Manual

J1939 API for Windows

Software Version 1.1

IXXAT

The expert for industrial and automotive communication

IXXAT

Headquarter
IXXAT Automation GmbH
Leibnizstr. 15
D-88250 Weingarten

Tel.: +49 (0)7 51 / 5 61 46-0
Fax: +49 (0)7 51 / 5 61 46-29
Internet: www.ixxat.de
e-Mail: info@ixxat.de

US Sales Office
IXXAT Inc.
120 Bedford Center Road
USA-Bedford, NH 03110

Phone: +1-603-471-0800
Fax: +1-603-471-0880
Internet: www.ixxat.com
e-Mail: sales@ixxat.com

Support

In case of unsolvable problems with this product or other IXXAT products please contact IXXAT in written form by:

Fax: +49 (0)7 51 / 5 61 46-29
e-Mail: support@ixxat.de

Copyright

Document number: 4.02.0287.20000
Version: 1.2

1 Introduction

The J1939 API software provides a comfortable programming interface for the rapid development of J1939 applications on a Windows PC in various languages (C, C++ and Python). The programming interface is based on the IXXAT VCI driver and is therefore useable with all IXXAT CAN interfaces. Furthermore it is possible to implement several applications on one CAN controller which can also communicate with each other. This allows the simulation of complete J1939 networks on one PC and is therefore excellent capable for testing and commissioning of control units.

The J1939 API provides a full implementation of the SAE J1939 protocol based on the IXXAT J1939 protocol software. Therefore it supports the transport protocol for transmission of J1939 messages with up to 1785 data bytes, network management for address claiming and monitoring of cyclic received messages.

The signals of a message (parameter group) are interpreted based on a XML configuration file, which can be generated by the J1939 Designer. The J1939 Designer, which is also part of the J1939 API delivery, allows the definition of application-specific messages.

The J1939 API provides DLLs and header files for incorporation into the various language environments. The C++ interface presents a simple C++ class. The C interface presents a collection of functions, including Create() and Delete() functions and the instance is maintained using an object handle. By means of a python interface class a python API is provided.

An important condition for working with the J1939 API is that you are familiar with the basic concepts of the SAE J1939 protocol. An overview of the most relevant SAE J1939 specifications is given in chapter 3 ‘Specifications of the J1939 Protocol’.

1.1 Definitions, Acronyms, Abbreviations

API	Application Programming Interface
BLOB	Binary Large Object
CAN	Controller Area Network
DLL	Dynamic Link Library
DTC	Diagnostic Trouble Code
PG	Parameter Group
PGN	Parameter Group Number
SAE	Society of Automotive Engineers
SP	Suspect Parameter
SPN	Suspect Parameter Number
VCI	Virtual CAN Interface
XML	Extensible Markup Language

2 Installation

The J1939 API is delivered as a setup executable, which installs the API libraries, the configuration tool, the demo applications, the online documentation and if required the optional python support.

2.1 Requirements

The J1939 API is developed for use on a Windows PC. The demo applications were developed in Microsoft Visual Studio 2008. Therefore at least the Visual Studio 2008 Express Edition, which can be downloaded free of charge from the Microsoft website, is required to open the enclosed project files.

Before installing the API, the VCI V3 driver must be installed, which allows to access the IXXAT CAN interface[1]. To install the CAN interface, please read the hardware installation manual. For installation of the VCI V3, please consult the VCI installation manual.

The API provides an optional Python interface, which requires Python 2.5 or newer. If Python is not installed on the Windows PC, the API will be installed without Python support.

2.2 Setup

To install the API, insert the program CD supplied into the CD drive of your PC and start the installation program by running the executable setup file. Follow the instructions of the installation program.

[1] To use the API, at least one IXXAT CAN interface (e.g. an USB-to-CAN) is required.

3 Specifications of the J1939 Protocol

The SAE J1939 protocol is described in detail in several specification documents. The structure of the documents is based to a large extent on the OSI layer model. A corresponding specification was created for each layer in the OSI layer model. The diagram below shows the assignment of each specifiation document to the according layer of the OSI model.

