IT 21 104
Examensarbete 15 hp
November 2021

# Design and Implementation of CANopen - SAE J1939 Communication Gateway

Peng Sun

UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet**
**UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

## Abstract

# Design and Implementation of CANopen - SAE J1939 Communication Gateway

*Peng Sun*

Since the introduction of CAN (Controller Area Network), it has become the de-facto standard communication bus for in-vehicle networks. Several higher layer protocols are developed based on CAN to fit various application domains. SAE J1939 is the standard for communication in heavy-duty vehicles. CANopen was initially designed for industrial automation, but quickly has been adopted in other industrial domains, including vehicles. Due to the various application areas of CANopen, many CANopen devices are available on the market. In order to integrate these off-the-shelf devices into a J1939 network, a communication gateway is required.

This thesis project briefly presents the CAN bus and two higher layer protocols – SAE J1939 and CANopen. As the existing proposals of CANopen – J1939 gateway device profiles do not fit the purpose of integrating arbitrary or unmodified CANopen devices into the J1939 network, we design a communication gateway to serve this purpose. The gateway software is prototyped on an STM32F107 development board using the C programming language. Several mapping table structures are proposed and implemented to bridge the two protocols. These mapping tables can be changed by the existing calibration tools over the CAN bus during the runtime. In this way, the gateway can be configured to support different connection setups of the connected devices.

Handledare: Dr. Meng Liu
Ämnesgranskare: Justin Pearson
Examinator: Johannes Borgström
IT 21 104
Tryckt av: Reprocentralen ITC

# Contents

# Abbreviations

| | |
|---|---|
| CAN | Controller Area Network |
| ISO | International Organization for Standardization |
| SAE | Society of Automotive Engineers |
| CiA | CAN in Automation |
| ECU | Electronic Control Unit |
| OSI model | Open Systems Interconnection model |
| PDU | Process Data Unit |
| CA | Controller Application |
| PGN | Parameter Group Number |
| SA | Source Address |
| PG | Parameter Group |
| SPN | Suspect Parameter Number |
| OD | Object Dictionary |
| COB | Communication OBject |
| EDS | Electronic Data Sheet |
| DCF | Device Configuration File |

# List of figures

# List of tables

# 1 Introduction

Controller Area Network (CAN) [1] is widely used in automotive industries and other industrial domains for communication between different nodes. Several higher layer protocols, such as SAE J1939 [2], CANopen [3], DeviceNet [4], and NMEA2000 [5], are developed based on CAN for various application areas.

SAE J1939 is mainly designed for heavy-duty vehicles and communication between electric vehicles and charging stations [6]. CANopen, on the other hand, is designed for industrial automation.

Both J1939 and CANopen communities are aware of the need for intercommunicating with each other. In the J1939 standard, two proprietary messages are reserved for supporting CANopen-like behaviors [7]. Several application notes are published in the draft version by the CANopen community to extend CANopen devices with a J1939-like interface [8] [9] [10]. However, both approaches require modification in the software of CANopen devices and J1939 nodes. To allow intercommunication between CANopen devices and J1939 nodes without changing their software, a communication gateway is required.

In this thesis project, a configurable communication gateway is designed and implemented in C. The gateway software is prototyped on an STM32F107 ARM development board. Configuration of the gateway is achieved by manipulating the data in the pre-defined CANopen – J1939 mapping tables via external calibration tools.

## 1.1 Purpose

This thesis project aims to design and implement a CANopen to J1939 communication gateway that enables communication between the J1939 and CANopen protocols. J1939 nodes can seamlessly communicate with CANopen devices in the same way they communicate with other J1939 nodes through the gateway and vice versa. Additionally, configuration methods are proposed to allow configuration changes without rebuilding and updating the gateway software.

## 1.2 Delimitations

This thesis project will focus on integrating CANopen devices with the gateway in a J1939 network. CANopen devices are typically sensors that transmit data to the J1939 network. This thesis project does not cover other use cases that might require a different gateway design.

## 1.3 Outline

In chapter 2, we introduce the basic concepts of CAN, as well as J1939 and CANopen. Most of the essential parts of both J1939 and CANopen protocols are covered. Based on these theories, the system and software architecture of the gateway is designed in chapter 3. A use-case diagram is also provided. Development environment, target hardware, and software components are introduced in chapter 4. Chapter 5 describes the test results for demonstrating the use of the gateway. Finally, in chapter 6, the thesis project is summarized and compared with related work.

# 2 Theory

## 2.1 Controller Area Network

Robert Bosch GmbH introduced Controller Area Network (CAN) in 1986 [11]. The intention was to enable more efficient, flexible, and robust message handlings between electronic control units (ECU) in vehicles. Before the introduction of CAN, communications of the in-vehicle network were mostly point-to-point [12], which increases the complexity of the wire harness when the number of in-vehicle ECUs grows. CAN, on the other hand, allow ECUs to communicate via a twisted-pair cable.

CAN standard was revised several times after its introduction. In 1991, Bosch published CAN 2.0, which later was adopted as an international standard - ISO 11898:1993 - Road vehicles — Controller area network (CAN) [1] , and it is still widely used today.

If we refer to the OSI reference model [13], the following layers are covered by the latest version of ISO 11898.

- Physical Layer – ISO 11898-2:2016 [14] and ISO 11898-2:2006 [15]
- Data Link Layer – ISO 11898-1:2015 [16]

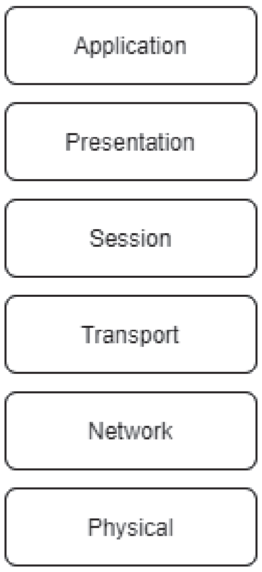


Figure 1: 7-Layer OSI reference model

On Physical Layer, high-speed CAN messages can be transferred at the highest 1Mbps with at least 30 nodes at a length of 40 meters with twisted shielded or unshielded wires, or 125Kbps at 500 meters. More nodes may be added if the electronic characteristics of the nodes do not reduce the signal quality [14].

On Data Link Layer, two different CAN message formats are defined – standard CAN (aka CAN 2.0A) and extended CAN (aka CAN 2.0B).

The upper layers are defined in other higher layer protocols based on CAN for different application areas, such as SAE-J1939, CANopen.

### 2.1.1 Standard CAN message format

Standard CAN message format has an 11-bit message ID, and the message frame is formatted according to the following table:

Table 1: Standard CAN frame

| | SOF | Message Identifier | RTR | IDE | r0 | DLC | Data Field | CRC | ACK | EOF |
|---|---|---|---|---|---|---|---|---|---|---|
| Bits | 1 | 11 | 1 | 1 | 1 | 4 | 0-64 | 16 | 2 | 7 |

The following list describes the detailed bit fields of the standard CAN frame.

- **SOF** - Start of the frame.
- **Message identifier** - ID of the message. Unlike other communication protocols, message identifier is also used for arbitrating CAN messages, which will be explained in 2.1.3.
- **RTR** - Remote Transmission Request. If RTR bit is dominant (0), it indicates a Remote frame required by other nodes. However, the remote frame is not recommended due to several issues, such as message collision found after the introduction of CAN. Details can be found in the application note issued by CAN in Automation – CAN remote frames – Avoiding of usage [17]. Requesting CAN messages is recommended to be implemented on the application level.
- **r0** - Reserved for future usage.
- **IDE** - Identifier Extension. If it is dominant (0), it is a standard message. If recessive (1), an extended message is being transmitted.

- **DLC** - Data Length Code, telling the length of the data being transmitted.

- **Data field** carries the actual data transmitted on CAN. There is, however, no specification on how data, which is longer than 8 bytes, is transferred in ISO 11898. It is up to the higher layer protocols to define how data over 8 bytes should be transferred.

- **CRC** - Checksum field for validating the message.

- **ACK** - Acknowledgement with a 1-bit delimiter. The sending nodes start to send a recessive bit (1). When the receiving nodes correctly receive the message, the receiving nodes will answer with a dominant bit (0). If the sending nodes do not receive the dominant bit (0), this message is considered not received.

- **EOF** - End of the frame.

### 2.1.2 Extended CAN message format

Extended CAN format has a 29-bit message ID, and the message frame is formatted according to the following table:

Table 2: Extended CAN format

| | SOF | Message Identifier | SRR | IDE | Message Identifier | RTR | r0 | r1 | DLC | Data Field | CRC | ACK | EOF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bits | 1 | 11 | 1 | 1 | 18 | 1 | 1 | 1 | 4 | 0-64 | 16 | 2 | 7 |

Apart from standard CAN, extended CAN also includes the following fields:

- **SRR** - Substitute Remote Request. This bit is always recessive and is used for message arbitration when arbitrating with a standard message (arbitration mechanism is explained in 2.1.3) when the first 11 bits of message identifiers are identical for a standard message and an extended message.

- **r1** - Another bit reserved for future usage.

### 2.1.3 Message arbitration

Priority of transmitting a CAN message on CAN bus is determined by the message identifiers – the smaller ID, the higher priority. When the CAN bus is Idle, all nodes on the bus can start to transmit messages simultaneously. A

CAN node can be both transmitter and receiver. The arbitration mechanism is as follows:

> If the bus is idle, start to transmit SOF, then the message ID. A node will keep transmitting messages until:
>
> - When the bit being transmitted on the bus is dominant (0) and a node is about to transmit a recessive bit (1). This node will stop transmitting the message and start to receive the message on the bus until the bus is idle. Then it will make a new try to resend the message.
>
> - When the bit being transmitted on the bus is recessive (1) and a node is about to transmit a dominant bit (0), this node wins the arbitration, and all other nodes transmitting recessive bit (1) stop transmitting messages on the bus.

Here is an example:

> Node A is transmitting a message with ID 0010. Node B is transmitting a message with ID 0011. Node C is transmitting a message with ID 1000. Please note that a standard CAN ID is 11 bits. So, there are seven zeros in front of each message IDs.

Table 3: Example of CAN message arbitration

| Bus | Node A | Node B | Node C | |
|---|---|---|---|---|
| 0…0 | 0…0 | 0…0 | 0…0 | All nodes keep transmitting on the bus. |
| 0 | 0 | 0 | 1 | Node C stops transmitting the message. |
| 0 | 0 | 0 | Rx | |
| 1 | 1 | 1 | | |
| 0 | 0 | 1 | | Node B stops transmitting the message. |
| … | … | Rx | | Node A wins the arbitration, continue to transmit messages on the bus. |

## 2.2 SAE J1939

The SAE J1939 standard [2] was initially introduced for heavy-vehicle applications. In passenger cars, most in-vehicle communications use proprietary CAN messages, meaning that the manufacturers define the contents and the interpretation of the CAN messages. However, for commercial vehicles, a common agreement on in-vehicle communication is more appropriate for, e.g., data loggers for fleet management or communication between trucks and trailers. Therefore, J1939 was created and became the de-facto branch standard.

J1939 is based on extended CAN (CAN 2.0B). The reason for choosing extended CAN message format is to adapt the concepts in SAE J1587 [18] – a previous communication standard for heavy-duty vehicles based on serial communications – SAE J1708 [19]. Adaptation of the old communication standard is achieved by assigning functions to parts of the message ID, such as differentiating the ways of interpreting the message content or identifying the source addresses of the messages. Apart from the basis of CAN, J1939 supports peer-to-peer communication. It also supports transmitting data frames that are longer than 8 bytes.

### 2.2.1 SAE J1939 message format

The J1939 Message format is based on extended CAN message format (see 2.1.2)

The 29-bits message ID, together with the data field (up to 8 bytes), are defined as PDU (Protocol Data Unit) in SAE J1939/21 [20], which is the core of J1939.

The message ID is further defined in SAE J1939/21 [20]:

Table 4: SAE J1939 PDU

| Message identifier (29 bits) | | | | Data field |
|---|---|---|---|---|
| **PDU fields** | **P** | **PGN** | **SA** | 0-8 bytes |
| Bits | 3 | 18 | 8 | |

The following list describes the detailed bit fields of the J1939 message identifier:

- **P** - Priority. The priority is ranged from 000 to 111. 000 is defined as the highest priority.

- **SA** - Source Address. In total, it is allowed to have up to 253 addresses of controller applications (CAs). The SA is unique in the network, meaning that no ECUs may share the same SA. Note that one ECU can include several CAs, thus several SAs.

- **PGN** - Parameter Group Number. It points to parameter tables defined in SAE J1939/71 [7] for interpreting data in the data field.

### 2.2.2 Parameter group number

Parameter Group Number (PGN) in Table 4 contains the following fields:

Table 5: SAE J1939 Parameter group number [20]

| PGN (18 bits) | | | | |
|---|---|---|---|---|
| **PGN fields** | **EDP** | **DP** | **PF** | **PS** |
| Bits | 1 | 1 | 8 | 8 |

The following list describes the detailed bit fields of the J1939 PGN fields:

- **EDP** - Extended Data Pager, reserved for future extensions. All current J1939 messages have this bit set to 0.
- **DP** - Data Page, used for creating another page of PDU.
- **PF** - PDU Format. PF values 0 to 239 are defined as PDU1 (peer-to-peer communications), while PF values 240 to 255 are defined as PDU2 (broadcast communications).
- **PS** - PDU Specific. It contains DA (Destination Address) if it is PDU1 or Group Extension if it is PDU2.

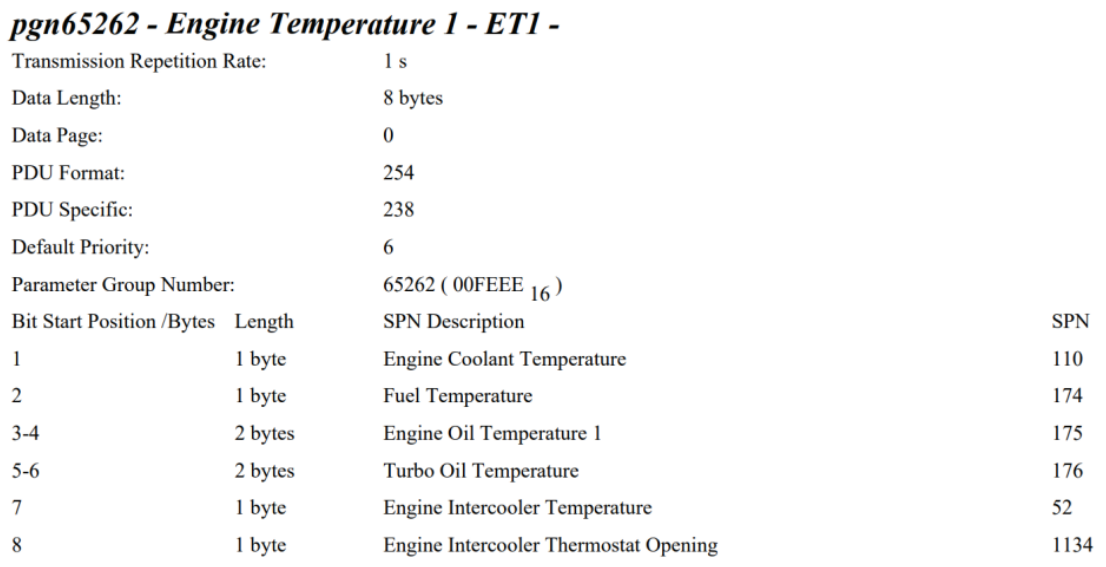
The total number of PGNs is calculated as:

(240 + 16*256)*2 = 8672, where 240 is the total number of PDU1, 16 is the total number of PDU2, 256 is the total number of Group Extension of PDU2, 2 is Data Pager (either 0 or 1).

