LANTRONIX®

Application Note

CAN Applications with AVL Devices

Part Number APP-0026
Revision A November 2019

Intellectual Property

Lantronix is a registered trademark of Lantronix, Inc. in the United States and other countries.

Patented: www.lantronix.com/legal/patents/; additional patents pending.

All trademarks and trade names are the property of their respective holders.

Contacts

Lantronix, Inc.
7535 Irvine Center Drive, Suite 100
Irvine, CA 92618, USA
Toll Free: 800-526-8766
Phone: 949-453-3990
Fax: 949-453-3995

Technical Support
Online: www.lantronix.com/support

Sales Offices
For a current list of our domestic and international sales offices, go to the Lantronix web site at www.lantronix.com/about/contact

Disclaimer

All information contained herein is provided “AS IS.” Lantronix undertakes no obligation to update the information in this publication. Lantronix does not make, and specifically disclaims, all warranties of any kind (express, implied or otherwise) regarding title, non-infringement, fitness, quality, accuracy, completeness, usefulness, suitability or performance of the information provided herein. Lantronix shall have no liability whatsoever to any user for any damages, losses and causes of action (whether in contract or in tort or otherwise) in connection with the user’s access or usage of any of the information or content contained herein. The information and specifications contained in this document are subject to change without notice.

Revision History

Date	Rev.	Comments
September 2008	1.0.0	Initial version.
June 2008	1.0.1	Corrected color codes of “Vehicle installation cable" - see table in page 8
January 2009	1.0.2	Improved Figure 1 - typical High & Low signals on CANBus. Added more information about the CAN Bus throughout this document
October 2014	1.0.3	Added hint in chapter 1.2
November 2019	A	Initial Lantronix document. Added Lantronix document part number, logo, contact information, and links.

For the latest revision of this product document, please check our online documentation at www.lantronix.com/support/documentation.

