CANedge1 Docs

Release FW 01.04.02

Mar 31, 2021

CONTENTS

0.1 CANedge1 documentation

0.1.1 About this manual

0.1.1.1 Purpose

This manual describes the functionality of the CANedge1 (firmware 01.04.02) with focus on:

1. Hardware & installation
2. Configuration
3. Firmware upgrade

This manual does not provide details on available software/API tools.

0.1.1.2 Other documentation

The CANedge1 Intro and CANedge2 Intro provide practical guides for getting started. The intros also detail software & API tools that can be used with the CANedge.

0.1.1.3 Notation used

The following notation is used throughout this documentation:

Admonitions

Note: Used to highlight supplementary information

Warning: Used if incorrect use may result in major loss of data and/or time

Danger: Used if incorrect use may result in damage to the device, personal injury or death

Number bases

When relevant, the base of a number is written explicitly as x_y, with y as the base.

The following number bases are used throughout this documentation:

- Binary ($y = 2$). Example: The binary number 10101010 is written as 10101010_2
- Decimal ($y = 10$). Example: The decimal number 170 is written as 170_{10}
- Hexadecimal ($y = 16$). Example: The hexadecimal number AA is written as AA_{16}

The value of a number is the same regardless of the base (e.g. the values in above examples are equal $10101010_2 = 170_{10} = AA_{16}$). However, it is sometimes more convenient to represent the number using a specific base.

0.1.2 Legal information

0.1.2.1 Usage warning

Warning: Carefully review the below usage warning before installing the product

The use of the CANedge device must be done with caution and an understanding of the risks involved. The operation of the device may be dangerous as you may affect the operation and behavior of a CAN bus system.

Improper installation or usage of the device can lead to serious malfunction, loss of data, equipment damage and physical injury. This is particularly relevant when the device is physically connected to a CAN based application that may be controlled via the CAN bus. In this setup you can potentially cause an operational change in the system, turn on/off certain modules and functions or change to an unintended mode.

While the device supports a high degree of security in regards to wireless data transfer and over-the-air updates, it is recommended that these features are used with caution. Incorrect usage of this functionality can result in a device being unable to connect to your server. Further, changing transmit messages over-the-air should be done with extreme caution.

The device should only be used by persons who are qualified/trained, understand the risks and understand how the device interacts with the system in which it is integrated.

0.1.2.2 Terms & conditions

Please refer to our general terms & conditions.

0.1.2.3 Electromagnetic compatibility

The CANedge has been tested in accordance with CE, FCC and IC standards.

Certificates are available in the online documentation.

The device is in conformity with all provisions of Annex II of Council Directive 2014/30/EU, in its latest amended version, referred to EMC directive.

The device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

The device complies with the requirements set forth in the Innovation, Science and Economic Development Canada (ISED) Rules and Regulations ICES-003 Class B and the measurement procedure according to CAN/CSA CISPR 22-10.

Specifically, it is in conformity with the following standards:

```
EN 55032:2015 - Electromagnetic Compatibility of Multimedia Equipment
EN 55024:2010+A1:2015 - IT equipment. Immunity characteristics. Limits and methods of␣
↪measurement
FCC Rules and Regulations Part 15 Subpart B: 2018
ICES-003: Issue 6 January 2016
```

0.1.2.4 Voltage transient tests

The CANedge has passed below ISO 7637-2:2011 tests, performed by TÜV SÜD:

```
ISO 7637-2:2011: Voltage transient emissions test on supply lines
ISO 7637-2:2011: Transient immunity test on supply lines
```

0.1.2.5 Contact details

For any questions regarding our products, please contact us:

CSS Electronics
EU VAT ID: DK36711949
Soeren Frichs Vej 38K (Office 35), 8230 Aabyhoej, Denmark
contact[AT]csselectronics.com
+45 91252563
www.csselectronics.com

0.2 Specification

0.2.1 CAN-bus (x2)

- Physical
  - Two physical CAN-bus interfaces
  - Industry standard DB9 (D-sub9) connectors
- Transceiver
  - Compliant with CAN Protocol Version 2.0 Part A, B and ISO 11898-1
  - Compliant with ISO CAN FD and Bosch CAN FD
  - Ideal passive behavior when unpowered (high impedance / no load)
  - Protection: ±16kV HBM ESD, ±15kV IEC ESD, ±70 V bus fault, short circuit
  - Common mode input voltage: ±30V
  - TXD dominant timeout (prevents network blocking in the event of a failure)
  - Data rates up to 5Mbps[1]
- Controller
  - Based on MCAN IP from Bosch
  - Bit-rate: Auto-detect (from list[2]), manual simple (from list[3]) or advanced (bit-timing)
  - 128 standard CAN ID + 64 extended CAN ID filters (per interface)
  - Advanced filter configuration: Range, mask, acceptance, rejection, down sampling
  - Configurable transmit messages, single shot or periodic (up to 128/64 regular/extended)
  - Support for Remote-Transmission-Request (RTR) frames
  - Silent mode: Restricted (acknowledge only) or monitoring (transmission disabled)
  - Supports all CAN based protocols (J1939, CANopen, OBD2, NMEA 2000, ...)[4]
- Application
  - Control reception (logging) and transmission states using CAN-bus *control message*
  - Heartbeat signal to broadcast device time, space left of SD-card and rx/tx state

0.2.2 LIN-bus (x2)

- Physical
  - Two physical LIN-bus interfaces
  - Industry standard DB9 (D-sub9) connectors
- Transceiver
  - Protection: ±8kV HBM ESD, ±1.5kV CDM, ±58V bus fault
  - Supports 4V to 24V applications
  - TXD dominant timeout (prevents network blocking in the event of a failure)
  - Data rates up to 20kbps

[1] Supported FD bit-rates: 1M, 2M, 4M
[2] Bit-rate list: 5k, 10k, 20k, 33.333k, 47.619k, 50k, 83.333k, 95.238k, 100k, 125k, 250k, 500k, 800k, 1M
[3] Bit-rate list: 5k, 10k, 20k, 33.333k, 47.619k, 50k, 83.333k, 95.238k, 100k, 125k, 250k, 500k, 800k, 1M, 2M, 4M
[4] The device logs raw data frames

- Controller
  - Support for both publisher and subscriber modes
  - Automatic[5] and custom frame lengths
  - Classic and Extended checksum formats
  - Configurable transmit messages, single shot or periodic

0.2.3 Logging

- Extractable industry grade micro SD-card (8-32GB)
- Standard FAT file system (can be read directly by a PC)
- Logging to industry standard `.MF4` (ASAM MDF4) file format
- Simultaneous logging from 2 x CAN-bus + 2 x LIN-bus
- Message time stamping with 50 us resolution
- Supports cyclic logging mode (oldest log file is deleted when memory is full)
- Log file splitting based on file size or time
- Session number (power cycle) log file naming (groups log files)
- Globally unique ID for each device ensures unique log file naming
- Power safe (device can be disconnected during logging without corrupting data)
- High-performance logging[6]

0.2.4 Real-time clock (RTC)

- High precision real-time clock retains date and time when device is off

0.2.5 Electrical

- Device supply
  - Channel 1 voltage supply range: +7.0V to +32V DC[7]
  - Reverse voltage protection[8]
  - Transient voltage event protection on supply lines[9]
  - Consumption:
    - CANedge1: 0.8W
    - CANedge2: 1W
- Secondary port output supply[10]
  - Configurable output supply on connector 2 (`CH2`), fixed 5V up to 1.5A[11]
  - Supports power out scheduling to control the output state based on time of day

[5] Data lengths are defined by bits 4 and 5 of the LIN identifier
[6] See the performance tests
[7] The device is supplied from connector 1 (`CH1`)
[8] Up to 24V
[9] The transient voltage protection is designed to clamp low energy voltage events. High energy voltage events may overheat and destroy the input protection
[10] Can be used to supply external devices
[11] The 5V output can be used to power WiFi hotspots, sensors, small actuators, external LEDs, etc.

0.2.6 Mechanical

- Status indicated using external LEDs
- Robust aluminum enclosure
- Dimensions: 50.2 x 83.4 x 24.5 mm (L x W x H)[12]
- Weight: 100g
- Operating temperature: -25 degC to +70 degC

0.2.7 Connectivity

See the CANedge2 for wireless data transfer.

[12] Without external antenna

0.3 Hardware

0.3.1 Installation

This section outlines the installation requirements that shall be satisfied.

0.3.1.1 Supply quality

The nominal voltage shall be kept within specifications at all times. The device is internally protected against low energy voltage events which can be expected as a result of supply wire noise, ESD and stub-wire inductance.

If the supply line is shared with inductive loads, care should be taken to ensure high energy voltage events do not reach the device. Automotive environments often include several sources of electrical hazards, such as load dumps (disconnection of battery while charging), relay contacts, solenoids, alternator, fuel injectors etc. The internal protection circuitry of the device is not capable of handling high energy voltage events directly from such sources.

0.3.1.2 Grounding

ISO 11898-2 tolerates some level of ground offset between nodes. To ensure the offset remains within range, it is recommended to use a single point ground reference for all nodes connected to the CAN-bus. This may require the ground wire to be carried along with data wires.

If a secondary CAN-bus network is connected to *Channel 2*, care must be taken to ensure that the ground potentials of the two networks can safely be connected through the common ground in the device.

0.3.1.3 Cable shielding

Shielding is not needed in all applications. If shielding is used, it is recommended that a short pig-tail be crimped to the shield end at each connector.

0.3.1.4 CAN ISO 11898-2

ISO 11898-2 defines the basic physical requirements of a high-speed CAN-bus network. Some of these are listed below:

- Max line length (determined by bit-rate)
- Line termination (120 ohm line termination at each end of data line)
- Twisted data lines
- Ground offsets in range -2V to +7V

0.3.1.5 CAN-bus stub length

It is recommended that the CAN-bus stub length is kept short. The stub length is defined as the length from the ”main” data line wires to the connection point of the CAN-bus nodes.

0.3.1.6 Enclosure caution

The device is not intended for use without the enclosure.

Warning: Opening the enclosure can permanently damage the device due to e.g. ESD (electrostatic discharge) - and improper handling may void the warranty

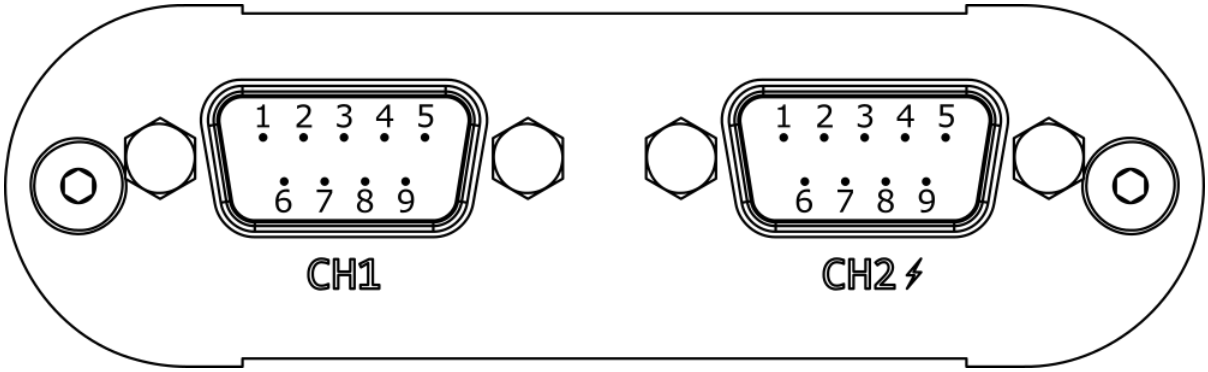
0.3.1.7 Mounting

The device should be mounted in a way that minimizes vibration exposure and accounts for the IP rating of the device.