Many PGNs are pre-defined in SAE J1939/71 [7]. There are also PGNs that are reserved for manufacturer-specific usage, allowing manufacturers to define proprietary messages.

In each Parameter Group (PG), the period time and default priority of the CAN message is provided, as well as how to interpret the 8-bytes data frame, which is detailed in SPNs (Suspect Parameter Number, see 2.2.3).

Taking an example from SAE J1939/71– PGN65262: Engine Temperature 1:


***pgn65262 - Engine Temperature 1 - ET1 -***

| | |
|---|---|
| Transmission Repetition Rate: | 1 s |
| Data Length: | 8 bytes |
| Data Page: | 0 |
| PDU Format: | 254 |
| PDU Specific: | 238 |
| Default Priority: | 6 |
| Parameter Group Number: | 65262 ( 00FEEE $_{16}$ ) |

| Bit Start Position /Bytes | Length | SPN Description | SPN |
|---|---|---|---|
| 1 | 1 byte | Engine Coolant Temperature | 110 |
| 2 | 1 byte | Fuel Temperature | 174 |
| 3-4 | 2 bytes | Engine Oil Temperature 1 | 175 |
| 5-6 | 2 bytes | Turbo Oil Temperature | 176 |
| 7 | 1 byte | Engine Intercooler Temperature | 52 |
| 8 | 1 byte | Engine Intercooler Thermostat Opening | 1134 |

Figure 2: PGN65262 - Engine Temperature 1 - ET1 - [7]

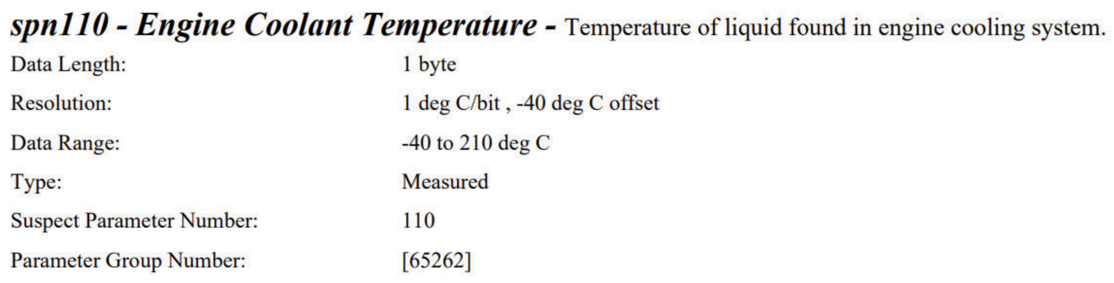
PDU format is 254, meaning that this message is a broadcast message. In the second half of PGN65262, the bit start position and length of SPN belonging to this PG are defined.

### 2.2.3 Suspect parameter number

SPNs are assigned to parameter groups, allowing the application to convert from raw value to physical value.

Taking SPN110 in Figure 2, for example, when a node receives a message with PGN65262, it knows the first byte in the data frame represents SPN110 - the engine coolant temperature by the definition of PG.

SPN110 is further defined in SAE J1939/71 [7]:


***spn110 - Engine Coolant Temperature -*** Temperature of liquid found in engine cooling system.

| | |
|---|---|
| Data Length: | 1 byte |
| Resolution: | 1 deg C/bit , -40 deg C offset |
| Data Range: | -40 to 210 deg C |
| Type: | Measured |
| Suspect Parameter Number: | 110 |
| Parameter Group Number: | [65262] |

Figure 3: SPN110 - Engine Coolant Temperature [7]

Assuming the first byte of the data frame of PGN65262 is 34, we can then calculate that the engine coolant temperature is -6 degrees Celsius by using the formula – raw value*resolution+offset.

### 2.2.4 Diagnostics

In J1939, diagnostics are reported by a series of pre-defined diagnostic messages, of which DM1 is the most important one. It reports all active Diagnostics Troubleshooting Codes (DTCs). A DTC contains the malfunctioning SPN, Failure Mode Indicator (FMI), Occurrence Counter (OC), and SPN Conversion Method (CM).

For example, a DTC containing SPN110, FMI4, OC2, CM4 can be interpreted as:

> The engine temperature sensor (SPN110) is malfunctioning. The reason for malfunctioning is that the sensor's voltage is below normal (FMI4). It occurred twice (OC2). The SPN is transmitted according to conversion method 4 (CM4) – little endianness.

All diagnostic-related standards are specified in SAE J1939/73 – Application Layer – Diagnostics [21].

### 2.2.5 Network management

Network management supported by J1939 [22], apart from the network management in CANopen (described in 2.3.3), is not for waking up or resetting nodes. It is mainly designed for claiming addresses. Similar to the plug-n-play concept in computers, J1939 allows a node to dynamically claim its CA (Controller Application, see 2.2.1) address. Typically, CA addresses are pre-defined in each node. However, dynamical CA addresses may be applicable when connecting extra ECUs, such as ECUs in third-party trailers, external diagnostic tools into the network, etc. If a CA address claiming conflict exists, a CA with higher priority wins.

It is worth noticing that network management is not mandatory if there is no need for ECU extension.

### 2.2.6 Transport protocol

All data transportation introduced so far is based on CAN2.0B, which allows transferring a maximum of 8 bytes' data in a CAN message. TP (Transport Protocol) is designed for transporting more than 8 bytes' data. There are two types of transport protocols – Broadcast Announcement Messages (BAM) for broadcasting messages to the entire network and Connection Mode Messages (CM) for peer-to-peer communications.

Both BAM and CM are initialized by a Transport Protocol Connection Management message (TP.CM), telling the receiver how many bytes of data the sender will send. Then it is followed by the rest of the data. Below is an illustration of how BAM works. For CM, it requires the responder node to send an acknowledgment message for each received package.

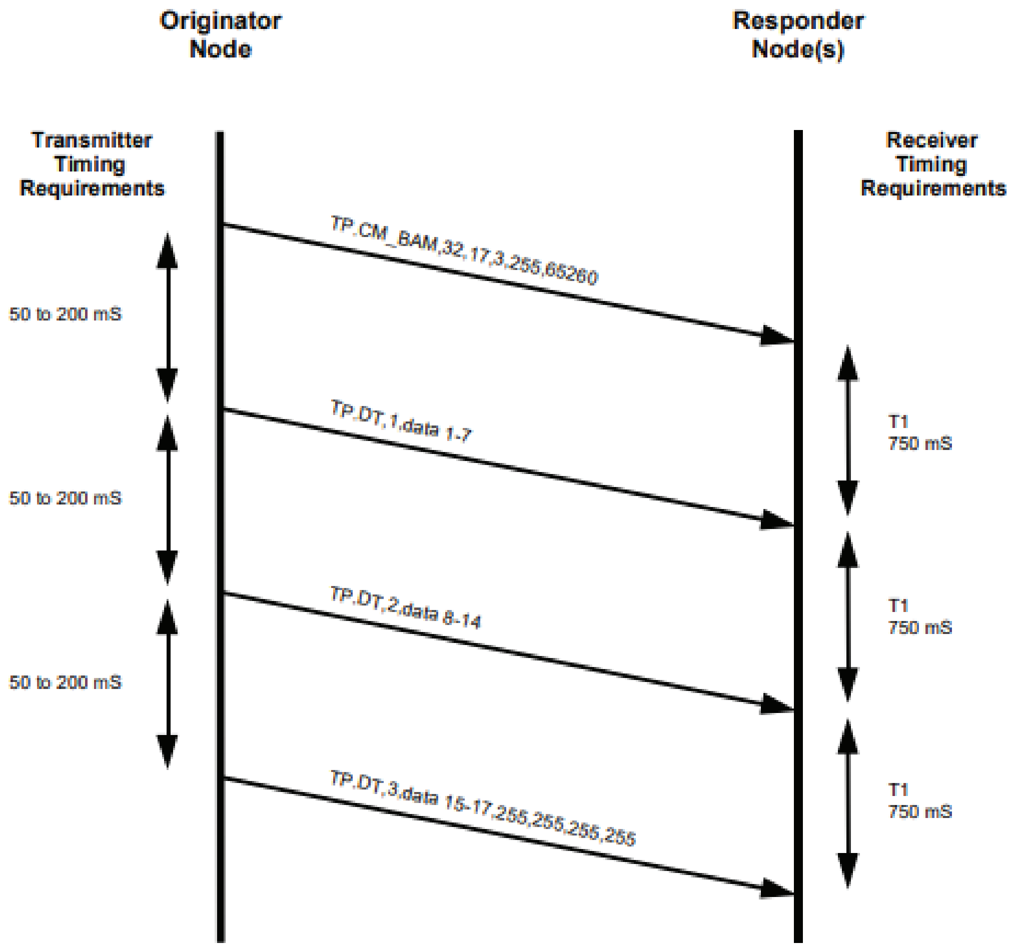


Figure 4: Broadcast data transfer [20]

## 2.3 CANopen

The CANopen standard was released by CAN in Automation (CiA) [3] and initially designed for industrial automation, such as stepper motor control in robot arms. It was later applied in many application areas such as medical devices, lift control, building door control, laboratory automation, municipal vehicles, etc. Due to the large number of application areas, CANopen is more complex than J1939.

### 2.3.1 Communication models

There are three communication models in CANopen to fit different scenarios [3]:

- Master/Slave communication model: This model is typically used when a master node controls other nodes in the network, such as waking up the network or requesting diagnostic data from slave nodes.

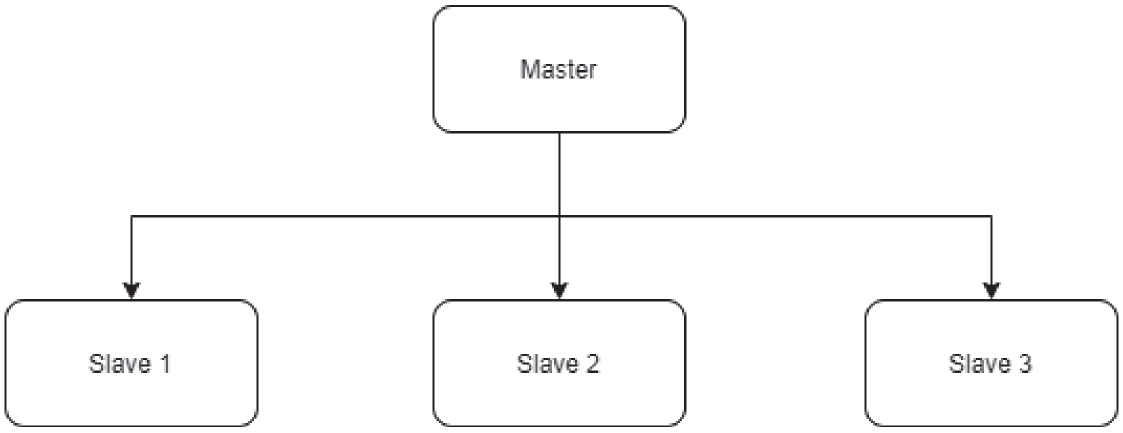
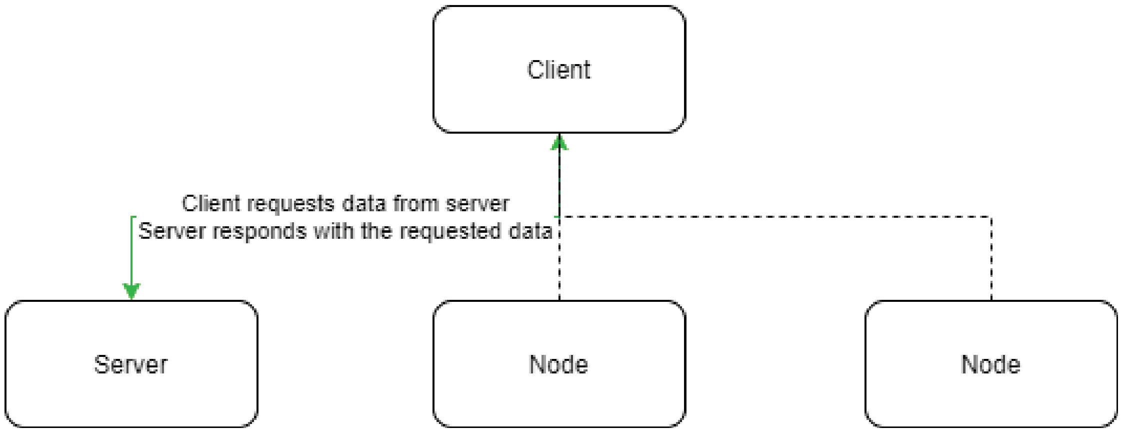


Figure 5: Master/Slave communication model

- Server/Client communication model: When a client requests data from a server, the server answers with the requested data. E.g., a host controller as a client requests temperature data from a server.


Client
Client requests data from server
Server responds with the requested data
Server
Node
Node
Figure 6: Server/Client communication model

- Producer/Consumer communication model: This model reminds the J1939 protocol. A producer can broadcast the data to consumers or send it when a consumer notifies that it is time to provide it.

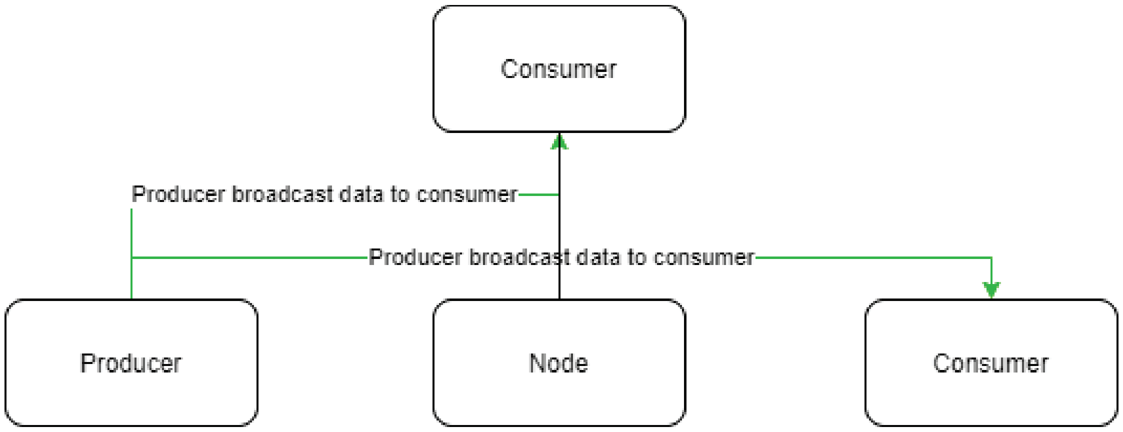

Consumer
Producer broadcast data to consumer
Producer broadcast data to consumer
Producer
Node
Consumer

Figure 7: Producer/Consumer communication model

It is worth noticing that there is no fixed role in all communication models. Depending on the network setup and purposes for information exchange, a node can be both a server and a client.