Table of Contents

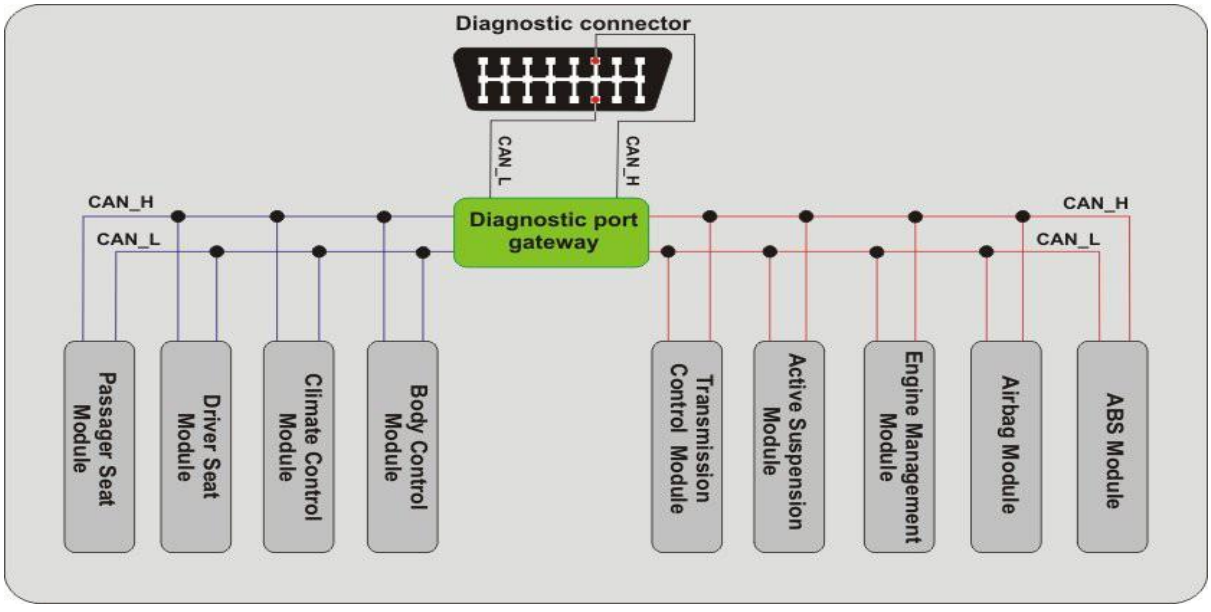
1: About This Document

This application note provides information how to connect your device with CAN Bus option to an external CAN bus and read the data on a CAN bus. The CAN bus can be a vehicle CAN bus on a car or truck. STEPPIII device is used as an example in this application note.

Main feature of the STEPPIII and FOX with CAN bus option is that they directly store received messages with selected CAN message identifiers into a message buffer. The user provides a list of CAN message identifiers that should be received by the CAN interface. The STEPPIII or FOX device automatically scans every incoming CAN message from the CAN bus and when there is an identifier match, the message is copied into the associated receive buffer. The user provides then the position of bits and bytes to be read out of the 8-byte data the identifier provides for finding out states of specific components in-vehicle (e.g. when the doors are locked or opened).The read out values are then stored into different storage slots and when a storage slot changes its value the corresponding event is occurred. With the help of such events you are able to sent these values to a TCP server for evaluating. Starting from the software revision 2.6.1, the STEPPIII and FOX devices have a limit of 25 such storage slots. That means STEPPIII and FOX can catch up to 25 messages identifiers from the CAN bus stream.

Introduction

Controller Area Network (CAN) was initially created for automotive applications. The goal was to make automobiles more reliable and safe. The CAN bus allows multiple devices to be linked together on the same bus. A typical vehicle architecture is illustrated in figure below.



The diagnostic port gateway provides the link between the diagnostic connector and the vehicle networks. In this example, two internal networks are shown, one (in red) associated with safety critical modules such as engine management and the other (in blue) associated with lower priority modules such as body control.

The aim of this application note is to explain some of the basics of CAN and show how to configure your STEPPIII/FOX to read out such specific information on the CAN bus. The CAN bus consists of two-wire data line to which all vehicle components are connected. CAN implements a serial data transmission using the two bus signal levels CAN_High and CAN_Low. CAN, like most modern networks, is serial based. This means that the information travels along the CAN Bus one bit at a time. On the CAN_High a redundant signal is transferred which is inverted compared to CAN_Low line (see figure below). When CAN_High goes high, in the same time, CAN_Low goes low, in the same proportion.

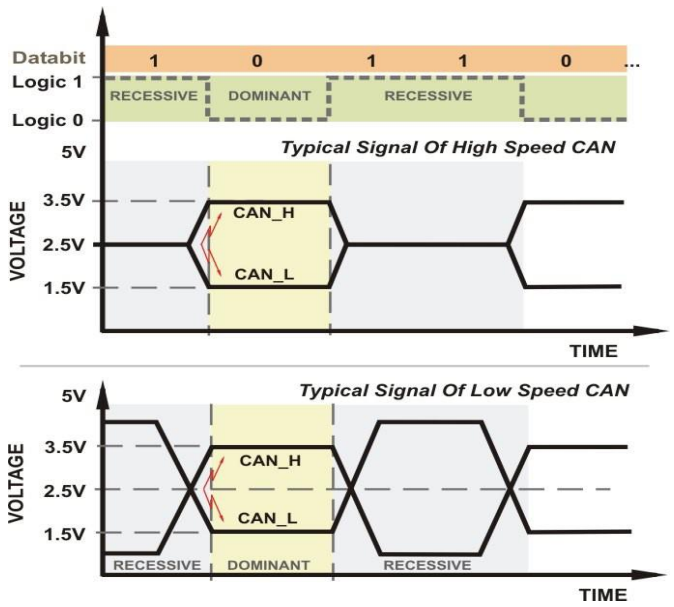


Figure 1: Typical signal of High & Low speed CAN.

