits to define the parameter group number (PGN). The PGN for PDU1 still uses 18 bits, but only the upper 10 bits are used to denote values as the lower 8 bits are always set to zero.

Exercise 1.1. [Evaluate] Show that the number of PGNs available in J1939 based on the two different PDUs is 17,344. (Hint: if the Data Page and Extended Data Page bits are omitted, then there are only 4336 PGNs.)

Table 1.1 on the following page includes an example of a PDU 1 message and an example of a PDU 2 message. The table contains the bit positions across the top row that are indexed from zero. The presentation of the bits in the table from left to right is the order in which they are recieved on the network. In other words, the bit in postion 28 appears on the nework first. This means the first three bits recieved by an ECU are the priority bits. These are designated by the network designers to create priorities for arbitration should any two nodes try to transmit at the same time. The lowest numerical value will have the highest priority. Since there are 3 bits available, there are a total of $2^3 = 8$ priority levels, usually noted as 0-7. To determine the priority using a computer program, we need to extract the portion of the message ID corresponding to the priority

Bit	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Byte	Priority			EDP	DP	PDU Format								PDU Specific								Source Address							
PDU1	Priority			PGN (10 Most Signficant Bits)										Destination Address								Source Address							
Binary	1	1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	1
Hex	1			C		E			C			0				3			0			B							
PDU2	Priority			Parameter Group Number (18 bits)																		Source Address (8 bits)							
Binary	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	1
Hex	1			8		F			E			F				1			3			1							

Table 1.1: CAN Message Identifier for J1939. The Protocol Data Unit (PDU) is the defintion of the CAN frame.

Bit	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Byte	Priority			EDP	DP	PDU Format								PDU Specific								Source Address							
Message Binary	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	1
Message Hex	1			8		F				E				F				1				3				1			
Mask Binary	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mask Hex	1			C		0				0				0				0				0				0			
AND Result	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AND Result	1			8		0				0				0				0				0				0			

Table 1.2: An example of using the AND boolean operation to mask the priortity bits of a J1939 message.

and then perform a right shift until the priority fiels is represented in bit positions 0, 1, and 2. This procedure is called *masking* and *bit-shifting*.

1.2.1 Masking

The process of masking is the process of applying a boolean AND operation to select only the bits of interest. A logic AND operation says that when both logic values are true then the result is true. If any arguments are false, then the result of an AND is false. The application is a programmer can block any bits using a 0 and accept any bits using a 1. A logic AND will keep the values of the passing bits. If we apply this logic to the examples in Table 1.1, we could isolate the bits responsible for the different fields in the J1939 message ID. The example in Table 1.2 shows the process of performing the AND operation on an example message. The mask is 0x1C000000 and the message is 0x18FEF131. The result of the AND operation is 0x18000000.

1.2.2 Bit Shifting

Once the bit fields are masked, the value needs to be determined. However, the position of the masked bits are retained. If we converted the masked result shown in Table 1.2

directly to a value, would get a priority of 402,653,184, which is clearly not an allowable value for the priority. For the computer to calculate the value, the binary needs to be shifted to the right 26 positions. This means the binary of 0b0001 1000 0000 0000 0000 0000 0000 0000 becomes 0b0110, which is the integer of 6. This makes for a valid priority. After working J1939 messages, you will come to recognize that any message that starts with a 0x18 will have a priority of 6.

Implementing a mask followed by a bit shift in a Python computer program makes use of so-called bit-wise operators (AND, OR, XOR) and bit shifts as shown in the following code snippet.

```
#!/usr/bin/env python3
id = 0x18FEF131
PRIORITY_MASK = 0x1C000000
priority = (id & PRIORITY_MASK) >> 26
print(priority)
```

Implementing the same process in a complete Arduino sketch would be the following:

```
// Arduino (Teensy) example sketch to mask and bit-shift
#define PRIORITY_MASK 0x1C000000

void setup(){
  // This is empty for this example
}

void loop(){
  uint32_t id = 0x18FEF131; // Normally this would be from CAN
  // Mask and bit shift
  uint8_t priority = (id & PRIORITY_MASK) >> 26;
  Serial.println(priority);
  delay(1000); // Wait 1 second
}
```

In the Arduino sketch, the parentheses matter because the bit shift operator takes precedence over the AND operator. This means

```
uint8_t priority = id & (PRIORITY_MASK >> 26);
```

and

```
uint8_t priority = id & PRIORITY_MASK >> 26;
```

would both evaluate to the same thing. In this case the output is 1 because the mask is bit shifted before the AND operation takes place. Always be sure to include the parentheses around the masking operation first.