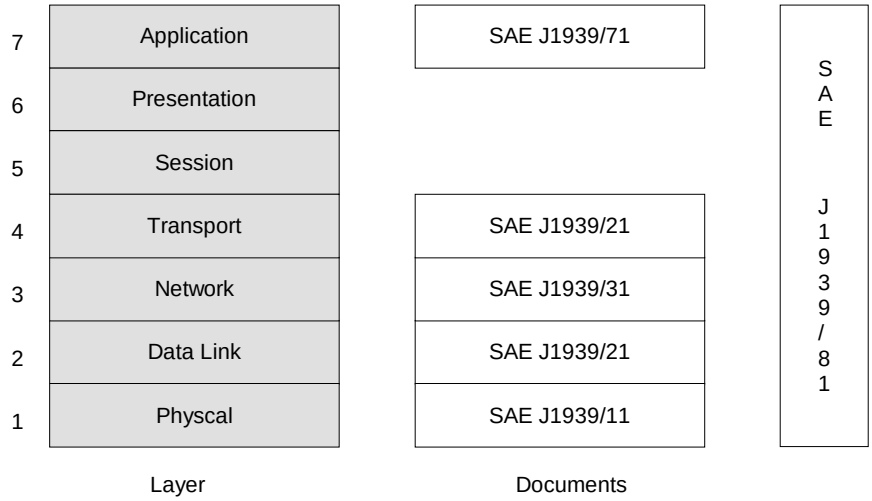

Fig. 3.1: J1939 protocol in OSI layer model

SAE J1939/11 defines a CAN high-speed bus interface in accordance with ISO/DIS 11898.

SAE J1939/21 describes the data communication via CAN based on the specification CAN2.0B. Only the extended format is used with this specification. Besides the segmentation and use of the 29-bit CAN identifier, the specification essentially describes the fragmented transmission of large data blocks.

SAE J1939/31 describes the functionality of a bridge. This functionality is not implemented in this protocol software.

SAE J1939/71 describes the actual data of a message. It is also shown that each J1939 message has a unique reference number (Parameter Group Number).

SAE J1939/81 describes the functionality of the network management. The network management can be regarded as an independent unit that includes functions within all layers – from layer 7 to layer 1. This is why this block is displayed as an independent functional block.

The above-mentioned specifications, which are not necessary for the use of this product, can be obtained from the SAE website (www.sae.org).

4 Getting Started

In the following sections, the loading of the API is discussed, organized by the implementation environment. The technique differs based on the environment. The source of the initialization arguments used in the examples is discussed in the next chapter.

4.1 Loading into an ANSI-C Project

The C-wrapper for the J1939 API is delivered as a header file (j1939api_c.h), a library file (j1939api_c.lib) and a DLL (j1939api_c.dll) and is loaded by including the header-file and linking the library. The DLL is automatically linked with the library file from the system directory.

The API instance is created via the function J1939Api_Create() which, on successful initialization of the API, delivers a handle via reference to the instance. All subsequent functionality of the API require the inclusion of the instance handle as the first argument of the function. In this way, the object-oriented functionality of the class is exposed in a non-object-oriented way. The instance is deleted using the J1939Api_Destroy() command or with the unloading of the DLL.

The C-wrapper requires the j1939api_types.h header-file for the the definition of required types, defines, structures and macros and the j1939api_err.h header-file for return value and error message definitions.

For more information on the use of the J1939 API in C, see chapter 7 ‘Demo Applications’, or refer directly to the demo project provided with the installation.

4.2 Loading into a C++ Project

Like the C-Wrapper, the C++-wrapper for the J1939 API is delivered as a header file (j1939api_cpp.h), a library file (j1939api_cpp.lib) and a DLL (j1939api_cpp.dll) and is loaded by including the header-file and linking the library. The DLL is automatically linked with the library file from the system directory.

The API instance is handled as a standard C++ class object. The object is created using the standard constructor and then initialized via the function Initialize(). The functions are provided as part of the objects class. The instance is deleted using the ‘delete’ keyword or by allowing the instance to go out of scope.

The C++-wrapper requires the j1939api_types.h header-file for the definition of required types, defines, structures and macros and the j1939api_err.h header-file for return value and error message definitions.

For more information on the use of the J1939 API in C++, see chapter 7 ‘Demo Applications’, or refer directly to the demo project provided with the installation.

4.3 Loading into a Python Script

For development using the python wrapper of the API, a Python installation of version 2.5 or newer is required. The wrapper is delivered as a python interface class (J1939api.py), a wrapper class (J1939_wrap.py) and a library (_J1939_wrap.pyd). The wrapper can be loaded by importing the file j1939.py which is installed with the API. To load the wrapper class, import the module ‘ixxat.j1939api.J1939api’ and create and initialize the object using the J1939() constructor.

For more information on the use of the J1939 API in Python, see chapter 7 ‘Demo Applications’, or refer directly to the demo project provided with the installation.

5 Message Interpretation

The J1939 API provides message data on the parameter level. Therefore the suspect parameters of a parameter group (J1939 message) are interpreted via a XML configuration file, which can be generated by the J1939 Designer.