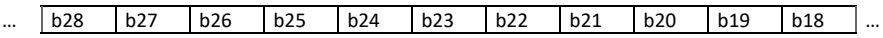
The figure above shows the physical (dominant and recessive) levels with a CAN High and Low speed Transceivers. It is important to know that the voltage levels from each of the two CAN lines to ground or to the vehicle chassis are not the important ones. Important to CAN is the voltage between the two lines or their difference voltage. The recessive bus level (logic "1") is characterized by a difference voltage of 0 V. Both communication lines are on the potential voltage of 2.5 V. With the dominant bus level (logic "0") the CAN_H line accepts a potential voltage of 3.5 V and the CAN_L line of 1.5 V. The difference voltage is 2 V.

Each automotive manufacturer has created his own CAN Protocol. A CAN bus protocol consists of an identifier and up to eight data bytes. The protocol developed by a manufacturer defines what data signals they add to an identifier and how the signal data is organised within the data bytes for each CAN message. The CAN bus connection point on your vehicle can be either behind the radio or under the dashboard. For more information, how to get the identifiers on your vehicle CAN bus and how the data is organised in it as well as the CAN connection point please contact your vehicle manufacturer or your local vendor.

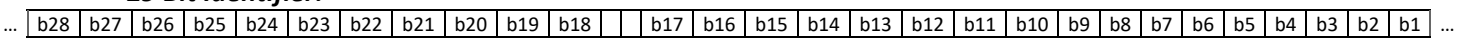
There are two different CAN messages: the standard and the extended message. The only difference between them is that the standard message supports an 11-bits identifier, and the extended one supports a 29-bits identifier, made up of the 11-bit identifier and an 18-bit extension identifier. The distinction between CAN standard and CAN extended is made by using the IDE bit, which is transmitted as dominant in case of an 11-bit message, and transmitted as recessive in case of a 29-bit message. A CAN that supports extended messages are also able to send and receive messages in CAN standard.

The structure of both 11-bits and 29-bits messages is given below:

11 Bit Identifier:



29 Bit Identifier:



↑ ↑

SRR -Substitute RTR bit for 29 bit ID **IDE - Identifier Extension (dominant =11 bit ID, recessive = 29 bit ID)**

A simplified structure of a CAN bus message is shown in table that follow with example identifier and data. In this example, the identifier is a 11-bits identifier with a value of hexadecimal 0x1AC. For example, the 4-th and 5-th data bytes correspond to engine RPM (Revolutions Per Minute), while the 1-th and 2-th data bytes correspond to door lock etc.

	Identifier segment	Data segment							
		0 to 7 bytes							
Example 11 bits	0x1AC	F3	14	1F	99	75	F1	F2	F5
Example 29 bits	0x0FC00860	E1	23	AF	2	55	BC	C1	A1

For more detailed information about the CAN Bus, visit the website: http://en.wikipedia.org/wiki/Controller_Area_Network.

How to Read the Data out of a CAN Bus

For the evaluation of the CAN bus data, a FOX or STEPPIII device with the CAN bus option is required. These devices can be supplied (upon request) with a CAN-bus interface that supports either High-Speed CAN-Bus or Low-Speed CAN-Bus.

The first option **Low-Speed CAN-Bus controller CAN1** (Type TJA1054A) inside the FOX or STEPPIII device is compatible to:

- ✓ ISO 11898-3 (CAN fault-tolerant (low-speed)),
- ✓ Two transmission lines (total termination=100 Ohm across the CAN_H and CAN_L wire - should be measured* at the end of the CAN bus lines in vehicle),
- ✓ SAE J2411 Single-wire CAN (SWC),
- ✓ Supporting a baudrate of up to 125 Kbit/s,
- ✓ for car applications.