1.2.3 Parsing a J1939 Identifier

Given the example in Table 1.2 on the preceding page, we needto determine the following:

- Message Priority
- Parameter Group Number
- Destination Address
- Source Address

To find these, we need to pull out the necessary elements using bitshifting and masking while determining if the message is a PDU1 or PDU2.

1.2.3.1 Using Arduino or C++

A complete Arduino Sketch to perform this operation is listed in Listing 1.1 on the next page. This listing starts as many programs do with defined constants. The #define tag simply replaces everywhere the name on the left is found with the value on the right. This way, there are no uses of the so-called magic numbers in your code. Every constant is defined and has a name. This makes code much more readable. Notice the #defines do not need a semicolon, whereas the other lines in C/C++ do need them.

The serial monitor output for this sketch is repeating every second.

```
Priority, PGN, DA, SA
6, 65265, 255, 49
7, 60416, 3, 11
```

This example computed the results from parsing the CAN identifier according to J1939. We went from one 29-bit word to four data parameters.

1.2.3.2 Parsing a J1939 message in Python

Often a data file with CAN is available to parse that came from a J1939 enabled network. Let's build a function that can parse the CAN ID based on J1939. The output will be a Python dictionary, which is useful to be used in many other contexts. The Python listing is shown in Listing 1.2 on page 7.

The program starts with a comment statement known in the unix world as a "she-bang." This symbol enables a shell to open the correct interpreter to run the file. In this case, we want to use the python3 interpreter. This line has no effect when running on Windows. The next lines setup defined constants. Python is an interpreted language, so it doesn't have compiler primitves like #define. Instead, each contant is assigned to a variable name at the top level (i.e. a global constant) and it can be used in all the functions. The first function is called main(). This function is called by the last line. Notice the second to last line tests to see if the name assigned to the module is called ___main___. This only happens when the script is run as the top level script (i.e. it is called directly). We can use this feature for testing testing the script, but don't have to worry about main() executing if we import the module to use the intended function, which is parseJ1939id().

The printed results from the Python script in Listing 1.2 on page 7 is as follows:

Listing 1.1: Arduino C++ code to demonstrate parsing two types of J1939 Protocol Data Units.

```
// Arduino (Teensy) Example Sketch to Parse J1939 IDs
#define PRIORITY_MASK          0x1C000000
#define PDU_FORMAT_MASK        0x00FF0000
#define PDU_SPECIFIC_MASK      0x0000FF00
#define SOURCE_ADDRESS_MASK    0x000000FF
#define DATA_PAGE_MASK       0x01000000
#define EXT_DATA_PAGE_MASK     0x02000000
#define PDU1_PGN_MASK          0x03FF0000
#define PDU2_PGN_MASK          0x03FFFF00
#define PRIORITY_BIT_SHIFT     26
#define PDU_FORMAT_BIT_SHIFT 16
#define PDU2_THRESHOLD         240
#define PGN_BIT_SHIFT          8
#define DEST_BIT_SHIFT         8
#define GLOBAL_SOURCE_ADDR     0xFF

//Create a function to parse J1939 CAN IDs
void parse_j1939_id(uint32_t id,
                    uint32_t &pgn,
                    uint8_t &da,
                    uint8_t &sa,
                    uint8_t &priority){
    sa = id & SOURCE_ADDRESS_MASK;
    priority = (id & PRIORITY_MASK) >> PRIORITY_BIT_SHIFT;
    uint8_t pf = (id & PDU_FORMAT_MASK) >> PDU_FORMAT_BIT_SHIFT;
    if (pf < PDU2_THRESHOLD) { // PDU 1 format uses values lower than 240
        da = (id & PDU_SPECIFIC_MASK) >> DEST_BIT_SHIFT;
        pgn = (id & PDU2_PGN_MASK) >> PGN_BIT_SHIFT;
    }
    else { // PDU 2 format
        da = GLOBAL_SOURCE_ADDR;
        pgn = (id & PDU1_PGN_MASK) >> PGN_BIT_SHIFT;
    }
}

void setup(){
}

void loop(){
    uint32_t pgn; // Parameter Group Number
    uint8_t da;  // Destination Address
    uint8_t sa;  // Source Address
    uint8_t priority;

    //Normally this would come from CAN
    uint32_t user_input_id = 0x18FEF131;
    parse_j1939_id(user_input_id, pgn, da, sa, priority);
    Serial.printf("%d, %d, %d, %d\n",priority,pgn,da,sa);

    user_input_id = 0x1CEC030B;
    parse_j1939_id(user_input_id, pgn, da, sa, priority);
    Serial.printf("%d, %d, %d, %d\n",priority,pgn,da,sa);
    delay(1000);
}
```