0.3.2 Connector

0.3.2.1 Pinout

The CANedge uses two D-sub9 connectors for supply, 2 x CAN, 2 x LIN and a 5 V Supply Output.



Make sure to refer to the pinout matching the product hardware revision (see the label).

Hardware revision $\geq$ 00.01

Pin #	Channel 1 (CH1)	Channel 2 (CH2)
1	NC	5V Supply Output
2	CAN 1 L	CAN 2 L
3	GND	GND
4	LIN Data 1	LIN Data 2
5	NC	NC
6	GND (optional)	GND (optional)
7	CAN 1 H	CAN 2 H
8	NC	NC
9	Supply & LIN1 VBAT	LIN2 VBAT

Hardware revision = 00.00

The hardware 00.00 pinout can be found here.

Supply

The supply (CH1 pin 9) is used to power the device. The supply is internally protected against reverse polarity and low-energy voltage spikes.

Refer to the *Electrical Specification* for more details on the device supply.

Warning: The supply line must be protected against high-energy voltage events exceeding device limits

GND

All GND (ground) pins are connected internally.

5 V Supply Output

The +5 V output can be used to power external devices. The power can be toggled via the device configuration. The output can deliver 1.5 A @ 5 V continuously.

Danger: Connecting external input power to this pin can permanently damage the device

Warning: External protection (such as clamp diodes) must be installed if inductive loads are connected to the 5V Supply Output

CAN L/H

Warning: CAN-bus requires no common reference (ground). However, it is recommended that GND (ground) is carried along with CAN-L/H to prevent that the common-mode voltage is exceeded (resulting in transceiver damage)

LIN VBAT

The LIN-bus positive reference. Supports systems operating from 4V to 24V.

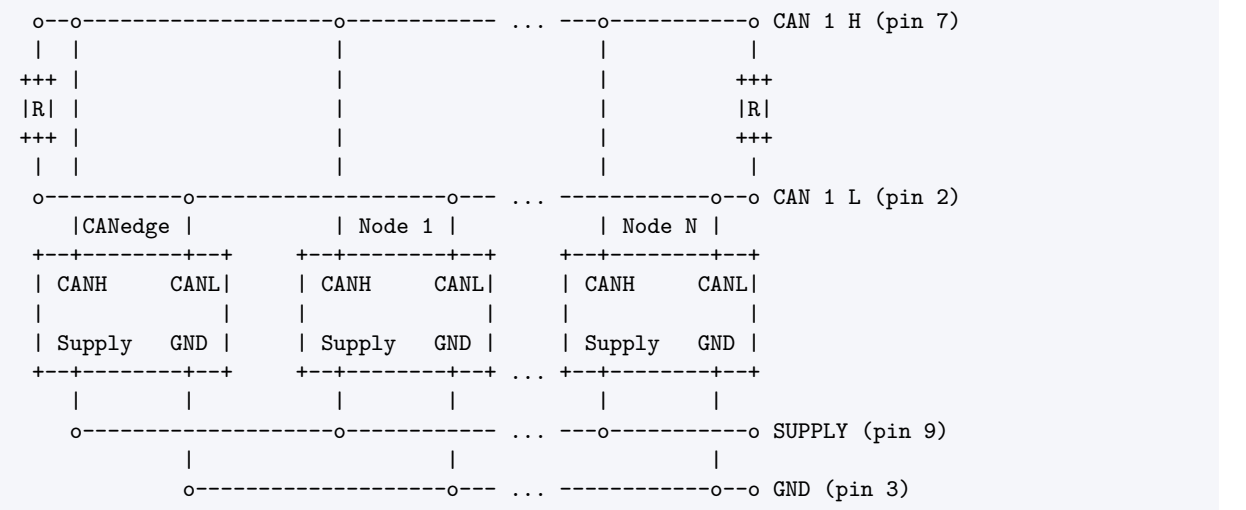
- LIN1 VBAT: Pin is shared with device supply and shares the supply input protection circuit
- LIN2 VBAT: Tolerates voltage spikes up to 48V. Spikes above this can damage the interface

LIN Data

LIN-bus single-wire data line referenced to LIN VBAT.

0.3.2.2 Basic wiring example

Below example illustrates how the CANedge CAN-bus 1 (channel 1) can be connected.



0.3.3 Enclosure

The CANedge uses a robust aluminium enclosure - PDF drawings and 3D STEP files can be found in the online documentation.

0.3.4 SD card

The CANedge uses an extractable SD-card to store device configuration and log files.

Warning: Never extract the SD-card while the device is on. Remove power first and wait a few seconds for the device to turn off.

0.3.4.1 File organization

The files stored on the SD-card are organized as illustrated by below example:

```
/
├── config-XX.XX.json
├── schema-XX.XX.json
├── device.json
└── LOG/
    └── [DEVICE_ID]
        ├── 00000001
        │   ├── 00000001.MF4
        │   ├── 00000002.MF4
        │   ├── ...
        │   └── ...
        ├── 00000002
        │   ├── 00000001.MF4
        │   ├── 00000002.MF4
        │   ├── ...
        │   └── ...
        ├── ...
        └── ...
```

- `config-XX.XX.json`: Configuration File (device configuration)
- `schema-XX.XX.json`: Rule Schema (configuration rules)

- `device.json`: Device File (device information)
- `LOG/`: Path for log files

For more information on the specific files, refer to the documentation menu.

Log files

Log files are stored in `LOG/[DEVICE_ID]/[SESSION_NUMBER]/[SPLIT_NUMBER].[FILE_EXTENSION]`. For more information on the structure of the log files, refer to the *log file* section.

0.3.4.2 Type

The CANedge uses a specifically selected industrial graded SD-card with special timing constraints to ensure safe shutdown when power is lost.

Warning: The device cannot be guaranteed to work if the pre-installed SD-card is replaced

Exchanging SD cards between devices

The SD card is not "locked" to the device. If the card is replaced, be aware of the following points:

- Critical device specific data are **not** stored on the SD card (e.g. device ID)
- Non-critical meta data are stored on the SD card (e.g. log file session number)
- If the card is replaced by a card from another device, it is recommended to clear the card first
- It may be necessary to update the configuration file if it contains device specific settings

0.3.4.3 Lifetime

The SD-card memory wears as any other flash based memory. The industrial graded SD-cards provided with the CANedge have the following guaranteed minimum endurance numbers[1]:

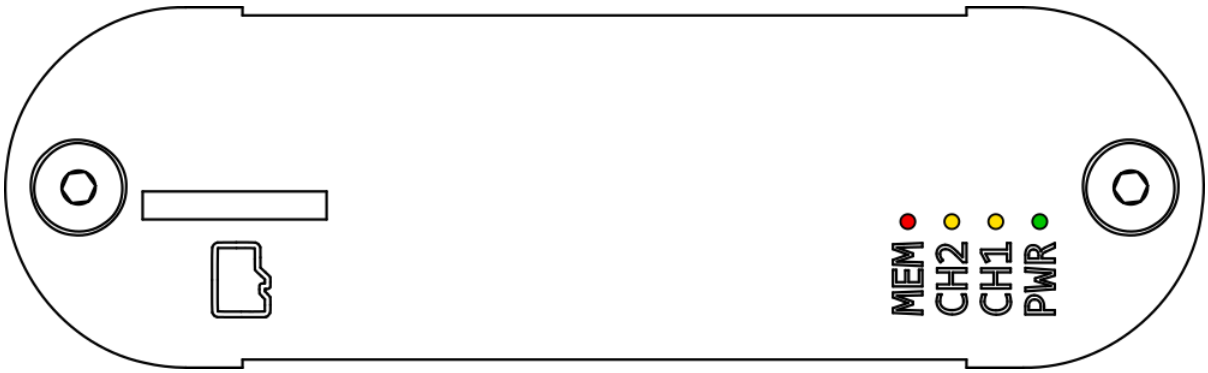
Size [GB]	TBW	Lifetime @ 1MB/sec [years][2]	Lifetime @ 1MB/min [years]
8	24	0.8	47.9
32	96	3.2	191.5

0.3.5 LED

0.3.5.1 Location

The LEDs are located at the back of the device as illustrated below.

[1] TBW: Terabytes Written
[2] A constant logging rate of 1 MB/sec is likely much much higher than in any practical logging use-case



0.3.5.2 Overview

LED Short Name	LED Color	Main Function
PWR	Green	Power
CH1	Yellow	Bus activity on connector 1 (CH1)
CH2	Yellow	Bus activity on connector 2 (CH2)
MEM	Red	Memory card activity

PWR

The *Power* LED is constantly on when the device is in normal operation. An exception is when the firmware is being updated (for more information go to the *Firmware* section).

CH1 / CH2

The *Channel 1* / *Channel 2* LEDs indicate bus activity on Channel 1 and 2 respectively.

MEM

The *Memory* LED indicates activity on the memory card. Config file parsing, message logging, file upload etc. all generate activity on the memory card.

0.3.6 Label

A unique label is attached to the back of each device. Examples of the labels are illustrated below.

Note: The data matrix can be scanned to simplify installation of a new device

The label holds the following information:

- Device type: `CANedge1`
- Production data in format YYYYWW (WW = week number): `201930`
- Hardware version: `00.00`
- Unique device ID: `EA9B650D`
- Data matrix (ECC200) containing production date and device ID: `201930;EA9B650D;XXXXXXXXXXXX`

0.4 Configuration

0.4.1 General

This page documents the *general* configuration.

0.4.1.1 Configuration file fields

This section is autogenerated from the Rule Schema.

Device `device`

Meta data `device.properties.meta`

Optional meta data string. Displayed in device file and log file headers. Example: Site1; Truck4; ConfigRev12

type: `string`	minLength: `0`	maxLength: `30`

Security `security`

Server public key `security.properties.kpub`

Server / user ECC public key in base64 format. Shall match the encryption used for all protected fields.

type: `string`	minLength: `0`	maxLength: `100`

Debug `debug`

Debug functionality for use during installation and troubleshooting.

System log `debug.properties.syslog`

System events logged to the SD-card. The log levels are listed in order of increasing amount of information logged. Should only be enabled if needed during installation or troubleshooting.

type: `integer`	default: `0`	options: `Disable (0): [0] Error (1): [1] Warning (2): [2] Info (3): [3]`

0.4.1.2 Configuration explained

This section contains additional information and examples.