** The vehicle key must be off to accurately measure the resistance of the CAN bus.*

The second option **High-Speed CAN-Bus controller CANH** (Type TLE6250) inside the FOX or STEPPIII device is compatible to:

- ✓ ISO 11898-2 (high-speed)
- ✓ Two wire differencial bus
- ✓ The CAN bus must be terminated at both ends by a 120-ohm resistor to prevent signal reflections. Use an ohm meter to check wiring on the vehicle side. With the power off, verify 60-ohm across CAN_H and CAN_L (if only one resistor is installed or no resistor is installed you will read 120-ohm or 0-ohm instead of 60-ohm),
- ✓ Supporting a baudrate of up to 1Mbit/s,
- ✓ for truck applications (CAN Gateway).

Hint: The first option (**Low-Speed CAN-Bus controller CAN1**) can also be used for one wire applications with up to 125 kBit/s bus speed and the second option (**High-Speed CAN-Bus controller CANH**) is designed for use in dual wire applications with up to 1Mbit/s bus speed.

Each of AVL devices provides two CAN bus signal pins CAN_High and CAN_Low. A cable shipped with the device can be used to interface your device with the CAN bus of your vehicle. The polarity of the CAN_High and CAN_Low lines must be observed when connecting your FOX/STEPPIII device to the two-wire CAN lines of your vehicle. Additionally, a manufacture special connector at the end of the shipped cable is required for connecting your FOX/STEPPIII to the CAN bus interface on your vehicle. Please contact your local vendor to get more information.

The CAN interface lines for STEPPIII and FOX:

STEPPIII		
	DI0 (Pin 10 on the 16pin connector)	as CAN_H line
	DI1 (PIN 12 on the 16pin connector)	as CAN_L line
FOX uses on the end of the 8-pin cable:		
	I/O2 (Pin 5 - Yellow - on 8pin connector of cable)	as CAN_L line
	I/O3 (Pin 6 - Green - on 8pin connector of cable)	as CAN_H line

The shipped 16-wires cable called "**Vehicle installation cable**" (for STEPPIII device only, while the FOX device provides an 8pin connector at the end of the external cable - for the pinout of this cable, refer to the "**FOX_EvalKit_Getting_Started.pdf**"), which can be used for in-vehicle installation, has different color codes.

Table below lists the wire colors and their meaning of this cable.

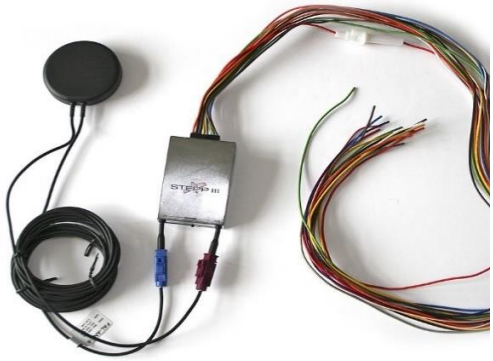


Figure 2: Vehicle mounting cable

To use this cable, first strip off about 2 cm of the outer insulation the end of the wires your application uses, then connect the end with connector to the STEPPIII device and finally, connect the other stripped ends DI0 (grey) to the CAN_High and DI1 (white) to the CAN_Low of the the CAN bus on your vehicle, and when you are sure the CAN connection is properly made, apply power to the device by connecting the GND-pin first and then VCC-pin.

The following table lists the color codes on the Vehicle mounting cable:

WIRE COLOR	NAME	Meaning
Orange - White	VBO	Do not use , leave disconnected.
Orange	IN0	Analog / digital Input (default = analog)
Green - White	GND	-
Lilac	IN1	Analog / digital Input (default = analog)
Brown - White	OUT0	Output
Black	IN2	Analog / digital Input (default = digital)
Yellow - White	OUT1	Output
Yellow	IN3	Analog / digital Input (default = digital)
Red - White	OUT2	Output
Grey	DI0	**CAN_H *(dominant HIGH)***
Black - White	OUT3	Output
White	DI1	**CAN_Low *(dominant LOW)***
Blue	IGN	Digital input
Green	DiWu	Digital input
Red	VCC (2A fuse-protected)	Input voltage (+10.8...35.0V DC)
Brown	GND	-

In order to get specific information from a CAN bus, e.g. RPM information, as trigger or control signal, this information must be extracted from the CAN bus data stream using CAN message identifiers.