5.1 Configuration File

The configuration file contains standard and proprietary parameter group definitions in XML format. Every paramater group consists of one or more suspect parameters and is identified by the PGN. The definition of a parameter group contains, besides the suspected parameters, the data length of the parameter group in bytes, which can be variable (e.g. if ASCII parameters are included). The suspect parameters, which are identified by the SPN, are also defined in the configuration file. The definition of a suspect parameter includes the required information for parameter interpretation, like parameter type (integer, float, ASCII, binary, bit field, BLOB), unit, resolution and offset.

5.2 J1939 Designer

The J1939 Designer is a powerful tool for generating XML configuration files for the J1939 Windows API and the J1939 canAnalyser Module, as well as source code for the J1939 Micro and Standard Protocol Software. The delivery of the J1939 API includes the J1939 Designer for Windows API (full version of Editor and API Code Generator together with demo versions of the J1939 Micro and Standard Stack Generators are included). The Editor of the J1939 Designer allows the definition of proprietary messages in an easy and comfortable way. By pressing the ‘Generate’ button, the API Code Generator will be started and a XML configuration file with the defined proprietary messages will be generated. In case that the J1939 Micro or Standard Protocol Software is used, the defined proprietary messages can also be included in the corresponding application. See the J1939 Designer Manual for further information.

6 Interface to the Application

The interfaces to the application for the various programming language environments do not differ with respect to provided functionality, instead in the calling style, instance maintenance and parameter structure. To this point, a general description of the functionality of the J1939 API is made in this chapter, and can be applied to all supported programming language interfaces.

For a specific interface description of each programming language, please refer to the corresponding online documentation, which is located under the ‘doc’ directory. The online documentation can also be accessed via the ‘Start’ menu, under ‘IXXAT->J1939->API->Doc’. Because the online documentation contains HTML-Frames and JavaScript for navigation, JavaScript has to be enabled in your browser, to ensure that the online documentation is displayed properly.

6.1 General Structures

The message and parameter structures, which are used to send and receive J1939 messages, take in a central part in the J1939 API. Therefore their parameters are described in detail in the following.

6.1.1 Message Structure

The entry *Pgn* contains the 17-bit parameter group number (16-bit PGN plus page bit) of the message. This functions as both an identifier for the message in the J1939 network and instructs the API how to encode or decode the message.

The entry *Priority* contains the 3-bit priority value of the message. This entry is only relevant for message transmission and not passed by the stack with reception messages.

The entry *RemoteDevice* contains the 8-bit device address associated with the message. For received messages, this is the originating device (source). For messages to be transmitted, this is the target device (destination). The address for the device controlled by the API is set during initialization as the first parameter of the initialization function (device address).

The entry *MsgType* contains message type information, currently consisting of the reception message type and a message flag, which signals if the message was sent global or specific. The reception message types ‘DATA’ (mapped data message), ‘RAW’ (raw data message), ‘REQ’ (request message) and ‘ERR’ (error message) are supported.

The entry *ParamCount* defines the number of suspect parameters in the parameter group. If the number of parameters passed to the send function does not equal that expected by the XML configuration file, an encoding error is returned. For raw messages, this entry is instead a byte-length indicator. For request messages, this is always 0. For error messages, this is always 2.

The entry *Parameters* contains the suspect parameter values as an array of 32-bit unsigned integers. For raw data messages, the individual bytes can be retrieved using specific macros (see chapter 7 ‘Demo Applications’). When supported messages are exchanged, ASCII strings can be passed via a 32-bit pointer to a single-byte width character array or string. For request messages, the parameters are unused. For error messages, the element with index 0 contains the 16-bit error value from the J1939 API software and the element with index 1, the 32-bit additional value info.

6.1.2 Parameter Structure

The entry *Name* contains the parameter name, as specified by the J1939 standard and represented by an ASCII string.

The entry *Resolution* contains the multiplier for converting between the real parameter value and the 32-bit unsigned integer passed via the message structure. This value is represented by a double type, but can represent an integer, double or no-value (ASCII) depending on the parameter type.

The entry *Offset* contains the offset for converting between the real parameter value and the 32-bit unsigned integer passed via the message structure. This value is represented by a double type, but can represent an integer, double or no-value (ASCII) depending on the parameter type.

The entry *ValueMin* contains the minimum allowable value for the real parameter value. This value is represented by a double type, but can represent an integer, double or no-value (ASCII) depending on the parameter type.

The entry *ValueMax* contains the maximum allowable value for the real parameter value. This value is represented by a double type, but can represent an integer, double or no-value (ASCII) depending on the parameter type.

