J1939

Stack and SDK for esd CAN Hardware

Software Manual

to Product C.1130.10,
C.1130.11, C.1130.15, C.1130.09

NOTE

The information in this document has been carefully checked and is believed to be entirely reliable. esd makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. esd reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of esd gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of esd gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd.eu
Internet: www.esd.eu

USA / Canada:
esd electronics Inc.
525 Bernardston Road
Suite 1
Greenfield, MA 01301
USA

Phone: +1-800-732-8006
Fax: +1-800-732-8093
E-mail: us-sales@esd-electronics.com
Internet: www.esd-electronics.us

Trademark Notices

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Document file:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\J1939 SDK\CAN-J1939-Stack_Software_Manual_12.odt
Date of print:	2011-09-20

Software version (Obj):	from Rev. 1.0.0
Software version (Src):	from Rev. 1.0.0

Document History
Technical changes are marked in the document history with an additional "**!**".

!	Revision	Chapter	Page	Changes versus previous version	Date
	1.0	all	all	First Version	2009-09-15
	1.0a	1, 3.1	8, 23	Added infos for Linux Version	2010-05-18
		3.1	23	Minor changes to Quick Start descriptions	
		8	64	Reworked Headers / Minor changes	
	1.1	-	-	Released version.	2010-07-06
	1.2	1	7-10	Updated stack descriptions	2011-09-20

Technical details are subject to change without further notice.

This page is intentionally left blank.

Contents

1. esd J1939 SDK Contents....7
    1.1 Windows Object License for esd CAN hardware....8
    1.2 Linux Object License for esd CAN hardware....8
    1.3 Source License....9
    1.4 CAN-Tools....10
        1.4.1 CANreal with J1939 plug in....10
        1.4.2 J1939 Device Simulator & Monitor....11

2. Overview of J1939....13
    2.1 Physical Layer (J1939/11)....13
    2.2 Network Management (J1939/81)....15
        2.2.1 Device NAME....15
        2.2.2 Device ADDRESS....16
    2.3 Message Format and Transport (J1939/21)....18
        2.3.1 Parameter Group Number (PGN)....18
        2.3.2 Protocol Data Unit (PDU)....19
        2.3.3 Message Types and Transport Protocols....20
    2.4 Application Layer (J1939/7x)....21

3. esd j1939 Stack....22
    3.1 Quick Start....23
        3.1.1 Windows .dll Version....23
        3.1.2 Linux .so Version....23
        3.1.3 Source Code Version....23
    3.2 Examples....24
        3.2.1 Send PGN with callback....24
        3.2.2 Send PGN without callback....24
        3.2.3 j_setCallback_PGNSend....25
        3.2.4 j_setCallback_PGNReceived....26
        3.2.5 j_setCallback_ClaimEvent....27
        3.2.6 j_setCallback_PGNAnnounce....28
        3.2.7 j_setCallback_RequestReceived....29
        3.2.8 Complete Application....30

4. Data Structure Index....35
    4.1 Data Structures....35

5. File Index....36
    5.1 File List....36

6. Data Structure Documentation....37
    6.1 cb_ClaimEventInfo_t Struct Reference....37
    6.2 cb_DataReceivedInfo_t Struct Reference....38
    6.3 cb_DataSendInfo_t Struct Reference....40
    6.4 cb_PGNAnnounceInfo_t Struct Reference....41
    6.5 cb_RequestReceivedInfo_t Struct Reference....42
    6.6 dataSendInfo_t Struct Reference....43

7. File Documentation, Source Code Version....44
    7.1 j1939CAN.h File Reference....44
        7.1.1 Detailed Description....44
    7.2 j1939defs.h File Reference....45
        7.2.1 Detailed Description....45
        7.2.2 Define Documentation....45

7.3 j1939stack.h File Reference....48
7.3.1 Detailed Description....48
7.3.2 Functions....49
7.3.3 Macros....49
7.3.4 Typedef Documentation....50
7.3.5 Function Documentation....50

8. Library Versions....64
8.1 Defines in Library Versions....64
8.2 Differences between library and source code version....64

9. Index....65

10. Reference....70

1. esd J1939 SDK Contents

The esd J1939 SDK is available in different forms:

Type	Properties	Order No.
J1939 Stack for Windows	J1939 Stack for Windows (object code, runtime licence) for esd CAN hardware as Win32 library, incl. CANreal, J1939 plug in, J1939 DSM, esd CAN Windows driver licence, example source code	C.1130.10
J1939 Stack for Linux	J1939 Stack for Linux (object code, runtime licence) for esd CAN hardware as shared library (32/64 bit), incl. J1939 DSM (32/64 bit), esd CAN Linux driver licence, example source code	C.1130.11
J1939 Stack (Source Code)	J1939 Stack (source code, project licence) for microcontrollers (SoC with CAN support)	C.1130.15
J1939 Starter Kit	J1939 Starter Kit CAN-USB/2 interface module, complete wiring for two CAN nodes, incl. J1939 Stack for Windows (order no. C.1130.10)	C.1130.09
Manuals:		
J1939-Stack ME	Software Manual in English (this manual)	C.1130.21

Table 1: Order information

1.1 Windows Object License for esd CAN hardware

Also called “Windows .dll Version”, includes:

- Windows driver license
- J1939 Stack DLL
- CANreal with J1939 plug in
- J1939 Device Simulator & Monitor (DSM)
- Example source code

No CAN hardware specific code is required, the J1939 stack uses the esd NTCAN API to access the hardware:

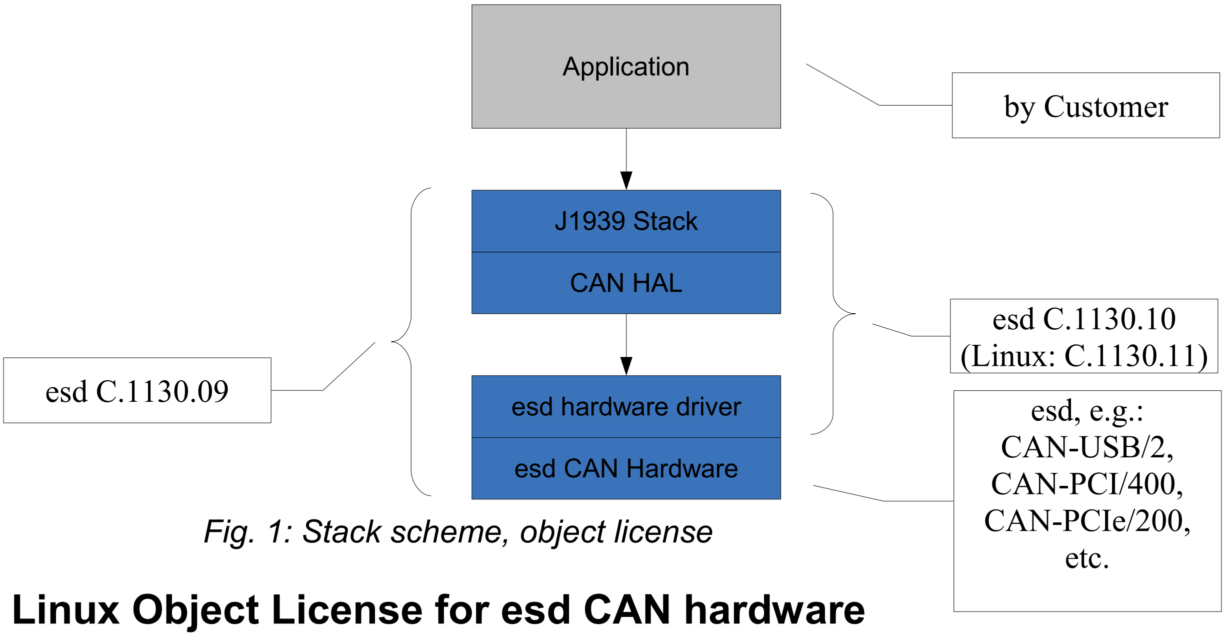

Application
by Customer
J1939 Stack
CAN HAL
esd C.1130.10
(Linux: C.1130.11)
esd C.1130.09
esd hardware driver
esd CAN Hardware
esd, e.g.:
CAN-USB/2,
CAN-PCI/400,
CAN-PCIe/200,
etc.

Fig. 1: Stack scheme, object license

1.2 Linux Object License for esd CAN hardware

Also called “Linux .so Version”, includes:

- Linux driver license
- J1939 Stack shared library
- J1939 Device Simulator & Monitor[1] (DSM)
- Example source code

No CAN hardware specific code is required, the J1939 stack uses the esd NTCAN API to access the hardware, see Fig. 1.

---

[1] GTK+ application (Available on most desktop environments, such as GNOME, KDE, Xfce, etc.)

1.3 Source License

Also called “Source Code Version”, for embedded CPUs with CAN controller and timer (SoC with CAN support), includes:

- J1939 Stack source code
- Example source code

Features:

- Written in ANSI-C
- Easy adaptation to other target systems due to well defined abstraction layer
- For big/little endian systems, CPU independent
- Many settings can be adapted to the requirements of the application and the available hardware resources by a simple configuration file at compile time, see section 7.2

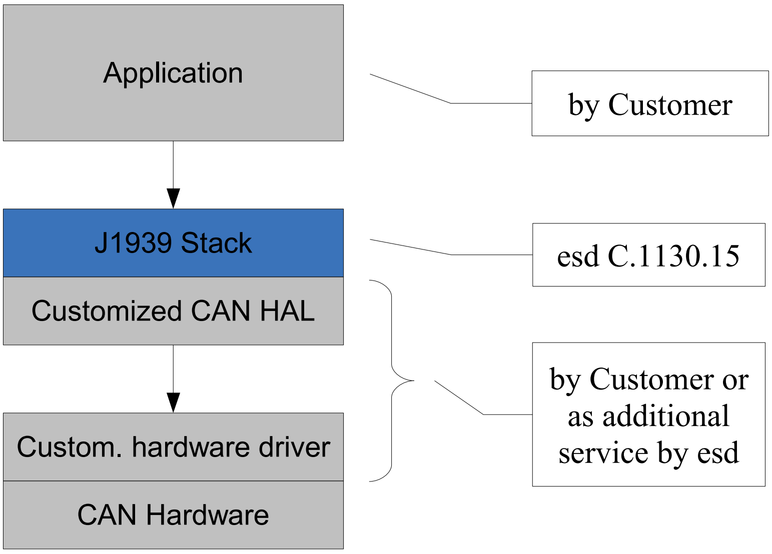

Application
by Customer
J1939 Stack
esd C.1130.15
Customized CAN HAL
by Customer or as additional service by esd
Custom. hardware driver
CAN Hardware

Fig. 2: Stack scheme, source license

The stack's CAN HAL (a single .c file) has to be adapted according to your hardware.