Using PFAL commands provided for CAN applications, the user defines the CAN communication baudrate of the CAN Bus in the vehicle and provides a list of CAN message identifiers (up to 25 messages identifiers currently available) that should be received by the CAN interface. When an identifier match is detected, the message is copied into the associated receive buffer. To find out states that an identifier provides (e.g. when the doors are locked or opened), the user have to provide the position of bits and bytes to be read out of the 8-byte data. The read out values are then automatically stored into different storage slots inside the device and when a storage slot changes its value the corresponding event is occurred. These events can be used to send out these values to a TCP server for further evaluation. On the remote server the received data may be graphically displayed to show e.g. vehicle diagnostics, Ignition, Door locks, Windows state etc.

In the table below are given some configuration settings to demonstrate how your STEPPIII or FOX device can be configured to get out information from a Low speed CAN bus and to send it to a remote server. For High speed CAN application, just replace the entry "**std**" in the

*"$PFAL,Sys.CAN.Msg.Add,**std**,xxx"* and *"$PFAL,Sys.Can.Var.Add,0,number,event,**std**,xxx,x,x,x,x,xxx"* by "***ext***".

How to add the required message identifiers, read out their data and send values to a remote server when they change?	
Command syntax	**$PFAL,*Sys.Can.Enable,*<*baudrate*>,<*mode*>**
Description	STEPPIII and FOX devices use the command above to activate the **CAN** interface. **CAN** interface will be enabled after rebooting the system.
Command syntax	**$PFAL,*Sys.CAN.Msg.Add,*<*type*>,<*identifier*>[,<*mask*>]**
Description	STEPPIII and FOX devices use the command above to add a CAN message identifier into the buffer that should be received by the CAN interface. The *CAN-Bus controller* built-in the device automatically scans every incoming CAN message from the bus and when there is an identifier match, the message is copied into the associated receive buffer. To read the data stored in this buffer identifier use the command "**$PFAL,Sys.CAN.Var.Add..**.".The <*mask*> entry is optional and can be used when you don't know exactly the message identifier, otherwise leave it empty (if used, please refer to the examples "**How to use <*mask*> in this command"**). In all examples in this table the <*mask*> entry is not used. **Hint:** If you want to read 10 different message identifiers (e.g. door lock, gear direction, vehicle speed, RPM etc.) you have to execute this command 10 times with corresponding identifier. If a message identifier provides 2 different information (e.g. door lock, door state) and you need both information, then first execute this command just one time and then execute the command "**$PFAL,Sys.CAN.Var.Add..."** 2 times by defining the same <*msg_identifier*> and different byte and bit order (see example with "ID=**39F"** below). **How to use <*mask*> in this command:** **Example 1 - for standard CAN:** - Let's assume that following message identifiers are on the CAN Bus data steam: A: id=19A; B: id=290; C: id=39E; D: d=89F; E: id=6F9; F: d=19F; G: d=392; - The PFAL command below with "id=**39B**" and "mask=**FFFFFFF0**" **$PFAL,Sys.CAN.Msg.Add,std,39B,FFFFFFF0 //** receives all message identifiers on the CAN bus (in this example all 3-bytes messages) starting with ID "**39**" (i.e. **390** to **39F**) and filters out all other messages. - messages **C** and **G** would be received, because they start with "**39**" and the last 4 bits are don't care, - all other messages **A, B**, **D**, **E** and **F** would be filtered out because they don't start with "**39**" (the first 2 digits in these message IDs do not match the required '**39**'). **Example 2 - for extended CAN:** - The PFAL command below with "id=**00FEF1**" and "mask=**FFFFFF00**" **$PFAL,Sys.CAN.Msg.Add,ext,00FEF1,FFFFFF00 //** receives all message identifiers on the CAN bus (in this example all 6-bytes messages) starting with "**00FE**" (i.e. "**00FE00"** to "**00FEFF"**) and filters out all other messages.

