READING OBD DATA FROM A LIGHT VEHICLE

1. Introduction

On-board diagnostics (OBD) is a term referring to a vehicle's self-diagnostic and reporting capability. In most countries, this capability is a requirement to comply with emissions standards and to detect failures that may increase the vehicle tailpipe emissions.

OBD systems give the repair technician access to the status of the various vehicle sub-systems. Modern OBD-II implementations typically use a standardised CAN Bus communications port to provide real-time data and diagnostic trouble codes which allow malfunctions within the vehicle to be rapidly identified.

This application note describes how to retrieve standard OBD information from a light vehicle that complies with the OBD-II standard. For heavy vehicles, see the Senquip application notes on J1939.

Further details on the Senquip scripting language can be found in the Device Scripting Guide.

2. OBD-II Introduction

OBD-II is an improvement over OBD-I in both capability and standardisation. The OBD-II standard specifies the type of diagnostic connector and its pinout, the electrical signalling protocols available, and the messaging format. It also provides a candidate list of vehicle parameters to monitor along with how to encode the data for each.

As shown in Figure 1, you can determine whether your car has OBD-II by knowing where and when it was purchased.

Figure 1 - Introduction of OBD-II

2.1. Physical Interface

While the location of the OBD-II port is not standardised, there are some common places where they are found:

- **Beneath the steering column**. Depending on the car model, the OBD port may be to the left, in the middle, or to the right of the underside of the steering wheel.
- **To the left or right of the car's dashboard**. You'll usually find it closer to the dashboard, a bit further away from the rest of the wheel.

- **Between the transmission and cup holder**. Usually located near the bottom of the transmission or cup holder.
- **Near the music or navigation system**. Sometimes, the OBD port may be placed somewhere next to the music or navigation system.
- **Beneath the glove compartment**. In some cases, the port may be placed on the passenger's side, placed beneath the glove compartment.

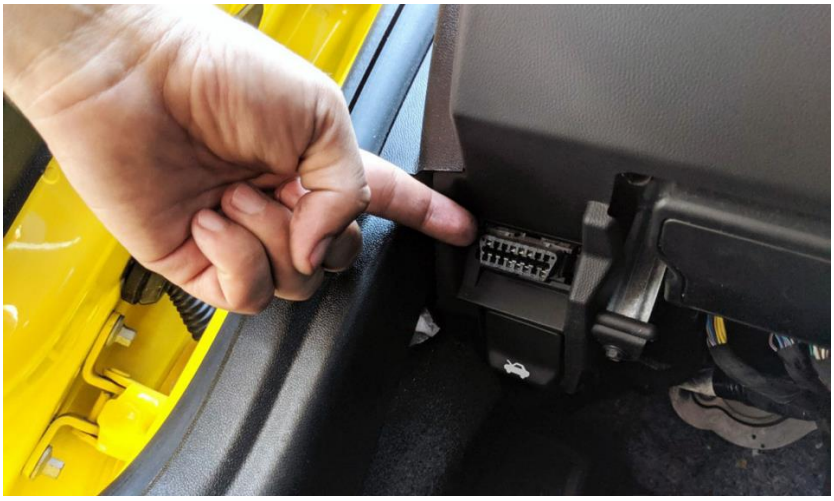


Figure 2 - Typical OBD-II Socket Location

The OBD-II specification provides for a standardised hardware interface, the female 16-pin (2x8) J1962 connector, where type A is used for 12-volt vehicles and type B for 24-volt vehicles. Type A and B connectors are the same except type B connectors have the centre groove split in two.