This HAL basically consists of a few functions to receive and send CAN frames that have to be implemented – as a help this file still contains the esd implementations that were used for the NXP LPC2292 and Fujitsu MB90543 microcontrollers as well as the use of the esd NTCAN API.

As an additional service esd also offers the implementation for your specific hardware.

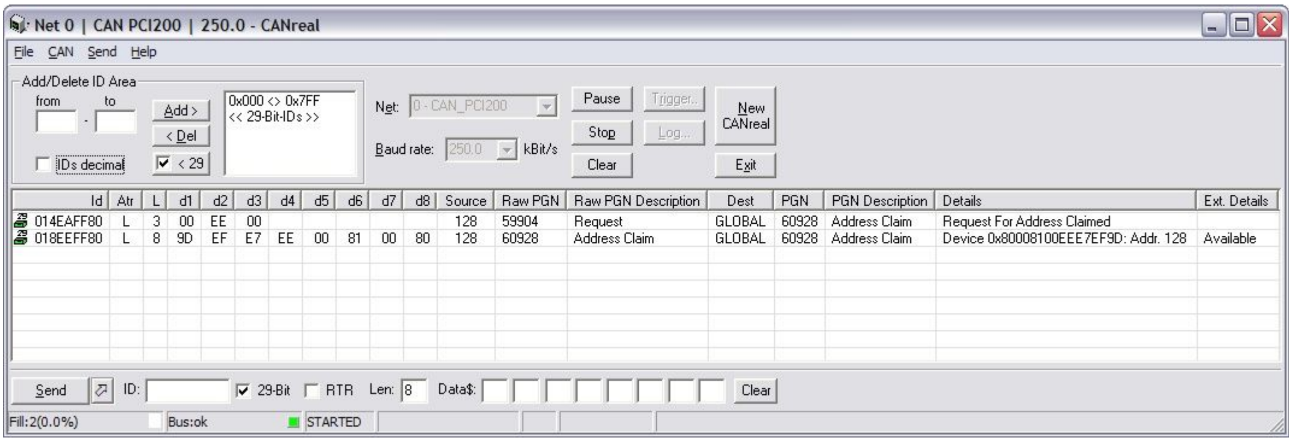
1.4 CAN-Tools

1.4.1 CANreal with J1939 plug in

For Windows and esd CAN hardware only. CANreal is also included in esd's CAN SDK. See documentation there. Features:

- Display and recording of CAN message frames with high resolution time stamps
- Protocol interpreter for J1939
- Supports message ID filtering
- Multiple instances of the software on the same or different channels can run at the same time
- Supports transmission of user defined CAN message frames

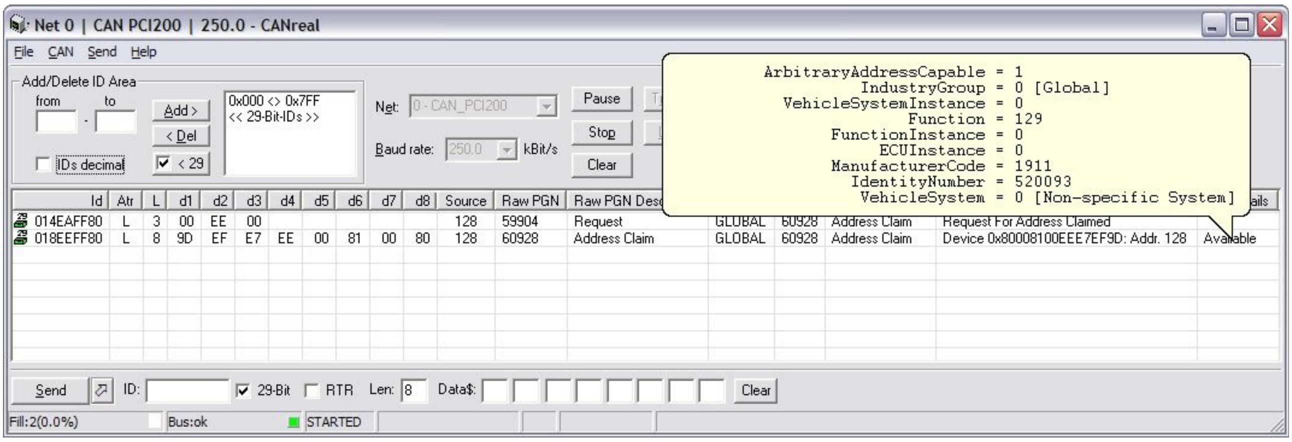
The J1939 plug in adds several columns to the CANreal display that interpret the CAN Id, etc.:


Net 0 | CAN PCI200 | 250.0 - CANreal
File CAN Send Help
Add/Delete ID Area
from
to
Add >
< Del
☐ IDs decimal
☑ < 29
0x000 <> 0x7FF
<< 29-Bit-IDs >>
Net: 0 - CAN_PCI200
Baud rate: 250.0 kBit/s
Pause
Stop
Clear
Trigger..
Log...
New CANreal
Exit
Send
ID:
☑ 29-Bit
☐ RTR
Len: 8
Data$:
Clear
Fill:2(0.0%)
Bus:ok
STARTED

Id	Atr	L	d1	d2	d3	d4	d5	d6	d7	d8	Source	Raw PGN	Raw PGN Description	Dest	PGN	PGN Description	Details	Ext. Details
014EAFF80	L	3	00	EE	00						128	59904	Request	GLOBAL	60928	Address Claim	Request For Address Claimed	
018EEFF80	L	8	9D	EF	E7	EE	00	81	00	80	128	60928	Address Claim	GLOBAL	60928	Address Claim	Device 0x80008100EEE7EF9D: Addr. 128	Available

Fig. 3: CANreal with J1939 plug in

For most PGNs even a data interpretation is available:


Net 0 | CAN PCI200 | 250.0 - CANreal
File CAN Send Help
Add/Delete ID Area
from
to
Add >
< Del
☐ IDs decimal
☑ < 29
0x000 <> 0x7FF
<< 29-Bit-IDs >>
Net: 0 - CAN_PCI200
Baud rate: 250.0 kBit/s
Pause
Stop
Clear
Send
ID:
☑ 29-Bit
☐ RTR
Len: 8
Data$:
Clear
Fill:2(0.0%)
Bus:ok
STARTED

```
ArbitraryAddressCapable = 1
          IndustryGroup = 0 [Global]
VehicleSystemInstance = 0
               Function = 129
       FunctionInstance = 0
            ECUInstance = 0
       ManufacturerCode = 1911
         IdentityNumber = 520093
          VehicleSystem = 0 [Non-specific System]
```

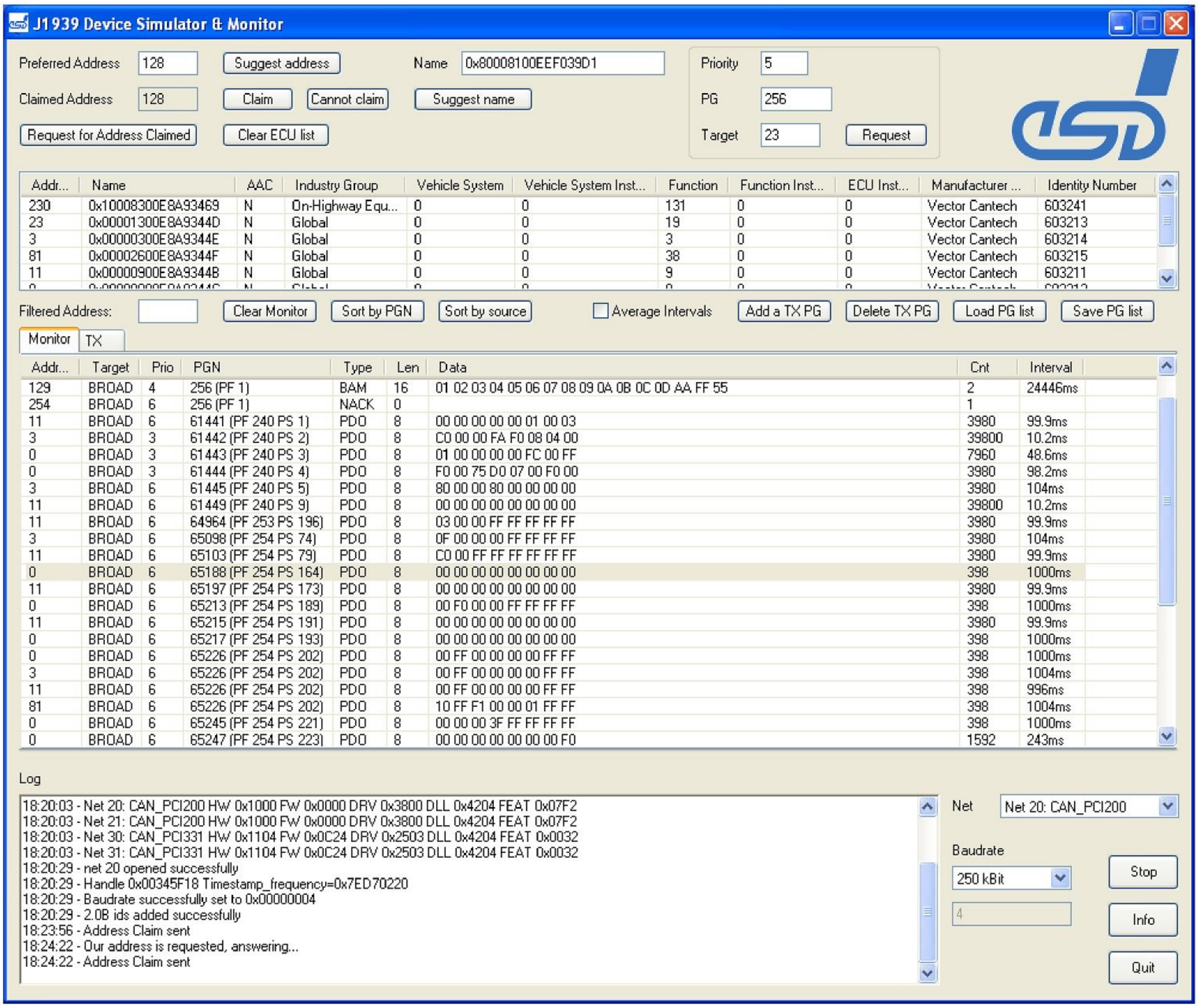
Id	Atr	L	d1	d2	d3	d4	d5	d6	d7	d8	Source	Raw PGN	Raw PGN Desc					ails
014EAFF80	L	3	00	EE	00						128	59904	Request	GLOBAL	60928	Address Claim	Request For Address Claimed	
018EEFF80	L	8	9D	EF	E7	EE	00	81	00	80	128	60928	Address Claim	GLOBAL	60928	Address Claim	Device 0x80008100EEE7EF9D: Addr. 128	Available