### 2.3.2 Object dictionary

Object Dictionary (OD) is the essential part of CANopen. An object dictionary is a lookup table containing the profiles of the device, including objects such as functionality, application data, and configurations. OD is indexed with a 16-bit index and an 8-bit subindex. CiA defines many device profiles to fit different application areas. An object with the same index in two device profiles may represent other functionalities

The objects in OD are mainly grouped as follows:

Table 6: Object groups in OD [23]

| Index (hex) | Object group |
|---|---|
| 0000$_h$ | Reserved |
| 0001$_h$ – 009F$_h$ | Static and complex data types |
| 00A0$_h$ – 0FFF$_h$ | Reserved |
| 1000$_h$ – 1FFF$_h$ | Communication profiles (e.g., DS 301, DS 302) |
| 2000$_h$ – 5FFF$_h$ | Manufacturer-specific device profiles |
| 6000$_h$ – 9FFF$_h$ | Standardized device profiles |
| A000$_h$ – FFFF$_h$ | reserved |

Here is an example of how part of the object dictionary of the Danfoss DST T92C Temperature transmitter looks like:

Table 7: Example of object dictionary [24]

| Index (HEX) | Sub Index | Name | Type | Access | Default | Comment |
|---|---|---|---|---|---|---|
| 1005 | 00 | COB-ID SYNC | Unsigned32 | rw | 0x80 | |
| 1008 | 00 | Manufacturer Device Name | String | ro | DST T92C CANopen | |
| 1009 | 00 | Manufacturer Hardware Version | String | ro | "x.xxrx" | |
| 100A | 00 | Manufacturer Software Version | String | ro | "x.xxrx" | |

Table 8: Example of object dictionary [24]

| Index (HEX) | Sub Index | Name | Type | Access | Default | Comment |
|---|---|---|---|---|---|---|
| 6110 | | Ai_Sensor_Type | | | | |
| | 00 | Number of entries | Unsigned8 | ro | 1 | |
| | 01 | Ai_Sensor_Type_1 | Unsigned16 | ro | 100 | 100 = temperature sensor |
| 6124 | | Ai_Input_Offset | | | | |
| | 00 | Number of entries | Unsigned8 | ro | 1 | |
| | 01 | Ai_Input_Offset_1 | Float32 | rw | 0 | Temperature offset; will be added to the current temperature value; Min./max. value=min./max. temperature of DST T92C |
| 6125 | | Ai_Input_Autozero | | | | |
| | 00 | Number of entries | Unsigned8 | ro | 1 | |
| | 01 | Ai_Input_Autozero_1 | Unsigned32 | wo | | Autozero for temperature 0x6F72657A (ASCII: "zero") |
| 6130 | | Ai_Input_PV | | | | |
| | 00 | Number of entries | Unsigned8 | ro | 1 | |
| | 01 | Ai_Input_PV_1 | Float32 | ro | | actual temperature value |

Each entry in the lookup table contains the index, subindex, data type, access, and the default value of the objects.

If two or more devices in the CANopen network use the same OD in Table 7 and Table 8, the following operations can be performed:

- Requesting data from the indexed object 0x100A with subindex 0x00, we can get the software version in this device.
- Writing data to the indexed object 0x6124 with subindex 0x00, we can configure the temperature offset.
- Requesting data from the indexed object 0x6130 with subindex 0x01, we can get the actual temperature value from the sensor.

Some objects in the OD are pre-defined in the standard. Furthermore, there are also many manufacturer-defined objects.

In CiA 306 [25] [26] and CiA311 [27], a human-readable data exchange file – EDS (Electronic Data Sheet) is defined in ini-file format or XML format. The EDS contains the definition of the OD in a device. It can be customized to a DCF (Device Configuration File) by adding configured values of the objects. DCF allows the host controller or configurator to communicate with or configure a device.

All the communications between devices are conducted through communication objects. The following figure shows how a device implementing CANopen protocol handles the communication with other devices via CAN:

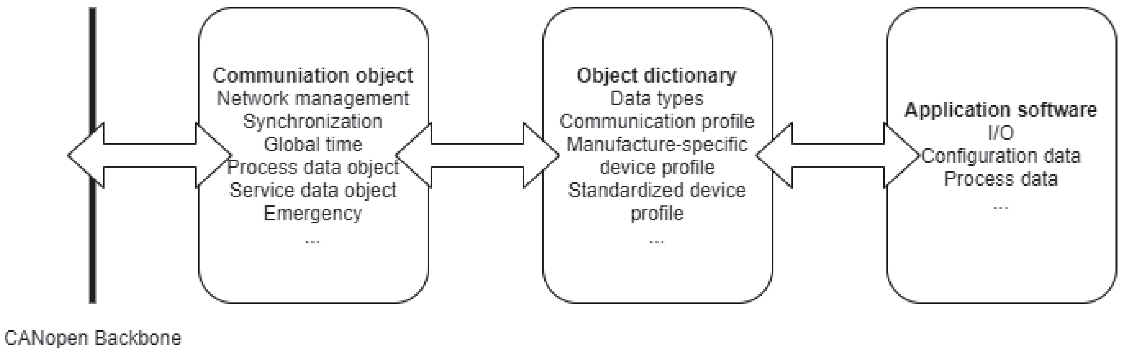


Figure 8: Overview of CANopen implementation in a device [28]

## 2.3.3 Communication object

CANopen is a higher layer protocol based on CAN2.0A. The 11-bit CAN ID is also known as COB-ID (Communication Object Identifier), which is defined as follows:

Table 9: Communication object ID format

| COB-ID (11 bits standard CAN Id ) | | |
|---|---|---|
| **COB id fields** | **Function Code** | **Node Id** |
| Bits | 4 | 7 |

- **Function Code –** It defines the different types of communication objects.

- **Node Id** - A unique identifier of the node to which the function code is connected.

The following table is an overview of all types of communication objects defined in CiA301 [3].

Table 10: Overview of communication objects in CANopen

| Communica-tion Object | Function Code (bin) | Node ID (dec) | COB-ID (hex) | OD Index for configuration |
|---|---|---|---|---|
| NMT | 0000 | 0* | 0 | - |
| SYNC | 0001 | 0 | 0x80 | 0x1005 |
| EMCY | 0001 | 1-127 | 0x81-0xFF | 0x1014 |
| TIME | 0010 | 0 | 0x100 | 0x1012 |
| TPDO1 | 0011 | 1-127 | 0x181-0x1FF | 0x1800 |
| RPDO1 | 0100 | 1-127 | 0x201-0x27F | 0x1400 |
| TPDO2 | 0101 | 1-127 | 0x281-0x2FF | 0x1801 |
| RPDO2 | 0110 | 1-127 | 0x301-0x37F | 0x1401 |
| TPDO3 | 0111 | 1-127 | 0x381-0x3FF | 0x1802 |
| RPDO3 | 1000 | 1-127 | 0x401-0x47F | 0x1402 |
| TPDO4 | 1001 | 1-127 | 0x481-0x4FF | 0x1803 |
| RPDO4 | 1010 | 1-127 | 0x501-0x57F | 0x1403 |
| TSDO | 1011 | 1-127 | 0x581-0x5FF | - |
| RSDO | 1100 | 1-127 | 0x601-0x67F | - |
| HEARTBEAT | 1110 | 1-127 | 0x701-0x77F | - |

*A communication object with node ID 0 indicates it is a broadcast object. Otherwise, it is a peer-to-peer object.

- **NMT** – Network management. This communication object is used in the Master/Slave communication model, allowing a master device to change the state of slave devices, such as pre-operational, operational, stopped.

- **SYNC** – Synchronization. This communication object is also used in the Master/Slave communication model. A slave device can be configured to respond to the SYNC object from the master device.

- **EMCY** – Emergency. It is a typical diagnostic object. When a device experiences failures, it may send an EMCY object to report the failure.

- **TIME** – Timestamps. This communication object allows synchronization of global time over the network.

- **TPDO1 ~ TPDO3** – Transmit process data object 1~ 3. This type of object is similar to J1939, allowing a node to broadcast messages and receive broadcast messages on the CAN bus in real-time, either periodically or on request. The data to be broadcasted is the defined objects in the object dictionary. The objects included in TPDOx can be either pre-defined or configured. There are 4 TPDO messages for one device.

- **RPDO1 ~ RPDO3** – Receive process data object 1 ~ 3.
- **TSDO** - Transmit service data object. IT is typically used when a device wants to change or access objects defined in the object dictionary.
- **RSDO** – Receive service data object.
- **HEARTBEAT** – A node can broadcast its state in order to be able to perform liveness-check over the network.

Since PDO (Process Data Object) and SDO (Service Data Object) are the essential parts in CANopen, we will explain them in the following subchapters.

### 2.3.4 Process data object

The PDO service is designed for transmitting and receiving data over the network in real-time via TPDO and RPDO messages. The objects transmitted in a TPDO message could be pre-defined or configured by changing the corresponding data entries in the object dictionary. Taking Danfoss DST T92C Temperature transmitter [24] as an example, one pre-defined TPDO message exits. The configuration of the TPDO can be checked via the indexed objects 0x1800 and 0x1A00, which are object entries related to the TPDO1 configuration.

Table 11: Example of TPDO1 configuration [24]

| **Index (HEX)** | **Sub Index** | **Name** | **Type** | **Access** | **Default** | **Comment** |
|---|---|---|---|---|---|---|
| 1800 | | Transmit PD01 parameter | | | | |
| | 00 | Number of entries | Unsigned8 | ro | 5 | |
| | 01 | COB ID used by PDO | Unsigned32 | rw | 0x181 | (0x00000180 + Node-ID) |
| | 02 | Transmission type | Unsigned8 | rw | 0x01 | Only 0x01 (sync) or 0xFF (async) with delta and/or event timer |
| | 03 | Inhibit time | Unsigned16 | rw | 0 | |
| | 04 | Reversed | Unsigned8 | rw | 0 | |
| | 05 | Event timer | Unsigned16 | rw | 1000 | 1000:default value of DS404 |
| 1A00 | | Transmit PDO1 mapping | | | | |
| | 00 | Number of entries | Unsigned8 | rw | 2 | |
| | 01 | PDO mapping for the 1. application object to be mapped | Unsigned32 | rw | 0x91300120 | Temperature as int32: 0x91300120 Temperature as float32: 0x61300120 Status temperature as uint8: 0x61500108 Meaning of status bits (if set): Bit 0: temperature value invalid Bit 1: positive overload Bit 2: negative overload |
| | 02 | PDO mapping for the 2. application object to be mapped | Unsigned32 | rw | 0x61500108 | |
| | 03 | PDO mapping for the 3. application object to be mapped | Unsigned32 | rw | 0 | |
| | 04 | PDO mapping for the 4. application object to be mapped | Unsigned32 | rw | 0 | |

Table 11 shows that TPDO1 is a pre-defined TPDO message (has default values in the entries). The data field in the message carries two objects – object index 0x9130, subindex 0x01, and object index 0x6150, subindex 0x01, which are temperature value (int32) and status temperature (int8). The whole TPDO1 message is illustrated in the following table:

Table 12: Example of TPDO1 message format [24]

| | **ID** | **DLC** | **Byte 0** | | | **Data** | | **Byte 7** |
|---|---|---|---|---|---|---|---|---|
| TPDO | 0x180 + Node ID | 5 | 0xA0 (temp. LSB) | 0x86 | 0x01 | 0x00 (temp. MSB) | 0x00 (temp. status) | not available |

The configuration of TPDO1 can be changed by TPDO mapping – writing 1 to the object index 0x1A00, subindex 01. Then this message will only carry one object. If we then write 0x61300120 to the object index 0x1A00, sub index 01, this message will only contain 4 bytes data - measured temperature (float32).

It is also possible to change other properties of the TPDO1 message, such as the period time, transmission type, and timeout, by changing the data in the indexed object 0x1800.

PDO messages can be either synchronous or asynchronous. Synchronous PDOs will only transmit messages after receiving one or x times SYNC message from the master device. Asynchronous PDOs will transmit the messages triggered by internal events. In both modes, a message can be either cyclic or acyclic.

In practice, PDO is mainly used for sending and receiving real-time data (similar to J1939). If a device would like to read data from an arbitrary object or write data to an object in another device, SDO comes in handy.

## 2.3.5 Service data object

SDO provides a universal way of accessing and changing objects defined in a device's object dictionary. The following is an example of how to request pressure value (index 0x6130, subindex 0x01) from Danfoss T92C Temperature transmitter:

Table 13: Example of SDO request and acknowledgment message format. [24]

| | **ID** | **DLC** | **Data** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | **Byte 0** | | | | | | | **Byte 7** |
| Command | 0x600 + Node ID | 8 | SDO Re-quest 0x40 | Index LSB 0x30 | Index MSB 0x61 | Sub Index 0x01 | Not used | | | |
| Answer | 0x580 + Node ID | 8 | SDO Ack. 0x43 | Index LSB 0x30 | Index MSB 0x61 | Sub Index 0x01 | Data LSB | Data | Data | Data MSB |

Compared with PDO, the SDO message has 4 bytes overhead to specify the SDO type and object index, which will increase the busload. Therefore, PDO is more suitable for handling real-time data. However, PDO is limited to transferring a maximum of 8 bytes' data. If we would like to transfer data larger than 8 bytes, SDO is the only option. (Similar to the J1939 transport protocol (TP) described in 2.2.6)

Here is how writing data to an object looks like:

Table 14: Example of SDO write and acknowledgment message format [24]

| | ID | DLC | Data | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Byte 0 | | | | | Byte 7 |
| Command | 0x600 + Node ID | 8 | SDO write 0x2F | Index LSB 0x00 | Index MSB 0x18 | Sub Index 0x02 | Trans-mission Type 0xFF | Not used |
| Answer | 0x580 + Node ID | 8 | SDO Ack. 0x60 | Index LSB 0x00 | Index MSB 0x18 | Sub Index 0x02 | Not used | |

In the above example, the transmission type of TPDO1 (0x1800, subindex 0x02) is configured from default 0x01 (sync) to 0xff (async).

# 3 Design

Our application scenario is to integrate devices (typically sensors) communicating in CANopen protocol into an SAE J1939 vehicle network. Therefore, we need to design a communication gateway which:

- works as a CANopen master device to perform the network management with the connected slave devices, receive data and diagnostic information, and request data when needed.
- works as a J1939 node in the vehicle network so that the data and diagnostic information from the CANopen slave devices can be transmitted to other J1939 nodes in the vehicle.