Command syntax	**$PFAL,*Sys.CAN.Var.Add,*** *<variable_slot>,<variable_type>,<notification>,<msg_type>,<msg_identifier>,<start_byte>,<start_bit>,<stop_byte>,<stop_bit>,<byte_order>*
Description	STEPPIII and FOX devices use the command above to read out specific data attached to a CAN message identifier (e.g. ***211*** *provides information about the door lock). The <byte_order> defines how the data bytes should be read,* ***MSB*** *(the most significant byte) is always on the left and* ***LSB*** *(the least significant byte) on the right.*

Let's see an example how to configure your device to read the data out of a standard CAN bus :

Let's suppose you want to read the data attached to the following identifiers in a standard CAN message (11-bits) and then send this data out to a remote server when their value changes:

Identifiers ***Data added to it***

211 *and* ***e.g. byte 0 (MSB)*** *provides information about the door lock.*

39F *and* ***e.g. byte 0 (MSB)*** *provides information about the gear direction.*

39F *and* ***e.g. bytes 1*** *and* ***2 (LSB)*** *provide information about the vehicle speed.*

15B *and* ***e.g. bytes 1*** *and* ***2 (LSB)*** *provide information about the RPM.*

479 *and* ***e.g. bytes 6*** *and* ***5 (MSB)*** *provide information about the door state.*

65A *and* ***e.g. bytes 1, 2*** *and* ***3 (LSB)*** *provide information about the milage.*

Steps to be done:

1) - First, activate the CAN interface and define the baudrate the CAN bus uses (e.g. **$PFAL,Sys.Can.Enable,100K,RO**)

// by default the baudrate is set to ***100K*** *(for Low-Speed CAN bus option) and* ***250K*** *(for High Speed CAN Bus option).*

2) - Thereafter, add a message ID into the associated buffer (e.g. **$PFAL,Sys.CAN.Msg.Add,std,211**) ***//std=****standard CAN;*
ext=*extended CAN;*

3) - Read out the data attached to the message identifier "**211**" by specifying bits, bytes and byte order containing the required data *(e.g.* **$PFAL,Sys.Can.Var.Add,0,number,event,std,211,0,0,0,7,MSB**)

4) - Finally, configure an alarm that sends out the contents of the ***Slot0*** via TCP when its contents changes (*door lock changes its value)* (e.g. **$PFAL,CNF.Set,AL45=SYS.Can.e0:TCP.Client.Send,8,"<sfal.event.text text='doorlock change to &(CAN0)'>")**

5) - Follow the steps **2**, **3** and **4** to add other message identifiers, read out their data and send out these values out when they change.

*For more information about the PFAL commands added in this application note, refer to the manual "****steppIII_fox_bolero_lt_PFAL_Configuration_Command_Set.pdf****".*

Below you will find some examples how to read the data added to some message identifiers using PFAL commands. ***The red marked bits in the examples below are the bits which will be read from the data that the message identifier provides.***

How to add the required message identifiers, read out their data and send values to a remote server when they change?

