PARSING A CAN BUS MESSAGE WITH A SCRIPT

1. Introduction

The Controller Area Network (CAN bus) is a robust communication standard used widely in vehicles and industrial systems. It allows modules to exchange data efficiently without a central controller.

Many Senquip devices include a CAN interface, enabling direct access to vehicle and machinery data such as engine speed, fuel rate, and temperature. By parsing these CAN messages in a script, users can extract specific parameters, scale them into engineering units, and dispatch them to the Senquip Portal for monitoring, alerts, or control logic.

This application note discusses the parsing of CAN messages within a script. Two methods are discussed, with the second offering massive efficiency gains.

The application note assumes you have access to the scripting feature, that the Senquip device is subscribed to a Hosted plan, and that you are familiar with the scripting environment on the Senquip Portal. For further details on scripting on Senquip device, see the scripting guide.

2. Introduction to J1939

SAE J1939 is a standard developed by the Society of Automotive Engineers (SAE) that defines communication for vehicle networks in trucks, buses, agricultural equipment, mining equipment and other heavy machinery.

J1939 data messages, or packets, contain a header and data. The header contains an identifier which is made up of:

- The priority of the message
- A message type indicator (PGN)
- Source address

The PGN contained in the identifier describes the message function and the data that will be contained in that message. J1939 attempts to define standard PGNs to encompass a wide range vehicle purposes. Some example PGN's include:

PGN	Description
pgn61442	Electronic Transmission Controller 1 - ETC1
pgn61444	Electronic Engine Controller 1 - EEC1
pgn64978	ECU Performance - EP
pgn65031	Exhaust Temperature - ET
pgn65102	Tachograph - TCO1
pgn65203	Fuel Information (Liquid) - LFI

Each PGN may contain multiple parameters. For example, PGN 61444, Electronic Engine Controller 1, contains the following parameters:

- engine torque demand,

- drivers demand,
- actual engine torque,
- engine speed,
- and more.

Each parameter is known as an SPN (Suspect Parameter Number). A PGN identifies a message that contains multiple SPNs.

A description of each PGN and how the SPNs are arranged within that message is given in the J1939 specification. SAE publish a spreadsheet of all J1939 PGNs, SPNs, Manufacturer codes and more.

For example, PGN 61444, Electronic Engine Controller 1 has the following description and structure.

PGN 61444 Electronic Engine Controller 1

Transmission repetition rate	Engine speed dependent
Data length	8
Extended data page	0
Data page	0
PDU format	240
PDU specific	4
Default priority	3

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

To extract engine speed (SPN190), from the PGN 61444 message, we need to extract bytes 4 and 5. For a full description of SPN190, we again refer to the specification.

The definition of the SPN190 message – Engine Speed is "actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders."

SPN 190 – Engine Speed

Data length	2 bytes
Resolution	0.125 rpm/bit, zero offset
Data range	0 to 8,031.875 rpm
Type	Measured
Suspect Parameter Number	190
Parameter Group Number	[61444]

From the full description, we see that the engine speed contained in bytes 4 and 5 of the PGN needs to be multiplied by 0.125 to determine the RPM. The process of determining the RPM now becomes:

- Capture the Electronic Engine Controller 1 message (PGN 61444)
- Extract SPN 190 from bytes 4 and 5
- Multiply the SPN by 0.125

These operations are perfect for a script running on a Senquip device.

You may have noticed that we are looking to capture PGN 61444, however capturing is done based on CAN bus identifiers. Senquip has a simple Excel CAN calculator to assist with the conversion from PGN to CAN identifier and back.

The CAN identifier for J1939 is 29 bits and is made up of 3 bits of priority, 18 bits of PGN and 8 bits of source address. The priority is defined in the specification and is 3 for this message. The PGN we know is 61444. The source address is the address of the module sending the message and is assumed to be 0 in this case. The source address can be obtained by doing a scan of all messages on the bus and determining the source address from the last 8 bits.

Part	Bits	Value	Binary
Priority	3	3	011
PGN	18	61444	00 1111 0000 0000 0100
Source Address	8	0	0000 0000

By combining the 3 components of the identifier, we get 011 00 1111 0000 0000 0100 0000 0000 or in decimal, 217056256, or hex CF00400.

This is the number that will be captured by the CAN peripheral on the Senquip device. In our script, we will write a small function to extract the PGN from the CAN identifier to make the script more readable, and to ensure we capture the message, even if we have the source address wrong.