## 3.1 System architecture

The overall system architecture for the gateway is:

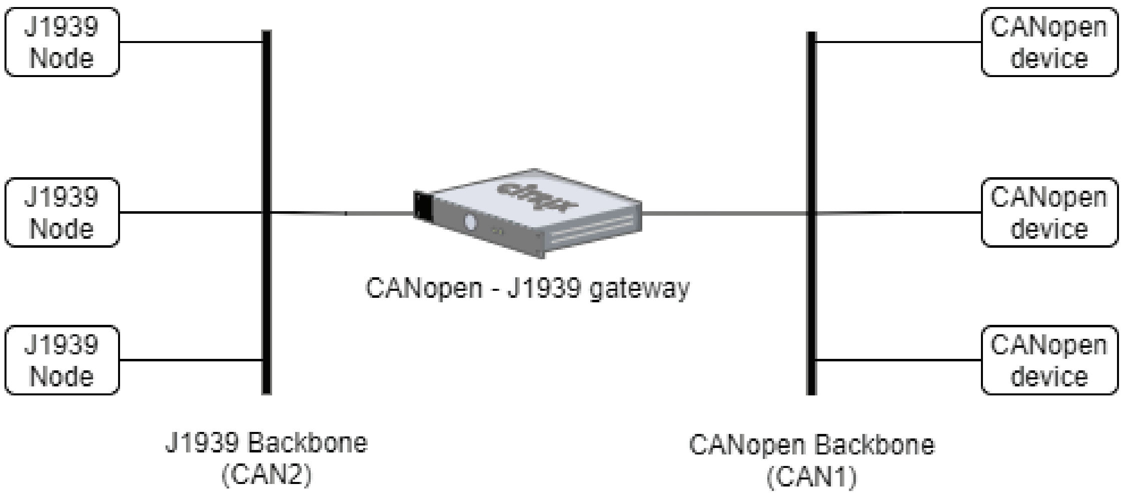


Figure 9: Overview of CANopen - J1939 system architecture

CANopen device can be connected directly into the J1939 network since CAN 2.0A and CAN 2.0B can coexist on the same physical communication bus [2]. However, it is inefficient – e.g., messages such as NMT and SYNC in CANopen have a smaller ID. Thus, it will always win the arbitration against J1939 messages. A communication bus overloaded with these CANopen messages will then cause a considerable delay of the J1939 messages. Therefore, it is desirable to add two separate CAN channels in the gateway to avoid overriding situations.

## 3.2 Software architecture

The Software architecture of the gateway is illustrated in Figure 10:

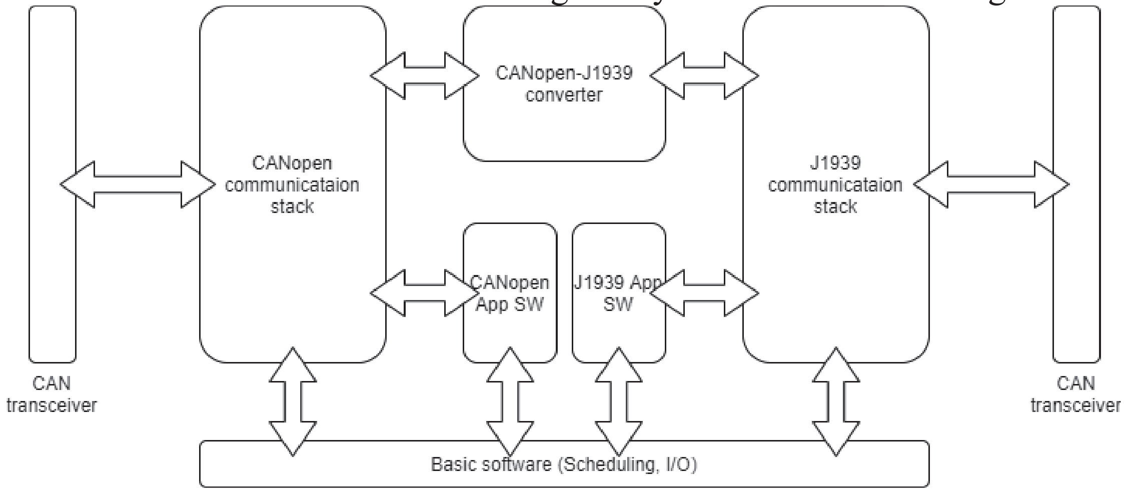

CANopen-J1939 converter
CANopen communicataion stack
J1939 communicataion stack
CANopen App SW
J1939 App SW
CAN transceiver
CAN transceiver
Basic software (Scheduling, I/O)

Figure 10: Software architecture of CANopen - J1939 gateway

The components in the software architecture are described as follows:

- Basic Software provides scheduling service, memory management, I/O, power management, etc.
- CANopen & J1939 communication stacks implement all functions according to the protocol standards. Various commercial and open-source software packages are available for the CANopen & J1939 communication stacks implementation, such as CANopen Stack Project [29], Open source J1939 stack [30], and SAE J1939 protocol stack from STMicroelectronics [31].
- CANopen – J1939 converter connects the SPNs in J1939 to the objects in the CANopen object dictionary. It also performs the data transformation, such as transforming from *Celsius* to *Kelvin,* when needed. It also converts the diagnostic information in CANopen EMCY into the proper format in the J1939 DM1 message.
- CANopen application software handles the rest of the CANopen communications, such as initializing slave devices, sending SYNC messages.
- J1939 application software handles the rest of J1939 communications, such as handling requested PGN messages and response to network management messages.

## 3.3 Use cases

The use cases of CANopen – J1939 gateway is illustrated in the figure below:

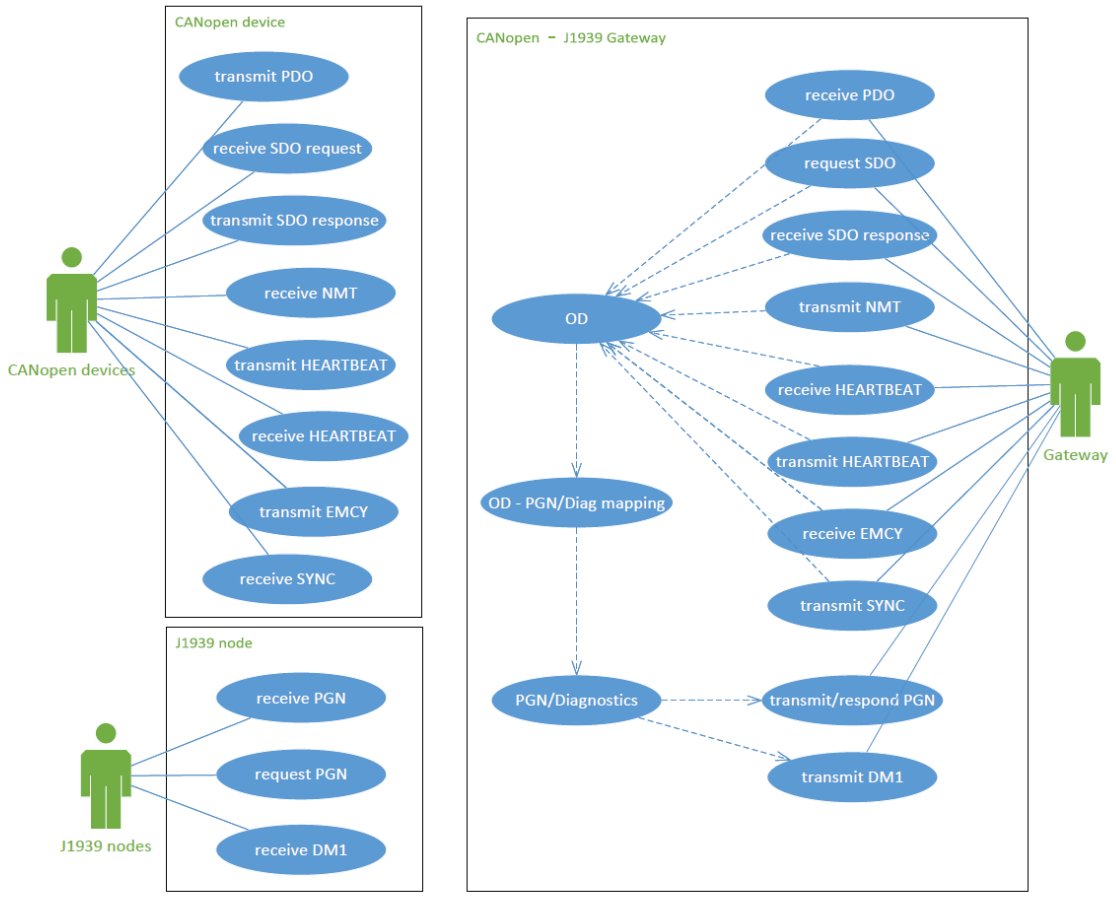


Figure 11:Use case diagram of CANopen - J1939 gateway

In order to fulfill the minimal requirements of the gateway, the following CANopen features shall be supported to communicate with other CANopen devices:

- NMT – the gateway shall be possible to change the state of the connected CANopen devices.
- SYNC – the gateway shall broadcast the SYNC message so that the slave devices can transmit PDO messages accordingly. If the slave devices transmit PDO messages in asynchronous mode, a SYNC message is not required.
- HEARTBEAT – the gateway shall broadcast and listen to the heartbeat messages for liveness detection in the network. The corresponding concept is, however, not found in J1939. Therefore, this should be optional.
- PDO – the gateway shall handle RPDO messages from the slave devices according to the configuration.

- SDO – the gateway shall handle SDO communications with the connected devices, such as requesting data and handling the response. Changing data in the device is not necessarily needed since the configuration can be performed offline by parsing a modified DCF (Device Configuration File) in a connected configurator.
- EMCY – the gateway shall be able to send an emergency message when malfunctioning is detected.

Moreover, the following J1939 features shall be supported in the gateway in order to communicate with other J1939 nodes:

- PGN – the gateway shall pack and unpack SPNs in PGNs and respond to PGN requests.
- Diagnostics – the gateway shall be able to broadcast diagnostic information through DM1 message.
- Network management – the gateway shall support dynamic address claiming.

## 3.4 CANopen – J1939 converter

We need an internal software module to convert CANopen objects to SPNs and transform the diagnostic information from EMCY to DM1, illustrated as *CANopen – J1939 converter* in Figure 10.

In order to convert CANopen objects to SPNs, we need to define a mapping table:

Table 15: Example of CANopen to J1939 mapping table

| **CANopen** | | | **J1939** | | |
|---|---|---|---|---|---|
| **Node ID** | **Object index** | **Object sub-index** | **Source address** | **SPN** | **Transform function** |
| 1 | 0x9130 | 0x01 | 2 | 110 | TrOD9130ToSPN110 |
| ... | ,,, | ,,, | ,,, | ,,, | ,,, |

With the help of the table above, we can easily convert the temperature from the Danfoss T92C sensor to engine coolant temperature in J1939. Transform function *TrOD9130ToSPN110* transforms the temperature from signed int32 to unsigned int32 with an offset of -40 and a resolution of 1 degree Celsius.

For diagnostics, the following mapping table shall be defined:

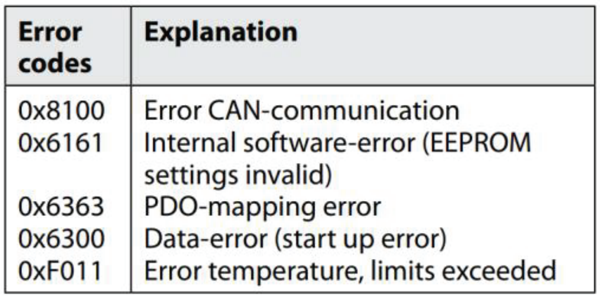
Table 16: Example of CANopen to J1939 diagnostics mapping table

| **CANopen** | | **J1939** | | |
|---|---|---|---|---|
| **Node ID** | **Error code** | **Source address** | **SPN** | **FMI** |
| 2 | 0x8100 | 2 | 110 | 11* |
| 2 | 0x6161 | 2 | 110 | 11 |
| 2 | 0x6363 | 2 | 111 | 11 |
| 2 | 0x6300 | 2 | 111 | 19* |
| 2 | 0xF011 | 2 | 112 | 19 |

*FMI11 – other error, FMI19 – data error.

The error code for Danfoss T92C sensor is explained in Table 17:

Table 17: Error code explanation of Danfoss T92C



| **Error codes** | **Explanation** |
|---|---|
| 0x8100 | Error CAN-communication |
| 0x6161 | Internal software-error (EEPROM settings invalid) |
| 0x6363 | PDO-mapping error |
| 0x6300 | Data-error (start up error) |
| 0xF011 | Error temperature, limits exceeded |

## 3.5 Configuration of the mapping tables

The two mapping tables described in 3.4 are the core of the communication gateway. They allow converting data in CANopen OD to J1939 SPNs, as well as translating error codes in CANopen into DTCs (Diagnostic Troubleshooting Code) in J1939.

In order to make the mapping tables configurable, they need to be stored in a well-defined memory structure. The value of the mapping tables can be changed by external tools via the XCP service [32] on CAN. XCP is a standardized calibration protocol for monitoring and altering data in ECUs. The following figure illustrates how the configuration of the gateway is done:

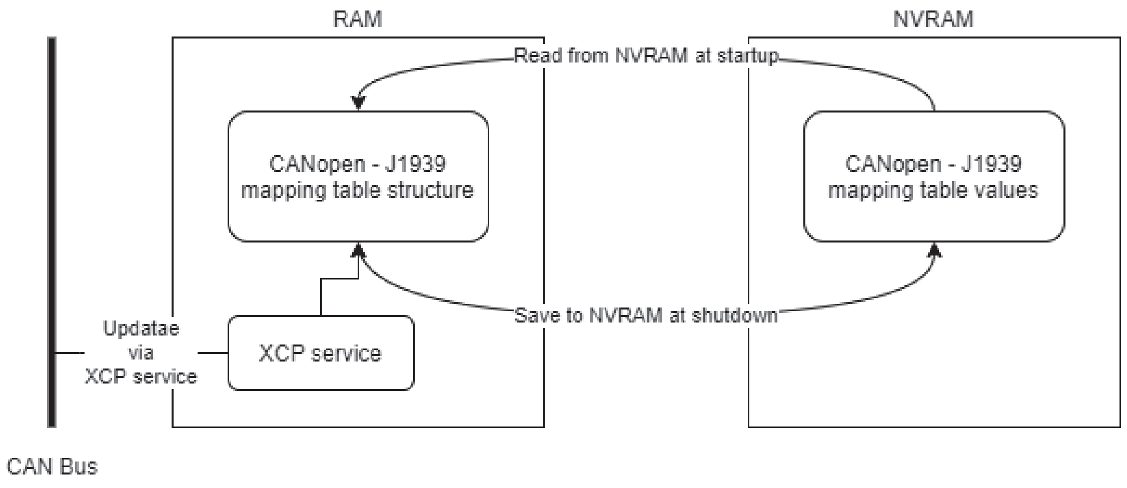

RAM
NVRAM
Read from NVRAM at startup
CANopen - J1939 mapping table structure
CANopen - J1939 mapping table values
Save to NVRAM at shutdown
Updatae via XCP service
XCP service
CAN Bus

Figure 12: Configuration of CANopen - J1939 mapping table

As shown in the above figure, the mapping table is initialized during the startup of ECU by copying the values from NVRAM (Non-Volatile Random-Access Memory) to RAM. Read and write of the mapping table is performed in the RAM. The configuration is saved and ready for being loaded next time by storing the mapping table values back to the NVRAM.