Device meta data

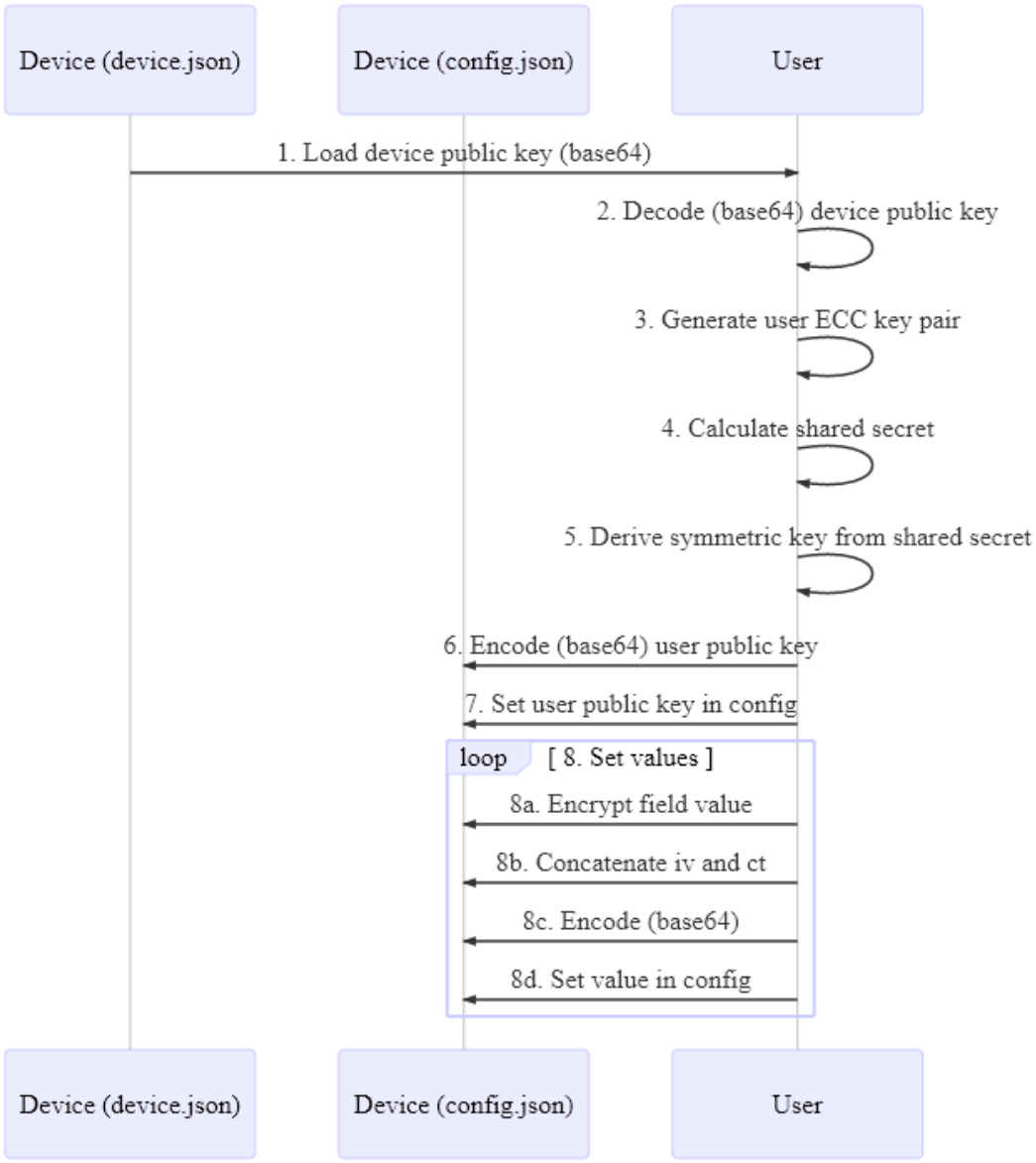
The device meta data is an optional string copied to the `device.json` file and log file headers.

Security

Some configuration field values can be encrypted to hide sensitive data stored in the Configuration File (passwords etc.). In this section, we provide a technical summary and provide resource suggestions for implementing the encryption.

The field encryption feature uses a key agreement scheme based on Elliptic Curve Cryptography (ECC) (similar to the one used in a TLS handshake). The scheme allows the device and user to compute the same shared secret, without exposing any secrets. The shared secret is in turn used to generate a symmetric key, which is used to encrypt / decrypt protected field values.

The following sequence diagram illustrates the process of encrypting configuration fields:



Below we explain the sequence:

1. Load device public key field (`kpub`) from the `device.json` file
2. Decode the device public key (base64)
3. Generate random user key pair (public and private) using curve `secp256r1`
4. Calculate shared secret using device public key and user private key

5. Derive shared symmetric key using HMAC-SHA256 with "config" as data and shared secret as key. Use the first 16 bytes of the output
6. Encode user public key (used by the device to calculate the same shared symmetric key for decryption)
7. Set the encoded user public key in the device configuration file
8. Use AES-128 CTR to encrypt protected fields using the symmetric key. The resulting initialization vector (iv) and cipher text (ct) are concatenated (iv + ct), base64 encoded and stored in the configuration file

Note: The symmetric key shall match the public key set by the user in the configuration and protected fields shall be encrypted with this symmetric key

Note: By storing the symmetric key it is possible to change specific protected fields - without updating the user public key (and in turn all other protected fields)

Encryption tools

Tools are provided with the CANedge for use in encrypting secure fields - see the CANedge Intro.

Example Python code

You can batch-encrypt passwords across multiple devices using e.g. Python. Below we provide a basic code sample to illustrate how Python can be used to encrypt plain-text data. The example code is tested with `Python 3.7.2` and requires the `pycryptodome` crypto library:

`Python example code`

0.4.2 Logging

This page documents the *logging* configuration

0.4.2.1 Configuration file fields

This section is autogenerated from the Rule Schema file.

File `file`

File split size (1 to 512 MB) `file.properties.split_size`

Log file split size in MB. When the file split size is reached a new file is created and the logging continues. Closed log files can be pushed to a server if WiFi is available. Small split sizes may reduce performance.

type: `integer`	default: `50`	minimum: `1`	maximum: `512`

File split time period (0 to 86400 seconds, 0 = disable) `file.properties.split_time_period`

Log file split time period in seconds relative to midnight (00:00:00). When a split time is reached a new file is created and the logging continues. Closed log files can be pushed to a server if WiFi is available. Small split time periods may reduce performance.

type: `integer`	default: `0`	minimum: `0`	maximum: `86400`	multipleOf: `10`

File split time offset (0 to 86400 seconds) `file.properties.split_time_offset`

Log file split time offset in seconds. This value offsets the split_time_period relative to midnight (00:00:00). The set value shall be less than the split_time_period value.

type: `integer`	default: `0`	minimum: `0`	maximum: `86400`	multipleOf: `10`

Cyclic logging `file.properties.cyclic`

With cycling logging mode enabled the oldest log file is deleted when the memory card becomes full, allowing the logging to continue.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

Compression `compression`

Level `compression.properties.level`

Window size used during optional compression. Larger window sizes yield potentially better compression rates, but may reduce logging performance. Compressed log files need to be decompressed prior to processing.

type: `integer`	default: `0`	options: `Disable: [0] 256 bytes window: [256] 512 bytes window: [512] 1024 bytes window: [1024]`

Encryption `encryption`

State `encryption.properties.state`

Optional log file encryption. Encrypted log files need to be decrypted prior to processing. Decryption requires your encryption password in plain form - if this is lost, the encrypted data cannot be recovered.

type: `integer`	default: `0`	options: `Disable: [0] Enable: [1]`

Error Frames `error_frames`

State `encryption.properties.state`

Should error frames be logged. Can negatively impact performance, as potentially many more frames needs to be written to the log.

type: `integer`	default: `0`	options: `Disable: [0]`

0.4.2.2 Configuration explained

This section contains additional information and examples.

File split

File splitting can be based on file size or file size and time:

- `split_time_period = 0`: Split based on size only
- `split_time_period > 0`: Split based on both size and time - whichever is reached first

Limits

The file system limits should be considered when configuring the split size and time:

- SD-card size
- Max `1024` sessions
- Max `256` splits (log files) in each session

Above limits result in a maximum of `1024*256=262144` log files if fully utilised.

If the session count limit is reached, the logger will either:

- Stop logging if cyclic logging is disabled[1]
- Delete the oldest session if cyclic logging is enabled

If SD-card becomes full (no more space), the logger will either:

- Stop logging if cyclic logging is disabled[1]
- Delete the oldest split file from the oldest session if cyclic logging is enabled

Compression

Log files can be compressed on the device during logging using a variant of the LZSS algorithm based on heatshrink. Compressed files will have `*.MFC` as file extension. A high window size improves compression rates, but may cause message loss on very busy networks.

The table below lists results for J1939 and OBD data with different window size configurations[3]:

Window size (Bits) Data	J1939 % (range)	OBD % (range)
8	49.7 (47.1-51.4)	32.0 (30.3-32.8)
9	49.5 (46.3-51.6)	30.2 (29.6-31.1)
10	41.4 (38.9-45.5)	30.0 (29.6-30.8)

Decompression can be done using an implementation of LZSS or via the CANedge tools (see the CANedge Intro).

Note: The split size set in `split_size` considers the size of the compressed data. I.e. if the split size is 10 MB, the resulting file sizes become 10 MB regardless if compression is used or not.

[1] If files are offloaded through WiFi in the case of the CANedge2, the logging will resume
[3] Compressed size in percentage of original. Lower is better.

Encryption

Log files can be encrypted on the device. Here, data is streamed through an AES-GCM encryption scheme prior to persistence on the SD card. Encrypted files have `*.MFE` as file extension.

Note: It is recommended to use a 40+ character password for proper encryption

Decryption can be done using an implementation of the PBKDF2 algorithm or via CANedge tools (see the CANedge Intro).

0.4.3 Real-Time-Clock

This page documents the *real-time-clock* configuration

0.4.3.1 Configuration file fields

This section is autogenerated from the Rule Schema file.

Real-Time Clock (RTC) “ “

Time synchronization method `properties.sync`

Internal real-time-clock synchronization method. The real-time-clock is maintained when the device is off.

type: `integer`	default: 2	options: `Retain current time: [0] Manual update: [1] Automatic update (requires WiFi): [2]`

Time zone (UTC-12 to UTC+14) `properties.timezone`

Adjustment in full hours to the UTC time. Includes daylight savings time if applicable.

type: `integer`	default: `0`	minimum: `-12`	maximum: `14`

Adjustment (-129600 to 129600 seconds) `properties.adjustment`

Adjustment in seconds to the UTC time. Can be used for fine tuning the internal time.

type: `integer`	default: `0`	minimum: `-129600`	maximum: `129600`

0.4.3.2 Configuration explained

This section contains additional information and examples.

The CANedge uses a real-time clock (RTC) with battery backup, which allows it to retain the absolute date & time when the device is not powered. The RTC enables the CANedge to add absolute timestamps to recorded messages. The RTC is set to UTC time during production.

Time-zone changes and minor adjustments can be done via the `timezone` and `adjustment` fields.

Synchronization methods (`sync`)

The RTC time can either be *retained*or *manually set*.

Manual update

Manually changing the RTC is *only needed* if the RTC time is completely lost (e.g. after a battery replacement). The following sequence explains how the RTC can be manually set:

1. Select the manual ´sync´ method and set the current UTC time
2. Power on the device and wait a few seconds to allow the device to read the manually set time
3. Power off the device
4. Change the ´sync´ method to *retain* current time
5. Power on the device again
6. Verify that the new absolute time is now correctly retained across power cycles
7. Set timezone (`timezone`) and do minor adjustments (`adjustment`) if needed

0.4.4 Secondary port

This page documents the *secondary port* configuration

0.4.4.1 Configuration file fields

This section is autogenerated from the Rule Schema file.

Power schedule `power_schedule`

The daily power schedule is defined by a number of power-on from/to intervals. Define no power-on intervals to keep always off. Define one interval with from/to both set to 00:00 to keep always on. Time format is HH:MM (1 minute resolution)

type: `array`	default: `[]`	maxItems: 5

From `power_schedule.items.properties.from`

Power-on FROM time in format HH:MM. Shall be before power-on TO time. E.g. at midnight 00:00

type: `string`	default: `00:00`

To `power_schedule.items.properties.to`

Power-on TO time in format HH:MM. Shall be after power-on FROM time. E.g. at midday 12:00.

type: `string`	default: `00:00`

0.4.4.2 Configuration explained

This section contains additional information and examples.

Note: Power out scheduling has resolution of 1 min and 1 min tolerance

Note: Power out scheduling uses adjusted local time (as set in the configuration)

Example: Secondary port power is scheduled to be on daily in the interval `00:00-04:00` and `12:00-16:00`. Secodary port configuration:

```
"secondaryport": {
  "power_schedule": [
    {
      "from": "00:00",
      "to": "04:00"
    },
    {
      "from": "12:00",
      "to": "16:00"
    }
  ]
}
```

The power out is turned off when the time changes from 03:59 to 04:00 and 15:59 to 16:00.

0.4.5 CAN

This page documents the *CAN* configuration.

The configuration of CAN Channel 1 and CAN Channel 2 is identical.