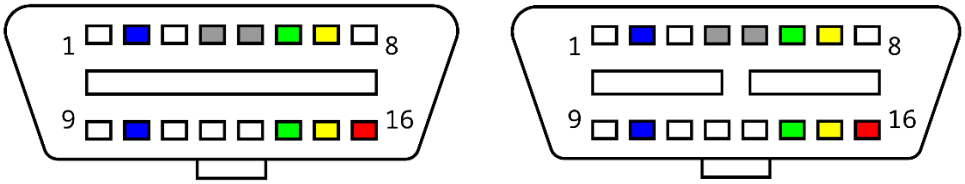


Figure 3 - Female Type A Connector (left) and Female Type B Connector (right)

The pinout from type A and B are the same and are shown below. Since 2008, CAN bus (ISO 15765) has been the mandatory protocol for OBD-II in all cars sold in the US and since then, most of the rest of the world. In this application note, we will focus on extracting data from the vehicle using the CAN Bus on pins 6 and 14.

Pin	Description	Pin	Description
1	Manufacturer Discretionary	9	Manufacturer Discretionary
2	SAE J1850 Bus + (VPW / PWM)	10	SAE J1850 Bus - (PWM-only)
3	Manufacturer Discretionary	11	Manufacturer Discretionary
4	Chassis Ground	12	Manufacturer Discretionary
5	Signal Ground	13	Manufacturer Discretionary
6	CAN High (ISO 15765-4 and SAE J2284)	14	CAN Low (ISO 15765-4 and SAE J2284)
7	ISO 9141-2 / ISO 14230-4 K Line	15	ISO 9141-2 / ISO 14230-4 L Line (Optional)
8	Manufacturer Discretionary	16	Vehicle Battery Power: Type A 12V/4A, Type B 24V/2A

2.2. OBD-II Protocol

SAE standard J1979 defines the OBD-II protocol and defines a range of standard Parameter IDs (PIDs) that can be logged across most cars. This means that you can easily get human-readable OBD-II data from your car on speed, RPM, throttle position and more. Manufacturers also define additional PIDs specific to their vehicles; these are more complex to decode. A list of standard PIDs can be found here.

OBD-II Modes

There are 10 diagnostic services, or modes, described in the latest OBD-II standard. These modes are provided by the ECM to allow functions such as clearing diagnostic codes or monitoring live data. Manufacturers are not required to support all modes and they are allowed to create additional modes if required.

Mode (hex)	Description
01	Show current data
02	Show freeze frame data
03	Show stored Diagnostic Trouble Codes
04	Clear Diagnostic Trouble Codes and stored values
05	Test results, oxygen sensor monitoring (non CAN only)
06	Test results, other component/system monitoring (Test results, oxygen sensor monitoring for CAN only)
07	Show pending Diagnostic Trouble Codes (detected during current or last driving cycle)
08	Control operation of on-board component/system
09	Request vehicle information
0A	Permanent Diagnostic Trouble Codes (DTCs) (Cleared DTCs)

When a request is made using a mode, the response will contain the mode with 0x40 added. So for instance if a mode 0x01 PID is requested, the response mode will be 0x41.

This application note will focus on the request of current data, mode 1.

OBD-II Parameter IDs (PID)

Each mode has associated PIDs used to request specific data. For instance, to request engine speed using mode 1 (show current data), PID 0x0C (engine speed) is used. The table below shows a few common PIDs and how to decode the data that is returned when they are requested. A full set of service mode 1 PIDs is given in Appendix 1.

PID (hex)	PID (dec)	No. bytes returned	Description	Min	Max	Unit	Formula
04	4	1	Calculated engine load	0	100	%	A/2.55
05	5	1	Engine coolant temperature	-40	215	°C	A-40
0C	12	2	Engine speed	0	16,383.75	rpm	(256*A + B)/4
0D	13	1	Vehicle speed	0	255	km/h	A

Figure 4- Example PID Decriptions

The references A, B, C, D in the formula table are byte positions in the returned data. Note for instance that engine load return as single byte and so only A is used in the formula, whereas engine speed returns 2 bytes and so A and B are used in the formula to calculate speed.

The first byte to be returned is A, followed by B, C, and D. When referring to bits in each byte, quantities like C4 means bit 4 from data byte C. Each bit is numbered from 0 to 7, with 7 being the most significant bit and 0 is the least significant bit.