Fig. 4: CANreal with J1939 plug in

1.4.2 J1939 Device Simulator & Monitor

This tool allows to monitor J1939 traffic and to create J1939 messages to act like a device.

- Simulates a J1939 ECU
- Multiple instances of the software on the same or different channels can run at the same time
- Monitors complete PGN traffic on the bus
- Tx messages can be set up for cyclic transmission or for transmission on request only
- Transmission of PGN can be triggered manually
- Manually sending of requests
- Log shows all user interaction and anomalies in the J1939 protocol parsing
- Supported operating system: Windows, Linux (as GTK+ application)



Preferred Address: 128
Claimed Address: 128
Name: 0x80008100EEF039D1
Priority: 5
PG: 256
Target: 23

Addr...	Name	AAC	Industry Group	Vehicle System	Vehicle System Inst...	Function	Function Inst...	ECU Inst...	Manufacturer ...	Identity Number
230	0x10008300E8A93469	N	On-Highway Equ...	0	0	131	0	0	Vector Cantech	603241
23	0x00001300E8A9344D	N	Global	0	0	19	0	0	Vector Cantech	603213
3	0x00000300E8A9344E	N	Global	0	0	3	0	0	Vector Cantech	603214
81	0x00002600E8A9344F	N	Global	0	0	38	0	0	Vector Cantech	603215
11	0x00000900E8A9344B	N	Global	0	0	9	0	0	Vector Cantech	603211

Filtered Address:
☐ Average Intervals

Addr...	Target	Prio	PGN	Type	Len	Data	Cnt	Interval
129	BROAD	4	256 (PF 1)	BAM	16	01 02 03 04 05 06 07 08 09 0A 0B 0C 0D AA FF 55	2	24446ms
254	BROAD	6	256 (PF 1)	NACK	0		1	
11	BROAD	6	61441 (PF 240 PS 1)	PDO	8	00 00 00 00 00 01 00 03	3980	99.9ms
3	BROAD	3	61442 (PF 240 PS 2)	PDO	8	C0 00 00 FA F0 08 04 00	39800	10.2ms
0	BROAD	3	61443 (PF 240 PS 3)	PDO	8	01 00 00 00 00 FC 00 FF	7960	48.6ms
0	BROAD	3	61444 (PF 240 PS 4)	PDO	8	F0 00 75 D0 07 00 F0 00	3980	98.2ms
3	BROAD	6	61445 (PF 240 PS 5)	PDO	8	80 00 00 80 00 00 00 00	3980	104ms
11	BROAD	6	61449 (PF 240 PS 9)	PDO	8	00 00 00 00 00 00 00 00	39800	10.2ms
11	BROAD	6	64964 (PF 253 PS 196)	PDO	8	03 00 00 FF FF FF FF FF	3980	99.9ms
3	BROAD	6	65098 (PF 254 PS 74)	PDO	8	0F 00 00 00 FF FF FF FF	3980	104ms
11	BROAD	6	65103 (PF 254 PS 79)	PDO	8	C0 00 FF FF FF FF FF FF	3980	99.9ms
0	BROAD	6	65188 (PF 254 PS 164)	PDO	8	00 00 00 00 00 00 00 00	398	1000ms
11	BROAD	6	65197 (PF 254 PS 173)	PDO	8	00 00 00 00 00 00 00 00	3980	99.9ms
0	BROAD	6	65213 (PF 254 PS 189)	PDO	8	00 F0 00 00 FF FF FF FF	398	1000ms
11	BROAD	6	65215 (PF 254 PS 191)	PDO	8	00 00 00 00 00 00 00 00	3980	99.9ms
0	BROAD	6	65217 (PF 254 PS 193)	PDO	8	00 00 00 00 00 00 00 00	398	1000ms
0	BROAD	6	65226 (PF 254 PS 202)	PDO	8	00 FF 00 00 00 00 FF FF	398	1000ms
3	BROAD	6	65226 (PF 254 PS 202)	PDO	8	00 FF 00 00 00 00 FF FF	398	1004ms
11	BROAD	6	65226 (PF 254 PS 202)	PDO	8	00 FF 00 00 00 00 FF FF	398	996ms
81	BROAD	6	65226 (PF 254 PS 202)	PDO	8	10 FF F1 00 00 01 FF FF	398	1004ms
0	BROAD	6	65245 (PF 254 PS 221)	PDO	8	00 00 00 3F FF FF FF FF	398	1000ms
0	BROAD	6	65247 (PF 254 PS 223)	PDO	8	00 00 00 00 00 00 00 F0	1592	243ms

Log

```
18:20:03 - Net 20: CAN_PCI200 HW 0x1000 FW 0x0000 DRV 0x3800 DLL 0x4204 FEAT 0x07F2
18:20:03 - Net 21: CAN_PCI200 HW 0x1000 FW 0x0000 DRV 0x3800 DLL 0x4204 FEAT 0x07F2
18:20:03 - Net 30: CAN_PCI331 HW 0x1104 FW 0x0C24 DRV 0x2503 DLL 0x4204 FEAT 0x0032
18:20:03 - Net 31: CAN_PCI331 HW 0x1104 FW 0x0C24 DRV 0x2503 DLL 0x4204 FEAT 0x0032
18:20:29 - net 20 opened successfully
18:20:29 - Handle 0x00345F18 Timestamp_frequency=0x7ED70220
18:20:29 - Baudrate successfully set to 0x00000004
18:20:29 - 2.0B ids added successfully
18:23:56 - Address Claim sent
18:24:22 - Our address is requested, answering...
18:24:22 - Address Claim sent
```

Net: Net 20: CAN_PCI200
Baudrate: 250 kBit
4

Fig. 5: J1939 DSM

When it's running all claim messages are monitored and devices details are listed. All send PGN data is listed, including those sent by transport protocol and to other devices. Also allows to send own claim/cannot claim/PGN data messages. PGN data can be sent periodically and also by different transport protocols.

Quick Start:

1. Select CAN net and baudrate and click start (lower right corner of the window)
2. Enter a “preferred address” and click “claim” (upper left) to send the “address claimed” message. (On conflict another address is claimed automatically)
3. Now the software is known as a regular device in the J1939 network
4. Now you can, for example:
    - **Request list of all devices**: click “Request for Address Claimed” (upper left). Device list is updated for each device that answers with its address information
    - **Request a PGN**: enter “PG” and “target” (upper right) and click “request”. All devices offering that PGN should answer; seen in “Monitor” tab.
    - **Send own PGN**: click “add a TX PG” (below device list) and enter desired data. Data is sent periodically as entered. And also when requested by another device. And also by appropriate transport protocol when requested (depending on request, to global or to specific device)

2. Overview of J1939

SAE J1939 [SAE193900] is a Controller Area Network (CAN) [ISO11898] based protocol that has been developed to provide a standard architecture by which multiple Electronic Control Units (ECUs) on a (mostly light- or heavy-duty) vehicle can communicate. It is based on the extended frame format (29-bit identifier) of the CAN 2.0B specification using a fixed baud rate of 250 Kbit/s.
This chapter is intended to introduce several J1939 terms which are necessary for an understanding of the functional description of the J1939 Stack. For more details about the various layers of the J1939 protocol you have to refer to the SAE J1939 documents:

- The physical layer (J1939/11 [SAE193911]) describes the electrical interface to the bus.
- The data link layer (J1939/21 [SAE193921]) describes the rules for constructing a message, accessing the bus, and detecting transmission errors.
- The network layer (J1939/31 [SAE193931]) describes mechanisms and services to connect several J1939 networks.
- The application layer (J1939/71 [SAE193971]and J1939/73) defines the specific data contained within each message sent across the network.
- The network management layer (J1939/81 [SAE193981]) is concerned with the management of source addresses and with the detection and reporting of network related errors.

2.1 Physical Layer (J1939/11)

The physical layer of a J1939 network is based on [ISO11898]. For J1939 a shielded twisted pair wire with a maximum length of 40 m is defined for the CAN communication. The CAN baud rate is fixed to 250 Kbit/s and the maximum number of ECUs is limited to 30 for one segment. Several segments can be connected using specialized interconnection ECUs which services are described in [SAE193931].

The sophisticated fault confinement features of the CAN bus are fully supported. CAN nodes are able to distinguish between permanent failure and temporary disturbances.
Within each CAN node an 8-bit transmit error counter and an 8-bit receive error counter are used. If one of the five error types CRC error, stuff error, form error, bit error or Acknowledgement (ACK) error is detected, the corresponding error counter is increased.
If a reception or transmission is completed successfully, the corresponding error counter is decremented. Consequently permanent failures result in large counts, whereas temporary disturbances result in small counts that recover back to zero in a running system.