The CAN configuration is split into the following sections:

0.4.5.1 General

This page documents the *general* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

General ‘‘ ‘‘

CAN bus general configuration

Reception (rx) initial state `properties.rx_state`

The initial state of CAN-bus reception. Can be changed using the control signal.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

Transmission (tx) initial state `properties.tx_state`

The initial state of CAN-bus transmissions. Can be changed using the control signal.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

Configuration explained

This section contains additional information and examples.

0.4.5.2 Physical

This page documents the *physical* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

Mode `mode`

Device CAN bus mode. Configures how the device interacts with the bus. In Normal mode, the device can receive, acknowledge and transmit frames. In Restricted mode, the device can receive and acknowledge, but not transmit frames. In Bus Monitoring mode, the device can receive, but not acknowledge or transmit frames. It is recommended to always use the most restrictive mode possible.

type: `integer`	default: `1`	options: `Normal (receive, acknowledge and transmit): [0] Restricted (receive and acknowledge): [1] Monitoring (receive only): [2]`

Automatic retransmission `retransmission`

Retransmission of frames that have lost arbitration or that have been disturbed by errors during transmission.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

CAN FD specification `fd_spec`

Configures the CAN FD specification used by the device. Shall match the specification used by the CAN bus network.

type: `integer`	default: `0`	options: “ ISO CAN FD (11898-1): [0]“ `non-ISO CAN FD (Bosch V1.0. ): [1]`

Bit-rate configuration mode `bit_rate_cfg_mode`

Configures how the CAN bus bit-rate is set. Modes Auto-detect and Bit-rate support all standard bit-rates. Non-standard bit-rate configuration can be set using Bit-timing. It is recommended to set the bit-rate manually if it is known.

type: integer	default: 0	options: Auto-detect: [0] Bit-rate (simple): [1] Bit-timing (advanced): [2]

Configuration explained

This section contains additional information and examples.

Bit-rate configuration

The input clock to the CAN-bus controllers is set to 40MHz (480MHz prescaled by 12).

Bit-rate modes `Auto-detect` and `Bit-rate (simple)` support the following list of bit-rates[1]:

Bitrate	BRP	Quanta	Seg1	Seg2	SJW
5k	100	80	63	16	4
10k	50	80	63	16	4
20k	25	80	63	16	4
33.333k	10	120	95	24	4
47.619k	8	105	83	21	4
50k	10	80	63	16	4
83.333k	4	120	95	24	4
95.238k	4	105	83	21	4
100k	5	80	63	16	4
125k	4	80	63	16	4
250k	2	80	63	16	4
500k	1	80	63	16	4
800k	1	50	39	10	4
1M	1	40	31	8	4
2M	1	20	15	4	4
4M	1	10	7	2	2

In `Auto-detect` mode, the device will attempt to determine the bit-rate from the list of detectable bit-rates. Depending on factors such as data patterns, bit-rate deviation etc. it may not always be possible to detect the bit-rate automatically.

Warning: It is recommended to set the bit-rate manually when possible

Warning: Bit-rate auto-detect cannot be used to detect a CAN FD switched bit-rate

In mode `Bit-timing (advanced)`, the bit-rate timing can be set directly. The following equations can be used to calculate the bit-timing fields:

- Input clock: $CLK = \frac{480000000}{12} = 40000000 = 40\text{MHz}$
- Quanta: $Q = 1 + SEG_1 + SEG_2$
- Bit-rate: $BR = \frac{CLK/BRP}{Q}$
- Sample point: $SP = 100 \cdot \frac{1+SEG_1}{Q}$

Example: Matching bit-timing settings based on different input clock frequency (`CLK`).

Settings to match (based on a 80MHz input clock):

[1] All bit-rate configurations use a sample point (SP) of 80%

- Bit-rate: 2M
- Quanta: 40
- SEG1: 29
- SEG2: 10
- Sample point: 75%

Above settings are based on an input clock with frequency:

$$CLK = BR \cdot Q = 2000000 \cdot 40 = 80\text{MHz}$$

The CANedge uses a 40MHz input clock. To obtain a bit-rate of 2M with a 40MHz input clock, the number of quanta is calculated as:

$$Q = \frac{CLK/BRP}{BR} = \frac{40000000/1}{2000000} = 20$$

To obtain a sampling point of 75%, SEG1 is calcualted as:

$$SEG_1 = \frac{SP \cdot Q}{100} - 1 = \frac{75 \cdot 20}{100} = 14$$

Now, SEG2 is calculated as:

$$SEG_2 = Q - SEG_1 - 1 = 20 - 14 - 1 = 5$$

The equivalent bit-timing settings using the 40 MHz input clock of the CANedge becomes:

- BRP: 1
- SEG1: 14
- SEG2: 5

0.4.5.3 Filter

This page documents the *filter* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

Receive filters " "

Filter remote request frames `properties.remote_frames`

Controls if remote request frames are forwarded to the message filters. If 'Reject' is selected, remote request frames are discarded before they reach the message filters.

type: `integer`	default: `0`	options: `Reject: [0] Accept: [1]`

ID filters `properties.id`

Filters are checked sequentially, execution stops with the first matching filter element. Max 128 11-bit filters and 64 29-bit filters.

type: `array`	minItems: `1`	maxItems: `192`

Name `properties.id.items.properties.name`

Optional filter name.

type: `string`	maxLength: `16`

State `properties.id.items.properties.state`

Disabled filters are ignored.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

Type `properties.id.items.properties.type`

Action on match, accept or reject message.

type: `integer`	default: `0`	options: `Acceptance: [0] Rejection: [1]`

ID format `properties.id.items.properties.id_format`

Filter ID format. Filters apply to messages with matching ID format.

type: `integer`	default: `0`	options: `Standard (11-bit): [0] Extended (29-bit): [1]`

Filter method `properties.id.items.properties.method`

The filter ID matching mechanism.

type: `integer`	default: `0`	options: `Range: [0] Mask: [1]`

From (range) / ID (mask) (HEX) `properties.id.items.properties.f1`

If filter method is Range, this field defines the start of range. If filter method is Mask, this field defines the filter ID.

type: `string`	maxLength: `8`	default: `0`

To (range) / mask (mask) (HEX) `properties.id.items.properties.f2`

If filter method is Range, this field defines the end of range. If filter method is Mask, this field defines the filter mask.

type: `string`	maxLength: `8`	default: `7FF`

Configuration explained

This section contains additional information and examples.

The following uses a mix of binary, decimal and hexadecimal number bases. For more information on the notation used, refer to section *Number bases*.

Note: In the following, it is convenient to do some calculations using binary numbers (base 2). However, the configuration file generally accepts either decimal or hexadecimal numbers.

Filter processing

The filter elements in the list of filters are processed sequentially starting from the first element. Processing stops on the first filter match.

Example: A message matches filter element 3. Filter element 4 is not evaluated.

Messages matching no filters are rejected as default. Note that the default Configuration File has filters that accept all incoming CAN messages (both standard/extended CAN IDs).

Filter state

The *state* of filter elements can be *Enable* or *Disable*. Disabled filter elements are ignored, as if they are not in the list of filters. If there are no enabled filters in the list then all messages are rejected.

By disabling a filter element (instead of deleting the element) it can be easily enabled at a later time.

Filter types

Filter elements can be either *Acceptance* or *Rejection*:

- If a message passes an *Acceptance* filter it is accepted
- If a message passes a *Rejection* filter it is discarded
- If a message does not pass a filter, the next filter in the list is processed

The filter list can hold a combination of *Acceptance* and *Rejection* filter elements. The first matching filter element determines if a message is accepted or rejected. *Acceptance* and *Rejection* filters can be combined to generate a complex message filtering mechanism.

Example: A message matches acceptance filter 3. Rejection filter 4 is not evaluated. The message is accepted.

Example: A message matches rejection filter 2. The following filters are not evaluated. The message is rejected.

Example: A message does not match any filters. The message is rejected.

Filter method

Acceptance and *Rejection* filters can be defined by range or mask. In either case, both the message type (standard / extended) and ID are compared to the filter.

Filter range method

With the *Range* method, the filter defines a range of IDs which are compared to the message ID. Message IDs within the range (both start and end included) pass the filter.

Example: Standard ID filter with range from = 1, to = 10:

ID format	ID (DEC)	Pass
Standard	0	No
Standard	1	Yes
Standard	10	Yes
Standard	11	No
Extended	1	No

Filter mask method

With the *Mask* method, the filter defines an ID and Mask which are compared to the message ID.

A message passes a mask filter if the following condition is true[1]:

```
filter_id & filter_mask == message_id & filter_mask
```

Below a few examples to demonstrate the use of filters.

Example: Filter configuration which accepts one specific message ID: $2000 * 10 = 11111010000_2$. The filter ID is set to the value of the message ID to accept. The filter mask is set to all ones, such that all bits of the filter are considered, as given in (1).

$$\begin{array}{ll} \text{Filter ID} & 11111010000_2 \\ \text{Filter mask} & \underline{\&11111111111_2} \\ \text{Masked filter} & 11111010000_2 \end{array} \quad (1) \qquad \begin{array}{ll} \text{Message ID} & 11111010000_2 \\ \text{Filter mask} & \underline{\&11111111111_2} \\ \text{Masked ID} & 11111010000_2 \end{array} \quad (2)$$

To test if the message passes the filter, we apply the filter mask to the message ID as given in (2). The masked filter and the masked ID are equal - the message passes the filter.

Example: Filter configuration which accepts two message IDs:

- $2000_{10} = 11111010000_2$
- $2001_{10} = 11111010001_2$

Note that the two binary numbers are identical except for the rightmost bit. To design a filter which accepts both IDs, we can use the mask field to mask out the rightmost bit - such that it is not considered

[1] `&` is used as the bitwise AND operation

when the filter is applied. In (1) the mask is set such that the rightmost bit is not considered (indicated by red color).

Filter ID	11111010000_2		Message ID	11111010001_2	
Filter mask	$\&11111111110_2$	(1)	Filter mask	$\&11111111110_2$	(2)
Masked filter	11111010000_2		Masked ID	11111010000_2	

To test if the messages pass the filter, we apply the mask to the message ID 11111010001_2 as given in (2). The masked filter and the masked ID are equal - the message passes the filter. Note that both 11111010000_2 and 11111010001_2 passes the filter, as the rightmost bit is not considered by the filter (the rightmost bit is masked out).

Example: J1939 - filter configuration which accepts PGN 61444 (EEC1) messages.

J1939 message frames use 29-bit CAN-IDs. The Parameter Group Number (PGN) is defined by 18 of the 29 bits. The remaining 11 bits define the priority and source address of the message. It is often useful to configure a filter to accept a specific PGN regardless of the source address and the priority - this can be done using the filter mask (to ignore the source and priority).

Below, the left red bits represent the 3-bit priority, the green bits the 18-bit PGN and the right red bits the 8-bit source address of the 29-bit CAN-ID.

$$00011111111111111111100000000_2 = 3FFFF00_{16}$$

Message ID bits in positions with zero bits in the filter mask are ignored. By using $3FFFF00_{16}$ as filter mask, the source and priority are ignored.

To specifically accept PGN 61444 $F004_{16}$ messages, the message ID is set to $F00400_{16}$ - note the the final 8-bit 00_{16} represents the source address which is ignored by the filter mask (these bits can be set to any value).

Filter mask $3FFFF00_{16}$ can be used for all J1939 PGN messages. To accept specific PGNs, the message ID is adjusted. To accept one specific PGN (as in the example above), the message ID is set to the specific PGN with 00_{16} appended to represent the ignored source address field.

Filter list examples

Below examples demonstate how filters can be combined into a list of filters.