A								B								C								D							
A7	A6	A5	A4	A3	A2	A1	A0	B7	B6	B5	B4	B3	B2	B1	B0	C7	C6	C5	C4	C3	C2	C1	C0	D7	D6	D5	D4	D3	D2	D1	D0

OBD-II Frame Details

OBD uses standard format CAN where the ID has 11 bits. This in contrast with heavy vehicle J1939 that uses the 29 bit extended format. CAN bit rates are typically 500bps in light vehicles, with 250kbps being seen in some vans and other larger vehicles.

A standard OBD frame consists of an identifier and CAN data. When requesting PIDs, the broadcast identifier 0x7DF is used. The ECU will reply with an identifier in the range 0x7E8 to 0x7EF. Note that 0x7E8 typically be where the main engine or ECU responds at.

Request Identifier: 0x7DF
Response Identifier: 0x7E8 to 0x7EF

The CAN data field consists of a message length, mode, PID, and data fields. The length field reflects the length in number of bytes of the remaining data. For the vehicle speed request shown below, identifier is 0x7DF which is the standard request identifier. The data field contains length of 2 as only mode and PID bytes follow. Mode 1 request messages always have length 2.

	Identifier	**Data Bytes**							
	Request	length	mode	PID					unused
Example	0x7DF	0x02	0x01	0x0D	X	X	X	X	X

Figure 5 - Vehicle Speed Request Message

The vehicle speed response comes from the ECU with identifier 0x7E8. Length is 3 because the data bytes contain the mode, PID, and 1 data byte for speed (see Figure 4).

	Identifier	**Data Bytes**							
	Response	length	Mode	PID	A	B	C	D	unused
Example	0x7E8	0x03	0x41	0x0D	0x1F	X	X	X	X

Figure 6 - Vehicle Speed Response Message

Vehicle speed is contained in byte A, in km/h or 0x1F = 31km/h.

Looking at a similar example for requesting engine speed, the request message is:

	Identifier	**Data Bytes**							
	Request	length	mode	PID					unused
Example	0x7DF	0x02	0x01	0x0C	X	X	X	X	X

Figure 7 - Engine Speed Request Message

Again, the engine speed response comes from the ECU with identifier 0x7E8. Length is 4 this time because the engine speed PID returns 2 bytes in addition to the mode and PID bytes.

	Identifier	**Data Bytes**							
	Response	length	mode	PID	A	B	C	D	unused
Example	0x7E8	0x03	0x41	0x0C	0x0C	0x36	X	X	X

Figure 8 - Engine Speed Response

Engine speed (from Figure 4) is contained in bytes A and B, in RPM with the formula (256*A + B)/4 being applied. Engine speed = 0x0C36/4 = 781.5RPM.

3. Senquip Device Connection

The Senquip device can be powered from the OBD-II connector and extracts data using the CAN peripheral. In the example below, a Senquip ORB is connected to a male OBD-II type A plug.