If an error is detected during reception, the Rx error counter is increased by 1. The error counter is increased by 8, if the first bit after transmission of the error flag is dominant, which suggests that the error is not detected by other nodes.

The Tx error counter is always increased by 8, if an error is detected while the node is transmitting. After a successful reception the Rx-error counter is decreased by 1 and after a successful transmission the Tx error counter is decreased by 1. Thus only the error counters of a node will increment rapidly, if a fault is local to this node.
Depending on the value of its error counters the node takes one of the three states *error active*, *error passive* or *bus off.*

Error Active: Regular operational state of the node, with both counts less than 128. In this state the node can participate in usual communication. If it detects any error during communication, an ERROR ACTIVE FLAG, consisting of 6 dominant bits, is transmitted. This blocks the current transmission.

Error Passive: When either counter exceeds 127, the node is declared error passive. This indicates that there is an abnormal level of errors. The node still participates in transmission and reception, but it has an additional time delay after a message transmission before it can initiate a new message transfer of its own. This extra delay for the error passive node which is known as suspended transmission results from transmission of 8 additional recessive bits at the end of the frame. This means that an error passive node loses arbitration to any error active node regardless of the priority of their Ids. When an error passive node detects an error during communication, it transmits an ERROR PASSIVE FLAG consisting of 6 recessive bits. These will not disturb the current transmission (assuming another node is the transmitter) if the error turns out to be local to the error passive node.

Bus-Off: When the transmit error count exceeds 255 the node is declared bus off. This indicates that the node has experienced consistent errors whilst transmitting. This state restricts the node from sending any further transmission. The node will eventually be re-enabled for transmission and become error active after it has detected 128 occurrences of 11 consecutive recessive bits on the bus which indicate periods of bus inactivity.

2.2 Network Management (J1939/81)

On a J1939 network each device has a unique device NAME and a unique device ADDRESS. This section describes the relation between the device NAME and the device ADDRESS.

2.2.1 Device NAME

On a J1939 network every ECU has a unique 64-bit device NAME which contains information about the vendor and the device function with the following format. For the fields with cursive typeface values predefined by SAE in appendix B of [SAE193900] have to be used, the other fields are manufacturer specific.

1 Bit	3 Bits	4 Bits	7 Bits	1 Bit	8 Bit	5 Bit	3 Bit	11 Bit	21 Bit
Arbitrary Address Capable	Industry Group	Vehicle System Instance	Vehicle System	Reserved	Function	Function Instance	ECU Instance	Manufacturer Code	Identity Number

Table 2: Format of J1939/81 device NAME

Arbitrary Address Capable:	Indicate the capability to solve address conflicts (see next chapter). Set to 1 if the device is *Arbitrary Address Capable*, set to 0 if it's *Single Address Capable*.
Industry Group:	One of the predefined J1939 industry groups.
Vehicle System Instance:	Instance number of a vehicle system to distinguish two or more device with the same Vehicle System number in the same J1939 network. The first instance is assigned to the instance number 0.
Vehicle System:	A subcomponent of a vehicle, that includes one or more J1939 segments and may be connected or disconnected from the vehicle. A Vehicle System may be made of one or more functions. The Vehicle System depends on the Industry Group definition.
Reserved:	This field is reserved for future use by SAE.
Function:	One of the predefined J1939 functions. The same function value (upper 128 only) may mean different things for different Industry Groups or Vehicle Systems.
Function Instance:	Instance number of a function to distinguish two or more devices with the same function number in the same J1939 network. The first instance is assigned to the instance number 0.
ECU Instance –	Identify the ECU instance if multiple ECUs are involved in performing a single function. Normally set to 0.
Manufacturer Code:	One of the predefined J1939 manufacturer codes.
Identity Number –	A unique number which identifies the particular device in a manufacturer specific way.

2.2.2 Device ADDRESS

On a J1939 network every device uses an 8-bit unique ADDRESS which is embedded as source and/or destination of a J1939 message.
The address numbers 0..253 are valid ECU addresses, *address* 254 is used as the *Null Address* for ECUs which haven't claimed or failed to claim an address and 255 is used as the *broadcast* or *global address*
Every device is assigned a *Preferred Address* at power-on from the range of the 254 possible *addresses* in a J1939 network, according to the following recommended layout:

Address	Preferred or Default Usage
0 to 127	Reserved for ECU type specific predefinitions in appendix B of [SAE193900].
128 to 247	Reserved for industry specific ECU types.
248 to 253	Reserved for special ECU types.
254	Null Address
255	Broadcast Address

Table 3: Preferred Address Assignment

Before a device can use this address for communication it has to verify that this address isn't in use by any other device by using one of the two possible *Address Claiming* strategies with the services described in [SAE193981].

1. **Forced Address Claim**: Send the *Address Claim* message with the *Preferred Address* and the device *Name* in it. Every active device on the network compares this *ADDRESS* with its own already assigned *ADDRESS* and replies with an own *Address Claim* message if it has a higher priority. The priority is derived from the unique device *Name*. The device with the lower priority has to solve the address conflict.

2. **Cooperative Address Claim**: Send a *Request for Address Claim* message whereupon every active device replies with its own *Address Claim* message which can be used by the initiating device to select an ADDRESS which isn't already in use by another device.

Every time an ECU starts the *address claim* procedure, an address conflict may occur for the initiating ECU or for an ECU which has already an assigned address because the initiating ECU has a higher priority.

The capability to solve this address conflict divides J1939 device into two groups:

- ***Single Address Capable*** ECUs, which are not able to change their *Address* after the initial successful assignment. This group can be divided into four sub-groups:
    - *Non-Configurable Address* devices, which have a fixed *address* that can only be changed by a firmware update.
    - *Service Configurable Address* devices, whose *address* can only be changed in a vendor specific way with proprietary tools.
    - *Command Configurable Address* devices, which are able to change their *address* via the *Commanded Address* message defined in [SAE193981] with the help of e.g. another ECU.
    - *Self-Configurable Address* devices, which choose their *address* based on an internal algorithm during the initial *address claim* procedure but which are unable to reclaim an address.

- ***Arbitrary Address Capable*** ECUs, which are able to claim an *address* during the *initial address* claim procedure and later on if it's necessary to *reclaim* an address because of a conflict.

  To give *Arbitrary Address Capable* devices the possibility not to claim the already assigned address of a *Single Address Capable* device, even if it has the higher priority, the capability to solve address conflicts is also indicated in the device *NAME* (see previous chapter) and an *Arbitrary Address Capable* device can claim an unused *address* if it supports the cooperative address claim method described above.

2.3 Message Format and Transport (J1939/21)

On a J1939 network the content of each message is described by its Parameter Group (PG), independent of the message type (command, requests, service specific, etc.) and the message length.
Parameter Groups for management and control of the network are defined in [SAE193921], [SAE193931] and [SAE193981]. Besides vendor specific PGs most of communication in a vehicle is based on application specific PGs which are defined in [SAE193971].

2.3.1 Parameter Group Number (PGN)

Each PG is assigned a unique Parameter Group Number (PGN). The PGN is represented by a 24-bit value with the following format, where only 18 bits are in use:

6 Bit	1 Bit	1 Bit	8 Bit	8 Bit
0	Reserved	Data Page (DP)	PDU Format (PF)	PDU Specific (PS)

Table 4: Format of the Parameter Group Number (PGN)

Reserved: This field is reserved for future use by SAE.

Data Page: Bit to select one of the two possible data pages for PGN.
At the moment only PGNs for page 0 are defined.

PDU Format: Defines the the format of the Process Data Unit (PDU),described below,
used for communication of this PG.

PDU Specific: Defines a PDU specific parameter that depends on the PDU format of the PG.

The J1939 distinguishes between two types of PGNs:

- Specific PGNs for Parameter Groups that are directed to a specific device in a peer-to-peer manner. For these PGNs the PF field is smaller than 240 and the PS field is set to 0.
- Global PGNs for Parameter Groups which are directed to all devices in a broadcast manner. For these PGNs the PF field is greater than 239 and the PS field is part of the PGN.

This PGN structure allows a total number of 4336 PGNs (240 specific PGNs and 16 * 256 global PGNs) per *Data Page* or 8672 for both pages. That the majority of the PGNs is of the global type shows that the J1939 network is based on (often unsolicited or periodic) broadcasts which gives any device the possibility to use the PG data without an explicit request.

2.3.2 Protocol Data Unit (PDU)

J1939 messages are structured as Protocol Data Units (PDU) which consists of the seven fields *Priority, Reserved, Data Page, PDU Format, PDU specific, Source Address* and data fields. These fields encompass the complete PGN and all fields but the data are located in the 29-Bit of the CAN identifier in Extended Frame Format, which means they are part of the arbitration process of the CAN message on the bus:

3 Bit	1 Bit	1 Bit	8 Bit	8 Bit	8 Bit
Priority	Reserved	Data Page (DP)	PDU Format (PF)	PDU Specific (PS)	Source Address (SA)
Priority	Parameter Group Number (PGN)				Source Address

Table 5: Structure of 29-Bit CAN Identifier

Priority: Priority of the message as 3-bit parameter. A value of 0 is the highest priority. The higher priority is typically used for high-speed messages. PGNs defined in [SAE193971] is assigned a default priority.

Reserved: This field is reserved for future use by SAE. It should always be set to 0.

Data Page: Bit used as data page selector.
At the moment all defined messages are defined in *Data Page* 0.

PDU Format: The PDU Format defines if this is a Specific PGN with a destination address (*PDU1 Format* with a PF value in the range from 0 to 239) which is transmitted peer-to-peer or a Global PGN (*PDU2 format* with a PF value in the range from 240 to 255) which is transmitted in a broadcast manner.

PDU Specific: Defines a *PDU Specific* parameter that depends on the PDU format of the PG.

- **PDU1 Format:**
  The PF field contains the *Destination Address* (DA).
  PDU1 Format messages can be requested or send as unsolicited messages.

- **PDU2 Format:**
  The PF field contains the *Group Extension* (GE), which expands the number of possible broadcast PGs that can be represented by the PF . *PDU2 Format* messages can be requested or send as unsolicited messages.

Source Address:
Defines the *address* of the ECU transmitting this PDU. Because every ECU claims an individual *address,* this field guarantees the CAN bus related requirement that the identifier of a message is unique.