3. Capturing CAN Bus Message

Senquip ORB and QUAD devices have integrated CAN bus interfaces that can be configured to capture messages on a CAN bus. A typical automotive CAN network will contain hundreds of messages, all with their own identifiers. Senquip devices can capture all messages received or can filter only the required messages by filling in the *ID Capture List* field. For more details on available CAN filters, see the device User Guides.

In this example, we have configured the CAN bus to:

- Have a bitrate of 250kbps
- Capture CAN data for the first 5 seconds of every measurement cycle (*base interval*)
- Since this is the only other device on the CAN network and CAN messages must be acknowledged, Tx is enabled
- We will send the captured CAN messages to the Senquip Portal

Note: Every CAN message must be acknowledged by at least one receiver on the bus.
If no node sends an ACK, the transmitter assumes the message failed and automatically retries until it succeeds.

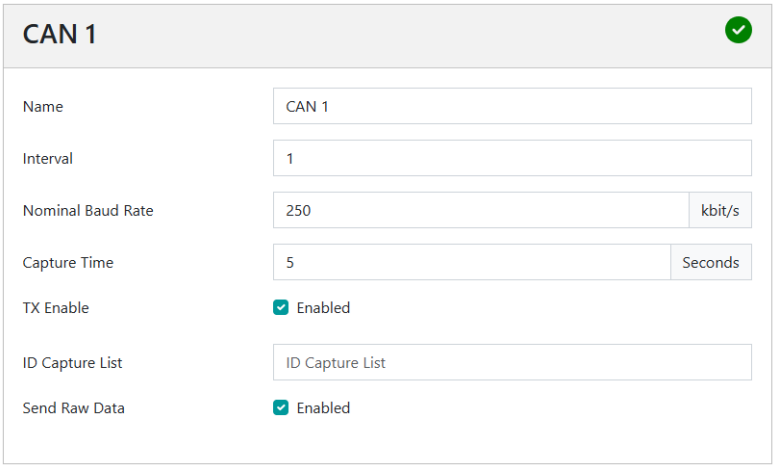

Figure 1 - CAN peripheral settings

A large amount of CAN data can be seen arriving on the Senquip Portal included in which is the Electronic Engine Controller 1 message.

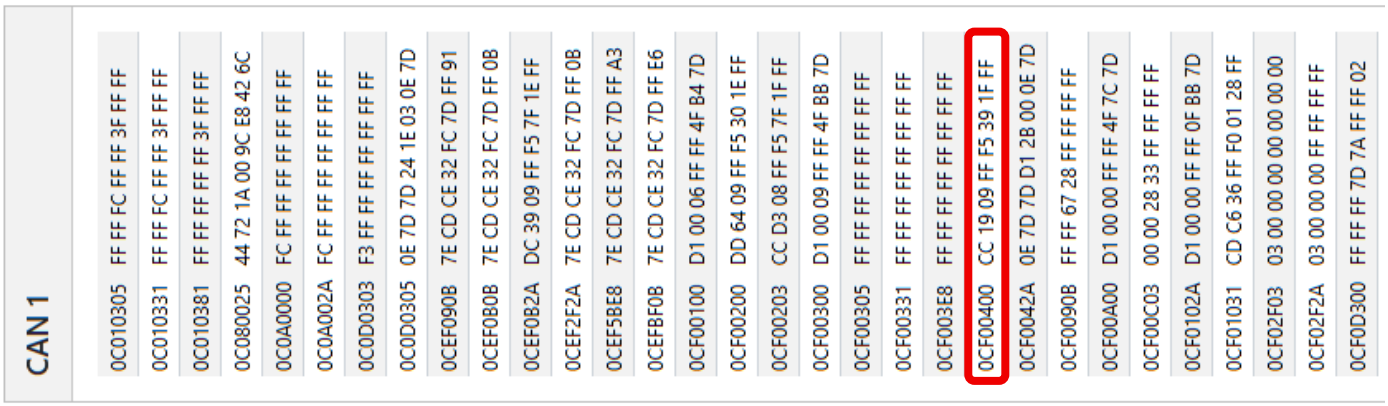

Figure 2 - CAN Data as Received on the Senquip Portal

Note: Once we have developed our application, it is suggested that the raw CAN feed be turned off to reduce the amount of data being sent by the Senquip device over the network.

Once the CAN peripheral is configured, captured messages appear as JSON data in the raw data packet under the key can1. These messages can then be parsed in a script.