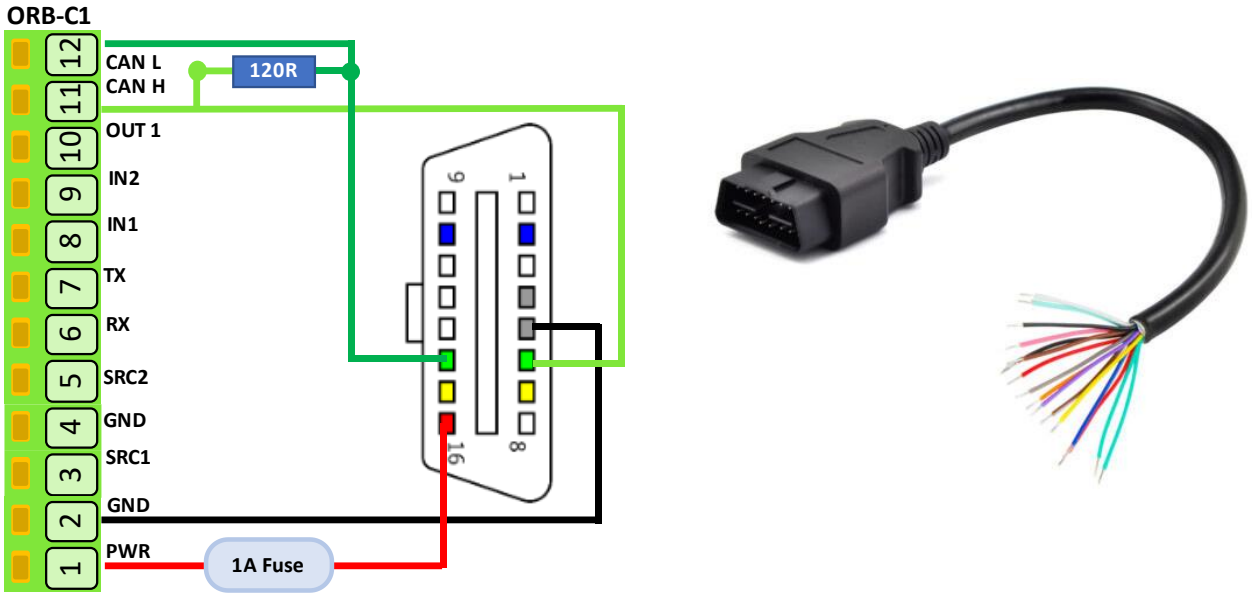


Figure 9 - Typical Senquip ORB OBD-II Connection

The Senquip ORB is configured with a base interval of 10 seconds. The CAN peripheral is set to be samples on each base interval and is configured for a baud rate of 500bps. The capture time is set as the same at the base interval to ensure that all CAN messages are captured. Because we are going to be requesting CAN data, transmitting on the bus is enabled. A CAN filter is specified so that only messages from the allowed response identifiers 0x7E8 to 0x7EF are received. Without the filter, hundreds of manufacturer specific messages are received. After an initial test, it was noticed that all responses were coming from 0x7E8 and 0x7E9 and so the filter was changed to allow 10 messages from each of these 2 addresses. The final filter is: 7E0, 7E1, 7E2, 7E3, 7E4, 7E5, 7E6, 7E7, 7E8*10, 7E9*10, 7EA, 7EB, 7EC, 7ED, 7EE, 7EF. The raw CAN messages are sent back to the Senquip Portal for debugging purposes. The raw data option will be turned off later to reduce data throughput.

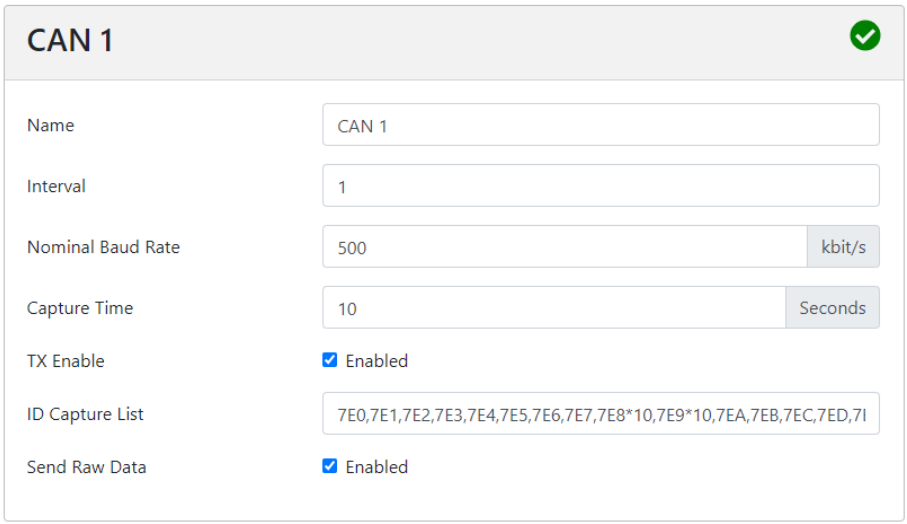


Figure 10 - Senquip ORB CAN Peripheral Settings.