The entry *Type* contains the parameter type, which specifies what types of values are contained in the preceding double type variables as well as how the parameter should be converted. The parameter types are adopted from the SAE J1939 specification and could be one of the following types: integer, float, ASCII, bit-field or binary (see also the example in chapter 7 ‘Demo Applications’).

The entry *Unit* contains the unit text, represented by an ASCII string.

6.2 General Functionality

To give a common overview of the functionality of the J1939 API, some general comments about initialization, message registration, message transmission, message reception and error handling are made in this chapter.

6.2.1 Initialization of the API

The initialization of the J1939 API always requires the following parameters: an 8-bit device address, a device name structure, the VCI V3 board index, the CAN line and the XML configuration file.

The *device address* is specified in the J1939 specification and is used for J1939 device identification. It will be included with all transmitted messages in the source address field.

The *device name* is also specified in the J1939 specification and is used for address claiming. It is included as the data field of the J1939 NAME message, which is sent upon initialization of the device with the J1939 network (see the J1939/81 specification for more information about the device name).

The *board index* specifies which CAN interface board to use. The VCI V3 maintains a current list of installed devices. The list can be retrieved using the function 'GetBoardList()'. The index corresponds to the index in the list returned by the function.

The *can line* specifies which CAN line to use on a particular CAN interface board. Boards can support a varying number of CAN lines.

The *configuration file* contains a list of supported PGs and SPs in XML format for usage with the J1939 API and information regarding the mapping and demapping of parameters to J1939 supported CAN messages. The XML file can be configured to include user proprietary messages using the provided configuration tool. An example configuration file is provided with the installation. The PGs not included in this file can still be transmitted and received, but must be transmitted and received as raw, or unmapped, messages. In this case, the 32-bit parameter array is handled rather as byte array memory, access to which can be handled using the provided macros or using direct memory copying or writing functions (see also the example in chapter 7 'Demo Applications').

The initialization must be performed for all instances of the API. The VCI V3 is developed so as to allow multiple instances and applications simultaneous access to the same hardware.

6.2.2 Registration / Deregistration of PGNs

Before J1939 messages can be received, they must be registered with the stack. This is done via the various registration functions. The PGN is message specific and defines the structure and contents of the data portion of the message.

Successful registration of a PGN will return a non-negative value. 'OK' is returned when the PGN is found within the loaded configuration file and indicates the registration of a mapped PGN. 'RAW' indicates a PGN was not found in the configuration file and will in all situations encode and decode messages in raw format.

PGNs cannot be cleared individually. The clearing of the entire list of registered PGNs is performed using the clear PGN function.

Information regarding supported PGNs and SPNs can be retrieved using the functions 'ReadSpnList()' and 'ReadSpnAttr()'.

6.2.3 Message Transmission

Messages are transmitted using the ‘Send()’ function. The message to be transmitted is passed as a message structure (see chapter 6.1.1 ‘Message Structure’) parameter along with a timeout value in milliseconds (-1 is infinite). The result of a timeout of the ‘Send()’ function is signalled via the return value.

For messages with supported PGNs, the suspect parameters are mapped according to the configuration file. Otherwise, the data field is filled byte-for-byte with the contents of the parameter array (see Chapter 7 ‘Demo Applications’). Setting the raw message type in the message type byte forces raw encoding of the message, even for supported messages.

The rest of the message type field is ignored, because the ‘Send()’ function only supports data messages and globally transmitted messages are specified by the destination address. Error messages are generated by the stack and therefore not transmitted over CAN and request messages originating at the API can be transmitted using the J1939 request message (PGN: 0xEA00, Data: <17-bit requested PGN>).

It is necessary for the successful encoding of mapped messages that the correct number of suspect parameters for the corresponding PG are passed to the function.

6.2.4 Message Reception

Messages are received using the ‘Receive()’ function. Because the messages are received asynchronously, the API stores them in a message reception queue. This means that it is necessary for the user application to regularly empty the reception queue. If this is not done queue overruns can quickly occur.

The receive function is passed a timeout value in milliseconds (-1 is infinite) and returns a message structure (see chapter 6.1.1 ‘Message Structure’), either as a return parameter or via a passed structure pointer. The receive function waits on the reception queue for the duration of the timeout. The result of a timeout of the receive function is signalled via the return value.

For data messages with supported PGNs, the parameters are mapped according to the configuration file. Otherwise, the data field is mapped directly to the 32-bit parameter array and accessed either via the provided macros or using direct memory copy or read functions (see example in chapter 7 ‘Demo Applications’.