$PFAL,Sys.CAN.Msg.Add,std,211
$PFAL,Sys.Can.Var.Add,0,number,event,std,211,0,0,0,7,MSB

First command adds a standard message identifier "**211**" into the associated receive buffer and second command stores the value *(e.g. door lock state)* into the ***slot 0*** read out of **byte 0** of the 8-bytes datastream starting from the MSB *(most significant byte)*. *Whenever the value in the **slot 0** changes, the corresponding event is occurred. This event enables you to capture and sent out these values via TCP.*
*Some automotive manufacturers organise their signal data within the data bytes to be read in different directions. That's way, the entries **MSB** or **LSB** at the end of the command **"Sys.Can.Var.Add"** defines which byte in the 8-bytes datastream should be read first. Therefore, if you set **MSB,** the data will be read from left to right, while if you set **LSB,** the data will be read from right to left as represented in table below. If the reading direction does not match with the direction given by the automotive manufacturer you will get a wrong value which does not correspond to the value that you will have to read.*
This example is represented in table form below.

	Identifier	**Data**							
	11 bit	***0 to 7 bytes (represented in bitwise)***							
		byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0 (←**LSB)**
		(MSB→) byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
Example	**211**	11110011	10100	11111	10011001	1110101	11110001	11110010	11110101

(→ ←) The arrow direction indicates reading direction of bytes.

$PFAL,Sys.CAN.Msg.Add,std,39F
$PFAL,Sys.Can.Var.Add,1,number,event,std,39F,0,0,0,7,MSB

First command adds a standard message identifier "**39F**" into the associated receive buffer and second command stores the value *(e.g. (e.g. gear direction)* into the ***slot 1*** read out of **byte 0** of the 8-bytes datastream starting from the MSB *(most significant byte)*.
This example is represented in table form below.

	Identifier	**Data**							
	11 bit	**0 to 7 bytes (represented in bitwise)**							
		byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0 (←**LSB)**
		(MSB→) byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
Example	**39F**	010111	111101	11111	1011101	10111	1001110	1001100	10100011

(→ ←) The arrow direction indicates reading direction of bytes.

$PFAL,Sys.Can.Var.Add,2,number,state,std,39F,1,0,2,7,LSB

This command stores the value (*e.g. Speed values*) into the ***slot 2*** read out of **bytes 1** and **2** of the 8-bytes datastream starting from the LSB *(last significant byte)*.
This example is represented in table form below.

	Identifier	**Data**							
	11 bit	***0 to 7 bytes (represented in bitwise)***							
		byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0 (←**LSB**)
		(**MSB→**) byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
Example	**39F**	10010111	11101	10001	11011101	1010111	**10010111**	**10011110**	10101011

(→ ←) The arrow direction indicates reading direction of bytes.

$PFAL,Sys.CAN.Msg.Add,std,15B
$PFAL,Sys.Can.Var.Add,3,number,event,std,15B,1,4,2,3,LSB

First command adds a standard message identifier "**15B**" into the associated receive buffer and second command stores the value *(e.g. RPM)* into the ***slot 3*** read out of ***bits 4 - 7 of*** **byte 1** and ***bits 0 - 3 of*** **byte 2** of the 8-bytes datastream starting from the LSB *(last significant byte)*. *This example is represented in table form below.*

	Identifier	**Data**							
	11 bit	***0 to 7 bytes (represented in bitwise)***							
		byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0 (←**LSB**)
		(**MSB→**)byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
Example	**15B**	10010111	11101	10001	11011101	1010111	1001**0111**	**1001**1110	10101011

(→ ←) The arrow direction indicates reading direction of bytes.

How to add the required message identifiers, read out their data and send values to a remote server when they change?

$PFAL,Sys.CAN.Msg.Add,std,479
$PFAL,Sys.Can.Var.Add,4,number,event,std,479,1,3,2,2,MSB

First command adds a standard message identifier "**479**" into the associated receive buffer and second command stores the value (*e.g. door state)* into the ***slot 3*** read out of ***bits 3 - 7 of*** **byte 1** and ***bits 0 - 2 of*** **byte 2** of the 8-bytes datastream starting from the MSB *(most significant byte)*. *In this example bits of bytes 6 and 5 will be read.*
This example is represented in table form below.