4. Parsing the CAN Data

Error! Reference source not found. shows the Electronic Engine Controller 1 message arriving at the Senquip Portal. N ote the id is shown as hex 0CF00400 as expected. By looking at the *Raw Data* window on the Senquip Portal device

page, or referring to the scripting guide, we note in Figure 3 that the JSON key for the CAN peripheral is can1. The key will be used to retrieve the data from the device raw data packet.

```
"can1": [
  {
    "data": "FFFFFF6813FFFFFF",
    "id": 217056256
  }
```

Figure 3 – CAN message in raw data

We will write a script to extract engine speed from the Engine Controller 1 message. We learned before that engine speed is in position 4 and 5 of the data field, 0x68 and 0x13 in this case. Taking the decimal form of the HEX value 0x1368 (Intel byte order), we get 4968 in decimal. To arrive at the RPM, we conduct a scaling of this value using the offset 0 and the scale 0.125 RPM/bit. The physical engine speed is found to be 621 RPM.

Once parsed and converted to km/h, the engine speed will be dispatched to the Senquip Portal for display and storage.

5. A Simple to Understand Method for Extracting SPNs

We start by including the required libraries and include a simple function that extracts the message PGN from the identifier.

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

// Extract the PGN from the CAN ID
function pgn(id)
{
  return((id >> 8) & 0x0003FFFF);
}
```

In the data handler, that runs at the end of each measurement cycle, we:

1) Convert the passed JSON measurements structure to an object.
2) Check that CAN data was captured.
3) Start a loop that runs through each CAN message identifier
4) Each identifier is checked against a string of if/else statements, checking for the PGN 61444.
5) If the PGN is found, we take the data, which is a string “FFFFFF6813FFFFFF”
6) Extract 4 characters starting at character 6, “6813”
7) Interpret them as a hex number in little endian format 0x1368.
8) Scale the number as per the SPN definition of 0.125 RPM per bit.
9) Dispatch the RPM to the Senquip Portal using the standard key ‘eng_speed’.
10) If there is no CAN data, dispatch a warning to the Senquip Portal.

```
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++) {

      // Look for the specific PGN:
      if (pgn(obj.can1[i].id) === 61444) {

        // Extract 4 Hex characters (2 bytes):
        let spn190 = SQ.parse(obj.can1[i].data, 6, 4, -16) * 0.125;

        // Convert to a signed 16-bit value and scale/offset the result:
        SQ.dispatch('eng_speed', spn190);
      }
    }
  } else {
    SQ.dispatch_event(1, SQ.WARNING, "No CAN data available");
  }
}, null);
```

Figure 4 - Script to parse the electronic engine controller 1 message

Note: Typically, when we create a new variable in a script, we would dispatch it to the Senquip Portal using a custom parameter (cp1….cp50) as the key. For values which are commonly used, such as typically measured J1939 parameters, Senquip has created standard keys. For a full list of available keys, see the Senquip scripting guide.

Engine RPM is now shown of the Senquip Portal.

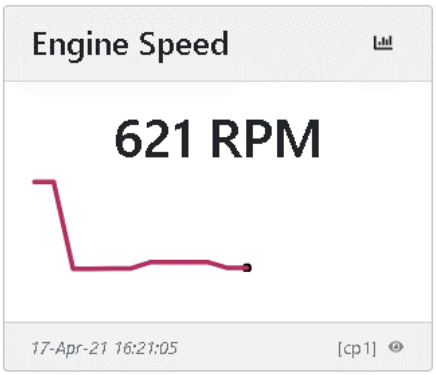

Figure 5 - RPM shown on Senquip Portal widget

This script considered the decode of just a single SPN. In reality, dozens of SPNs are likely to be decoded. Appendix 1 contains a version of the full script that has been extended to extract more SPNs from the captured CAN data.

6. A More Efficient Script Structure

While the scripting engine is lightweight, unnecessary processing within the data handler will raise CPU load, especially where many CAN messages are captured, and many PGNs are to be extracted.

This chapter introduces an alternate method for decoding the captured CAN messages that provides a significant performance advantage over the previously described algorithm that uses if/else handling.

6.1. Method 1 – Sequential if/else Processing

In the traditional approach, each incoming CAN frame is tested against a sequence of if and else if statements:

```
if (pgn(id) === 61444) { ... }
else if (pgn(id) === 65262) { ... }
else if (pgn(id) === 65263) { ... }
...
```

Each message must traverse the chain until a matching condition is found. If there are N defined PGN handlers and M messages received per second, the processor performs roughly M × N conditional checks. This makes the method O(M x N) in computational complexity per message.

As both the number of supported PGNs (N) and the number of incoming messages (M) increase, the CPU load grows proportionally.

6.2. Method 2 – Structured Dispatch Table

The alternative approach replaces the conditional chain with a lookup table that maps each PGN to a handler function:

```
let can_handler = {
  "61444": function(o,i){ ... },
  "65262": function(o,i){ ... },
  "65263": function(o,i){ ... },
};

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);
  if (obj.can1) {
    for (let i = 0; i < obj.can1.length; i++) {
      let id = JSON.stringify(pgn(obj.can1[i].id));
      if (isdef(can_handler[id])) {
        can_handler[id](obj, i);
      }
    }
  }
}, null);
```

Here, the PGN is computed once per frame, and the matching handler is called directly using an object. This single table lookup is 1 per message, meaning the total workload scales only with the number of messages resulting in O(M) total complexity.

6.3. Performance Comparison

Here the worst case scenarios are considered, where the CAN message being parsed is always the last one in the if/else chain. In reality, efficiency gains will be lower than stated below.

Number of CAN Messages (M)	PGN Handlers (N)	Estimated Operations = N×M	Estimated Operations = M	Relative Efficiency
1	10	10 comparisons	1 lookup	10× faster
10	10	100 comparisons	10 lookups	10× faster
40	10	400 comparisons	40 lookups	10× faster
40	40	1,600 comparisons	40 lookups	≈ 40× faster

Specifically looking at the example in the appendices, assuming:

- 40 CAN messages
- 10 PGNs handled
- Each message requires PGN extraction and dispatch

Method 1:

- Each message checks up to 10 if-else conditions
- Worst-case: 40 × 10 = **400 PGN comparisons**

Method 2:

- Each message does **1 map lookup**
- Total: 40 × 1 = **40 PGN lookups**

Efficiency Gain

Efficiency Ratio = Method 1 comparisons / Method 2 lookups = 480 / 40 = 10 times more efficient. It also improves readability, modularity, and scalability—making it the preferred approach for larger datasets.

Appendix 2 contains a script that decodes the same messages as that in Appendix 1 but does so using this more efficient algorithm.

6.4. Filters

The addition of filters can again improve script efficiency. The default behavior of the CAN peripheral with no filters in place, is to return one CAN message from each unique identifier. In the example above, we are parsing 10 PGNs but are receiving 40 CAN messages, each with different CAN identifiers.

We know the PGNs that we are looking for and likely know the CAN identifiers. By adding a filter list, we can reduce the number of received CAN messages from 40 to 10, a four times efficiency gain on top of the 10 times already achieved.

Number of CAN Messages (M)	PGN Handlers (N)	Estimated Operations = N×M (no filters | filters)	Estimated Operations = M (no filters / filters)	Relative Efficiency
1	10	10 | 10 comparisons	1 | 1 lookup	10× faster
10	10	100 | 100 comparisons	10 | 10 lookups	10× faster
40	**10**	**400 | 100 comparisons**	**40 | 10 lookups**	**40× faster**
40	40	1,600 | 1600 comparisons	40 | 40 lookups	≈ 40× faster

For more information on CAN filters, see the Senquip device User Guides.

7. Conclusions

Parsing CAN messages within a Senquip script provides a flexible and powerful way to extract key engine, transmission, and system parameters directly from a J1939 network. The examples in this application note have shown how individual SPNs can be decoded, scaled, and dispatched to the Senquip Portal for viewing and analysis.

Two scripting methods for decoding CAN data were examined. The traditional if/else approach is simple to understand and effective for a small number of PGNs, but its processing cost increases proportionally with both the number of captured messages and the number of PGNs handled. This O(M × N) growth can quickly consume unnecessary CPU resources as networks become more complex.

By contrast, the structured dispatch-table method performs a single lookup per message, reducing complexity to O(M). This makes it far more scalable and predictable as message rates and PGN counts increase. In real-world applications—such as a Senquip device decoding 40 CAN messages with a dozen active PGNs, the structured approach performs roughly 12 times fewer PGN-selection operations, translating to a massive gain in efficiency.