4. Requesting PIDs in a Script

A script will be written to request the following PIDs at a 5 second interval:

- 0x0C: Engine speed
- 0x0D: Vehicle speed
- 0x04: Engine load
- 0x05: Coolant temperature
- 0x11: Throttle position
- 0x1F: Run time since start
- 0x2F: Fuel level
- 0xA6: Odometer

Required libraries are included and a function is written to extract the PID from a CAN message. PID request messages are sent using the repeat send function.

```
load('senquip.js');
load('api_timer.js');

function pid(data) {return(data.slice(4,6));}

//request messages
 CAN.tx(1, 0x7DF, "\x02\x01\x0C\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(0), 3000); // request engine rpm
 CAN.tx(1, 0x7DF, "\x02\x01\x0D\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(1), 3100); // request vehicle speed
 CAN.tx(1, 0x7DF, "\x02\x01\x04\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(2), 3200); // request engine load
 CAN.tx(1, 0x7DF, "\x02\x01\x05\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(3), 3300); // request engine coolant temp
 CAN.tx(1, 0x7DF, "\x02\x01\x11\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(4), 3400); // throttle position
 CAN.tx(1, 0x7DF, "\x02\x01\x1F\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(5), 3500); // run since start
 CAN.tx(1, 0x7DF, "\x02\x01\x2F\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(6), 3600); // fuel level
```

Figure 11 shows a typical scan of the CAN Bus as shown in Senquip Portal in the raw data widget. In the messages, see that the responding address is 0x7E8 for all messages. The 1st byte tells us how many of the subsequent bytes are valid data. The 2nd byte contains the response to a mode 1 request, 0x41. The 3rd byte is the PID, and the subsequent valid bytes are data.

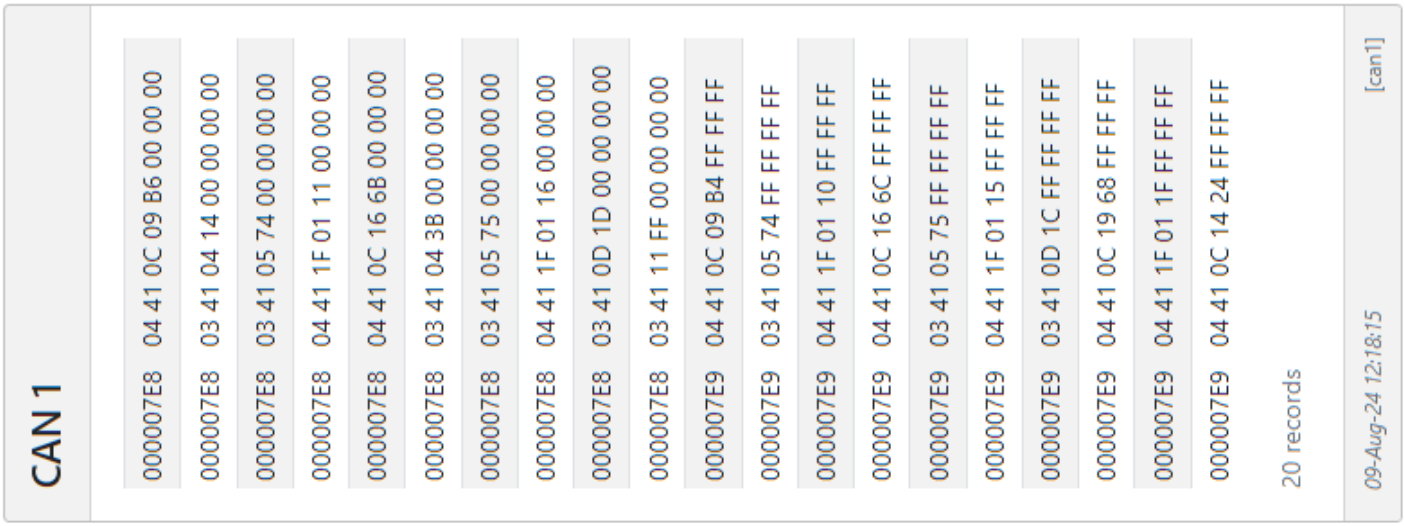
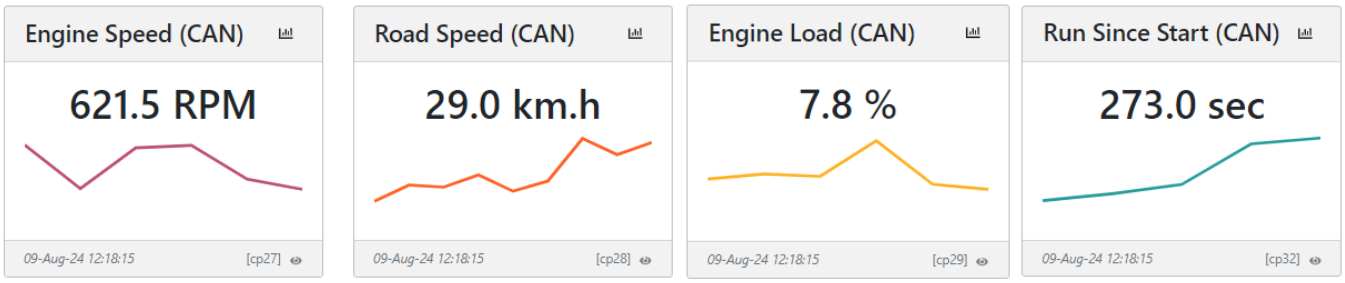