2.3.3 Message Types and Transport Protocols

J1939 defines the following five message types:

1. **COMMAND:** Send a PDU to a specific device or as broadcast to command to perform a certain action based on the PGN of this PDU. The data fields of this PDU contain the commanded data.

2. **REQUEST**: Send a PDU to request information globally or from a specific device. The PGN of this PDU has a predefined fixed value [SAE193921]and the PGN being requested is contained in the first three bytes of the data fields.

3. **BROADCAST/RESPONSE:** Unsolicited broadcast of information or the response to a COMMAND or REQUEST. The PDU contains the PGN of the of the parameter and the data fields the parameter data.

4. **ACKNOWLEDGEMENT**: Positive or negative acknowledgement to a COMMAND or REUEST as handshake mechanism between transmitting and receiving devices. The PGN of this PDU has a predefined fixed value [SAE193921] and data fields contain protocol specific data. Possible types of acknowledgement are the *Positive Acknowledgement* (ACK), the *Negative Acknowledgement* (NACK) and *Access Denied*.

5. **GROUP FUNCTION**: This type of message is intended for groups of special functions (network management functions, multi-packet transport functions, etc.). The PGN of this PDU has group specific predefined fixed values [SAE193921] and the data fields are used in a group specific way.

The majority of PGs in a J1939 network require only one PDU as the data fits into the 8 data bytes of a CAN frame. PGs with 9 up to 1785 bytes of data are sent as multi-packet messages for which [SAE193921] defines two types of transport protocols:

- **Broadcast Announce Message** (TP_BAM): The message data is directed to all devices on the J1939 network (global destination address) split up into multiple packets which are sent as PDUs with predefined PGNs. The delay time between consecutive PDUs of a TP_BAM message is device specific and shall be between 50 and 200 ms according to [SAE193921].

- **Connection Management** (TP_CM): The transmitting and receiving device create a virtual peer-to-peer channel where the message data is transferred with a protocol supporting a hand-shake mechanism. The message data is split up into multiple packets which are sent as PDUs with predefined PGNs. According to [SAE193921] the originator of the transfer has to guarantee a maximum time of 200 ms between consecutive packets.

2.4 Application Layer (J1939/7x)

The Application Layer [SAE193971] defines (industry specific) application parameter in detail. For every parameter the following properties are defined:

- Parameter name and description
- A 19-bit parameter specific number, the so called Suspected Parameter Number (SPN)
- The parameter type (measured value, status, etc.)
- The parameter resolution (unit, scaling, offset)
- The parameter range
- The parameter data size (in bits or bytes)

In order to optimize the PDU and CAN bandwidth usage several SPNs are grouped in a PG based on their function, transmission rate or subsystem. For this purpose the Application Layer documents contain Parameter Group Definitions with the following properties:

- A parameter group name
- The PDU Format (PF) field (PDU1 Format or PDU2 Format)
- The PDU Specific (PS) field according to the PDU format (Destination Address (DA) or Group Extension (GE).
- The Data Page (0 or 1).
- The default priority of the message
- The message transmission type (cyclic with transmission repetition rate or requested)
- The data length in bytes.
- The start bit position and length of every embedded SPN in the data field of the CAN frame.
- The transport protocol to use if the data size of a single SPN exceeds 8 bytes or is variable.

The combination of *PDU Format*, *PDU Specific* and *Data Page* is the PGN assigned to this PG. New parameter definitions have to be registered at SAE. Once a parameter has assigned an SPN and this SPN is assigned to a PG this assignment isn't changed in future revisions of the Application Layer document.

3. esd j1939 Stack

This stack allows quick development of applications supporting the SAE J1939 protocol.

It offers especially:

- **Sending of PGN data**
  No need to care about BAM or RTS/CTS: done automatically depending on data size and destination. Optionally done by callback, to send even larger amount of data with a minimum of resource usage.
  Automatic broadcasts: stack can automatically broadcast PGNs in a given interval.

- **Receiving PGN data**
  Done in convenient callback function for easy differentiating between sources and types (complete data, data chunk, interruption etc.) Filtering by PG Number and/or source address possible. Splitting to extra callback for Diagnostic Messages possible.

- **Network Management**
  Automatic handling of address claiming procedures. All four address configuration types possible. (Non Configurable, Service Configurable, Command Configurable and Self Configurable)

- **Multiple Devices**
  Even multiple devices in a single software instance are possible. Activated simply by changing a value in a `#define` .Interface remains unchanged except for an additional `deviceNumber` parameter in every function.

- **Configuration**
  Resource specific features controllable by defines:
  Max number of possible simultaneously transport protocol transfers (Separated by incoming and outgoing).
  Max number of queued BAMs. (BAM queue can even be set to consider message's priority when full)
  Max number of automatic broadcasts.
  Filter functions are excluded from build when defined to be unused.

- **Portability**
  ANSI C.
  Tested under little- and big endian systems.
  Simply adaptable to new systems usually just by adding some #includes and #defines. (Examples exist)

3.1 Quick Start

All code examples and documentations are also available in **HTML format**. This allows much easier navigation and is therefore **recommended**.

Installation also contains a "demo" folder with *Makefile* etc.

See installation folder / start menu.

3.1.1 Windows .dll Version

Depends on the following installed files: (besides esd CAN hardware and its driver installed)

- **j1939.h**
- **j1939.dll**
- **j1939.lib**

See j1939.h and Complete Application example (**j1939demo.c**) for available functions and their usage.

3.1.2 Linux .so Version

Depends on the following installed files: (besides esd CAN hardware and its driver installed)

- **j1939.h**
- **libj1939.so**

See j1939.h and Complete Application example (**j1939demo.c**) for available functions and their usage.

3.1.3 Source Code Version

Following files are needed by your application:

- **j1939defs.h**
- **j1939stack.h**
- **j1939stack.c**
- **j1939CAN.h**
- **j1939CAN.c**

Follow the steps below:

1. Adjust j1939defs.h (Chapter 7.2) to your needs/system.
2. Adapt j1939CAN.c (Chapter 7.1) to your system (if not yet included).
3. See j1939stack.h (Chapter 7.3) and Complete Application example for available functions and their usage. **Note**: Examples use an additional first parameter for device number (always 0 here), see also J_NUM_DEVICES

3.2 Examples

3.2.1 Send PGN with callback

Send PGN 0x2000 with 9 data bytes and priority 6 to address 128:

```
j_sendPGNByCallback(0, 128, 0x2000, 6, 9);
```

Stack gets data by callback:

```
void sendCallback(cb_DataSendInfo_t* sendInfos)
 {
    if (sendInfos->nBytes == 0) {
          // stack has finished collecting data (for that pgn)
          return;
      }

    switch (sendInfos->pgn) {
          case 0x2000:
            sendInfos->data = &dataForPGN2000[sendInfos->offset];
            break;
      }
 }
```

3.2.2 Send PGN without callback

Broadcast PGN 0x1100 with 10 data bytes:

```
sendInfos.pgn = 0x1100;
sendInfos.data = &data[0];
sendInfos.dataLen = 10;
sendInfos.destAddr = ADR_GLOBAL;
sendInfos.priority = 7;

j_sendPGN(0, &sendInfos);
```

Where `data` to send and `sendInfos` are defined as:

```
const uint8_t   data[10] = {...};

dataSendInfo_t  sendInfos;
```

3.2.3 j_setCallback_PGNSend

Set callback function:

```
    j_setCallback_PGNSend(0, &sendCallback);
```

Called function:

```
void sendCallback(cb_DataSendInfo_t* sendInfos)
{
    if (sendInfos->nBytes == 0) {
        /* stack has finished collecting data */
        /* could now free/update that data etc. */
        return;
    }

    switch (sendInfos->pgn) {
        /* stack needs new data for PGN: */

        case 0x1000:
            sendInfos->data = &dataToBroadcast[sendInfos->offset];
            break;

        case 0x3000:
            sendInfos->data = &testData[sendInfos->offset];
            break;
    }
}
```

3.2.4 j_setCallback_PGNReceived

Set callback function:

```
    j_setCallback_PGNReceived(0, &receiveCallback);
```

Called function:

```
int8_t receiveCallback(cb_DataReceivedInfo_t* rcvInfos)
{
    if (rcvInfos->status < 0) { /* multipacket transfer aborted */
        receiveBufferActive = 0;
        return 0; /* result is unused */
    }

    switch(rcvInfos->status) {

        /* Complete message at once: */
        case J_RCV_STATUS_SHORT_MSG:
            debugPrint_tntntn(LT_INFO, "received short PGN ", rcvInfos->pgn, "
                             (", rcvInfos->dataLen, " bytes) from ", rcvInfos->fromAddr);

            switch(rcvInfos->pgn) {
                /* handle PGNs */
            }
            return 0; /* result is unused */


        /* Multipacket Transfer, completed: */
        case J_RCV_STATUS_LONG_MSG_COMPLETE:
            receiveBufferActive = 0;
            debugPrint_tntntn(LT_INFO, "received long PGN ", rcvInfos->pgn, "(",
                                 rcvInfos->offset, " bytes) from ", rcvInfos->fromAddr);

            switch(rcvInfos->pgn) {
                /* handle PGNs */
            }
            return 0; /* result is unused */


        /* Multipacket transfer, data chunk: */
        /* only called for accepted transfers, needs callback set in j_setCallback_PGNAnnounce(). */
        case J_RCV_STATUS_LONG_MSG_CHUNK:
            if (!receiveBufferActive)
                return 1; /* abort if buffer not 'activated' (should not happen...)*/

            /* add chunk to receiveBuffer: */
            if ((rcvInfos->offset + rcvInfos->dataLen) <= RECEIVE_BUFFER_SIZE)
                memcpy(&receiveBuffer[rcvInfos->offset], rcvInfos->data, rcvInfos->dataLen);

            return 0; /* ok, continue transfer */
    }


    return 1; /* abort unhandled incoming PGN transfers */
}
```