# 4 Implementation

## 4.1 Development environment

The gateway software is implemented on a development board built on the STM32F107VC microcontroller to implement the gateway.

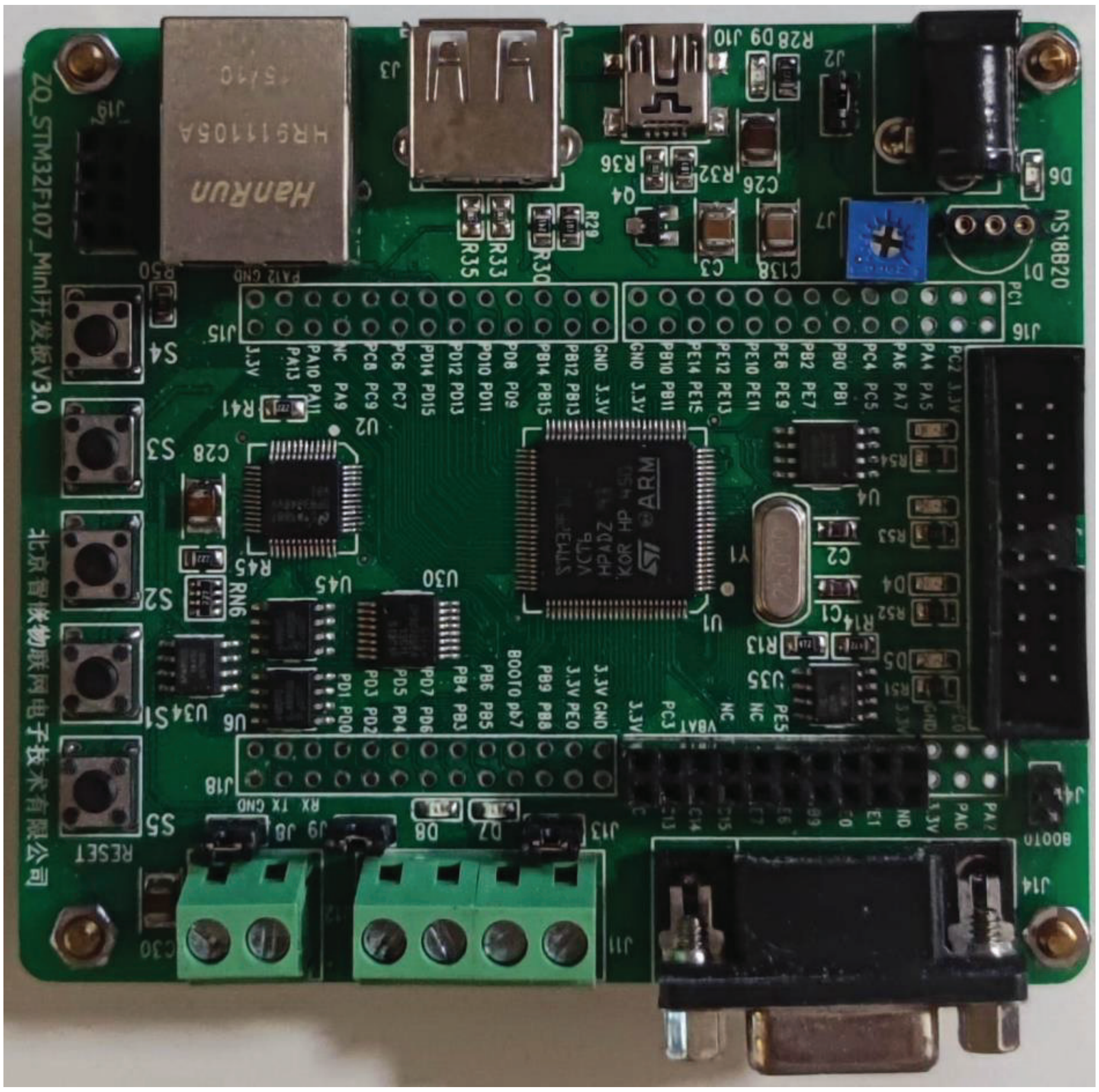


Figure 13:STM23F107VC development board

STM32F107VC has the following key features [33]:

- ARM 32-bit Cortex-M3 CPU

- 256 Kbytes Flash memory
- 64 Kbytes SRAM
- 2 CAN interfaces
- And much more…

The gateway software is implemented in C with FreeRTOS as the real-time operating system for scheduling the tasks. The whole project is developed in Keil uVision 5 IDE, which provides the whole development toolchain, including compiling the source code, linking object files, software downloading, and debugging via JTAG on the target. J-LINK on-board debugger from Segger is used for performing the on-board debug. Compared with outputting debug information via serial port, it allows setting breakpoints and stepping through the source code or assembly code on the target device. It also can read and write data in the memory and registers.

## 4.2 CANopen and J1939 communication stacks

Due to time constraints, we will not integrate the third-party software stacks for CANopen and J1939 since the integration requires much time. Instead, we implement a subset of CANopen and J1939 software stacks for demonstration purposes.

As discussed in 3.1, using two CAN channels is superior to allocating both CANopen and J1939 communications on one single CAN channel. We will assign CAN1 and CAN2 in Figure 9: Overview of CANopen - J1939 system architecture to the two CAN interfaces on the development board, respectively. The baud rate of CAN1 (CANopen) and CAN2 (J1939) are configured to 250 kbits/s.

The implementation of CANopen and J1939 communication stacks is trivial. We only need to follow the part of the standards required for the gateway. The most memory-consuming part is the PGN and SPN list in J1939 and object dictionary in CANOpen. In this thesis project, we only implement the necessary PGNs and objects in the OD for demonstration purposes.

The following parts have been implemented:

- J1939:
    - PGN transmit
    - PGN request/response

    - DM1 transmit
- CANopen:
    - PDO receive
    - SDO request/response
    - EMCY receive

Since the CANopen devices are configured as self-started, there is no need for waking up the CANopen devices. Therefore, NMT is not required for the demonstration. Similarly, as PDO communication is configured as “asynchronous,” SYNC is not needed. HEARTBEAT is only beneficial when liveness monitoring is required. This feature is not defined in the J1939 standard. Thus it is not necessary to implement HEARTBEAT.

## 4.3 CANopen to J1939 converter

For converting from CANopen objects to SPN, the function *TrOdToSpn* has been implemented. It takes the object index and subindex as inputs and returns the SPN and the transformed data with the help of the lookup table in Table 15.

Similarly, the function *TrEmcyToDm1* takes the CANopen error code as input and returns the DTC by looking up the error code in Table 16.

We maintain two mapping tables:

- Objects in CANopen to SPNs in J1939. This table allows us to match the objects with SPNs and perform the data transformation when required.
- Error code in CANopen to FMI in J1939 for converting the diagnostic information from CANopen devices to J1939 nodes.

The mapping tables are stored in a continuous area of memory. The idea is to copy the mapping tables from non-volatile memory to the continuous memory area. In this way, we can change the mapping tables by updating the data in the non-volatile memory without recompiling the source code.

## 4.4 Mapping table structures

The mapping table described in Table 15 is implemented as a structure array in C:

```
#define MAX_ENTRIES_OF_MAPPING_TABLE 40

typedef void (*transFunc)();

typedef struct {
    uint8_t canOpenNodeId;
    uint32_t canOpenObjIndex;
    uint8_t canOpenObjSubIndex;
    uint8_t j1939SA; //J1939 Source Address
    uint32_t j1939Spn;
    transFunc trFunc; //Transform function pointer
}canOpenJ1939MappingTable;


canOpenJ1939MappingTable mappingTable2[MAX_ENTRIES_OF_MAPPING_TABLE] =
{
    {2,0x9130, 0x01, 2, 110, TrOD9130ToSPN110},
    {3,0x9130, 0x01, 2, 174, TrOD9130ToSPN174},
    {4,0x9130, 0x01, 2, 175, TrOD9130ToSPN175},
    {5,0x9130, 0x01, 2, 146, TrOD9130ToSPN144},
    {6,0x9130, 0x01, 2, 146, TrOD9130ToSPN145},
    {7,0x9130, 0x01, 2, 146, TrOD9130ToSPN146}
};

void TrOD9130ToSPN110(int32_t obj9310, uint8_t* spn110)
{
    //transform function from signed int to unsigned int, deg C/bit, -
40 deg C offset
    *spn110 = (uint32_t)(obj9310/100 + 40);
}
```

The corresponding SPNs and transform functions can be found by searching objects' index and subindex in the table. Similarly, the mapped CANopen objects can be achieved by searching the SPNs in the table. The callback function `trFunc` converts raw data from CANopen to J1939.

The mapping table of diagnostic information in Table 16 is implemented similarly to the above example.

All mapping tables are stored in structure arrays to simplify the initialization shown in Figure 12. Since structures in C are stored in a continuous memory area, the initialization can be performed by using the `memcpy` function, copying the mapping table values from NVRAM to `mappingTable2` in the RAM.

# 5 Experimental results

We set up a system for demonstrating the gateway developed in this thesis project. The system consists of three temperature sensors and three pressure sensors.

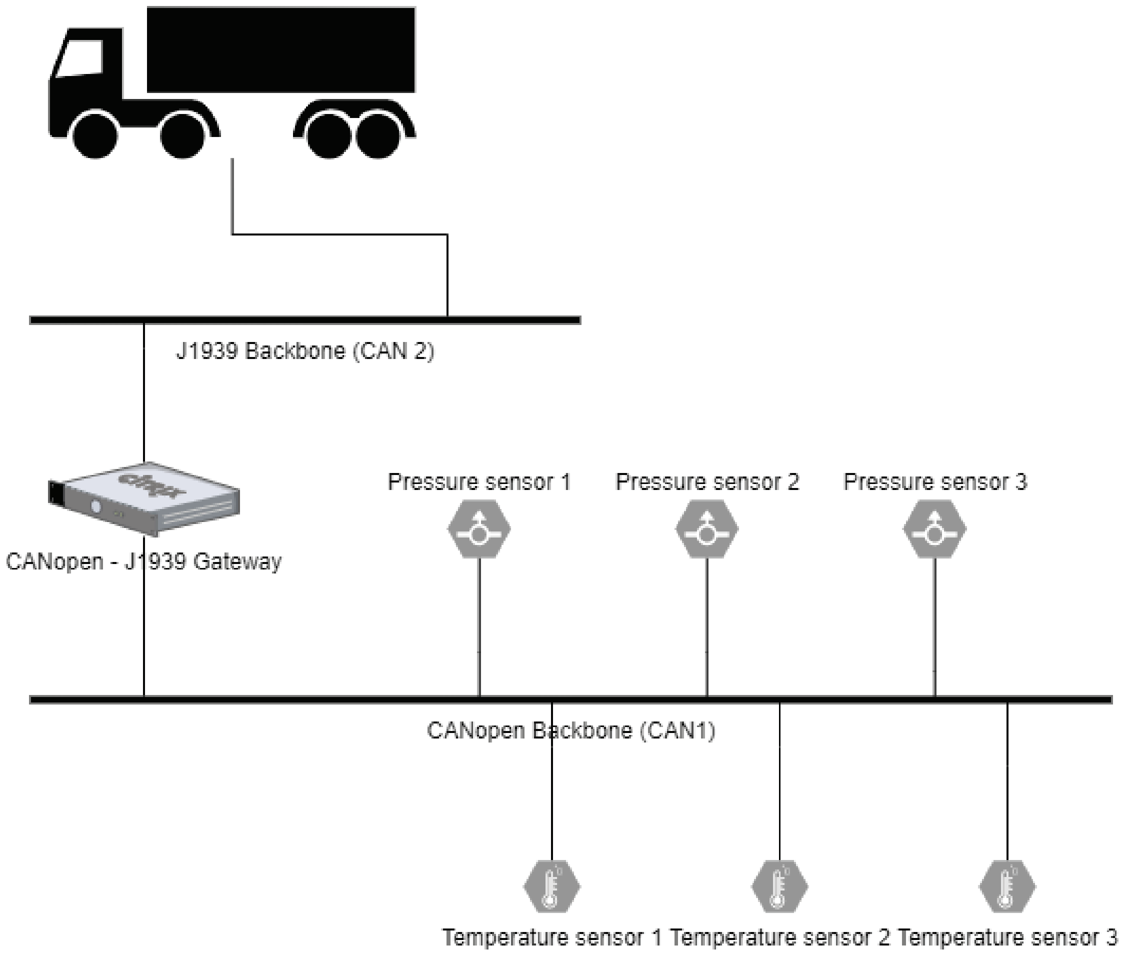


Figure 14: System setup for demonstrating the gateway.

The temperature and pressure sensors follow the CiA 404 - CANopen device profile for measuring devices and closed-loop controllers [34]. We choose to follow the protocols for temperature transmitter T92C [24] and pressure transmitter P92C [35] from Danfoss as examples.

CANPro Analyzer software is used for simulating the sensor communication and monitoring the communication with the gateway. The gateway is connected with the test PC via a CANalyst II device on the CAN bus.

All these sensors are pre-configured as self-starting (no NMT required). TPDOs are transmitted asynchronously with an event timer at 1000ms.

Sensors are configured and connected according to the following table:

Table 18: Mapping tables for demonstrating the CANopen-J1939 gateway

| **CANopen** | | | **J1939** | | | | |
|---|---|---|---|---|---|---|---|
| **Sensor** | **Device ID** | **Measure-ment** | **Source Address** | **SPN** | **Data pos.** | **PGN** | **Transmission Rate** |
| Temp. 1 | 2 | Engine Coolant | 0 (Engine 1) | 110 | 1 | 65262 | 1s |
| Temp. 2 | 3 | Fuel | 0 (Engine 1) | 174 | 2 | 65262 | 1s |
| Temp. 3 | 4 | Engine Oil | 0 (Engine 1) | 175 | 3-4 | 65262 | 1s |
| Press. 1 | 5 | Trailer Tire | 51(Tire pressure Controller) | 144 | 1-2 | 65146 | on request |
| Press. 2 | 6 | Drive Channel Tire | 51(Tire pressure Controller) | 145 | 3-4 | 65146 | on request |
| Press. 3 | 7 | Steer Channel Tire | 51(Tire pressure Controller) | 146 | 5-6 | 65146 | on request |

## 5.1 Test scenario 1 – TPDO to PGN

In this test scenario, three temperature sensors periodically (1s) send data to the communication gateway via TPDO message with ID 0x182, 0x183, and 0x184. The gateway transfers the corresponding data in an ET1 (engine temperature, PGN 65262) message on the J1939 bus.