Because the messages are received from the CAN network, there is no possibility for forcing raw encoding, and it is assumed that messages received over the network have the correct J1939 structure for their PGNs. A raw decoded message indicates this with the raw message type in the message type byte.

The reception of a request message is indicated with the request message type in the message type byte. In this case the message structure does not contain the PGN of the request message, which is always 0xEA00, but the re-

quested PGN. As response the data of the requested PGN has to be sent to the device which requested it.

The message type byte also contains a global message flag, which is set to indicate the reception of a global message.

6.2.5 HRESULTs / Error Exceptions

Every function of the API returns a value with the results of the function call. In most cases, this value should be 0 (OK). When cast as signed 32-bit integers, all error results are negative. The result values are represented in C/C++ code as HRESULTs (long integer) and in Python they are delivered as the value of a thrown exception. The result values can help the programmer evaluate the success of a function call and diagnose the cause of an error. With help of the 'GetHRESULTString()' or 'GetExceptionString()' function, the description to a HRESULT or an exception code can be determined.

6.2.6 Error Messages

Because the J1939 API maintains its own task, errors can occur unbound to a specific function call instance. Those errors are entered as messages into the message queue, with the msgtype set to 'ERR' and the error code entered into the first element of the message parameters array. An additional error information is entered into the second element. With help of the 'GetErrorString()' function, the error description to an error code can be determined.

7 Demo Applications

Demonstration applications are delivered with the API for all supported development languages. The demos themselves are essentially translations into the various languages of one program, which is described in this chapter.

The first sub-chapter describes the structure and the coding of the demonstration application, which can aid the user in writing their own code.

The second sub-chapter describes the steps necessary, and the expected results of running the demonstration application.

7.1 Application Structure

The demo application is split into three specific units: The *decode* unit is a library for the conversion and display of received messages to and in user format. The executable *demo1* maintains a reception polling loop that displays received messages and responds to request messages. The executable *demo2* maintains a loop for transmitting messages cyclically and displaying request response messages.

For all applications, in the case of errors in calling an API function, the *GetHRESULTString()* or *GetExceptionString()* function is used with the returned result code to retrieve a string description of the error.

The three units are discussed in more details in the following sub-chapters.

7.1.1 Decode

The decode library contains one function: *MsgDecoder()*. The function is passed access to the API instance on which the message was received, as well as the received message.

Initially the function determines the type of messages received. Request messages are not handled, since they must be responded to immediately by the *demo1* exectuable, and therefore are sorted out at that stage. Raw messages are displayed as a list of bytes from the parameter list. Error messages are retrieved using the first parameter list entry and the GetErrorString() function and displayed along with their additional information parameter.

Data messages must be further interpreted. This is accomplished using the ReadSpnList() and ReadSpnAttr() functions. Initially, the list of SPNs are retrieved from the API instance. For each SPN in the SPN list, the individual attributes are retrieved from the API instance, which then can be used for the conversion of the parameter value. For applicable parameter types, the conversion formula is:

$$\text{user / display value} = \text{parameter value} * \text{resolution} + \text{offset}$$

Since the parameters can represent a variaty of SPN types, the resolution and offset are maintained as double types in the parameter attribute structure. De-

pending on the language, casting is important for the correct calculation of user values.

- integer types are calculated by casting the resolution and offset to integers and performing the conversion. They are displayed along with their unit string.
- float types are calculated by casting the parameter value as a double and performing the conversion. They are displayed along with their unit string.
- ASCII types are returned as a pointer to a string which is located in the return parameter array. These should be cast as a string and displayed. Because they are stored in the parameter array, these strings must not be freed.
- binary types represent unitless values. They are converted like the integer types using unsigned integer castings, and are displayed in hexidecimal format without a unit string.
- BitField types represent data that is specified in the J1939 specification or by the user directly. There is, therefore, no conversion for these values and they are displayed in decimal format without a unit string. Full support for the retrieval of specified BitField values is not included in this version of the API.
- BLOBs are used for the transmission of binary data with variable length.

7.1.2 Demo1

The first demo executable performs the following functions.

Initially, the application retrieves and displays a list of available boards from the VCI (via the board list retrieval function). Following the successful retrieval of the board list, an instance of the API is created and initialized, using board index 0 and CAN line 0 (first CAN on first board), along with an arbitrary device NAME and an appropriate XML configuration file (one that at-least supports the PGNs used in this demonstration). The XML configuration file is located in the directory of the demonstration application.

Following successful initialization of the API, including waiting for stack activation by monitoring the stack status, the necessary PGNs for the demonstration are registered with the API (one standard supported PGN, one raw cyclical PGN at 1.1 second cycles, one request PGN and one user-defined PGN containing all supported parameter types) and a message is specified for transmission in response to request messages.