Listing 1.2: Python program to demonstrate parsing a 29-bit CAN ID based on J1939.

```
#!/usr/bin/env python3
PRIORITY_MASK          = 0x1C000000
PDU_FORMAT_MASK        = 0x00FF0000
PDU_SPECIFIC_MASK      = 0x0000FF00
SOURCE_ADDRESS_MASK    = 0x000000FF
DATA_PAGE_MASK         = 0x01000000
EXT_DATA_PAGE_MASK     = 0x02000000
PDU1_PGN_MASK          = 0x03FF0000
PDU2_PGN_MASK          = 0x03FFFF00

PRIORITY_OFFSET        = 26
PDU_FORMAT_OFFSET      = 16
DA_OFFSET              = 8
PGN_OFFSET             = 8
PDU2_THRESHOLD         = 240
GLOBAL_SOURCE_ADDR     = 0xFF

def main():
    print(parseJ1939id(0x18FEF131))
    print(parseJ1939id(0x1CEC030B))

def parseJ1939id(id):
    sa = id & SOURCE_ADDRESS_MASK
    priority = (id & PRIORITY_MASK) >> PRIORITY_OFFSET
    pf = (id & PDU_FORMAT_MASK) >> PDU_FORMAT_OFFSET
    if (pf < PDU2_THRESHOLD):  # See SAE J1939-21
        # PDU 1 format uses values lower than 240
        da = (id & PDU_SPECIFIC_MASK) >> DA_OFFSET
        pgn = (id & PDU1_PGN_MASK) >> PGN_OFFSET
    else:           # PDU 2 format
        da = GLOBAL_SOURCE_ADDR
        pgn = (id & PDU2_PGN_MASK) >> PGN_OFFSET
    return {'sourceAddress': sa,
            'priority': priority,
            'destinationAddress': da,
            'PGN': pgn}

if __name__ == '__main__':
    main()
```

```
{'source_address': 49, 'priority': 6, 'destination_address': 255, 'PGN': 65024}
{'source_address': 11, 'priority': 7, 'destination_address': 3, 'PGN': 60419}
```

The next step is to figure out what how to derive meaning for those four numbers. Specifically, we will label the addresses and PGNs with English names.

1.2.4 Source and Destination Addresses

The source addresses for J1939 are standardized by SAE. The Javascript Object Notation (JSON) in Listing 1.3 on the following page and Listing 1.4 on page 10 creates a dictionary lookup map the numerical values of the source or destination address to a description in English. The keys of the dictionary are the decimal values of the source address. These are the same for the destination address as well.

1.2.5 Using a Parameter Group Number (PGN)

Once the PGN is known based on the CAN ID, we can look up what the message data contains based on Suspect Parameter Numbers (SPNs). The SAE J1939-71 document contains many maps from PGN to SPN. Let's see how this works with an example. A log of CAN data was collected from a truck that went through a startup sequence, then the driver depressed the accelerator pedal and sped up the engine. Finally the engine was turned off. The data was collected using an embedded Linux device called the BeagleBone Black with a Truck Cape. The data is in a raw text form with the following format:

```
(unix timestamp) channel CAN_ID [N] B0 B1 ... BN
```