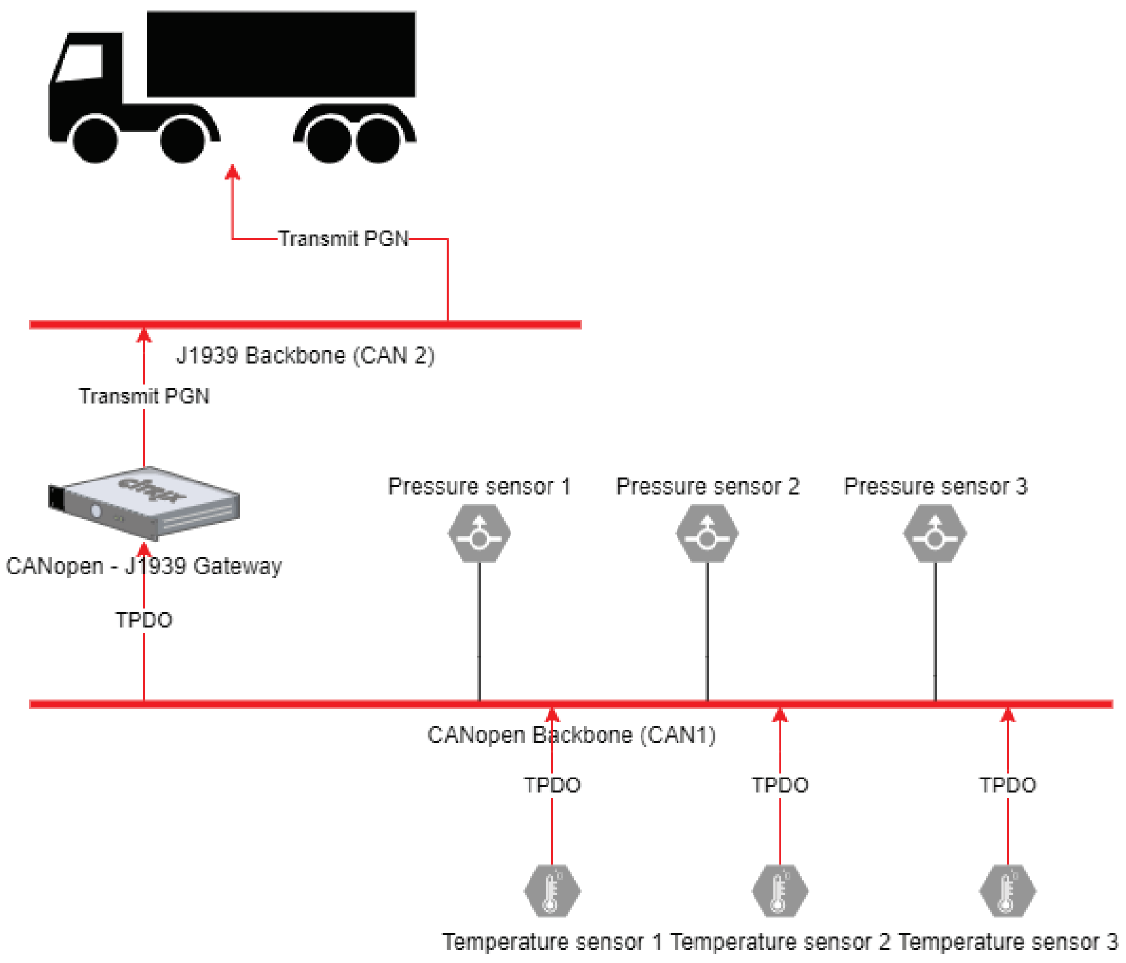


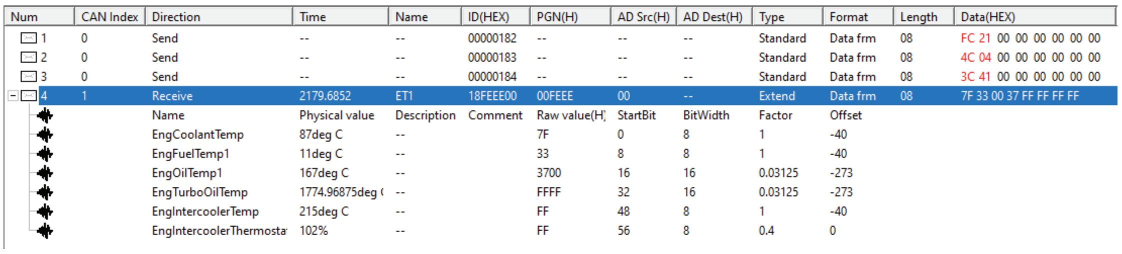
Figure 15: Signal flow for test scenario 1 - Transmit PGN

The transmitted data in the TPDOs and received PGN 65262 (0xFEEE) are shown in the following table:

Table 19: Overview of test scenario 1

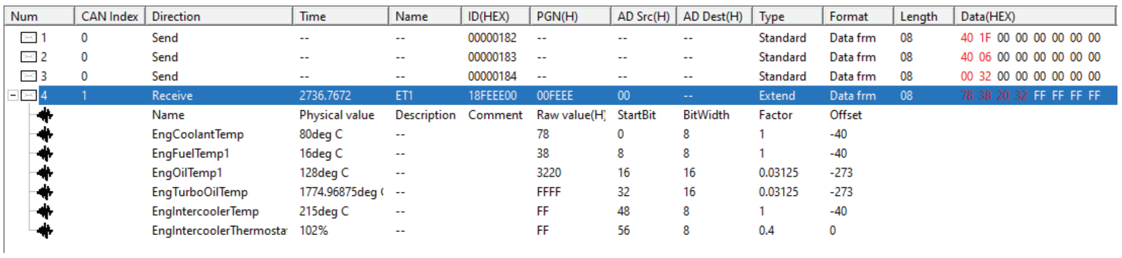
| **CANopen** | | | **J1939** | | | | |
|---|---|---|---|---|---|---|---|
| **Msg ID** | **Raw value** | **Resolu-tion** | **Raw value** | **Resolu-tion** | **Offset** | **Physi-cal value** | **Comments** |
| 0x182 | 0x21FC | 0.01 °C | 0X7F | 1 °C | -40 °C | 87 °C | Engine Coolant Temperature |
| 0x183 | 0x044C | 0.01 °C | 0x33 | 1 °C | -40 °C | 11 °C | Engine Fuel Temperature |
| 0x184 | 0x413C | 0.01 °C | 0x3700 | 0.3125 °C | -237 °C | 167 °C | Engine Oil Temperature |
| 0x182 | 0x1F40 | 0.01 °C | 0X78 | 1 °C | -40 °C | 80 °C | Engine Coolant Temperature |
| 0x183 | 0x0640 | 0.01 °C | 0x38 | 1 °C | -40 °C | 16 °C | Engine Fuel Temperature |
| 0x184 | 0x3200 | 0.01 °C | 0x3220 | 0.3125 °C | -273 °C | 128 °C | Engine Oil Temperature |

Here are the screenshots from CANPro Analyzer:



```
Num  CAN Index  Direction  Time                 Name         ID(HEX)   PGN(H)        AD Src(H)  AD Dest(H)  Type      Format    Length  Data(HEX)
1    0          Send       --                   --           00000182  --            --         --          Standard  Data frm  08      FC 21 00 00 00 00 00 00
2    0          Send       --                   --           00000183  --            --         --          Standard  Data frm  08      4C 04 00 00 00 00 00 00
3    0          Send       --                   --           00000184  --            --         --          Standard  Data frm  08      3C 41 00 00 00 00 00 00
4    1          Receive    2179.6852            ET1          18FEEE00  00FEEE        00         --          Extend    Data frm  08      7F 33 00 37 FF FF FF FF
                Name                   Physical value    Description  Comment  Raw value(H)  StartBit   BitWidth    Factor    Offset
                EngCoolantTemp         87deg C           --                     7F            0          8           1         -40
                EngFuelTemp1           11deg C           --                     33            8          8           1         -40
                EngOilTemp1            167deg C          --                     3700          16         16          0.03125   -273
                EngTurboOilTemp        1774.96875deg (   --                     FFFF          32         16          0.03125   -273
                EngIntercoolerTemp     215deg C          --                     FF            48         8           1         -40
                EngIntercoolerThermosta 102%             --                     FF            56         8           0.4       0
```

Figure 16: Screenshot of test scenario 1 - Engine Temperature



```
Num  CAN Index  Direction  Time                 Name         ID(HEX)   PGN(H)        AD Src(H)  AD Dest(H)  Type      Format    Length  Data(HEX)
1    0          Send       --                   --           00000182  --            --         --          Standard  Data frm  08      40 1F 00 00 00 00 00 00
2    0          Send       --                   --           00000183  --            --         --          Standard  Data frm  08      40 06 00 00 00 00 00 00
3    0          Send       --                   --           00000184  --            --         --          Standard  Data frm  08      00 32 00 00 00 00 00 00
4    1          Receive    2736.7672            ET1          18FEEE00  00FEEE        00         --          Extend    Data frm  08      78 38 20 32 FF FF FF FF
                Name                   Physical value    Description  Comment  Raw value(H)  StartBit   BitWidth    Factor    Offset
                EngCoolantTemp         80deg C           --                     78            0          8           1         -40
                EngFuelTemp1           16deg C           --                     38            8          8           1         -40
                EngOilTemp1            128deg C          --                     3220          16         16          0.03125   -273
                EngTurboOilTemp        1774.96875deg (   --                     FFFF          32         16          0.03125   -273
                EngIntercoolerTemp     215deg C          --                     FF            48         8           1         -40
                EngIntercoolerThermosta 102%             --                     FF            56         8           0.4       0
```

Figure 17: Screenshot of test scenario 1 - Engine Temperature

## 5.2 Test scenario 2 – PGN request to SDO request

In this test scenario, the pressure sensors on the CANopen bus only send the data on request.

We initialize a request command in the J1939 network, requesting PGN 65146 - tire pressure (in RQST message). Then the gateway sends three receive SDO messages with ID 0x605, 0x606, 0x607 to the pressure sensors. The requested SDOs are achieved by searching the corresponding SPNs in the CANopen-J1939 mapping table.

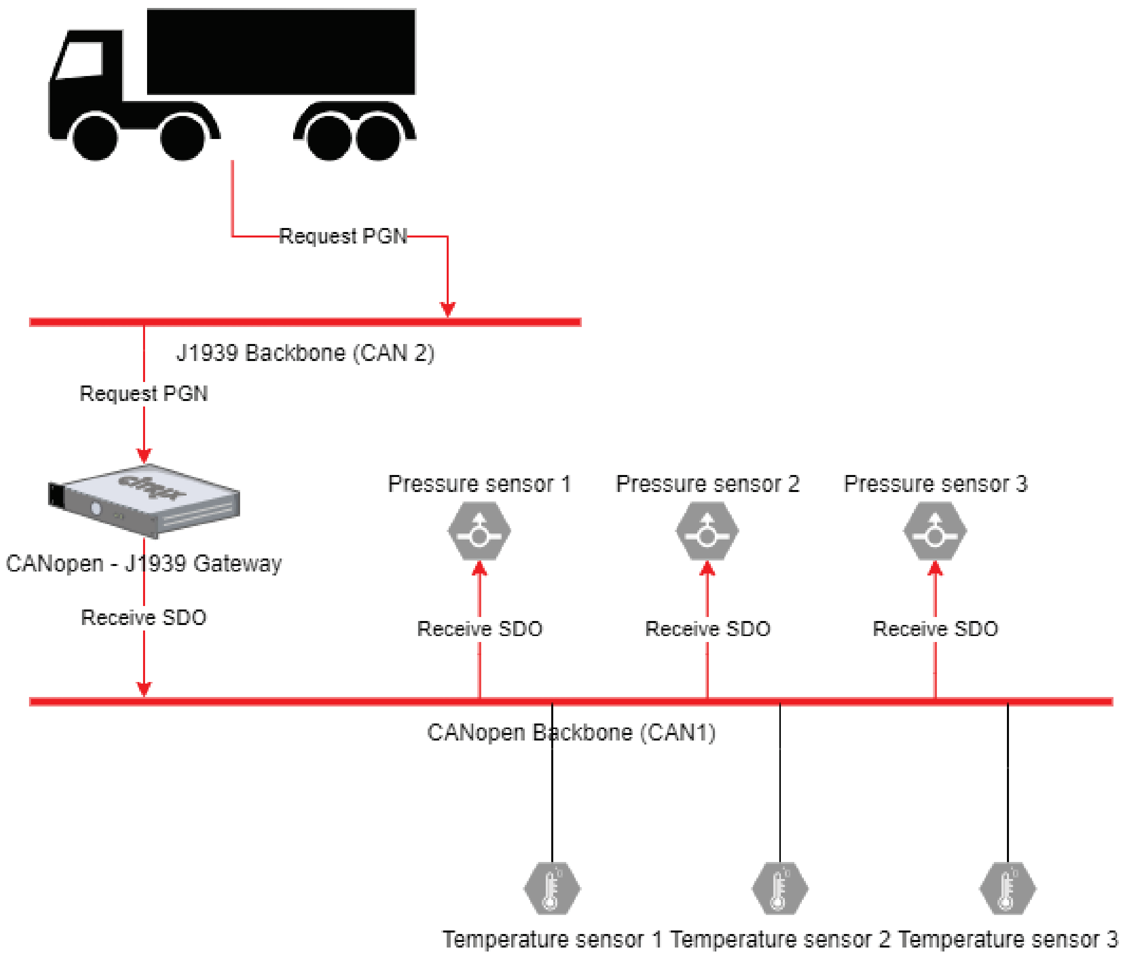


Figure 18: Signal flow for test scenario 2 - Request

The pressure sensors reply with the pressure values in transmit SDO messages with ID 0x585, 0x586, 0x587. The pressure values are converted to SPNs in the gateway. Finally, the gateway acknowledges with TP3 (tire pressure, PGN 65146) message on the J1939 bus.