Example: The filter list is setup to accept standard messages with **even** IDs in range $500_{10} - 1000_{10}$ (500, 502, ... 998, 1000):

The following two filters are used to construct the wanted filter mechanism:

- Rejection filter which rejects all odd message IDs
- Acceptance filter which accepts all message IDs in range $500_{10} - 1000_{10}$

The rejection filter is setup to reject all odd messages by using *Mask* filtering. The filter is setup with:

- Filter ID: $1_{10} = 00000000001_2$
- Filter Mask: $1_{10} = 00000000001_2$

Above rejection filter rejects all messages with the rightmost bit set (all odd IDs).

The acceptance filter is setup to accept all messages in range $500_{10} - 1000_{10}$ by using *Range* filtering. The filter is setup with:

- Filter from: 500_{10}
- Filter to: 1000_{10}

The filter list is constructed with the rejection filter first, followed by the acceptance filter.

Note that messages are first processed by the rejection filter (rejects all odd messages), then proccessed by the acceptance filter (accepts all message in range). If none of the filters pass/match, the default behavior is to reject the message. It is in this case important that the rejection filter is placed before the acceptance filter in the list (processing stops on first match).

Filter list test table:

Message ID	Filter elm 1	Filter elm 2	Result
498_{10}	Ignore	Ignore	Reject
499_{10}	Reject		Reject
500_{10}	Ignore	Accept	Accept
501_{10}	Reject		Reject
999_{10}	Reject		Reject
1000_{10}	Ignore	Accept	Accept
1001_{10}	Reject		Reject
1002_{10}	Ignore	Ignore	Reject

Message Prescaling

Message prescaling can be used to decrease the number of logged messages for a given message ID. Prescaling is applied to the messages accepted by the associated filter. The list of filters can be assigned a mixture of prescaler types.

Applying filters can dramatically reduce log file size, resulting in prolonged offline logging and reduced data transfer time and size.

The prescaling type can be set to:

- `None`: Disables prescaling
- `Count`: Prescales based on the number of messages
- `Time`: Prescales based on message period time
- `Data`: Prescales based on changes in the message data payload

The first message with a given ID is always accepted regardless of prescaling type.

Note: A maximum of 100 unique message IDs can be prescaled for each CAN-bus channel (the first 100 IDs received by the device). Additional unique IDs are not prescaled

Count

Count prescaling reduces the number of messages with a specific ID by a constant factor (prescaling value). A prescaling value of 2 accepts every 2nd message (with a specific ID), a value of 3 every 3rd and so on up to 256[2].

Count prescaling applied to ID 600_{10} with a scaling value of 3

[2] A scaling factor of 1 effectively disables prescaling

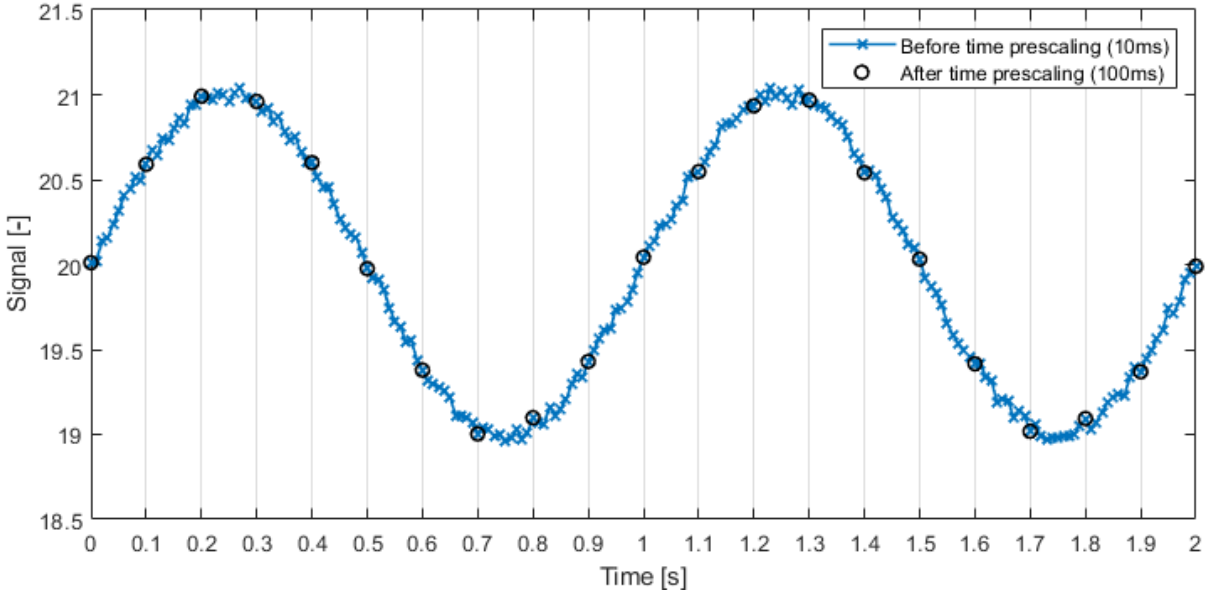
ID (DEC)	ID occurrences	Result
600_{10}	1	Accept
600_{10}	2	Reject
600_{10}	3	Reject
600_{10}	4	Accept
600_{10}	5	Reject

Time

Time prescaling sets a lower limit on time interval (period time) of a specific message ID. This is done by rejecting messages until at least the prescaler time has elapsed[3]. The prescaler timer is reset each time a message is accepted. The prescaling value is set in milliseconds[4] with a valid range 1-4194304 (0x400000).

This prescaler type is e.g. useful if a slowly changing signal (low frequency signal content) is broadcasted on the CAN-bus at a high frequency[5].

Example: A slowly changing temperature measurement broadcasted every 10 ms (100Hz). Prescaled to a minimum time interval of 100ms (prescaler value set to 100).



Example: Time prescaling applied to ID 700_{10} with a time interval of 1000ms selected.

ID (DEC)	Message timestamp [ms]	Prescaler timer [ms]	Result
700_{10}	**200**	0	Accept
700_{10}	700	500	Reject
700_{10}	1000	800	Reject
700_{10}	**1200**	1000 -> 0 (reset)	Accept
700_{10}	1300	100	Reject
700_{10}	**3200**	2000 -> 0 (reset)	Accept
700_{10}	**4200**	1000 -> 0 (reset)	Accept
700_{10}	**5200**	1000 -> 0 (reset)	Accept

[3] Note that messages are not *resampled* to a specific fixed period time
[4] It is not possible to do sub-millisecond time prescaling
[5] Higher frequency than needed to get a good representation of the signal content

Data

Data prescaling can be used to only accept messages when the data payload changes. A mask can be set to only consider changes in one or more specific data bytes. The mask works on a byte level. The mask is entered in hex up to 8 bytes long (16 hex characters). Each byte contains 8 bits, allowing for the mask to be applied to any of the maximum 64 data bytes (CAN FD).

This prescaler type is useful if only changes in data or parts of the data are to be logged.

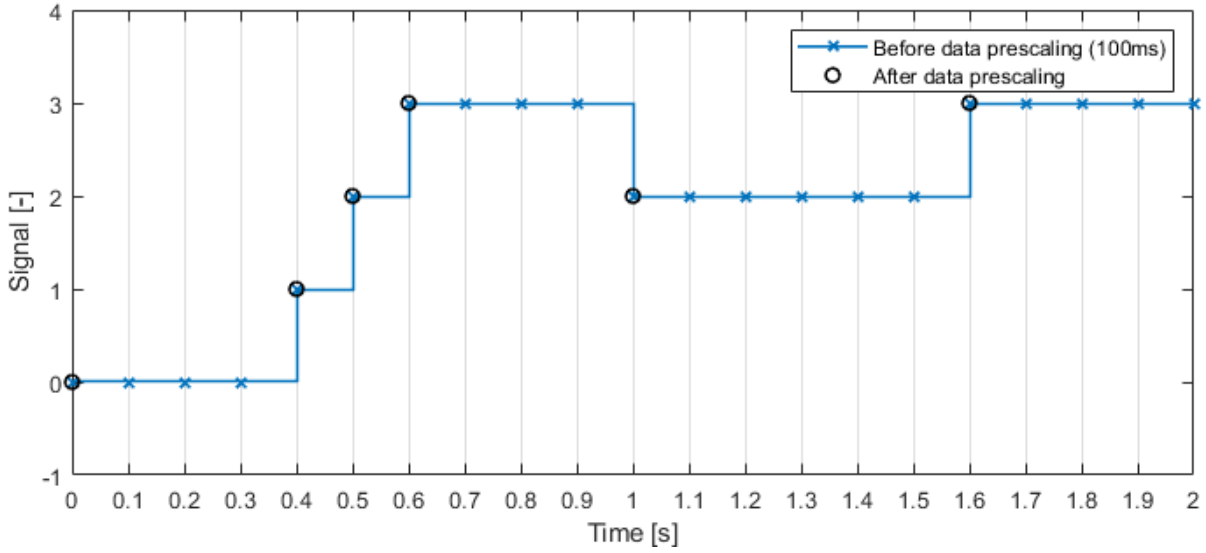
Examples of data masks:

- `""`: A empty mask triggers on any data change (equivalent to mask value `FFFFFFFFFFFFFFFF`)
- `1`: Triggers on changes to the first data byte (binary `1`)
- `2`: Triggers on changes to the second data byte (binary `10`)
- `3`: Triggers on changes to the first or second data byte (binary `11`)
- `9`: Triggers on changes to the first or fourth data byte (binary `1001`)
- `FF`: Triggers on changes to any of the first 8 data bytes (binary `11111111`)
- `100`: Triggers on changes to the 9th data byte (binary `100000000`)

If the data payload contains more data bytes than entered in the mask, then changes to the additional bytes are ignored by the prescaler.

Warning: Data prescaling assumes that a message with a specific ID always carries the same number of data bytes

Example: A discretely changing signal is broadcasted every 100 ms (10Hz). A data prescaler is used such that only changes in the signal are logged.



Example: Data prescaling applied to ID 800_{10} with empty mask (all changes considered). D0-D3 is a 4-byte payload (with D0 the first data byte).

ID (DEC)	D0	D1	D2	D3	Result
800_{10}	00	11	22	33	Accept
800_{10}	00	11	22	33	Reject
800_{10}	00	**BB**	22	33	Accept
800_{10}	**AA**	BB	22	33	Accept
800_{10}	AA	BB	22	**DD**	Accept
800_{10}	AA	BB	22	DD	Reject

Example: Data prescaling applied to ID 800_{10} with mask `1` (considering only changes to the 1st data byte). D0-D3 is a 4-byte payload (with D0 the first data byte).

ID (DEC)	D0	D1	D2	D3	Result
800_{10}	00	11	22	33	Accept
800_{10}	00	11	22	33	Reject
800_{10}	00	**BB**	22	33	Reject
800_{10}	**AA**	BB	22	33	Accept
800_{10}	AA	BB	22	**DD**	Reject
800_{10}	AA	BB	22	DD	Reject

Example: Data prescaling applied to ID 800_{10} with mask `8` (considering only changes to the 4th data byte). D0-D3 is a 4-byte payload (with D0 the first data byte).

ID (DEC)	D0	D1	D2	D3	Result
800_{10}	00	11	22	33	Accept
800_{10}	00	11	22	33	Reject
800_{10}	00	**BB**	22	33	Reject
800_{10}	**AA**	BB	22	33	Reject
800_{10}	AA	BB	22	**DD**	Accept
800_{10}	AA	BB	22	DD	Reject

Example: Data prescaling applied to ID 800_{10} with mask `9` (considering only changes to the 1st or 4th data byte). D0-D3 is a 4-byte payload (with D0 the first data byte).

ID (DEC)	D0	D1	D2	D3	Result
800_{10}	00	11	22	33	Accept
800_{10}	00	11	22	33	Reject
800_{10}	00	**BB**	22	33	Reject
800_{10}	**AA**	BB	22	33	Accept
800_{10}	AA	BB	22	**DD**	Accept
800_{10}	AA	BB	22	DD	Reject

0.4.5.4 Transmit

This page documents the *transmit* configuration.