In this entry the unix timestamp is the number of seconds from the unix epoch, which is midnight on January 1, 1970 UTC. The channel is based on the device settings in Linux. The CAN ID is in hexadecimal is 8 characters for 29-bit ids. The data length code (DLC) is denoted as N with N bytes of data following. There can be at most 8 bytes of data per CAN frame. An example from the text file is as follows:

```
(011.802193) can1 0CF00331 [8] FF FF FF FF FF FF FF FF
(011.822872) can1 18EEFF03 [8] 64 00 40 00 00 03 03 10
(011.840437) can1 18EAFF00 [3] EB FE 00
(011.842907) can1 18FE4A03 [8] 03 5F 4F FF FF F3 FF FF
(011.849202) can1 0CF00331 [8] FF FF FF FF FF FF FF FF
(011.852511) can1 18EEFF00 [8] F4 B8 4E 01 00 00 00 00
(011.853073) can1 18EEFF29 [8] F4 B8 4E 01 00 0C 00 00
(011.861682) can1 18F0010B [8] CF FF F0 FF FF DC FF FF
(011.871828) can1 18FEBF0B [8] 00 00 7D 7D 7D 7D FF FF
(011.877482) can1 18FEDF00 [8] 7D A0 28 7D 7D FF FF FC
(011.879632) can1 1CECFF31 [8] 20 17 00 04 FF EB FE 00
(011.888282) can1 0CF00400 [8] 0E 7D 7D 00 00 FE 00 7D
```

Listing 1.3: J1939 Source and Destination Addresses

```
{
    "0": "Engine #1",
    "1": "Engine #2",
    "2": "Turbocharger",
    "3": "Transmission #1",
    "4": "Transmission #2",
    "5": "Shift Console - Primary",
    "6": "Shift Console - Secondary",
    "7": "Power TakeOff - (Main or Rear)",
    "8": "Axle - Steering",
    "9": "Axle - Drive #1",
    "10": "Axle - Drive #2",
    "11": "Brakes - System Controller",
    "12": "Brakes - Steer Axle",
    "13": "Brakes - Drive axle #1",
    "14": "Brakes - Drive Axle #2",
    "15": "Retarder - Engine",
    "16": "Retarder - Driveline",
    "17": "Cruise Control",
    "18": "Fuel System",
    "19": "Steering Controller",
    "20": "Suspension - Steer Axle",
    "21": "Suspension - Drive Axle #1",
    "22": "Suspension - Drive Axle #2",
    "23": "Instrument Cluster #1",
    "24": "Trip Recorder",
    "25": "Passenger-Operator Climate Control #1",
    "26": "Alternator/Electrical Charging System",
    "27": "Aerodynamic Control",
    "28": "Vehicle Navigation",
    "29": "Vehicle Security",
    "30": "Electrical System",
    "31": "Starter System",
    "32": "Tractor-Trailer Bridge #1",
    "33": "Body Controller",
    "34": "Auxiliary Valve Control or Engine Air System Valve Control",
    "35": "Hitch Control",
    "36": "Power TakeOff (Front or Secondary)",
    "37": "Off Vehicle Gateway",
    "38": "Virtual Terminal (in cab)",
    "39": "Management Computer #1",
    "40": "Cab Display #1",
    "41": "Retarder, Exhaust, Engine #1",
    "42": "Headway Controller",
    "43": "On-Board Diagnostic Unit",
    "44": "Retarder, Exhaust, Engine #2",
    "45": "Endurance Braking System",
    "46": "Hydraulic Pump Controller",
    "47": "Suspension - System Controller #1",
    "48": "Pneumatic - System Controller",
```