Further gains are achievable with filters, with a typical scenario offering an additional 4 times efficiency gain by filtering only for the messages required to be parsed.

Beyond efficiency, the structured method also improves code readability, modularity, and maintainability. Adding or modifying a handler involves changing a single entry in the table rather than extending a long conditional chain, reducing the chance of logic errors.

In summary, when building scripts that parse multiple CAN messages or handle several PGNs, Senquip recommends adopting the dispatch-table structure. It not only improves performance but also produces cleaner, more maintainable code that is easier to scale as applications grow.

APPENDIX 1 – Sequential if/else Method

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

function pgn(id) { return((id >> 8) & 0x0003FFFF); }

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++){
      let pgn_no = pgn(obj.can1[i].id);

      if (pgn_no === 61444){ //Electronic Engine Controller 1
        let spn190 = SQ.parse(obj.can1[i].data, 6, 4, -16)*0.125; // engine speed
        SQ.dispatch('eng_speed', spn190, 1);
      }
      else if (pgn_no === 65262){ //Engine Temperature 1
        let spn110 = SQ.parse(obj.can1[i].data, 0, 2, -16, SQ.S16)-40; // engine coolant temp
        SQ.dispatch('coolant_temp', spn110, 1);
        let spn175 = (SQ.parse(obj.can1[i].data, 4, 4, -16, SQ.S16)*0.03125)-273; // oil temp
        SQ.dispatch('oil_temp', spn175, 1);
      }
      else if (pgn_no === 65263){ //Engine Fluid Level/Pressure 1
        let spn100 = SQ.parse(obj.can1[i].data, 6, 2, -16)*4/6.89; // oil pressure
        SQ.dispatch('oil_pressure', spn100, 1);
        let spn98 = SQ.parse(obj.can1[i].data, 4, 2, -16)*0.4; // oil level
        SQ.dispatch('oil_level', spn98, 1);
        let spn109 = SQ.parse(obj.can1[i].data, 12, 2, -16)*2; // coolant pressure
        SQ.dispatch('coolant_pressure', spn109, 1);
        let spn111 = SQ.parse(obj.can1[i].data, 14, 2, -16)*0.4; // coolant level
        SQ.dispatch('coolant_level', spn111, 1);
      }

      else if (pgn_no === 61443){ //Electronic Engine Controller 2
        let spn92 = SQ.parse(obj.can1[i].data, 4, 2, -16);// engine load
        SQ.dispatch('eng_load', spn92, 1);
      }

      else if (pgn_no === 65253){ // Engine Hours, Revolutions
        let spn247 = SQ.parse(obj.can1[i].data, 0, 8, -16)*0.05; // total engine hours
        SQ.dispatch('eng_hrs', spn247, 1);
      }

      else if (pgn_no === 65257){ // Fuel Consumption
        let spn182 = SQ.parse(obj.can1[i].data, 0, 8, -16)*0.05; // trip fuel
        SQ.dispatch('trip_fuel', spn182, 1);
        let spn250 = SQ.parse(obj.can1[i].data, 8, 8, -16)*0.05; // total fuel used
        SQ.dispatch('total_fuel', spn250, 1);
      }

      else if (pgn_no === 65276){ //Dash Display
        let spn96 = SQ.parse(obj.can1[i].data, 2, 2, -16)*0.4; // Fuel Level
        SQ.dispatch('fuel_level', spn96, 1);
      }

      else if (pgn_no === 65244){ // Idle Operation
        let spn236 = SQ.parse(obj.can1[i].data, 0, 8, -16)*0.5; // Total Idle Fuel Used
        SQ.dispatch('idle_fuel', spn236, 1);
        let spn235 = SQ.parse(obj.can1[i].data, 8, 8, -16)*0.05; // Total Idle Hours
        SQ.dispatch('idle_hrs', spn235, 1);
      }

     else if (pgn_no === 65272){ // Transmission Fluids
        let spn124 = SQ.parse(obj.can1[i].data, 2, 2, -16)*0.4; // Transmission oil level
        SQ.dispatch('trans leve', spn124, 1);
        let spn127 = SQ.parse(obj.can1[i].data, 6, 2, -16)/16; // Transmission oil pressure
        SQ.dispatch('trans_pressure', spn127, 1);
```

```
        let spn177 = (SQ.parse(obj.can1[i].data, 8, 4, -16, SQ.S16)*0.03125)-273; // Transmission oil temp
        SQ.dispatch('trans_temp', spn177, 1);
       }

      else if (pgn_no === 65266){ // Fuel economy
        let spn183 = SQ.parse(obj.can1[i].data, 0, 4, -16)*0.05; // current l/h consumption
        SQ.dispatch('fuel_rate', spn183, 1);
        let spn184 = SQ.parse(obj.can1[i].data, 4, 4, -16)*0.001953125; //current km/l economy
        SQ.dispatch('fuel_economy', spn184, 1);
      }
     }
   }
}, null);
```