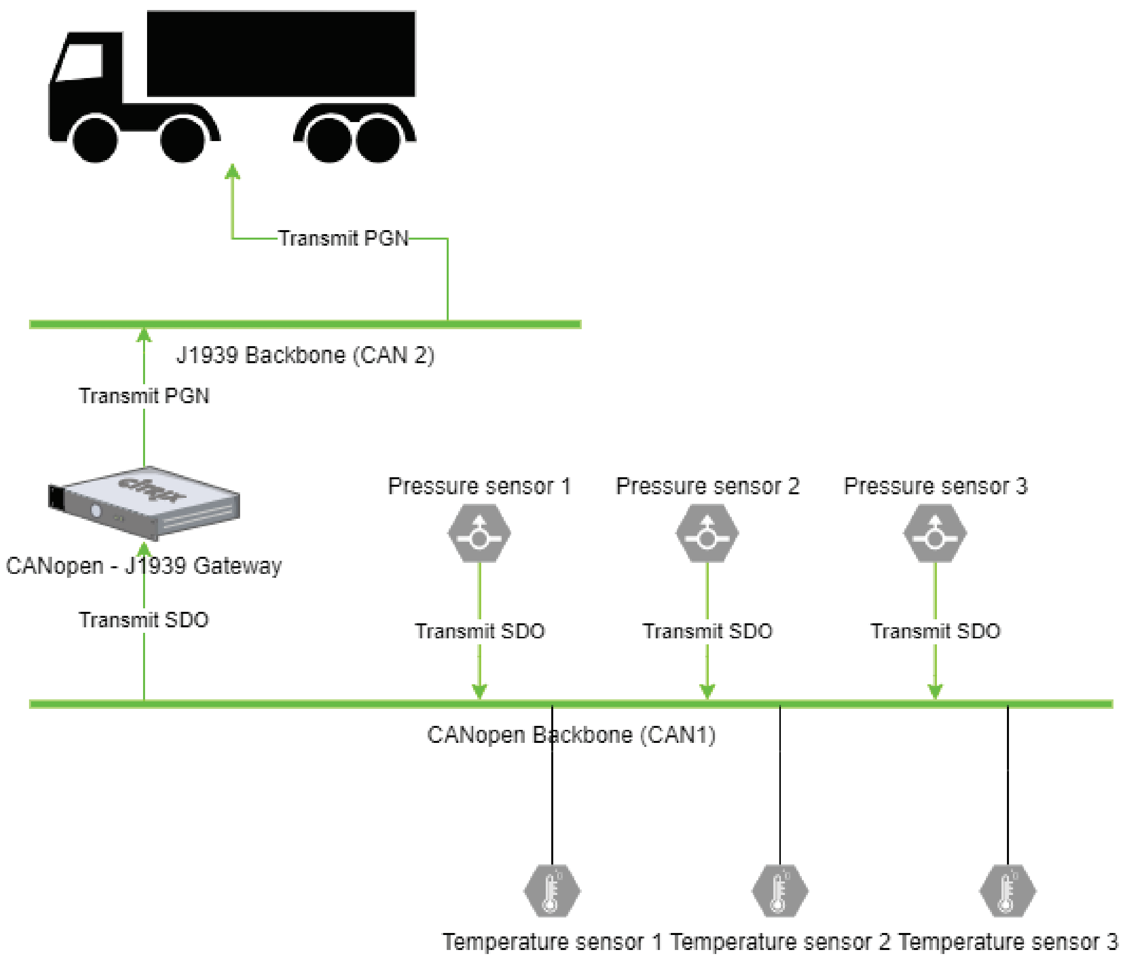


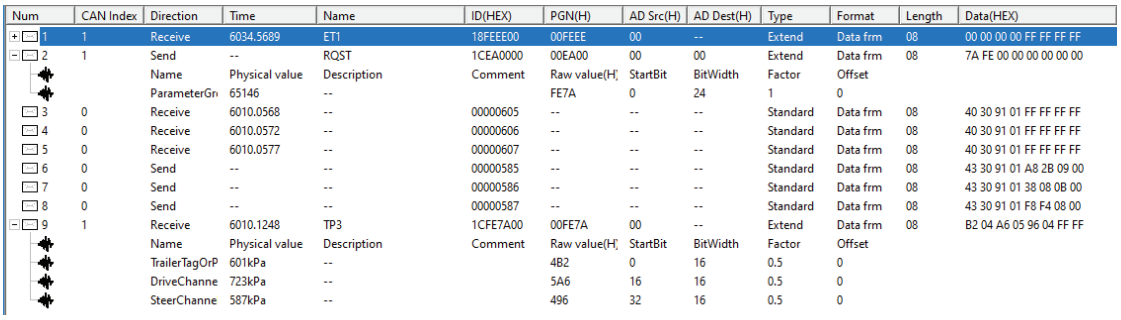
Figure 19: Signal flow for test scenario 2 - Response

The following table shows how SDO acknowledgment data is related to the ET1 message on J1939.

Table 20: Overview of test scenario 2

| **CANopen** | | | **J1939** | | | |
|---|---|---|---|---|---|---|
| **CANopen msg ID** | **CANopen raw value** | **CANopen resolution** | **J1939 raw value** | **J1939 resolution** | **Physical value** | **Com-ments** |
| 0x585 | 0x92BA8 | 1 Pa | 0X4B2 | 0.5 kPa | 601 kPa | Trailer tire pressure |
| 0x586 | 0xB0838 | 1 Pa | 0x5A6 | 0.5 kPa | 723 kPa | Driver tire pressure |
| 0x587 | 0x8F4F8 | 1 Pa | 0x496 | 0.5 kPa | 587 kPa | Steer tire pressure |

Here is the screenshot from CANPro Analyzer:



| Num | CAN Index | Direction | Time | Name | ID(HEX) | PGN(H) | AD Src(H) | AD Dest(H) | Type | Format | Length | Data(HEX) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Receive | 6034.5689 | ET1 | 18FEEE00 | 00FEEE | 00 | -- | Extend | Data frm | 08 | 00 00 00 00 FF FF FF FF |
| 2 | 1 | Send | -- | RQST | 1CEA0000 | 00EA00 | 00 | 00 | Extend | Data frm | 08 | 7A FE 00 00 00 00 00 00 |
|  |  | Name | Physical value | Description | Comment | Raw value(H) | StartBit | BitWidth | Factor | Offset |  |  |
|  |  | ParameterGr | 65146 | -- |  | FE7A | 0 | 24 | 1 | 0 |  |  |
| 3 | 0 | Receive | 6010.0568 | -- | 00000605 | -- | -- | -- | Standard | Data frm | 08 | 40 30 91 01 FF FF FF FF |
| 4 | 0 | Receive | 6010.0572 | -- | 00000606 | -- | -- | -- | Standard | Data frm | 08 | 40 30 91 01 FF FF FF FF |
| 5 | 0 | Receive | 6010.0577 | -- | 00000607 | -- | -- | -- | Standard | Data frm | 08 | 40 30 91 01 FF FF FF FF |
| 6 | 0 | Send | -- | -- | 00000585 | -- | -- | -- | Standard | Data frm | 08 | 43 30 91 01 A8 2B 09 00 |
| 7 | 0 | Send | -- | -- | 00000586 | -- | -- | -- | Standard | Data frm | 08 | 43 30 91 01 38 08 0B 00 |
| 8 | 0 | Send | -- | -- | 00000587 | -- | -- | -- | Standard | Data frm | 08 | 43 30 91 01 F8 F4 08 00 |
| 9 | 1 | Receive | 6010.1248 | TP3 | 1CFE7A00 | 00FE7A | 00 | -- | Extend | Data frm | 08 | B2 04 A6 05 96 04 FF FF |
|  |  | Name | Physical value | Description | Comment | Raw value(H) | StartBit | BitWidth | Factor | Offset |  |  |
|  |  | TrailerTagOrP | 601kPa | -- |  | 4B2 | 0 | 16 | 0.5 | 0 |  |  |
|  |  | DriveChanne | 723kPa | -- |  | 5A6 | 16 | 16 | 0.5 | 0 |  |  |
|  |  | SteerChanne | 587kPa | -- |  | 496 | 32 | 16 | 0.5 | 0 |  |  |

Figure 20: Screenshot of test scenario 2 - Tire pressure

## 5.3 Test scenario 3 – Diagnostics EMCY to DM1

In this test scenario, we simulate an emergency message (ID 0x82) on the CANopen bus with error code 0xF011 – temperature fault from temperature sensor 1. The gateway transmits a DM1 (active diagnostics) message on the J1939 bus.

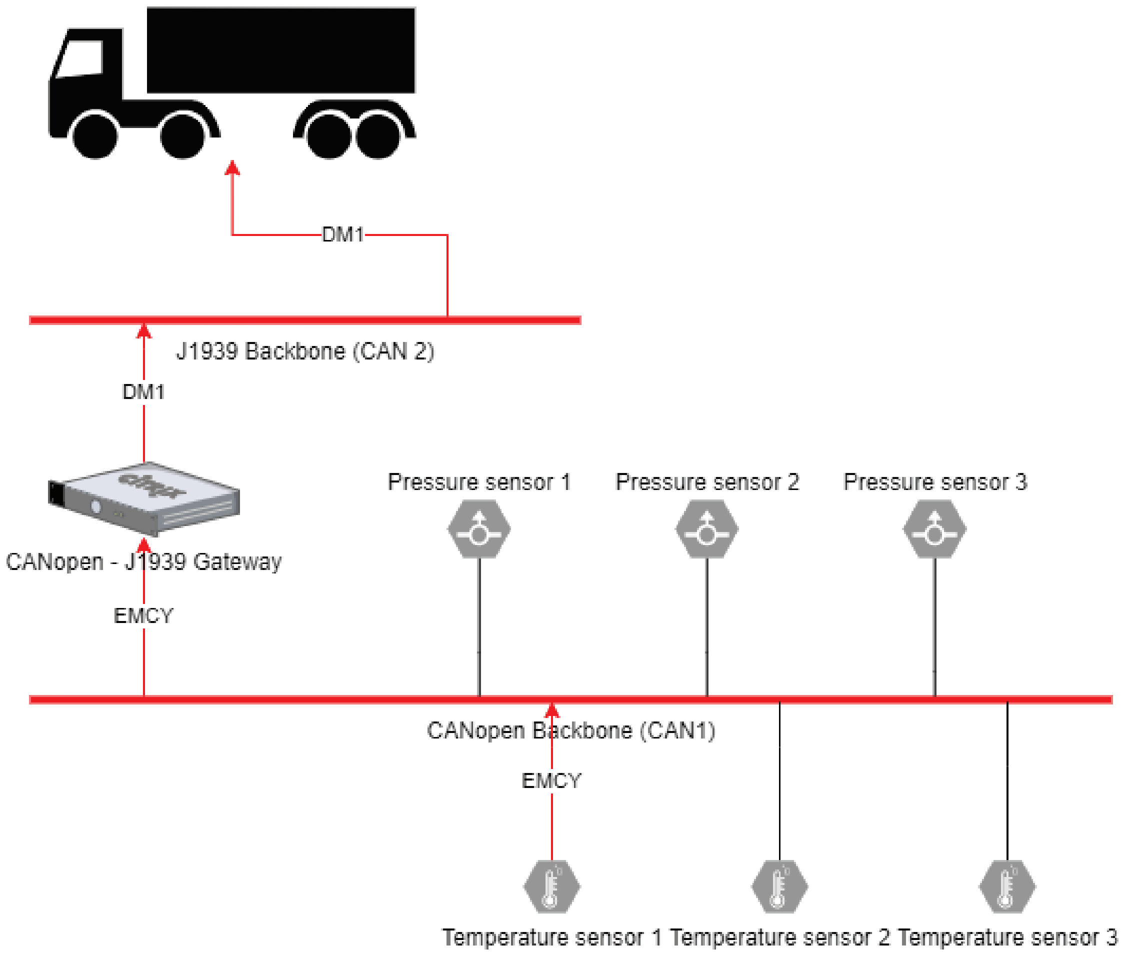
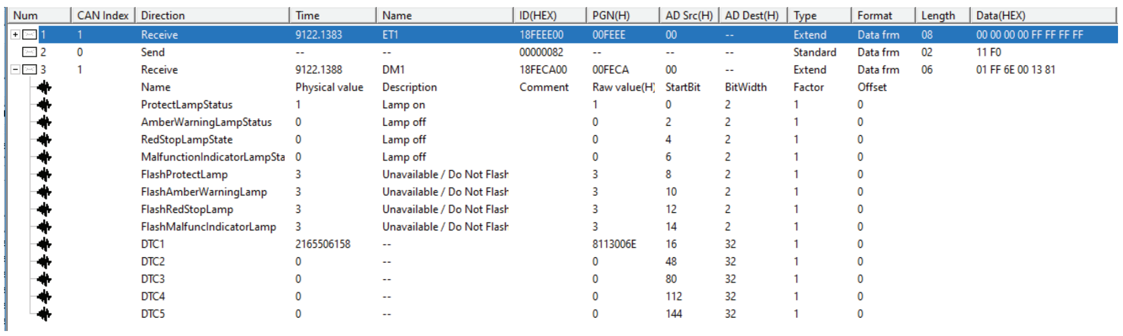


Figure 21: Signal flow for test scenario 3 - EMCY to DM1

The received DTC1 0x6E001381 in Figure 22 is interpreted as:

- Malfunctioning SPN: 110 (0x6E).
- FMI: 19 (0x13), indicating data error.
- Occurrence Counter: 1 (last 7 bits in 0x81)



| Num | CAN Index | Direction | Time | Name | ID(HEX) | PGN(H) | AD Src(H) | AD Dest(H) | Type | Format | Length | Data(HEX) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Receive | 9122.1383 | ET1 | 18FEEE00 | 00FEEE | 00 | -- | Extend | Data frm | 08 | 00 00 00 00 FF FF FF FF |
| 2 | 0 | Send | -- | -- | 00000082 | -- | -- | -- | Standard | Data frm | 02 | 11 F0 |
| 3 | 1 | Receive | 9122.1388 | DM1 | 18FECA00 | 00FECA | 00 | -- | Extend | Data frm | 06 | 01 FF 6E 00 13 81 |
|  |  | Name | Physical value | Description | Comment | Raw value(H) | StartBit | BitWidth | Factor | Offset |  |  |
|  |  | ProtectLampStatus | 1 | Lamp on |  | 1 | 0 | 2 | 1 | 0 |  |  |
|  |  | AmberWarningLampStatus | 0 | Lamp off |  | 0 | 2 | 2 | 1 | 0 |  |  |
|  |  | RedStopLampState | 0 | Lamp off |  | 0 | 4 | 2 | 1 | 0 |  |  |
|  |  | MalfunctionIndicatorLampSta | 0 | Lamp off |  | 0 | 6 | 2 | 1 | 0 |  |  |
|  |  | FlashProtectLamp | 3 | Unavailable / Do Not Flash |  | 3 | 8 | 2 | 1 | 0 |  |  |
|  |  | FlashAmberWarningLamp | 3 | Unavailable / Do Not Flash |  | 3 | 10 | 2 | 1 | 0 |  |  |
|  |  | FlashRedStopLamp | 3 | Unavailable / Do Not Flash |  | 3 | 12 | 2 | 1 | 0 |  |  |
|  |  | FlashMalfuncIndicatorLamp | 3 | Unavailable / Do Not Flash |  | 3 | 14 | 2 | 1 | 0 |  |  |
|  |  | DTC1 | 2165506158 | -- |  | 8113006E | 16 | 32 | 1 | 0 |  |  |
|  |  | DTC2 | 0 | -- |  | 0 | 48 | 32 | 1 | 0 |  |  |
|  |  | DTC3 | 0 | -- |  | 0 | 80 | 32 | 1 | 0 |  |  |
|  |  | DTC4 | 0 | -- |  | 0 | 112 | 32 | 1 | 0 |  |  |
|  |  | DTC5 | 0 | -- |  | 0 | 144 | 32 | 1 | 0 |  |  |

Figure 22: Screenshot of test scenario 3 - Active diagnostics

## 5.4 Test scenario 4 – Changing configurations of the mapping table

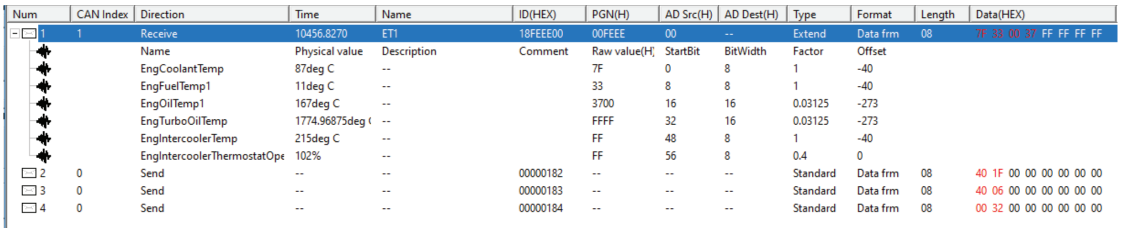
This test scenario will show the possibility of changing the mapping tables without recompiling the source code. Since we do not have Non-Volatile Random-Access Memory (NVRAM) on the development board, we create two mapping tables in the software. By changing the pointer of the mapping table via the on-board debugger, we can simulate the switch of the mapping tables. The initial value of the mapping tables is intended to be stored in the NVRAM. During the initialization of the gateway, the corresponding memory areas in the NVRAM shall be copied to the RAM area, in which the mapping table structures are stored.