	Identifier	**Data**							
	11 bit	***0 to 7 bytes (represented in bitwise)***							
		byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0 (←**LSB**)
		(**MSB→**) byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
Example	**479**	10100111	**11011**100	10**111**	11101	1110001	10111111	11111110	11100001

(→ ←) The arrow direction indicates reading direction of bytes.

$PFAL,Sys.CAN.Msg.Add,std,65A
$PFAL,Sys.Can.Var.Add,5,number,state,std,65A,1,0,3,7,LSB

First command adds a standard message identifier "**65A**" into the associated receive buffer and second command stores the value *(e.g. milage)* into the ***slot 3*** read out of **bytes 1, 2** and **3** of the 8-bytes datastream starting from the LSB *(last significant byte)*.
This example is represented in table form below.

	Identifier	**Data**							
	11 bit	***0 to 7 bytes (represented in bitwise)***							
		byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0 (←**LSB**)
		(**MSB**→) byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
Example	**65A**	10010111	11101	10001	11011101	**1010111**	**10010111**	**10011110**	10101011

(→ ←*) The arrow direction indicates reading direction of bytes.*

$PFAL,CNF.Set,AL45=SYS.Can.e0:TCP.Client.Send,8,"<sfal.event.text text='doorlock changed to &(CAN0)'>"

$PFAL,CNF.Set,AL46=SYS.Can.e4:TCP.Client.Send,8,"<sfal.event.text text='doorstate changed to &(CAN4)'>"

$PFAL,CNF.Set,AL47=SYS.Can.e1:TCP.Client.Send,8,"<sfal.event.text text='gearstate changed to &(CAN1)'>"

Alarm 45 reports an RMC protocol and the CAN value from the variable slot 0 to the TCP server whenever this value changes.
Alarm 46 reports an RMC protocol and the CAN value from the variable slot 4 to the TCP server whenever this value changes.
Alarm 47 reports an RMC protocol and the CAN value from the variable slot 1 to the TCP server whenever this value changes.

$PFAL,CNF.Set,AL48=SYS.Can.e3>1&SYS.TRIGGER.s_TRIP=low:SYS.TRIGGER_TRIP=high&TCP.Client.Send,8,"<sfal.event.tripstart>"

$PFAL,CNF.Set,AL49=SYS.Can.e3=0&SYS.TRIGGER.s_TRIP=high:SYS.TRIGGER_TRIP=low&TCP.Client.Send,8,"<sfal.event.tripstop dist='&(NavDist)'>"&TCP.Client.Send,8,"<sfal.event.text text='milage = &(CAN5) '>"

$PFAL,CNF.Set,AL50=SYS.TIMER.e_1SEC:MSG.Send.Serial,0,"SERIALCAN Speed=&(CAN2) kmh RPM=&(CAN3) u/min tacho=&(CAN5) km"

Alarm 48 reports the CAN value from the variable slot 3 to the TCP server whenever this value is greater than 1 and trigger *s_TRIP* is low.
Alarm 49 reports the CAN value from the variable slot 3 to the TCP server whenever this value is 0 and trigger *s_TRIP* is high.
Alarm 50 reports the CAN value from the variable slots 2 and 3 to the serial line every 1 second.
<sfal.event...> commands are supported only from our **D2Sphere** server.

doorstate changed to 0
doorstate changed to 1
gearstate changed to 0
gearstate changed to 2
gearstate changed to 1
doorstate changed to 8
doorstate changed to 9
doorstate changed to 2
doorstate changed to 5
doorstate changed to 3
doorstate changed to 4
doorstate changed to 60
doorlock changed to 8
milage = 46940

How to add the required message identifiers, read out their data and send values to a remote server when they change?

milage = 46957
milage = 47030
milage = 47046
milage = 47056
milage = 47065
milage = 47070

The values above ***(doorlock, door state** and **milage)*** are taken out of the ***D2Sphere*** server database the STEPPIII device has reported to server, based on the alarm configuration of **AL45, AL46** and **AL49.**