Configuration file fields

This section is autogenerated from the Rule Schema file.

Transmit messages " "

List of CAN bus messages transmitted by the device. Requires a CAN-bus physical mode supporting transmissions.

type: `array`	maxItems: `64`

Name `items.properties.name`

Optional transmit message name.

type: `string`	maxLength: `16`

State `items.properties.state`

Disabled transmit messages are ignored.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

ID Format `items.properties.id_format`

ID format of the transmit message.

type: `integer`	default: `0`	oneOf: `[{'type': 'integer', 'title': 'Standard (11-bit)', 'enum': [0]}, {'type': 'integer', 'title': 'Extended (29-bit)', 'enum': [1]}]`

Frame format `items.properties.frame_format`

Frame format of the transmit message.

type: `integer`	default: `0`	options: `Standard: [0] Standard RTR: [2] FD: [1]`

Bit-Rate Switch `items.properties.brs`

Determines if an FD message is transmitted using a switched bit-rate.

type: `integer`	default: `0`

Include in log `items.properties.log`

Determines if the transmitted message is included in the log file.

type: `integer`	default: `0`	options: `Disable: [0] Enable: [1]`

Period (10 ms steps) `items.properties.period`

Time period of the message transmission. 0: single shot, >0: periodic. Unit is ms.

type: `integer`	minimum: `0`	maximum: `4294967290`	multipleOf: `10`

Delay (10 ms steps) `items.properties.delay`

Offset message within the period or delay a single shot message. If multiple messages are transmitted by the device, it is recommended to offset each separately to reduce peak load on bus. If period > 0, delay < period. If single-shot, delay can be up to max value. Unit is ms.

type: `integer`	minimum: `0`	maximum: `4294967290`	multipleOf: `10`

Message ID (hex) `items.properties.id`

ID of message to transmit in hex. Example: 1FF.

type: `string`

Messages Data (hex) `items.properties.data`

Data bytes of message to transmit. RTR frames only use the number of bytes do determine the DLC. Example: 01020304 or 0102030405060708.

type: `string`	maxLength: `128`

Configuration explained

This section contains additional information and examples.

Period and delay

If multiple transmit messages are defined, it is recommended to spread them in time by using *delay*. It may not be possible to transmit all messages if they are to be transmitted simultaneously.

0.4.5.5 Heartbeat

This page documents the *heartbeat* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

State `state`

Enable to periodically transmit heartbeat signal.

type: `integer`	default: `0`	options: `Disable: [0] Enable: [1]`

ID Format `id_format`

ID format of heartbeat message.

type: `integer`	default: `1`	oneOf: `[{'type': 'integer', 'title': 'Standard (11-bit)', 'enum': [0]}, {'type': 'integer', 'title': 'Extended (29-bit)', 'enum': [1]}]`

ID (hex) `id`

ID of heartbeat message in hex. Example: 1FF.

type: `string`	default: `00435353`

Configuration explained

This section contains additional information and examples.

Note: The heartbeat cannot be disabled using the control signal

Note: The heartbeat feature requires a CAN-bus physical mode supporting transmissions

Payload format

The device can transmit a 1-second periodic heartbeat signal. The signal payload contains logging state (enabled/disabled), the device time and space left on the memory card in MB.

The interpretation of the 8-byte data payload of the heartbeat signal is given below:

Byte No.	0	1	2-5	6-7
Interpretation	Fixed 0xAA	State	Epoch time	Space left

- Byte 0 has the reserved value 0xAA
- The Epoch time is time-zone and offset adjusted
- Multi-byte fields should be interpreted MSB (Most-SignificantByte) first
- The `State` holds information on the current `rx_state` / `tx_state`:
  - 0: RX disabled, TX disabled
  - 1: RX enabled, TX disabled
  - 2: RX disabled, TX enabled
  - 3: RX enabled, TX enabled

Heartbeat with payload: `AA 03 5D 78 FB 8B 1D 93`

Byte No.	0	1	2-5	6-7
Interpretation	Fixed	State	Epoch time	Space left
Payload	0xAA	0x03	0x5D78FB8B	0x1D93

- Fixed: 0xAA
- State: RX and TX enabled
- Epoch time: $5D78FB8B_{16} = 1568209803_{10}$ -> 11/09/2019 13:50:03
- Space left: $1D93_{16} = 7571_{10}$ MB

Heartbeat with payload: `AA 00 5D 78 FB 8B 00 00`

Byte No.	0	1	2-5	6-7
Interpretation	Fixed	State	Epoch time	Space left
Payload	0xAA	0x00	0x5D78FB8B	0x0000

- Fixed: 0xAA
- State: RX and TX disabled
- Epoch time: $5D78FB8B_{16} = 1568209803_{10}$ -> 11/09/2019 13:50:03
- Space left: $0000_{16} = 0_{10}$ MB

0.4.5.6 Control

This page documents the *control* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

Control ‘‘ ‘‘

Control signal

Control reception (rx) state `properties.control_rx_state`

Control CAN-bus reception state (including logging)

type: `integer`	default: `0`	options: `Disable: [0] Enable: [1]`

Control transmission (tx) state `properties.control_tx_state`

Control CAN-bus transmission state (including logging)

type: `integer`	default: `0`	options: `Disable: [0] Enable: [1]`

Start signal `properties.start`

$ref: `#/definitions/control_signal`

Stop signal `properties.stop`

$ref: `#/definitions/control_signal`

Control signal object `properties.start, properties.stop`

ID Format `properties.id_format`

ID format of the control message.

type: `integer`	default: `1`	oneOf: `[{'type': 'integer', 'title': 'Standard (11-bit)', 'enum': [0]}, {'type': 'integer', 'title': 'Extended (29-bit)', 'enum': [1]}]`

Message ID (hex) `properties.id`

ID of the control message in hex. Example: 00435354.

default: `00435354`	type: `string`

Message ID mask (hex) `properties.id_mask`

ID mask of the control message in hex. Example: 1FFFFFFF.

type: `string`	default: `1FFFFFFF`

Data mask (hex) `properties.data_mask`

Data trigger mask (byte 0 to the left). Shall match the length of the control signal message.

type: `string`	maxLength: `16`	default: `FFFFFFFFFFFFFFFF`

Data trigger high (hex) `properties.data_high`

Data trigger high range (byte 0 to the left)

type: `string`	maxLength: `16`	default: `0100000000000000`

Data trigger low (hex) `properties.data_low`

Data trigger low range (byte 0 to the left)

type: `string`	maxLength: `16`	default: `0000000000000000`

Configuration explained

This section contains additional information and examples.

The control signal has a flexible configuration allowing for integration with many protocols. The control signal can be set to control the device message reception (effectively the logging) and / or the transmission (effectively the processing of the transmit list). The control signal can e.g. be used to start / stop logging based on some application parameters, such as speed, RPM or discrete events.

Control signal overview:

- A control signal can be setup for each CAN-bus channel
- One message ID is used for start and one for stop. These can be different or the same
- The message ID can be masked to trigger on a partial ID match (this allows for e.g. J1939 messages with unknown source)
- The data trigger can be masked to support CAN-bus frames with multiple signals
- The date trigger uses a high/low range, such that any value in the range will trigger (this allows for e.g. stop logging when speed is in range 0-10 km/h)
- File splitting is not affected by the control signal (i.e. the control signal does not force additional log file splits)

Note: The control signal can only be used if accepted by the CAN-bus filter

Note: The control signal data byte fields must match in length vs. the message data

Warning: The start/stop ranges shall be mutual exclusive (cannot both occur at the same time)

Examples

Example: Start / stop on OBD speed response message. The OBD CAN-bus is connected to two CAN-bus channels. *Channel 1* is configured to start / stop message reception based on the OBD speed response. *Channel 2* does not use the control signal feature.

Start trigger:

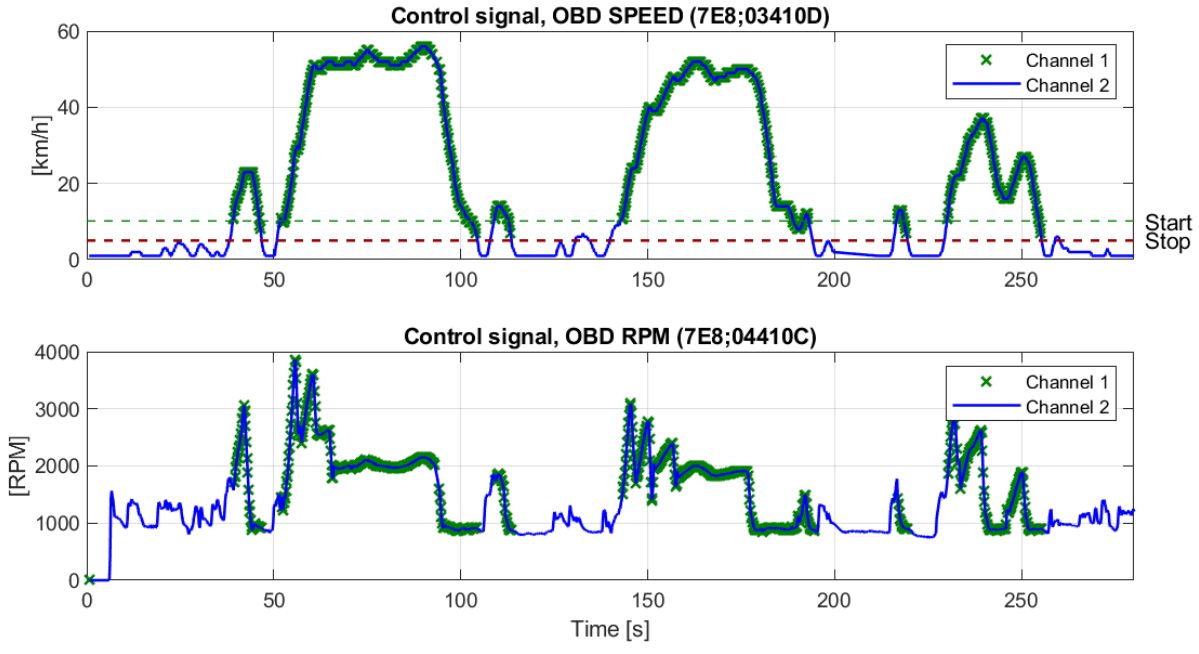
- High: $03410DFF00000000_{16} \Rightarrow 255$ km/h
- Low: $03410D0A00000000_{16} \Rightarrow 10$ km/h

Stop trigger:

- High: $03410D0500000000_{16} \Rightarrow 5$ km/h
- Low: $03410D0000000000_{16} \Rightarrow 0$ km/h

```
"start": {
  "id_format": 0,
  "id": "7E8",
  "id_mask": "7FF",
  "data_mask": "FFFFFFFF00000000",
  "data_high": "03410DFF00000000",
  "data_low": "03410D0A00000000"
},
"stop": {
  "id_format": 0,
  "id": "7E8",
  "id_mask": "7FF",
  "data_mask": "FFFFFFFF00000000",
  "data_high": "03410D0500000000",
  "data_low": "03410D0000000000"
}
```

Below plot illustrates that *Channel 1* message reception (and logging) starts when the speed signal goes above 10 km/h and stops below 5 km/h. *Channel 2* is shown for reference (no control signal).



Example: Start / stop on simple discrete value. In this example the start / stop is controlled via a single byte.