We use the same test setup as test scenario 1 in 0.

Firstly, we run the same test in test scenario 1. Then, we switch the mapping table. The temperature sensors are configured with a resolution of 0.005 degrees Celsius instead of 0.01 degrees Celsius. With the same raw values from the temperature sensors, the Physical values of the temperature are half of the ones in test scenario 1:

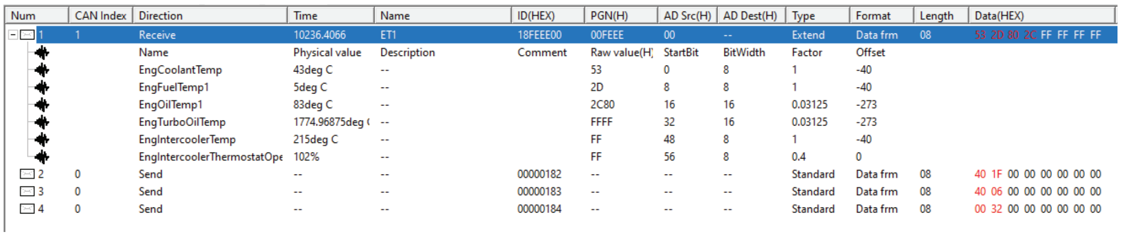
Table 21: Overview of test scenario 4

| CANopen msg ID | CANopen raw value | CANopen resolution | J1939 raw value | Physical value | Comments |
|---|---|---|---|---|---|
| 0x182 | 0x21FC | 0.01 °C | 0X7F | 87 °C | Engine Coolant Temperature |
| 0x183 | 0x044C | 0.01 °C | 0x33 | 11 °C | Engine Fuel Temperature |
| 0x184 | 0x413C | 0.01 °C | 0x3700 | 167 °C | Engine Oil Temperature |
| 0x182 | 0x21FC | 0.005 °C | 0X7F | 43 °C | Engine Coolant Temperature |
| 0x183 | 0x044C | 0.005 °C | 0x33 | 5 °C | Engine Fuel Temperature |
| 0x184 | 0x413C | 0.005 °C | 0x3700 | 83 °C | Engine Oil Temperature |



| Num | CAN Index | Direction | Time | Name | ID(HEX) | PGN(H) | AD Src(H) | AD Dest(H) | Type | Format | Length | Data(HEX) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Receive | 10456.8270 | ET1 | 18FEEE00 | 00FEEE | 00 | -- | Extend | Data frm | 08 | 7F 33 00 37 FF FF FF FF |
| | | Name | Physical value | Description | Comment | Raw value(H) | StartBit | BitWidth | Factor | Offset | | |
| | | EngCoolantTemp | 87deg C | -- | | 7F | 0 | 8 | 1 | -40 | | |
| | | EngFuelTemp1 | 11deg C | -- | | 33 | 8 | 8 | 1 | -40 | | |
| | | EngOilTemp1 | 167deg C | -- | | 3700 | 16 | 16 | 0.03125 | -273 | | |
| | | EngTurboOilTemp | 1774.96875deg ( | -- | | FFFF | 32 | 16 | 0.03125 | -273 | | |
| | | EngIntercoolerTemp | 215deg C | -- | | FF | 48 | 8 | 1 | -40 | | |
| | | EngIntercoolerThermostatOpe | 102% | -- | | FF | 56 | 8 | 0.4 | 0 | | |
| 2 | 0 | Send | -- | -- | 00000182 | -- | -- | -- | Standard | Data frm | 08 | 40 1F 00 00 00 00 00 00 |
| 3 | 0 | Send | -- | -- | 00000183 | -- | -- | -- | Standard | Data frm | 08 | 40 06 00 00 00 00 00 00 |
| 4 | 0 | Send | -- | -- | 00000184 | -- | -- | -- | Standard | Data frm | 08 | 00 32 00 00 00 00 00 00 |

Figure 23: Screenshot of test scenario 4 -  raw data resolution 0.01 degrees Celsius



| Num | CAN Index | Direction | Time | Name | ID(HEX) | PGN(H) | AD Src(H) | AD Dest(H) | Type | Format | Length | Data(HEX) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Receive | 10236.4066 | ET1 | 18FEEE00 | 00FEEE | 00 | -- | Extend | Data frm | 08 | 53 2D 80 2C FF FF FF FF |
| | | Name | Physical value | Description | Comment | Raw value(H) | StartBit | BitWidth | Factor | Offset | | |
| | | EngCoolantTemp | 43deg C | -- | | 53 | 0 | 8 | 1 | -40 | | |
| | | EngFuelTemp1 | 5deg C | -- | | 2D | 8 | 8 | 1 | -40 | | |
| | | EngOilTemp1 | 83deg C | -- | | 2C80 | 16 | 16 | 0.03125 | -273 | | |
| | | EngTurboOilTemp | 1774.96875deg ( | -- | | FFFF | 32 | 16 | 0.03125 | -273 | | |
| | | EngIntercoolerTemp | 215deg C | -- | | FF | 48 | 8 | 1 | -40 | | |
| | | EngIntercoolerThermostatOpe | 102% | -- | | FF | 56 | 8 | 0.4 | 0 | | |
| 2 | 0 | Send | -- | -- | 00000182 | -- | -- | -- | Standard | Data frm | 08 | 40 1F 00 00 00 00 00 00 |
| 3 | 0 | Send | -- | -- | 00000183 | -- | -- | -- | Standard | Data frm | 08 | 40 06 00 00 00 00 00 00 |
| 4 | 0 | Send | -- | -- | 00000184 | -- | -- | -- | Standard | Data frm | 08 | 00 32 00 00 00 00 00 00 |

Figure 24: Screenshot of test scenario 4 -  raw data resolution 0.005 degrees Celsius

# 6 Conclusion

This thesis project describes CAN, one of the most widely used communication buses in the automotive industry and industrial automation. Two higher layer protocols based on the CAN bus have been investigated. One of them is SAE J1939, which is primarily used in heavy-duty vehicles. The purpose is to standardize the communication protocol for data exchange between different nodes in heavy-duty vehicles. CANopen, on the other hand, is used in a wide variety of industrial automation applications. As CANopen is more widely used than J1939, a greater number of devices on the market are designed to support standardized CANopen. Such devices may not have a J1939 interface. As a result, a communication gateway is necessary in order to integrate CANopen devices into the J1939 network without modifying the existing CANopen devices and J1939 nodes.

We have proposed the architecture and design for a communication gateway that enables data and diagnostic information to be transferred from CANopen devices to J1939 nodes. In addition, J1939 nodes are also capable of requesting data from CANopen devices through the gateway. The CANopen - J1939 mapping table structure can be configured without recompiling the software by using external tools, providing some flexibility.

The results in chapter 5 show that the gateway fulfills the requirements of the use case scenarios defined in chapter 3. By adding a gateway, CANopen devices can communicate in the J1939 network without modifying any J1939 node or CANopen device to make the vehicle network aware that a node or a device is communicating in another protocol.

## 6.1 Related work

Zeltwanger proposed a CANopen truck gateway specification [36] in 2000. This proposal became a technical document – CiA 413 CANopen device profile for truck gateways [8]. However, the technical document is in draft specification proposal state and has never been released. The device profile only defines a static mapping from J1939 to CANopen to interchange information between towing and towed vehicles. It does not cover how diagnostic data or requesting data from a CANopen device. Since the mapping from J1939 to CANopen is static, it cannot be changed. The gateway software designed in this thesis project, however, allows changes of J1939 – CANopen mapping. Implementation of CiA 413 is discussed in [37].

At the time of writing, several device profiles have been proposed by CiA recently (December 2020). CiA 406 [9] enables the integration of encoder devices into the J1939 network, while CiA 409 [10] is for inclinometer CANopen devices in J1939. CiA 510 [38] specifies the mapping from J1939 to SDO and EMCY communication objects. These device profiles are, however, primarily designed for extending J1939 with some CANopen-like functionalities. By introducing two proprietary PGNs (CAM11, CAM12) in J1939, a J1939 node can configure data or send diagnostics to a CANopen device with an extra J1939 interface. Both CiA 406 and CiA 409 device profiles require modifications of the software in J1939 nodes and CANopen devices to communicate with each other.

# References

[1] ISO 11898:1993 - Controller area network (CAN), 1993.

[2] SAE International, "SAE J1939 - Serial Control and Communications Heavy Duty Vehicle Network," 2018.

[3] CAN in Automation, "CANopen additional application layer functions - Part 1: General definition, CiA 301 Version 4.2.0," 2021.

[4] International Electrotechnical Commission, "IEC 62026-3:2014 - Low-voltage switchgear and controlgear - Controller-device interfaces (CDIs) - Part 3: DeviceNet," 2014.

[5] International Electrotechnical Commission, "IEC 61162-3:2008 - Maritime navigation and radiocommunication equipment and systems - Digital interfaces - Part 3: Serial data instrument network," 2008.

[6] National Standard of The People's Republic of China, "GB/T 27930-2015: Communication Protocols between Off-Board Conductive Charger and Battery Management System for Electric Vehicle," 2015.

[7] SAE International, "SAE J1939/71 - Vehicle Application Layer," 2020.

[8] CAN in Automation, "CANopen device profile for truck gateways – Part 1: General definitions, CiA 413-1 version 3.0.0," 2011.

[9] CAN in Automation, "Encoder device profile - Part J: Mapping to J1939, CiA 406-J version 1.1.0," 2020.

[10] CAN in Automation, "Inclinometer device profile - Part J: Mapping to J1939, CiA 410-J version 1.1.0," 2020.

[11] U. Kiencke, S. Dais and M. Litschel, "Automotive Serial Controller Area Network," in *SAE International Congress and Exposition*, 1986.

[12] N. Navet and F. Simonot-Lion, “Trends in Automotive Communication Systems.,” *Networked Embedded Systems,* p. 13, 2009.

[13] ISO/IEC 7498-1:1994 Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model, 1994.

[14] ISO 11898-2:2016 - Road vehicles — Controller area network (CAN) — Part 2: High-speed medium access unit, 2016.

[15] ISO 11898-3:2006 - Road vehicles — Controller area network (CAN) — Part 3: Low-speed, fault-tolerant, medium-dependent interface, 2006.

[16] ISO 11898-1:2015 - Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling, 2015.

[17] CAN in Automation, “CAN remote frames - Avoiding of usage, CiA Application Note 802,” 2005.

[18] SAE International, “SAE J1587 - Electronic Data Interchange Between Microcomputer Systems in Heavy-Duty Vehicle Applications,” 2013.

[19] SAE International, “SAE J1708 - Serial Data Communications Between Microcomputer Systems in Heavy-Duty Vehicle Applications,” 2004.

[20] SAE International, “SAE J1939/21 - Data Link Layer,” 2021.

[21] SAE International, “SAE J1939/73 – Application Layer – Diagnostics,” 2020.

[22] SAE International, “SAE J1939/81 - Network Management,” 2017.

[23] Siemens, “CANopen Tutorial Version 2.0,” 07 2019. [Online]. Available: https://support.industry.siemens.com/cs/ww/en/view/109479771. [Accessed 02 11 2021].

[24] Danfoss, “Operation guide - Temperature transmitter CANopen DST T92C,” 01 2018. [Online]. Available: https://assets.danfoss.com/documents/67324/AQ267742448328en-000101.pdf. [Accessed 02 11 2021].

[25] CAN in Automation, “CANopen electronic data sheet (EDS), CiA 306 version 1.3.0,” 2005.

[26] CAN in Automation, “CANopen electronic device description - Part 1: Electronic data sheet (EDS) and device configuration file (DCF), CiA 306-1 version 1.4.0,” 2019.

[27] CAN in Automation, “CANopen XML schema definition, CiA 311 version 1.1.0,” 2011.

[28] H. Boterenbrood, “CANopen high-level protocol for CAN-bus,” *,* 2000.

[29] “CANopen Stack Project,” [Online]. Available: https://canopen-stack.org/. [Accessed 02 11 2021].

[30] “Open Source J1939 Stack,” [Online]. Available: https://github.com/stevinliang/open-j1939. [Accessed 02 11 2021].

[31] “SAE J1939 protocol stack,” [Online]. Available: https://www.st.com/en/embedded-software/j1939.html. [Accessed 02 11 2021].

[32] Association for Standardisation of Automation and Measuring Systems, “ASAM MCD-1 XCP - The Universal Measurement and Calibration Protocol,” 2017.

[33] STMicroelectronics, “DS6014 - Connectivity line, ARM®-based 32-bit MCU with 64/256 KB Flash, USB OTG, Ethernet, 10 timers, 2 CANs, 2 ADCs, 14 communication interfaces,” 2017.

[34] CAN in Automation, “CANopen device profile for measuring devices and closed-loop controllers, CiA 404 version 1.2.0,” 2002.

[35] Danfoss, “Operation guide - Pressure transmitter CANopen DST P92C,” 01 2018. [Online]. Available: https://assets.danfoss.com/documents/67325/AQ267732163301en-000202.pdf. [Accessed 02 11 2021].

[36] H. Zeltwanger, “Mapping J1939 Parameter to CANopen Object Dictionary,” in *International Off-Highway & Powerplant Congress & Exposition*, 2000.

[37] A. Åström, “J1939 – CANopen gateway - A CANopen gateway according to DSP-413,” 2006.

[38] CiA, “J1939 parameter groups for SDO and EMCY, CiA 510 version 1.1.0,” 2020.