APPENDIX 2 – Structured Dispatch Table

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

function pgn(id) { return ((id >> 8) & 0x0003FFFF); }

// PGN handler map
let can_handler = {
  "61444": function(object, index) { // Electronic Engine Controller 1
    let spn190 = SQ.parse(object.can1[index].data, 6, 4, -16) * 0.125; // engine speed
    SQ.dispatch('eng_speed', spn190, 1);
  },
0
  "65262": function(object, index) { // Engine Temperature 1
    let spn110 = SQ.parse(object.can1[index].data, 0, 2, -16, SQ.S16) - 40; // engine coolant temp
    SQ.dispatch('coolant_temp', spn110, 1);
    let spn175 = (SQ.parse(object.can1[index].data, 4, 4, -16, SQ.S16) * 0.03125) - 273; // oil temp
    SQ.dispatch('oil_temp', spn175, 1);
  },

  "65263": function(object, index) { // Engine Fluid Level/Pressure 1
    let spn100 = SQ.parse(object.can1[index].data, 6, 2, -16) * 4 / 6.89; // oil pressure
    SQ.dispatch('oil_pressure', spn100, 1);
    let spn98 = SQ.parse(object.can1[index].data, 4, 2, -16) * 0.4; // oil level
    SQ.dispatch('oil_level', spn98, 1);
    let spn109 = SQ.parse(object.can1[index].data, 12, 2, -16) * 2; // coolant pressure
    SQ.dispatch('coolant_pressure', spn109, 1);
    let spn111 = SQ.parse(object.can1[index].data, 14, 2, -16) * 0.4; // coolant level
    SQ.dispatch('coolant_level', spn111, 1);
  },

  "61443": function(object, index) { // Electronic Engine Controller 2
    let spn92 = SQ.parse(object.can1[index].data, 4, 2, -16); // engine load
    SQ.dispatch('eng_load', spn92, 1);
  },

  "65253": function(object, index) { // Engine Hours, Revolutions
    let spn247 = SQ.parse(object.can1[index].data, 0, 8, -16) * 0.05; // total engine hours
    SQ.dispatch('eng_hrs', spn247, 1);
  },

  "65257": function(object, index) { // Fuel Consumption
    let spn182 = SQ.parse(object.can1[index].data, 0, 8, -16) * 0.05; // trip fuel
    SQ.dispatch('trip_fuel', spn182, 1);
    let spn250 = SQ.parse(object.can1[index].data, 8, 8, -16) * 0.05; // total fuel used
    SQ.dispatch('total_fuel', spn250, 1);
  },

  "65276": function(object, index) { // Dash Display
    let spn96 = SQ.parse(object.can1[index].data, 2, 2, -16) * 0.4; // Fuel Level
    SQ.dispatch('fuel_level', spn96, 1);
  },

  "65244": function(object, index) { // Idle Operation
    let spn236 = SQ.parse(object.can1[index].data, 0, 8, -16) * 0.5; // Total Idle Fuel Used
    SQ.dispatch('idle_fuel', spn236, 1);
    let spn235 = SQ.parse(object.can1[index].data, 8, 8, -16) * 0.05; // Total Idle Hours
    SQ.dispatch('idle_hrs', spn235, 1);
  },

  "65272": function(object, index) { // Transmission Fluids
    let spn124 = SQ.parse(object.can1[index].data, 2, 2, -16) * 0.4; // Transmission oil level
    SQ.dispatch('trans_leve', spn124, 1);
    let spn127 = SQ.parse(object.can1[index].data, 6, 2, -16) / 16; // Transmission oil pressure
    SQ.dispatch('trans_pressure', spn127, 1);
    let spn177 = (SQ.parse(object.can1[index].data, 8, 4, -16, SQ.S16) * 0.03125) - 273; // Transmission oil
temperature
    SQ.dispatch('trans_temp', spn177, 1);
```

```
  },

  "65266": function(object, index) { // Fuel economy
    let spn183 = SQ.parse(object.can1[index].data, 0, 4, -16) * 0.05; // current l/h consumption
    SQ.dispatch('fuel_rate', spn183, 1);
    let spn184 = SQ.parse(object.can1[index].data, 4, 4, -16) * 0.001953125; // current km/l economy
    SQ.dispatch('fuel_economy', spn184, 1);
  }
};

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++) {
      let pgn_id = JSON.stringify(pgn(obj.can1[i].id));
      if (isdef(can_handler[pgn_id])) {
        can_handler[pgn_id](obj, i);
      }
    }
  }
}, null);
```