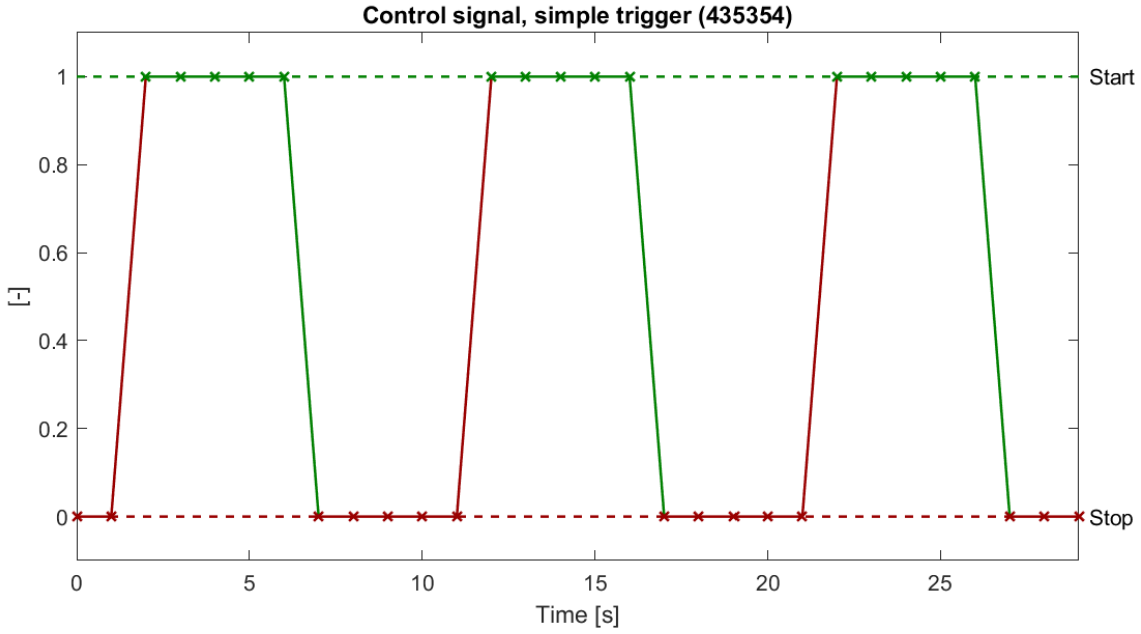
Start trigger:

- High: 01_{16} ⇒ on
- Low: 01_{16} ⇒ on

Stop trigger:

- High: 00_{16} ⇒ off
- Low: 00_{16} ⇒ off

```
"start": {
  "id_format": 1,
  "id": "00435354",
  "id_mask": "1FFFFFFF",
  "data_mask": "FF",
  "data_high": "01",
  "data_low": "01"
},
"stop": {
  "id_format": 1,
  "id": "00435354",
  "id_mask": "1FFFFFFF",
  "data_mask": "FF",
  "data_high": "00",
  "data_low": "00"
}
```



Example: Start / stop on J1939 broadcast speed. This can be used to start logging when a vehicle is moving.

Start trigger:

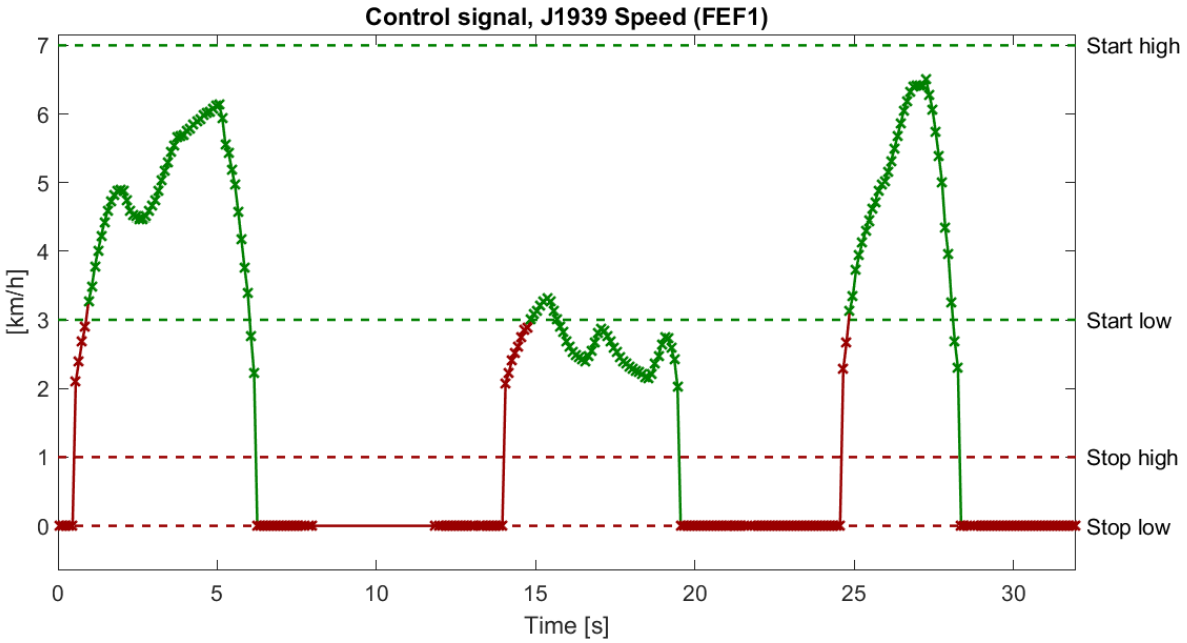
- High: 0000070000000000_{16} ⇒ 7 km/h
- Low: 0000030000000000_{16} ⇒ 3 km/h

Stop trigger:

- High: 0000010000000000_{16} ⇒ 1 km/h

- Low: $0000000000000000_{16} \Rightarrow 0$ km/h

```
"control": {
  "start": {
    "id_format": 1,
    "id": "CFEF100",
    "id_mask": "3FFFF00",
    "data_mask": "0000FF0000000000",
    "data_high": "0000070000000000",
    "data_low":  "0000030000000000"
  },
  "stop": {
    "id_format": 1,
    "id": "CFEF100",
    "id_mask": "3FFFF00",
    "data_mask": "0000FF0000000000",
    "data_high": "0000010000000000",
    "data_low":  "0000000000000000"
  }
}
```



0.4.6 LIN

The configuration of LIN Channel 1 and LIN Channel 2 is identical.

The LIN configuration is split into the following sections:

0.4.6.1 Physical

This page documents the *physical* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

Mode `mode`

Device LIN bus mode.

type: `integer`	default: `0`	options: `Subscriber: [0] Publisher: [1]`

Bit-rate `bit_rate`

type: `integer`	default: `19200`	options: `2400: [2400] 9600: [9600] 10400: [10400] 19200: [19200]`

Configuration explained

This section contains additional information and examples.

0.4.6.2 Frame Table

This page documents the *frame table* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

Name `name`

Optional frame name.

type: `string`	maxLength: `16`

Frame ID (hex) `id`

ID of frame in hex. Example: 0F.

type: `string`	maxLength: `2`

Frame Length (decimal) `length`

Length of the frame in decimal.

type: `integer`	minimum: `1`	maximum: `8`

Checksum Type `checksum_type`

Type of the checksum used on the LIN frame.

type: `integer`	default: `0`	options: `Enhanced: [0] Classic: [1]`

Configuration explained

This section contains additional information and examples.

The LIN controller expects default data lengths and checksums as explained in *LIN*. LIN-frames using a different configuration (length, checksum or both) can be explicitly configured using the *frame table*.

Note: LIN frames satisfying the default expected configuration do not need to be inserted in the *frame table*

0.4.6.3 Transmit

This page documents the *transmit* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

Name `name`

Optional transmit rule name.

type: `string`	maxLength: `16`

State `state`

Disabled transmit rules are ignored.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

Frame ID (hex) `id`

type: `string`	maxLength: `2`

Data (hex) `data`

type: `string`	maxLength: `16`

Configuration explained

This section contains additional information and examples.

The interpretation of the *transmit list* depends on the configuration of *LIN bus mode*:

Publisher mode

The number of bytes entered in the `data` field determines the interpretation of the transmission frame:

Length of `data` is zero

The transmit is a *SUBSCRIBE* frame, meaning that a *Subscriber* on the bus is expected to provide the data payload (satisfying the *frame table*).

Length of `data` is above zero

The transmit is a *PUBLISH* frame, meaning that the CANedge provides the data payload.

In *Publisher* mode, the CANedge schedules the frame transmissions configured by the `period` and `delay`.

Warning: Be aware that transmit uses `period` and `delay` to schedule transmissions. This is a different concept than what is used by *LDF* files.

Subscriber mode

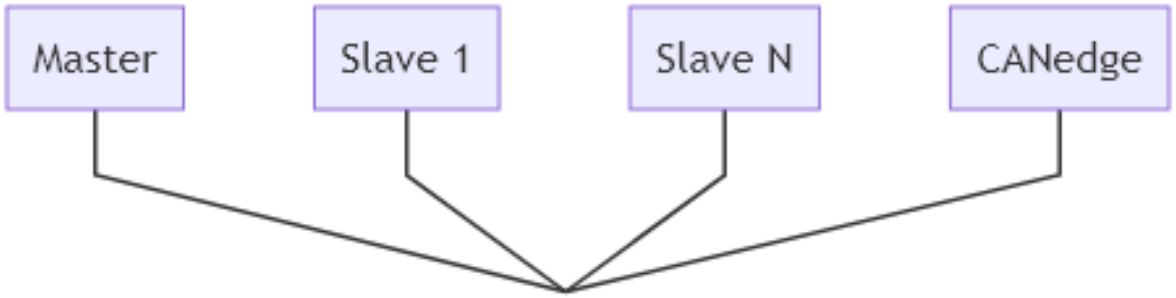
In *Subscriber* mode, the CANedge awaits a *SUBSCRIBE* frame with a matching ID from the bus *Publisher* node. The number of bytes provided shall satisfy the *frame table*.

Warning: If the transmit list contains multiple frames using the same ID, then only the first entry is used.

0.4.6.4 Topology

The LIN master's task is to schedule transmissions. The LIN slaves react to the messages scheduled by the master. The configuration of the LIN network shall ensure that each message has one producer.

The CANedge connects as a slave to the LIN-bus. A master node is required to schedule the transmissions. Below illustrates how the CANedge can be connected to the bus:



0.4.6.5 Data length

Unless configured otherwise, the device assumes that the length of the LIN frame data payload is always defined by the message ID (bits 5 and 6 of the identifier), as defined in the table below:

Message ID	Data length
00-31 (0x00-0x1F)	2
32-47 (0x20-0x2F)	4
48-63 (0x30-0x3F)	8

0.4.6.6 Checksum

Supports LIN 1.3 *classic* checksum and LIN 2.0 *enhanced* checksum format. By default, all frames except ID 0x3C and 0x3D will use enhanced checksum. This can be overridden in the configuration for each frame ID.

0.4.7 Connect

This page documents the *Connect* configuration.

The Connection configuration is split into the following sections:

0.4.7.1 WiFi

This page documents the *wifi* configuration

Configuration file fields

This section is autogenerated from the Rule Schema file.

Mode `mode`

The WiFi mode.

type: `integer`	default: `0`	readOnly: `True`	options: `Station mode: [0]`

Key format `keyformat`

The format of the key used for all entries in the access point list. Can be used to hide the WiFi passwords stored on the device.

type: `integer`	default: `0`	options: `Plain: [0] Encrypted: [1]`

Access points `accesspoint`

List of access points. Connects to the first available AP in list.

type: `array`	maxItems: `5`

SSID `accesspoint.items.properties.ssid`

Access point SSID (name of access point)

type: `string`	minLength: `0`	maxLength: `32`

Password `accesspoint.items.properties.pwd`

type: `string`	default: `(blank)`	maxLength: `64`

Minimum RSSI (received signal strength indicator), 0-100% `accesspoint.items.properties.minrssi`

Sets a minimum required access point signal strength. The device will not connect to the access point if the measured signal strength is below the value of this field.

type: `integer`	default: `0`	minimum: `0`	maximum: `100`

Configuration explained

This section contains additional information and examples.