The message reception is then repeatedly polled until a message is received (timeouts are ignored). Request messages are responded to directly. All other messages are forwarded to the *decode* unit for display.

For more information on the specific messages transmitted and received in the demonstration application, see Chapter 7.3 ‘Message Specifications’.

7.1.3 Demo2

The second demo executable performs the following functions.

Like the first, the second initially retrieves and displays the list of available boards and initializes an API instance using board index 0 and CAN line 1 (second CAN on first board). After the stack state goes active, the application initializes the set of messages to send to the first demo application (one standard supported PGN, one raw cyclical PGN, one request PGN and one user defined PGN).

The application cycles every 100 ms and within each cycle transmits one of each of these messages and checks the reception queue for a request response.

For more information on the specific messages transmitted and received in the demonstration application, see Chapter 7.3 ‘Message Specifications’.

7.2 Running the Application

Before starting the demos, the selected board index and CAN lines can be adapted. It’s also possible to run both demos on the same CAN line of the same board. But in this case some sort of remote CAN device is required for successful administration of the demo applications. This can be performed by starting an instance of the IXXAT canAnalyser or MiniMon or any other J1939 device on the same network as the demonstration application.

The functionality of the API also requires a valid XML configuration file. In the event that the initialization fails, the first recourse to ensure that the correct path is specified to find the XML file delivered with the demo applications. The C and C++ demo applications should automatically copy the delivered J1939api.xml to the target directory, where they can be accessed by directly clicking on the generated executables.

7.3 Message Specifications

To demonstration the functionality of the API over J1939, various message types are transmitted via the demo applications. The following messages are utilized in the demonstration process:

7.3.1 PGN 65128 (0xFE68) – Vehicle Fluids – VF

The Vehicle Fluids PGN is a straightforward J1939 message and is used to demonstrate the transmission and reception of a standard, supported, mapped message. Because of the PGN of the message, it must always be transmitted globally. In the demo, the priority is set to six and the message type to 'DATA'. The following is the specification of the PGN according to the XML configuration file:

PGN	65128	
Name	Vehicle Fluids	
Acronym	VF	
Length	8 bytes	
SPN List	SPN	Start-bit
	1638	1
	1713	9
	1857	11
	2602	17

7.3.1.1 SPN 1638 – Hydraulic Temperature

The SPN is specified by the XML configuration file as follows:

SPN	1638
Name	Hydraulic Temperature
Resolution	1.0
Data-Max	-
Bit-Length	8
Type	Float
J1939 Standard	true
Unit	"C"
Offset	-40.0
Data-Min	-

In the demo, the value of this SPN is set and transmitted as 240, which represents the value 200 degrees C.

7.3.1.2 SPN 1713 – Hydraulic Oil Filter Restriction Switch

The SPN is specified by the XML configuration file as follows:

SPN	1713
Name	Hydraulic Oil Filter Restriction Switch
Resolution	-
Data-Max	3
Bit-Length	2
Type	BitField
J1939 Standard	true
Unit	“bit”
Offset	-
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 2, which, as a BitField type, will not be converted, rather displayed as a unitless value. According to the J1939 specification, the value 2 indicates an error with the reading.

7.3.1.3 SPN 1857 – Winch Oil Pressure Switch

The SPN is specified by the XML configuration file as follows:

SPN	1857
Name	Winch Oil Pressure Switch
Resolution	-
Data-Max	3
Bit-Length	2
Type	BitField
J1939 Standard	true
Unit	“bit”
Offset	-
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 1, which, as a BitField type, will not be converted, rather displayed as a unitless value. Ac-

cording to the J1939 specification, the value 1 indicates an acceptable oil pressure.

7.3.1.4 SPN 2602 – Hydraulic Oil Level

The SPN is specified by the XML configuration file as follows:

SPN	2602
Name	Hydraulic Oil Level
Resolution	0.4
Data-Max	-
Bit-Length	8
Type	Float
J1939 Standard	true
Unit	"%"
Offset	0.0
Data-Min	-

In the demo, the value of this SPN is initially set and transmitted as 0, but is incremented every cycle, which is represented upon message decoding as the series beginning with 0% and increasing by 0.4% each transmission.

7.3.2 PGN 65226 (0xFECA) – Active DTCs – DM01

The DM01 is a diagnostic message and is used to demonstrate the transmission and reception of a PGN with variable data length. The length of the data field depends on the number of DTCs included in the message. In the demo, the priority is set to six and the message type to 'DATA'.