3.2.5 j_setCallback_ClaimEvent

Set callback function:

```
    j_setCallback_ClaimEvent(0, &claimEvent);
```

Called function:

```
void claimEvent(cb_ClaimEventInfo_t* infos)
{
    switch (infos->evtType) {
        case CLAIM_EVT_CLAIMED:
            ownAddress = infos->addr;
            debugPrint_tn(LT_INFO, "own address is now ", ownAddress);

            /* this is the best time to add autobroadcasts: */
            autoBroadcastId = j_addAutoBroadcast(0, 1000 * J_TIMER_TICKS_PER_MS, 0x1000, 6, 5);
            break;

        case CLAIM_EVT_FAILED:
            ownAddress = ADR_NULL;
            debugPrint(LT_WARN, "address claim failed/lost!");

            infos->addr = ADR_NULL; /* tell stack to give up (and send cannot claim) */

            break;

        case CLAIM_EVT_OTHER_CLAIM:
            /* another device claimed/lost its address */
            break;
    }

}
```

3.2.6 j_setCallback_PGNAnnounce

Set callback function:

```
    j_setCallback_PGNAnnounce(0, &receivedTPAnnounce);
```

Called function:

```
int8_t receivedTPAnnounce(cb_PGNAnnounceInfo_t* infos)
{
    if ((receiveBufferActive) || (infos->msgSize > RECEIVE_BUFFER_SIZE)) {
        debugPrint_tnt(LT_WARN, "Denied TP/BAM from ", infos->fromAddr,
                             " due to busy receive buffer");

        /* only receive buffer is busy: */
        return 1;  /* deny transfer */
    } else {
        receiveBufferActive = 1;
        return 0; /* accept transfer */
    }
}
```

The example receivedTPAnnounce() function shows handling with a single receive buffer. If you have multiple buffers, you could differentiate them later by using the customData member of the cb_PGNAnnounceInfo_t struct:

You would **set** customData in the announce callback, and **use** it the receive callbacks.

3.2.7 j_setCallback_RequestReceived

Set callback function:

```
    j_setCallback_RequestReceived(0, &receivedRequest);
```

Called function:

```
void receivedRequest(cb_RequestReceivedInfo_t* infos)
{
    switch (infos->pgn) {
        case 0x2000:
            {
                /* send PGN 0x2000 without callback: */
                dataSendInfo_t sendInfos;
                sendInfos.data = testData;
                sendInfos.dataLen = 8;
                sendInfos.destAddr = infos->toGlobal? ADR_GLOBAL : infos->fromAddr;
                sendInfos.pgn = 0x2000;
                sendInfos.priority = 6;
                j_sendPGN(0, &sendInfos);
            }
            break;

        case 0x3000:
            /* send PGN 0x3000 with callback: */
            j_sendPGNByCallback(0, infos->toGlobal? ADR_GLOBAL : infos->fromAddr, 0x3000, 6, 10);
            break;

        default:
            if (!(infos->toGlobal))
                j_sendNACK(0, infos->pgn);
    }
}


void claimEvent(cb_ClaimEventInfo_t* infos)
{
    switch (infos->evtType) {
        case CLAIM_EVT_CLAIMED:
            ownAddress = infos->addr;
            debugPrint_tn(LT_INFO, "own address is now ", ownAddress);

            /* this is the best time to add autobroadcasts: */
            autoBroadcastId = j_addAutoBroadcast(0, 1000 * J_TIMER_TICKS_PER_MS, 0x1000, 6, 5);
            break;

        case CLAIM_EVT_FAILED:
            ownAddress = ADR_NULL;
            debugPrint(LT_WARN, "address claim failed/lost!");

            infos->addr = ADR_NULL; /* tell stack to give up (and send cannot claim) */

            break;

        case CLAIM_EVT_OTHER_CLAIM:
            /* another device claimed/lost its address */
            break;
    }
```

3.2.8 Complete Application

A small example application: (For latest version please check installation's demo folder)

Features:

- Claims Address 100 (does not claim another on conflict)
- Broadcasts PGN 0x1000 with 5 data bytes every 1000 ms
- Sends PGN 0x2000 with 8 data bytes on request
- Sends PGN 0x3000 with 10 data bytes (multipacket) on request
- Uses extra callback for Diagnostic Messages. Pauses autobroadcast on DM13

Requirements:

Windows: j1939.h, j1939.lib and j1939.dll file
Linux: j1939.h and libj1939.so file

```
/************************************************************************/
/*                                                                      */
/*   Test/Demonstration program to use the ESD J1939 stack              */
/*                                                                      */
/*          Copyright 2008 esd - electronic system design gmbh          */
/*----------------------------------------------------------------------*/
/*                                                                      */
/*      Filename:      j1939demo.c                                      */
/*      Date:          2008-07-01                                       */
/*      Language:      ANSI-C                                           */
/*      Targetsystem:  N/A                                              */
/*                                                                      */
/*      Description:   Demonstration of J1939 API                       */
/*----------------------------------------------------------------------*/
/* Revision history:                                                    */
/*----------------------------------------------------------------------*/
/* 100,01jul08,    * Initial release                                    */
/************************************************************************/



#include "j1939.h"



#define                     OWN_PREF_ADDRESS                    100

static  const   uint8_t     ownName[8] =                        {0x83, 0x95, 0xFF, 0xEE, 0x00,
                                                                  0x82, 0x00, 0x80};

static          uint8_t     ownAddress =                        ADR_NULL;

#define                     RECEIVE_BUFFER_SIZE                 1785
static          uint8_t     receiveBuffer[RECEIVE_BUFFER_SIZE];
static          uint8_t     receiveBufferActive;

static  const   uint8_t     dataToBroadcast[5] =                {0x01, 0x02, 0x03, 0x04, 0x05};
static          int8_t      autoBroadcastId =                   -1;

static  const   uint8_t     testData[10] =                      {0x10, 0x20, 0x30, 0x40, 0x50,
                                                                  0x60, 0x70, 0x80, 0x90, 0xA0};


void        sendCallback(cb_DataSendInfo_t* sendInfos);
int8_t      receiveCallback(cb_DataReceivedInfo_t* rcvInfos);
int8_t      receiveDMCallback(cb_DataReceivedInfo_t* rcvInfos);
void        claimEvent(cb_ClaimEventInfo_t* infos);
```

```
int8_t      receivedTPAnnounce(cb_PGNAnnounceInfo_t* infos);
void        receivedRequest(cb_RequestReceivedInfo_t* infos);


int main(void)
{
    jtimer_t timeStart, timeNow;

    /* set the callbacks: */
    j_setCallback_RequestReceived(0, &receivedRequest);
    j_setCallback_PGNAnnounce(0, &receivedTPAnnounce);
    j_setCallback_ClaimEvent(0, &claimEvent);
    j_setCallback_PGNSend(0, &sendCallback);
    j_setCallback_PGNReceived(0, &receiveCallback);
    j_setCallback_DMReceived(0, &receiveDMCallback);

    /* init the stack/device (which inits CAN itself): */
    if (j_init(0, 0, J_CAN_BAUD_DEFAULT, OWN_PREF_ADDRESS, ownName)) {
        debugPrint(LT_ERROR, "j_init() failed!");
        return 1;
    }

    /* start main loop: */
    timeStart = GET_SYSTEM_TICK();
    while(1) {
        timeNow = GET_SYSTEM_TICK();

        can_receiveMessages(); /* receive new message for every used CAN net (here only net 0)*/
        j_processData(0, (jtimer_t)(timeNow - timeStart));

        SLEEP_MS(1);
        timeStart = timeNow;
    }

    return 0;
}


void sendCallback(cb_DataSendInfo_t* sendInfos)
{
    if (sendInfos->nBytes == 0) {
        /* stack has finished collecting data */
        /* could now free/update that data etc. */
        return;
    }

    switch (sendInfos->pgn) {
        /* stack needs new data for PGN: */

        case 0x1000:
            sendInfos->data = &dataToBroadcast[sendInfos->offset];
            break;

        case 0x3000:
            sendInfos->data = &testData[sendInfos->offset];
            break;
    }
}


int8_t receiveCallback(cb_DataReceivedInfo_t* rcvInfos)
{
    if (rcvInfos->status < 0) { /* multipacket transfer aborted */
        receiveBufferActive = 0;
        return 0; /* result is unused */
    }

    switch(rcvInfos->status) {

        /* Complete message at once: */
        case J_RCV_STATUS_SHORT_MSG:
            debugPrint_tntntn(LT_INFO, "received short PGN ", rcvInfos->pgn, "(",
                                    rcvInfos->dataLen, " bytes) from ", rcvInfos->fromAddr);
            switch(rcvInfos->pgn) {
                /* handle PGNs */
            }
            return 0; /* result is unused */


        /* Multipacket Transfer, completed: */
        case J_RCV_STATUS_LONG_MSG_COMPLETE:
            receiveBufferActive = 0;
            debugPrint_tntntn(LT_INFO, "received long PGN ", rcvInfos->pgn, "(",
                               rcvInfos->offset, " bytes) from ", rcvInfos->fromAddr);

            switch(rcvInfos->pgn) {
                /* handle PGNs */
            }
            return 0; /* result is unused */


        /* Multipacket transfer, data chunk: */
        /* only called for accepted transfers, needs callback set in
                                                                j_setCallback_PGNAnnounce(). */
        case J_RCV_STATUS_LONG_MSG_CHUNK:
            if (!receiveBufferActive)
                return 1; /* abort if buffer not 'activated' (should not happen...)*/

            /* add chunk to receiveBuffer: */
            if ((rcvInfos->offset + rcvInfos->dataLen) <= RECEIVE_BUFFER_SIZE)
                memcpy(&receiveBuffer[rcvInfos->offset], rcvInfos->data, rcvInfos->dataLen);

            return 0; /* ok, continue transfer */
    }


    return 1; /* abort unhandled incoming PGN transfers */
}

/*
   This callback example reacts on DM13: 'Stop Start Broadcast'
   It's _no_ example for a proper reaction on that message!, but an example
   for j_pauseAutoBroadcasts() and j_unPauseAutoBroadcasts() functions.
*/
int8_t receiveDMCallback(cb_DataReceivedInfo_t* rcvInfos)
{
    if (rcvInfos->status < 0) {
        receiveBufferActive = 0;
        return 0;
    }

    switch(rcvInfos->status) {
        case 0:
            debugPrint_tntntn(LT_INFO, "received short DM ", rcvInfos->pgn, "(",
                                    rcvInfos->dataLen, " bytes) from ", rcvInfos->fromAddr);
            switch(rcvInfos->pgn) {
                case PGN_DM13: /* 'Stop Start Broadcast' (SAE J1939-73) */
                    if (rcvInfos->dataLen == 8) {
                        switch (rcvInfos->data[0] & 0xC0) { /* Extract bits for
                                                                    'Current Data Link' */
                            case 0x00: /* stop broadcast */
                                debugPrint(LT_INFO, "Autobroadcasts paused.");
                                j_pauseAutoBroadcasts(0, 6000 * J_TIMER_TICKS_PER_MS);
                                break;

                            case 0x40: /* start broadcast */
                                debugPrint(LT_INFO, "continue with Autobroadcasts");
                                j_unPauseAutoBroadcasts(0);
                                break;
                        }
                    }
                    break;

                case PGN_DM7: /* 'Command Non-continuously Monitored Test' (SAE J1939-73) */
                    if (rcvInfos->destAddr != ADR_GLOBAL)
                        j_sendNACK(0, PGN_DM7);
                    break;
            }
            return 0;
        case 1:
            receiveBufferActive = 0;
            debugPrint_tntntn(LT_INFO, "received long DM ", rcvInfos->pgn, "(",
                                     rcvInfos->offset, " bytes) from ", rcvInfos->fromAddr);
            return 0;

        case 2:
            if (!receiveBufferActive)
                return 1;

            if ((rcvInfos->offset + rcvInfos->dataLen) <= RECEIVE_BUFFER_SIZE)
                memcpy(&receiveBuffer[rcvInfos->offset], rcvInfos->data, rcvInfos->dataLen);

            return 0;
    }

    return 1;
}

void receivedRequest(cb_RequestReceivedInfo_t* infos)
{
    switch (infos->pgn) {
        case 0x2000:
            {
                /* send PGN 0x2000 without callback: */
                dataSendInfo_t sendInfos;
                sendInfos.data = testData;
                sendInfos.dataLen = 8;
                sendInfos.destAddr = infos->toGlobal? ADR_GLOBAL : infos->fromAddr;
                sendInfos.pgn = 0x2000;
                sendInfos.priority = 6;
                j_sendPGN(0, &sendInfos);
            }
            break;

        case 0x3000:
            /* send PGN 0x3000 with callback: */
            j_sendPGNByCallback(0, infos->toGlobal? ADR_GLOBAL :
                                   infos->fromAddr, 0x3000, 6, 10);
            break;

        default:
            if (!(infos->toGlobal))
                j_sendNACK(0, infos->pgn);
    }
}


void claimEvent(cb_ClaimEventInfo_t* infos)
{
    switch (infos->evtType) {
        case CLAIM_EVT_CLAIMED:
            ownAddress = infos->addr;
            debugPrint_tn(LT_INFO, "own address is now ", ownAddress);

            /* this is the best time to add autobroadcasts: */
            autoBroadcastId = j_addAutoBroadcast(0, 1000 * J_TIMER_TICKS_PER_MS, 0x1000, 6, 5);
            break;

        case CLAIM_EVT_FAILED:
            ownAddress = ADR_NULL;
            debugPrint(LT_WARN, "address claim failed/lost!");

            infos->addr = ADR_NULL; /* tell stack to give up (and send cannot claim) */

            break;

        case CLAIM_EVT_OTHER_CLAIM:
            /* another device claimed/lost its address */
            break;
    }

}
```

```
int8_t receivedTPAnnounce(cb_PGNAnnounceInfo_t* infos)
{
    if ((receiveBufferActive) || (infos->msgSize > RECEIVE_BUFFER_SIZE)) {
        debugPrint_tnt(LT_WARN, "Denied TP/BAM from ", infos->fromAddr, " due to busy receive
                                                                                       buffer");
        /* only receive buffer is busy: */
        return 1;  /* deny transfer */
    } else {
        receiveBufferActive = 1;
        return 0; /* accept transfer */
    }
}
```

4. Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

cb_ClaimEventInfo_t (Used for callback set in j_setCallback_ClaimEvent()) 37

cb_DataReceivedInfo_t (Used for callback set in j_setCallback_PGNReceived(), j_setCallback_SpecialPGNReceived() and j_setCallback_DMReceived()) 38

cb_DataSendInfo_t (Used for callback set in j_setCallback_PGNSend()) 40

cb_PGNAnnounceInfo_t (Used for callback set in j_setCallback_PGNAnnounce()) 41

cb_RequestReceivedInfo_t (Used for callback set in j_setCallback_RequestReceived()) 42

dataSendInfo_t (Used to send data in j_sendPGN()) 43

5. File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

j1939CAN.h **(esd j1939 Stack CAN header file)** 44
j1939defs.h **(esd j1939 Stack definitions file)** 45
j1939stack.h **(esd j1939 Stack header file)** 48

6. Data Structure Documentation

6.1 cb_ClaimEventInfo_t Struct Reference

`#include <j1939stack.h>`

Data Fields

- uint8_t evtType
- uint8_t addr
- uint8_t deviceNum
- uint8_t * deviceName

Field Documentation

uint8_t cb_ClaimEventInfo_t::evtType

Type of claim event: `CLAIM_EVT_CLAIMED` (own claim successful), `CLAIM_EVT_FAILED` (own claim failed/lost) or `CLAIM_EVT_OTHER_CLAIM` (another devices claimed or lost its source address).

uint8_t cb_ClaimEventInfo_t::addr

Claimed or lost address. `CLAIM_EVT_FAILED:` changing `addr` will tell the stack to try to claim that new address.

uint8_t cb_ClaimEventInfo_t::deviceNum

Number of calling device. (Only needed if J_NUM_DEVICES > 1)

uint8_t* cb_ClaimEventInfo_t::deviceName

Only used with `CLAIM_EVT_OTHER_CLAIM:` Name of device that claimed/lost its address (8 bytes).

6.2 cb_DataReceivedInfo_t Struct Reference

`#include <j1939stack.h>`

Data Fields

- uint32_t pgn
- uint32_t customData
- int16_t offset
- uint8_t fromAddr
- uint8_t destAddr
- uint8_t dataLen
- int8_t status
- uint8_t deviceNum
- uint8_t * data

Field Documentation

uint32_t cb_DataReceivedInfo_t::pgn

PGN.

uint32_t cb_DataReceivedInfo_t::customData

May be used by application, see j_setCallback_PGNAnnounce().

int16_t cb_DataReceivedInfo_t::offset

Offset of new data. (Only defined with `status = 1`)

uint8_t cb_DataReceivedInfo_t::fromAddr

Source address.

uint8_t cb_DataReceivedInfo_t::destAddr

Destination address (own address or `ADR_GLOBAL`).

uint8_t cb_DataReceivedInfo_t::dataLen

Length of (new) data. (Undefined with `status < 1`)

int8_t cb_DataReceivedInfo_t::status

J_RCV_STATUS_SHORT_MSG : Complete message at once,
J_RCV_STATUS_LONG_MSG_COMPLETE : Multipacket Transfer complete,
J_RCV_STATUS_LONG_MSG_CHUNK : Multipacket Transfer data Chunk,
<**0** : Transfer aborted.

uint8_t cb_DataReceivedInfo_t::deviceNum

Number of calling device. (Only needed if J_NUM_DEVICES > 1)

uint8_t* cb_DataReceivedInfo_t::data

Pointer to newly received data. (Undefined with `status < 0`)

6.3 cb_DataSendInfo_t Struct Reference

`#include <j1939stack.h>`

Data Fields

- uint32_t pgn
- uint16_t offset
- uint16_t nBytes
- uint8_t destAddr
- uint8_t deviceNum
- const uint8_t * data

Field Documentation

uint32_t cb_DataSendInfo_t::pgn

PGN.

uint16_t cb_DataSendInfo_t::offset

Offset of requested data, set by stack.

uint16_t cb_DataSendInfo_t::nBytes

Stack sets number of bytes wanted, app 'answers' here with number of bytes available at data . See Example in j_setCallback_PGNSend().

uint8_t cb_DataSendInfo_t::destAddr

Destination. (Useful to differentiate data when sending different PGN data to the destinations)

uint8_t cb_DataSendInfo_t::deviceNum

Number of calling device. (Only needed if `J_NUM_DEVICES > 1`)

const uint8_t* cb_DataSendInfo_t::data

Pointer to requested data, **must** be set by application. (Set to `NULL` to abort sending)

6.4 cb_PGNAnnounceInfo_t Struct Reference

`#include <j1939stack.h>`

Data Fields

- uint32_t pgn
- uint32_t customData
- uint16_t msgSize
- uint8_t fromAddr
- uint8_t destAddr
- uint8_t deviceNum

Field Documentation

uint32_t cb_PGNAnnounceInfo_t::pgn

PGN.

uint32_t cb_PGNAnnounceInfo_t::customData

May be used by application, see j_setCallback_PGNAnnounce().

uint16_t cb_PGNAnnounceInfo_t::msgSize

Size of pending message.

uint8_t cb_PGNAnnounceInfo_t::fromAddr

Sender.

uint8_t cb_PGNAnnounceInfo_t::destAddr

Destination.

uint8_t cb_PGNAnnounceInfo_t::deviceNum

Number of calling device. (Only needed if J_NUM_DEVICES > 1)

6.5 cb_RequestReceivedInfo_t Struct Reference

`#include <j1939stack.h>`

Data Fields

- uint32_t pgn
- uint8_t fromAddr
- uint8_t toGlobal
- uint8_t deviceNum

Field Documentation

uint32_t cb_RequestReceivedInfo_t::pgn

PGN.

uint8_t cb_RequestReceivedInfo_t::fromAddr

Sender.

uint8_t cb_RequestReceivedInfo_t::toGlobal

1 if destination of request is `ADR_GLOBAL` .

uint8_t cb_RequestReceivedInfo_t::deviceNum

Number of calling device. (Only needed if J_NUM_DEVICES > 1)

6.6 dataSendInfo_t Struct Reference

```
#include <j1939stack.h>
```

Data Fields

- uint32_t pgn
- uint16_t dataLen
- uint8_t destAddr
- uint8_t priority
- const uint8_t * data

Field Documentation

uint32_t dataSendInfo_t::pgn

PGN.

uint16_t dataSendInfo_t::dataLen

Length.

uint8_t dataSendInfo_t::destAddr

Destination address or `ADR_GLOBAL` .

uint8_t dataSendInfo_t::priority

Priority. (0 to 7, where 0 is highest)

const uint8_t* dataSendInfo_t::data

Pointer to data to send.