Multiple WiFi access points

The device supports multiple access points. The device will attempt to connect to the access points in the prioritized order in which they are entered in the configuration file. If unable to connect to an access point or if the signal strength is below the minimum RSSI%, the device will attempt the next one from the list. Every time the device initiates a new WiFi connection, it will cycle through the list in this manner, starting from the top.

Using an RSSI% of 0 (default) means that the device will attempt to connect to the AP, even if the signal strength is practically non-existent.

0.4.7.2 S3 server

This page documents the *s3 server* configuration

Note: If a HTTPS (TLS) server `Endpoint` is used, refer to the Security Section for more information on how to setup certificates.

Configuration file fields

This section is autogenerated from the Rule Schema file.

Synchronization `sync`

This section configures how and when the device communicates with the S3 server.

Firmware, config and certificate `sync.properties.ota`

Configures how often the device looks for firmware-, config- and certificate-over-the-air updates. Small values may reduce performance. Time period may sometimes become longer if device is busy. Set to 0 to disable.

type: `integer`	default: `600`	minimum: `0`	maximum: `86400`	multipleOf: `5`

Heartbeat `sync.properties.heartbeat`

Configures how often the device transmits the heartbeat signal. Small values may reduce performance. Time period may sometimes become longer if device is busy. Set to 0 to disable.

type: `integer`	default: `300`	minimum: `0`	maximum: `86400`	multipleOf: `5`

Log files `sync.properties.logfiles`

Configures if the device pushes closed log files to the server. The log files are deleted from the device when successfully uploaded.

type: `integer`	default: `1`	options: `Disable: [0] Enable: [1]`

Server `server`

This section contains the server connection parameters.

Endpoint `server.properties.endpoint`

S3 server endpoint. Prefix with http:// to connect using standard http. Prefix with https:// to connect using SSL/TLS - requires support by the server and that the server certificate is loaded onto the device. Examples: http://192.168.0.1, https://s3.mydomain.com, https://s3.amazonaws.com, http://s3-us-east-2.amazonaws.com.

type: `string`	maxLength: `128`

Port `server.properties.port`

S3 server port. Examples: 80 (http), 443 (https), 9000 (custom).

type: `integer`	minimum: `0`	maximum: `65535`

Bucket name `server.properties.bucket`

S3 server bucket name. Examples: logbucket, fleetbucket, testbucket.

type: `string`	maxLength: `20`

Region `server.properties.region`

S3 server region. Example: us-east-1.

type: `string`	minLength: `0`	maxLength: `32`

Request style `server.properties.request_style`

Virtual-hosted-style or path-style S3 requests. Virtual hosted-style format: "http://[BUCKET-NAME].[DOMAIN]/[OBJECT-NAME]". Path-style format: "http://[DOMAIN]/[BUCKET-NAME]/[OBJECT-NAME]"

type: `integer`	default: `0`	options: `Path-style: [0] Virtual hosted-style: [1]`

AccessKey `server.properties.accesskey`

S3 server access key ID. Example: PRDDKN8R6PAAOGTEI53E

type: `string`	minLength: `3`	maxLength: `128`

SecretKey format `server.properties.keyformat`

The format of the secret key. Can be used to hide the secret key stored on the device.

type: `integer`	default: `0`	options: `Plain: [0] Encrypted: [1]`

SecretKey `server.properties.secretkey`

type: `string`

Signed payload `server.properties.signed_payload`

Include payload checksum in signature. Reduces device upload performance.

type: `integer`	default: `0`	options: `Off: [0] On: [1]`

Configuration explained

This section contains additional information and examples.

Request-style

S3 supports two different request styles *path* and *virtual hosted*. The device supports both styles.

With the virtual hosted style, the subdomain is specific to the bucket, which makes it possible to use DNS to map a specific bucket to an IP address.

Warning: Some S3 servers may only support one of the two request formats.

Path-style http header example:

```
GET /[BUCKET_NAME]/[OBJECT_NAME] HTTP/1.1
Host: [DOMAIN]
...
```

Virtual hosted-style http header example:

```
GET /[OBJECT_NAME] HTTP/1.1
Host: [BUCKET_NAME].[DOMAIN]
...
```

0.5 Device file

A Device File (`device.json`) is located in the root of the SD card with info on the device. The content of the Device File is updated when the device powers on.

```
{
  "id": "4F07A3C3",
  "type": "0000001D",
  "kpub": "l27UKi4ehjpxxEdmRstBk5UaqSGQYnfylzUNs9EOoJfDodvr/
↪PqNnMrz61IxzrBfFTmuhw2K2cJ4q60iFiYM8w==",
  "fw_ver": "01.01.02",
  "hw_ver": "00.01",
  "cfg_ver": "01.01",
  "cfg_name": "config-01.01.json",
  "cfg_crc32": "9ECC0C10",
  "sch_name": "schema-01.01.json",
  "log_meta": "Truck1",
  "space_used_mb": "36/7572"
}
```

Additional content may be added to the `device.json` in future firmware updates.

Fields explained

- `id`: Device unique ID number
- `type`: Device type (CANedge1 = `0000001D`, CANedge2 = `0000001F`)
- `kpub`: Device public key in Base-64 format
- `fw_ver`: Firmware version
- `hw_ver`: Hardware version
- `cfg_ver`: Configuration File version
- `cfg_name`: Configuration File name
- `cfg_crc32`: Configuration File checksum
- `sch_name`: Configuration Rule Schema name
- `log_meta`: Configurable device string (e.g. application name)
- `space_used_mb`: The SD-card used space of the total in MB (`[used]/[total]`)

0.6 Log file

This page documents the log files stored on the device SD-card.

0.6.1 Organization

Log files are organized by the following path structure:

`LOG/[DEVICE_ID]/[SESSION_NUMBER]/[SPLIT_NUMBER].[FILE_EXTENSION]`

The path is constructed from the following parts:

- `LOG`: Static directory name used to store log files
- `DEVICE_ID`: Globally unique device ID
- `SESSION_NUMBER`: Increased by one for each power cycle[1]
- `SPLIT_NUMBER`: Reset to 1 on each power cycle and increased by one for each file split
- `FILE_EXTENSION`: The file extension selected in the configuration (`MF4|MFC|MFE|MFM`)

For details on log file splits and related limits, see the *Logging Configuration* section.

File extension

The default extension is `MF4`. With compression/encryption enabled the extension changes:

Compression enabled	Encryption enabled	File extension
		`.MF4`
X		`.MFC`
	X	`.MFE`
X	X	`.MFM`

With both compression and encryption enabled, the data is first compressed, then encrypted.

For details on compression and encryption, see the *Logging Configuration* section.

Path example

Example: Log file path: `3B912722/00000004/00000189.MF4`

- `3B912722`: The unique ID of the device which generated the log file
- `00000004`: Generated during the 4th session / power cycle
- `00000189`: Is log file number 189 of the session (since power cycle)
- `MF4`: File type

0.6.2 MDF

The CANedge logs data in the industry standard MDF4 format, standardized by *ASAM*. MDF4 is a binary format which allows compact storage of huge amounts of measurement data. It is specifically designed for bus frame logging accross e.g. CAN-bus, LIN-bus and Ethernet. MDF4 is widely adopted by the industry and supported by many existing tools.

Specifically, the CANedge uses MDF version 4.11 (file extension: `*.MF4`).

[1] The session number is also increased by one if the number of splits in one session exceeds 256

Timestamps

Each record is timestamped with 50 us resolution.

Finalization & sorting

The CANedge stores log files as *unfinalized* and *unsorted* to enable power safety. Finalization[3] and sorting[4] can be done as a post-processing step to speed up work with the files.

Note: It may be necessary to finalize/sort a log file before it is loaded into some MDF tools

[3] The MDF file header includes information on how to finalize the MDF file before use
[4] Sorting refers to an organization of the log records which enable fast indexing. It is not related to sorting of timestamps.

0.7 Firmware

0.7.1 Firmware download

See the online documentation for the latest Firmware Files and changelog.

0.7.1.1 Mandatory update steps

Some upgrade steps are mandatory and cannot be skipped. Skipping mandatory update steps can result in unexpected behavior. The mandatory update sequence is illustrated below.

00.07.XX –> 01.02.XX

Warning: Firmware upgrade from 00.07.XX to 01.02.XX includes a one-time boot migration step. The migration step formats the SD-card resulting in loss of existing log files. The configuration and certificates are copied back to the card after migration

Firmware Files can be downloaded from the online documentation.

This page describes how to upgrade the device firmware.

0.7.2 Firmware versioning & naming

The device firmware versioning is inspired by the semantic versioning system.

Each firmware is assigned three two digit numbers: `MAJOR`, `MINOR`, `PATCH`:

- `MAJOR`: Incompatible changes (e.g. requires changes to the Configuration File)
- `MINOR`: New backwards-compatible functionality (e.g. new fields in the Configuration File)
- `PATCH`: Backwards-compatible bug fixes (e.g. no changes to the Configuration File)

The firmware files available for download are zipped with naming as follows:

`firmware-[MAJOR].[MINOR].[PATCH].zip`

Example:

`firmware-01.02.03.zip`

0.7.3 Firmware upgrade

The device supports in-the-field firmware upgrades.

Note: The firmware upgrade process is power safe (tolerates power failures). However, it is recommended to ensure that the process completes

0.7.3.1 Upgrade process

Upgrading initiates when the device is powered and has been prepared with a new Firmware File:

1. Power is applied to device
2. The green LED comes on
3. If the firmware is valid, the green LED blinks 5 times, else the red LED blinks 5 times
4. The green LED remains solid while the firmware is upgraded (~20 sec)
5. If the upgrade succeeds, the green LED blinks 5 times, else the red LED blinks 5 times
6. The upgraded firmware is started

Note: The green LED comes on later than usual when a firmware upgrade is initiated

Note: The device automatically removes any Firmware Files (`firmware.bin` or `firmware_wifi.bin`) when the upgrade has completed. Firmware Files should never be manually deleted during the upgrade process.

0.7.3.2 Configuration update

If a device is updated to a firmware version with a different `MAJOR` or `MINOR` number, then the Configuration File also needs updating (i.e. with an updated name and structure matching the new firmware). The Configuration File is named as described in the *Configuration* section. A default Configuration File and corresponding Rule Schema are contained in the firmware download zip.

To modify an existing Configuration File, it can be useful to load the new Rule Schema in an editor together with the old Configuration File. After making the necessary updates, save the modified Configuration File with a name matching the new version.

Note: The firmware can be upgraded without providing a new compatible Configuration File. In this case, the device will create a default Configuration File on the SD card

0.7.3.3 Upgrade from SD card

The firmware can be upgraded by placing a Firmware File on the SD and powering the device:

1. Download the firmware zip (Firmware File + Configuration File + Rule Schema)
2. Place the `firmware.bin` file on the SD card (root directory)
3. If `MAJOR`/`MINOR` is different, update the Configuration File and place it on the SD card root
4. Power on the device and wait for the upgrade process to complete

Note: An incompatible firmware image is deleted and does not break the device

Example: Current firmware: `01.01.01`, new firmware: `01.01.02`

1. Download `firmware-01.01.02.zip` and unzip it
2. Copy `firmware.bin` to the SD card
3. The `MAJOR` and `MINOR` versions are unchanged (no need to update the Configuration File)
4. Power on the device and wait for the upgrade process to complete

Example: Current firmware: `01.01.00`, new firmware: `01.02.00`

1. Download `firmware-01.02.00.zip` and unzip it
2. Copy `firmware.bin` to the SD card
3. Update the Configuration File (or use the default created by the firmware update)
4. Power on the device and wait for the upgrade process to complete