PGN	65226
Name	Active DTCs
Acronym	DM01
Length	Variable
SPN List	See SAE J1939/73 Standard

The DM01 consists of the lamp status (a) and a variable number of DTCs. Every DTC includes the SPN (b), FMI (c) and CM together with OC (d). The message forms looks as follows: a, b, c, d, b, c, d, b, c, d, … etc. In the demo application the number of DTCs varies between 0 and 3.

7.3.3 PGN 55040 (0xD700) – Binary Data Transfer – DM16

The DM16 is a diagnostic message and is used to demonstrates the data type BLOB. BLOBs are used for the transmission of binary data with variable length. The DM16 is part of the Memory Access Protocol (DM14 to DM18), but in the demo the other memory access messages are not considered, because they are not relevant to demonstrate binary large objects.

PGN	55040	
Name	Binary Data Transfer	
Acronym	DM16	
Length	Variable	
SPN List	SPN	Start-bit
	1650	1
	1651	9

7.3.3.1 SPN 1650 – Length of Raw Binary Data

The SPN is specified by the XML configuration file as follows:

SPN	1650
Name	Number of Occurrences of Raw Binary Data
Resolution	1
Data-Max	255 (0xFF)
Bit-Length	8
Type	Integer
J1939 Standard	TRUE
Unit	Count
Offset	0
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 5, which is number of occurrences of raw data (bytes 2-6).

7.3.3.2 SPN 1651 –Raw Binary Data

The SPN is specified by the XML configuration file as follows:

SPN	1651
Name	Raw Binary Data
Resolution	1
Data-Max	-
Bit-Length	8
Type	BLOB
J1939 Standard	TRUE
Unit	-
Offset	0
Data-Min	-

In the demo, all five bytes of the binary large object are set to 0xFF. The decoder does not display the binary data of the BLOB data type. It just signals that binary data has been received successfully.

7.3.4 PGN 59904 (0xEA00) – Request – RQST

The Request PGN simply contains the PGN of the desired response message and is used by the demo to demonstrate the reqeust process. The message is not transmitted globally. In the demo, the priority is set to six and the message type to ‘DATA’, however, due to the special nature of this message,it is received by the demo1 application as a request message type. The following is the specification of the PGN according to the XML configuration file:

PGN	59904	
Name	Request	
Acronym	RQST	
Length	3 bytes	
SPN List	SPN	Start-bit
	2540	1

7.3.4.1 SPN 2540 – Parameter Group Number (RQST)

The SPN is specified by the XML configuration file as follows:

SPN	2540
Name	Parameter Group Number (RQST)
Resolution	1
Data-Max	16777215 (0xFFFFFF)
Bit-Length	24
Type	Binary
J1939 Standard	true
Unit	“binary”
Offset	0
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 65213, which is the value of the desired response message, described in the next sub-chapter.

7.3.5 PGN 65213 (0xFEBD) – Fan Drive – FD

The Fan Drive PGN is a supported, mapped message which demonstrates the request capabilities and therefore transmitted from the demo1 application. This message is not transmitted globally, rather to the specific address received as the source of the request message. In the demo, the priority is set to six and the message type to ‘DATA’. The following is the specification of the PGN according to the XML configuration file:

PGN	65213	
Name	Fan Drive	
Acronym	FD	
Length	8 bytes	
SPN List	SPN	Start-bit
	975	1
	977	9
	1639	17
	4211	33
	4212	49

7.3.5.1 SPN 975 – Estimated Percent Fan Speed

The SPN is specified by the XML configuration file as follows:

SPN	975
Name	Estimated Percent Fan Speed
Resolution	0.4
Data-Max	-
Bit-Length	8
Type	Float
J1939 Standard	true
Unit	“%”
Offset	0.0
Data-Min	-

In the demo, the value of this SPN is set and transmitted as 120, which represents the value 48%.

7.3.5.2 SPN 977 – Fan Drive State

The SPN is specified by the XML configuration file as follows:

SPN	977
Name	Fan Drive State
Resolution	-
Data-Max	15
Bit-Length	4
Type	BitField
J1939 Standard	true
Unit	“bit”
Offset	-
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 2, which, as a BitField type, will not be converted, rather displayed as a unitless value. According to the J1939 specification, the value 2 indicates excessive engine air temperature.

7.3.5.3 SPN 1639 – Fan Speed

The SPN is specified by the XML configuration file as follows:

SPN	1639
Name	Fan Speed
Resolution	0.125
Data-Max	-
Bit-Length	16
Type	Float
J1939 Standard	true
Unit	“rpm”
Offset	0.0
Data-Min	-

In the demo, the value of this SPN is initially set and transmitted as 2, but is increased by 100 every request reception, which is represented upon message decoding as the series beginning with 0.250 rpm and increasing by 12.500 rpm each transmission.