Listing 1.4: J1939 Source and Destination Addresses (continued)

```
    "49": "Cab Controller - Primary",
    "50": "Cab Controller - Secondary",
    "51": "Tire Pressure Controller",
    "52": "Ignition Control Module #1",
    "53": "Ignition Control Module #2",
    "54": "Seat Control #1",
    "55": "Lighting - Operator Controls",
    "56": "Rear Axle Steering Controller #1",
    "57": "Water Pump Controller",
    "58": "Passenger-Operator Climate Control #2",
    "59": "Transmission Display - Primary",
    "60": "Transmission Display - Secondary",
    "61": "Exhaust Emission Controller",
    "62": "Vehicle Dynamic Stability Controller",
    "63": "Oil Sensor",
    "64": "Suspension - System Controller #2",
    "65": "Information System Controller #1",
    "66": "Ramp Control",
    "67": "Clutch/Converter Unit",
    "68": "Auxiliary Heater #1",
    "69": "Auxiliary Heater #2",
    "70": "Engine Valve Controller",
    "71": "Chassis Controller #1",
    "72": "Chassis Controller #2",
    "73": "Propulsion Battery Charger",
    "74": "Communications Unit, Cellular",
    "75": "Communications Unit, Satellite",
    "76": "Communications Unit, Radio",
    "77": "Steering Column Unit",
    "78": "Fan Drive Controller",
    "79": "Seat Control #2",
    "80": "Parking brake controller",
    "81": "Aftertreatment #1 system gas intake",
    "82": "Aftertreatment #1 system gas outlet",
    "83": "Safety Restraint System",
    "84": "Cab Display #2",
    "85": "Diesel Particulate Filter Controller",
    "86": "Aftertreatment #2 system gas intake",
    "87": "Aftertreatment #2 system gas outlet",
    "88": "Safety Restraint System #S",
    "89": "Atmospheric Sensor",
    "248": "File Server / Printer",
    "249": "Off Board Diagnostic-Service Tool #1",
    "250": "Off Board Diagnostic-Service Tool #2",
    "251": "On-Board Data Logger",
    "252": "Reserved for Experimental Use",
    "253": "Reserved for OEM",
    "254": "Null Address",
    "255": "GLOBAL (All-Any Node)"
}
```

PGN 61444 ***Electronic Engine Controller 1*** ***EEC1***

Engine related parameters

Transmission Repetition Rate:	engine speed dependent
Data Length:	8
Extended Data Page:	0
Data Page:	0
PDU Format:	240
PDU Specific:	4 PGN Supporting Information:
Default Priority:	3
Parameter Group Number:	61444 (0x00F004)

Start Position	Length	Parameter Name	SPN
1.1	4 bits	Engine Torque Mode	899
1.5	4 bits	Actual Engine - Percent Torque High Resolution	4154
2	1 byte	Driver's Demand Engine - Percent Torque	512
3	1 byte	Actual Engine - Percent Torque	513
4-5	2 bytes	Engine Speed	190
6	1 byte	Source Address of Controlling Device for Engine Control	1483
7.1	4 bits	Engine Starter Mode	1675
8	1 byte	Engine Demand – Percent Torque	2432

Figure 1.1: Data entry in SAE J1939 for the Electronic Engine Controller 1 message.

PGN 61444 (0xF004) is for the Electronic Engine Control 1 that contains the engine speed, which is shown as the bottom entry in the short section of the log above. This is a frequent message and is ubiquitus. It is responsible for displaying the engine speed on the instrument cluster. However, how do we translate the CAN message into some engineering value that is readable on an instrument or able to be plotted. Information from the SAE J1939-71 document as shown in Figure 1.1 and Figure 1.2 on the next page can help us decode the message. The complete J1939 database can be obtained from SAE.

In J1939, all positions are indexed from 1, whereas in C/C++ and Python, the starting index is zero. This is important when setting up constants for bit and byte positions in the different measurements. Our interest is in the engine speed, which is shown as SPN 190 in positions 4 and 5. The numbers in J1939 that are multiple bytes in length are represented in the big endian format, or reverse byte order. This means the least significant byte in the engine speed is in position 4 and the most significant byte is in postion 5. This is also known as the Intel format. This is opposite they way humans normally read numbers, which is from left to right. In Python, we can make use of the struct library. There is a format string for structs that enable multibyte numbers to be interpreted in the correct endianness. The listing of the format structures is available in the Python documentation.

Armed with the standard decoding strategy, we can now write a program to examine the log file for engine speeds. Listing 1.5 on page 13 is a complete example for creating the resulting plot of the engine speed shown in Figure 1.3 on page 15. When we look at Listing 1.5, you'll notice a carry over of the code in Listing 1.2 to parse CAN IDs into J1939 fields. We also had to add a function to parse the candump file format we used for our log file. This function relies on the structure of each line to produce all the parts of the CAN message. The data is returned as a dictionary and has the ability to handle

SPN 190 ***Engine Speed***

Actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders.