Figure 11 - OBD-II Response to Mode 1 Request

In the data handler, we loop through all the received CAN messages, checking if the identifier is that of the ECU. If so, we check the PID and apply the corresponding formula to extract the required parameter. Each parameter is dispatched to the Senquip Portal.



The script has been tested on the following vehicles:

- Toyota Camry
- Toyota Prado
- Mitsubishi Pajero

In some cases, the ECU will not respond to a request message. In the case of the fuel level request message, this could for instance, be because the fuel sender is mechanical and is not available on the CAN network.

```
SQ.set_data_handler(function(data) {
 let obj = JSON.parse(data);
 if (typeof obj.can1 !== "undefined") {
   for (let i = 0; i < obj.can1.length; i++) {
    if (obj.can1[i].id === 0x7E8 ||  obj.can1[i].id === 0x7E9){ // These are the ecu id's

      if (pid(obj.can1[i].data) === "0C"){ // engine speed
       let pid0c = SQ.parse(obj.can1[i].data, 6, 4, 16)*0.25;
       SQ.dispatch(1, pid0c);
      }
      else if (pid(obj.can1[i].data) === "0D"){ // vehicle speed
       let pid0d = SQ.parse(obj.can1[i].data, 6, 2, 16);
       SQ.dispatch(2, pid0d);
      }
      else if (pid(obj.can1[i].data) === "04"){ // engine load
       let pid04 = SQ.parse(obj.can1[i].data, 6, 2, 16)/2.55;
       SQ.dispatch(3, pid04);
      }
      else if (pid(obj.can1[i].data) === "05"){ // engine coolant temp
       let pid05 = SQ.parse(obj.can1[i].data, 6, 2, 16)-40;
       SQ.dispatch(4, pid05);
      }
      else if (pid(obj.can1[i].data) === "11"){ // throttle position
       let pid11 = SQ.parse(obj.can1[i].data, 6, 2, 16)/2.55;
       SQ.dispatch(5, pid11);
      }
      else if (pid(obj.can1[i].data) === "1F"){ // run since start
       let pid1f = SQ.parse(obj.can1[i].data, 6, 4, 16);
       SQ.dispatch(6, pid1f);
      }
      else if (pid(obj.can1[i].data) === "2F"){ // fuel level
       let pid2f = SQ.parse(obj.can1[i].data, 6, 2, 16)/2.55;
       SQ.dispatch(7, pid2f);
      }
     }
   }
}, null);
```