7.3.5.4 SPN 4211 – Hydraulic Fan Motor Pressure

The SPN is specified by the XML configuration file as follows:

SPN	4211
Name	Hydraulic Fan Motor Pressure
Resolution	0.5
Data-Max	-
Bit-Length	16
Type	Float
J1939 Standard	true
Unit	“kPa”
Offset	0.0
Data-Min	-

In the demo, the value of this SPN is set and transmitted as 32000, which represents the value 16,000 kPa.

7.3.5.5 SPN 4212 – Fan Drive Bypass Command Status

The SPN is specified by the XML configuration file as follows:

SPN	4212
Name	Fan Drive Bypass Command Status
Resolution	0.4
Data-Max	-
Bit-Length	8
Type	Float
J1939 Standard	true
Unit	“%”
Offset	0.0
Data-Min	-

In the demo, the value of this SPN is set and transmitted as 1000, which represents the value 400%.

7.3.6 PGN 61467 (0xF01B) – Undefined

The undefined PGN, 61467, is used to demonstrate the transmission and reception of an unsupported, raw message. Because of the PGN of the message, it must always be transmitted globally. In the demo, the priority is set to six and the message type to raw. The data of the message is set as an 8-byte array of the series 1..8.

7.3.7 PGN 45312 (0xB100) – MyMessage

The MyMessage PGN is a user defined message specifically defined for the demo application and containing one of every parameter type. Because of its length, it is transmitted as a segmented message. This message is not transmitted globally. In the demo, the priority is set to six and the message type to 'DATA'. The following is the specification of the PGN according to the XML configuration file:

PGN	45312	
Name	MyMessage	
Acronym		
Length	16 bytes	
SPN List	SPN	Start-bit
	520192	1
	520193	9
	520194	17
	520195	19
	520196	25

7.3.7.1 SPN 520192 – MyIntParam

The SPN is specified by the XML configuration file as follows:

SPN	520192
Name	MyIntParam
Resolution	1
Data-Max	255
Bit-Length	8
Type	Integer
J1939 Standard	false
Unit	"Volt"
Offset	0
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 1, which represents the value 1 Volt.

7.3.7.2 SPN 520193 – MyFloatParam

The SPN is specified by the XML configuration file as follows:

SPN	520193
Name	MyFloatParam
Resolution	0.1
Data-Max	25
Bit-Length	8
Type	Float
J1939 Standard	false
Unit	"°C"
Offset	0
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 2, which represents the value 0.2 degrees C.

7.3.7.3 SPN 520194 – MyBitFieldParam

The SPN is specified by the XML configuration file as follows:

SPN	520194
Name	MyBitFieldParam
Resolution	1
Data-Max	3
Bit-Length	2
Type	BitField
J1939 Standard	false
Unit	
Offset	0
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 3, which, as a BitField type, will not be converted, rather displayed as a unitless value. Because this parameter is only defined for this demo application, the value 3 has no defined meaning.

7.3.7.4 SPN 520195 – MyBinaryParam

The SPN is specified by the XML configuration file as follows:

SPN	520195
Name	MyBinaryParam
Resolution	1
Data-Max	15
Bit-Length	4
Type	Binary
J1939 Standard	false
Unit	
Offset	0
Data-Min	0

In the demo, the value of this SPN is set and transmitted as 4, which, as a binary type, will not be converted, rather displayed as a unitless value. Because this parameter is only defined for this demo application, the value 4 has no defined meaning.

7.3.7.5 SPN 520196 – MyStringParam

The SPN is specified by the XML configuration file as follows:

SPN	520196
Name	MyStringParam
Resolution	
Data-Max	
Bit-Length	104
Type	ASCII
J1939 Standard	false
Unit	
Offset	
Data-Min	

In the demo, the value of this SPN is set to the UINT32 casted address of the constant string, “HelloWorld!”.

8 Support

For more information on our products, FAQ lists and installation tips, please refer to the support area on our homepage (http://www.ixxat.de). There you will also find information on current product versions and available updates.

If you have any further questions after studying the information on our homepage and the manuals, please contact our support department. In the support area on our homepage you will find the relevant forms for your support request. In order to facilitate our support work and enable a fast response, please provide precise information on the individual points and describe your question or problem in detail.

If you would prefer to contact our support department by phone, please also send a support request via our homepage first, so that our support department has the relevant information available.