Data Length:	2 bytes	
Resolution:	0.125 rpm/bit, 0 offset	
Data Range:	0 to 8,031.875 rpm	Operational Range: same as data range
Type:	Measured	
Supporting Information:		
PGN reference:	61444	

Figure 1.2: Data entry in SAE J1939-71 for Engine Speed (SPN 190).

data length entries less than 8. The main function is called when this program is run as itself. If the python file is imported into another program, then the main function would not be called, but the functions would be available in memory.

The main function starts with setting up empty lists to keep track of the data from the file. To run this script successfully, you will need access to the log file called KWTruck.txt.

1.3 Examples and Exercises

Exercise 1.2. [Apply] Read Live Engine RPM Challenge

Using the hardware of your choice, connect to an engine contoller that is broadcasting non-zero engine RPM. Gather this data in a text format where each line has the following elements:

1. Timestamp in the number of seconds from the Unix epoch
2. CAN Channel
3. CAN Arbitration Identifier in hexadecimal format
4. CAN message data length code (DLC)
5. CAN data in hex format.

Interpret the raw CAN frames and extract information for Engine RPM, or J1939 SPN 190. Plot 20 seconds of changing RPM with matplotlib. Print the properly labeled plot to PDF and show it to your instructor.

Exercise 1.3. [Evaluate] Deconstruct the voltage trace of a CAN frame recorded with an ocilloscope. Decode the signal into J1939 SPNs with values and units.

Exercise 1.4. Man in the Middle

Listing 1.5: Python code to plot engine speed vs time for a CAN log.

```
#!/usr/bin/env python3
import struct
import matplotlib.pyplot as plt

PRIORITY_MASK       = 0x1C000000
PDU_FORMAT_MASK     = 0x00FF0000
PDU_SPECIFIC_MASK   = 0x0000FF00
SOURCE_ADDRESS_MASK = 0x000000FF
PDU1_PGN_MASK       = 0x03FF0000
PDU2_PGN_MASK       = 0x03FFFF00

PRIORITY_OFFSET    = 26
PDU_FORMAT_OFFSET  = 16
DA_OFFSET          = 8
PGN_OFFSET         = 8
PDU2_THRESHOLD     = 240

GLOBAL_SOURCE_ADDR = 0xFF
ENGINE_SA          = 0

CANDUMP_TIMESTAMP_ADDR = 0
CANDUMP_CHANNEL_ADDR   = 1
CANDUMP_ID_ADDR        = 2
CANDUMP_DLC_ADDR       = 3
DATA_START_ADDR        = 4

PGN_EEC1 = 61444
SPN190_SCALE = 0.125 #RPM/bit
SPN190_OFFSET = 0

def main():
    spn190_times = []
    spn190_values = []
    filename = 'KWTruck.txt'
    with open(filename,'r') as f:
        for line in f:
            # We knew this data file was from Linux using candump.
            j1939_frame = parse_candump_line(line)
            # Add the additional results from parsing the id
            j1939_frame.update(parseJ1939id(j1939_frame['id']))
            # PGN for electronic engine control 1 messsage
            if (j1939_frame['pgn'] == PGN_EEC1 and
                j1939_frame["source_address"] == ENGINE_SA):
                # Unpack the engine speed data in big endian format and
                # convert to engineering values
                rpm = (struct.unpack('<H',j1939_frame['data'][3:5])[0]
                        * SPN190_SCALE + SPN190_OFFSET)
                spn190_values.append(rpm)
                # Include the timestamp for time series data
                spn190_times.append(j1939_frame['timestamp'])
    #Plot the engine speed
    plt.plot(spn190_times,spn190_values,'-',label="Engine RPM")
    plt.xlabel("Time (sec.)")     plt.ylabel("Engine Speed (RPM)")
    plt.title("Engine Speed for {}".format(filename))
    plt.grid()
    plt.legend()
    plt.show()
    plt.savefig('EngineSpeedGraphFrom{}.pdf'.format(filename))
```