Although this application note has dealt with standard PIDs, most of all OBD-II PIDs in use are non-standard. For most modern vehicles, there are many more functions supported on the OBD-II interface than are covered by the standard PIDs, and there is relatively minor overlap between vehicle manufacturers for these non-standard PIDs.

5. Conclusions

On-board diagnostics (OBD) enables the reading of standard parameters from most vehicles over CAN Bus.

A simple script enables a Senquip device to request standard PIDs and to decode the responses.

Not all vehicles will respond to all PID requests. Additional information is available using non-standard PIDs.

6. Appendix I – Example Script to Request PIDs

```
load('senquip.js');
load('api_timer.js');

function pid(data) {return(data.slice(4,6));}

//request messages
 CAN.tx(1, 0x7DF, "\x02\x01\x0C\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(0), 3000); // request engine rpm
 CAN.tx(1, 0x7DF, "\x02\x01\x0D\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(1), 3100); // request vehicle speed
 CAN.tx(1, 0x7DF, "\x02\x01\x04\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(2), 3200); // request engine load
 CAN.tx(1, 0x7DF, "\x02\x01\x05\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(3), 3300); // request engine coolant temp
 CAN.tx(1, 0x7DF, "\x02\x01\x11\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(4), 3400); // throttle position
 CAN.tx(1, 0x7DF, "\x02\x01\x1F\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(5), 3500); // run since start
 CAN.tx(1, 0x7DF, "\x02\x01\x2F\xCC\xCC\xCC\xCC\xCC", 8, CAN.STD + CAN.TX_SLOT(6), 3600); // fuel level

SQ.set_data_handler(function(data) {
 let obj = JSON.parse(data);
 if (typeof obj.can1 !== "undefined") {
   for (let i = 0; i < obj.can1.length; i++) {
    if (obj.can1[i].id === 0x7E8 ||  obj.can1[i].id === 0x7E9){ // These are the id's of the ecu

      if (pid(obj.can1[i].data) === "0C"){ // engine speed
        let pid0c = SQ.parse(obj.can1[i].data, 6, 4, 16)*0.25;
        SQ.dispatch(1, pid0c);
      }
      else if (pid(obj.can1[i].data) === "0D"){ // vehicle speed
        let pid0d = SQ.parse(obj.can1[i].data, 6, 2, 16);
        SQ.dispatch(2, pid0d);
      }
      else if (pid(obj.can1[i].data) === "04"){ // engine load
        let pid04 = SQ.parse(obj.can1[i].data, 6, 2, 16)/2.55;
        SQ.dispatch(3, pid04);
      }
      else if (pid(obj.can1[i].data) === "05"){ // engine coolant temp
        let pid05 = SQ.parse(obj.can1[i].data, 6, 2, 16)-40;
        SQ.dispatch(4, pid05);
      }
      else if (pid(obj.can1[i].data) === "11"){ // throttle position
        let pid11 = SQ.parse(obj.can1[i].data, 6, 2, 16)/2.55;
        SQ.dispatch(5, pid11);
      }
      else if (pid(obj.can1[i].data) === "1F"){ // run since start
        let pid1f = SQ.parse(obj.can1[i].data, 6, 4, 16);
        SQ.dispatch(6, pid1f);
      }
      else if (pid(obj.can1[i].data) === "2F"){ // fuel level
        let pid2f = SQ.parse(obj.can1[i].data, 6, 2, 16)/2.55;
        SQ.dispatch(7, pid2f);
      }
    }
   }
 }

}, null);
```