Listing 1.6: Python code to plot engine speed vs time for a CAN log (cont).

```
59
60 def parseJ1939id(id):
61     sa = id & SOURCE_ADDRESS_MASK
62     priority = (id & PRIORITY_MASK) >> PRIORITY_OFFSET
63     pf = (id & PDU_FORMAT_MASK) >> PDU_FORMAT_OFFSET
64     if (pf < PDU2_THRESHOLD):  # See SAE J1939-21
65         # PDU 1 format uses values lower than 240
66         da = (id & PDU_SPECIFIC_MASK) >> DA_OFFSET
67         pgn = (id & PDU1_PGN_MASK) >> PGN_OFFSET
68     else:
69         # PDU 2 format
70         da = GLOBAL_SOURCE_ADDR
71         pgn = (id & PDU2_PGN_MASK) >> PGN_OFFSET
72     return {'source_address': sa,
73             'priority': priority,
74             'destination_address': da,
75             'pgn': pgn}
76
77 def parse_candump_line(line):
78     # Strip the newline characters and white space off the ends
79     # then split the string into a list based on whitespace.
80     data = line.strip().split()
81     # can_dump formats use parentheses to wrap the floating
82     # point timestamp. We just want the numbers so use [1:-1] to slice
83     time_stamp = float(data[CANDUMP_TIMESTAMP_ADDR][1:-1])
84     # physical CAN channel
85     channel = data[CANDUMP_CHANNEL_ADDR]
86     # determine the can arbitration identifier as an integer
87     can_id = int(data[CANDUMP_ID_ADDR],16)
88     # Data length code is a single byte wrapped in []
89     dlc = int(data[CANDUMP_DLC_ADDR][1])
90     # Build the data field as a byte array
91     data_bytes = b''
92     for byte_string in data[DATA_START_ADDR:]:
93         # Convert text into a single byte
94         data_bytes += struct.pack('B',int(byte_string,16))
95     #assert dlc == len(data_bytes)
96     can_frame = {'id':can_id,
97                  'dlc':dlc,
98                  'data':data_bytes,
99                  'timestamp':time_stamp,
100                 'channel':channel}
101    return can_frame
102
103 if __name__ == '__main__':
104     main()
```

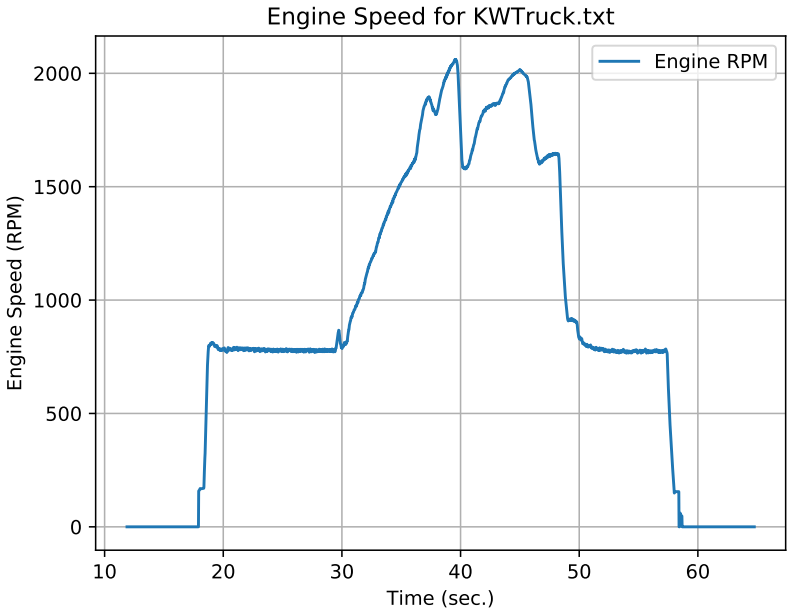

Figure 1.3: Engine speed graph produced by the Python source code in Listing 1.5 on page 13

Build a man-in-the middle board and box that takes CAN signals into one can channel and sends them out on another. Start a diagnostics session with a PC and RP1210 device to perform maintenance. Create a forwarding system that inspects and forwards network traffic in both directions. Attempt to hijack a a diagnostic session and affect a parameter change started with the PC diagnostics software.

Using DDEC Reports, try to prevent resetting the CPC clock during a data extraction on a CPC.