7. File Documentation, Source Code Version

7.1 j1939CAN.h File Reference

7.1.1 Detailed Description

This file is the stacks interface to the CAN bus. For an implementation for own hardware (with the Source Code Version) the `j1939CAN.c` file has to be adapted to contain the function described by this header.

7.2 j1939defs.h File Reference

7.2.1 Detailed Description

This file is used to set up several j1939 stack options or needed system includes. This file and `j1939CAN.c` should be the only stack files that have to be altered by application developer.

Defines

- J_DEBUG_OUTPUT
- J_NUM_DEVICES
- J_RECEIVE_ONLY_LISTED_PGN
- J_RECEIVE_PGN_SOURCE_SPECIFIC
- J_SPECIAL_CALLBACK_FOR_DM
- J_TP_MAX_RCV_SOCKETS
- J_TP_MAX_SEND_SOCKETS
- J_TP_BAM_QUEUE_SIZE
- J_TP_MAX_AUTO_BROADCASTS
- J_TP_USE_PRIORITY_FOR_BAM_QUEUE
- J_TIMER_SIZE_32_BIT
- J_TIMER_TICKS_PER_MS
- J_TEST_FOR_MESSAGES_FROM_SELF
- J_CAN_NET_COUNT
- J_USE_NTCAN_H
- J_MSG_TX_BUFFER_SIZE

7.2.2 Define Documentation

7.2.2.1 J_DEBUG_OUTPUT

Set to 1 to enable debug output. The *debugPrint* functions are defined in this file and might need additional implementations for your specific system. See also log types defines in this file.

7.2.2.2 J_NUM_DEVICES

Defines number of devices. If defined as greater than 1, then all functions need the device number as additional (first) parameter. First device will be 0, second 1, and so on.

7.2.2.3 J_RECEIVE_ONLY_LISTED_PGN

If set to 0, the callback given in j_setCallback_PGNReceived() is called for any PGN, else every PGN has its own callback, given in j_setCallback_SpecialPGNReceived(). Memory usage: about 8 bytes per item.

7.2.2.4 J_RECEIVE_PGN_SOURCE_SPECIFIC

If set to 0, the callback given in j_setCallback_PGNReceived() is called for PGNs from any sender, else only for PGNs from addresses set in j_setPGNSourceAddressFilter().

Remarks:

Only incoming PGNs are filtered, not requests, etc.

7.2.2.5 J_SPECIAL_CALLBACK_FOR_DM

If set to 1, an extra callback for Diagnostic Messages (DM) is available.

Set by j_setCallback_DMReceived().

7.2.2.6 J_TP_MAX_RCV_SOCKETS

Max. number of multipacket transfers at a time as **receiver** (includes both methods: RTS/CTS and BAM). Memory usage: about 24 bytes per item.

7.2.2.7 J_TP_MAX_SEND_SOCKETS

Max. number of multipacket transfers at a time as **sender** (only RTS/CTS transfers). Memory usage: about 24 bytes per item.

7.2.2.8 J_TP_BAM_QUEUE_SIZE

Max. number of queued (only one at a time possible) **BAM transfers** . Memory usage: about 16 bytes per item.

7.2.2.9 J_TP_MAX_AUTO_BROADCASTS

Max. number of items in autobroadcast list for j_addAutoBroadcast(). Memory usage: about 16 bytes per item.

7.2.2.10 J_TP_USE_PRIORITY_FOR_BAM_QUEUE

If defined as 1 and BAM-queue is full, then an item in queue is deleted when adding an item with better priority. If not needed set to 0 to save some cpu time.

7.2.2.11 J_TIMER_SIZE_32_BIT

If defined as 1, type of timer variables will be `uint32_t` , else `uint16_t` .

7.2.2.12 J_TIMER_TICKS_PER_MS

Defines the number of timer ticks within one millisecond.

7.2.2.13 J_TEST_FOR_MESSAGES_FROM_SELF

If j1939CAN.c is implemented so that own messages will also be received, then this needs to be defined as 1.

7.2.2.14 J_CAN_NET_COUNT

Define as max used can net +1

7.2.2.15 J_USE_NTCAN_H

Set to 1 if j1939CAN.c should include/use esd's ntcan.h header. If that is not available, an own implementation for receiving/sending can messages, etc. has to be done in j1939CAN.c

7.2.2.16 J_MSG_TX_BUFFER_SIZE

If defined as greater than 0 then TX messages are buffered when can_sendJMsg() fails. (Next j_processData() call will retry sending them)

7.3 j1939stack.h File Reference

7.3.1 Detailed Description

```
#include "j1939defs.h"
```

Data Structures

- struct cb_DataReceivedInfo_t
  Used for callback set in j_setCallback_PGNReceived(), j_setCallback_SpecialPGNReceived() and j_setCallback_DMReceived().
- struct cb_DataSendInfo_t
  Used for callback set in j_setCallback_PGNSend().
- struct cb_PGNAnnounceInfo_t
  Used for callback set in j_setCallback_PGNAnnounce().
- struct cb_RequestReceivedInfo_t
  Used for callback set in j_setCallback_RequestReceived().
- struct cb_ClaimEventInfo_t
  Used for callback set in j_setCallback_ClaimEvent().
- struct dataSendInfo_t
  Used to send data in j_sendPGN().

Macros

- j_sendAcknowledgment
- j_sendACK(pgn)
- j_sendNACK(pgn)

Typedefs

- typedef uint32_t jtimer_t

7.3.2 Functions

- int8_t j_sendPDU (const uint8_t priority, const uint8_t dp, const uint8_t pduFormat, const uint8_t pduSpecific, const uint8_t dataLen, const uint8_t data0, const uint8_t data1, const uint8_t data2, const uint8_t data3, const uint8_t data4, const uint8_t data5, const uint8_t data6, const uint8_t data7)
- int8_t j_compareNameTo (const uint8_t *otherName)
- int8_t j_sendPGN (const dataSendInfo_t *sendInfos)
- int8_t j_sendPGNByCallback (const uint8_t destAddr, const uint32_t pgn, const uint8_t priority, const uint16_t dataLen)
- int8_t j_sendRequest (const uint8_t destAddr, const uint32_t pgn)
- int8_t j_sendRequestForAddressClaimed (const uint8_t destAddr)
- int8_t j_addAutoBroadcast (const jtimer_t interval, const uint32_t pgn, const uint8_t priority, const uint16_t dataLen)
- void j_removeAutoBroadcast (const uint8_t id)
- void j_pauseAutoBroadcasts (const jtimer_t duration)
- void j_unPauseAutoBroadcasts ()
- int8_t j_init (const uint8_t canNetNum, const jcanbaud_t canBaudRate, const uint8_t autoClaimAddr, const uint8_t *ownName)
- void j_finish ()
- void j_processData (const jtimer_t ticksDelta)
- void j_startAddrClaim (const uint8_t addr)
- void j_setCallback_ClaimEvent (cb_ClaimEvent_t *function)
- void j_setCallback_RequestReceived (cb_RequestReceived_t *function)
- void j_setCallback_PGNAnnounce (cb_PGNAnnounce_t *function)
- void j_setCallback_PGNSend (cb_PGNSend_t *function)
- void j_setCallback_PGNReceived (cb_PGNReceived_t *function)
- int8_t j_setCallback_SpecialPGNReceived (uint32_t pgn, cb_PGNReceived_t *function)
- void j_setPGNSourceAddressFilter (const uint8_t addr, const uint8_t allow)
- void j_setCallback_DMReceived (cb_PGNReceived_t *function)

7.3.3 Macros

7.3.3.1 j_sendAcknowledgment(contrByte, groupFunct, pgn) j_sendPDU(6, 0, 232, 255, 8, contrByte, groupFunct, 0xFF, 0xFF, 0xFF, pgn & 0xFF, (pgn >> 8) & 0xFF, (pgn >> 16) & 0xFF)

Sends an acknowledgement with specified control byte.

7.3.3.2 j_sendACK(pgn) j_sendAcknowledgment(0, 0, pgn)

Sends a positive acknowledgement (control byte = 0).

7.3.3.3 j_sendNACK(pgn) j_sendAcknowledgment(1, 0, pgn)

Sends a negative acknowledgement (control byte = 1).

7.3.4 Typedef Documentation

typedef uint32_t jtimer_t

Type of all timestamps, etc. See J_TIMER_SIZE_32_BIT.

7.3.5 Function Documentation

7.3.5.1 int8_t j_sendPDU (const uint8_t *priority*, const uint8_t *dp*, const uint8_t *pduFormat*, const uint8_t *pduSpecific*, const uint8_t *dataLen*, const uint8_t *data0*, const uint8_t *data1*, const uint8_t *data2*, const uint8_t *data3*, const uint8_t *data4*, const uint8_t *data5*, const uint8_t *data6*, const uint8_t *data7*)

Description:

Sends a PDU. Almost all other functions/macros use this to send, therefore this should not be needed directly.

Return values:

0 When successfully copied to sendqueue.
else Error.

Macros:

- j_sendAcknowledgment()
- j_sendACK()
- j_sendNACK()

Remarks:

Fails without valid source address.

7.3.5.2 int8_t j_compareNameTo (const uint8_t * *otherName*)

Description:

Compares current own name to given other. Needed by stack for address claiming. App could also need this to test for own name on receipt of 'Commanded Address' message.

Parameters:

otherName Pointer to other name (8 bytes).

Return values:

0	Names equal.
1	Other name has higher priority.
-1	Other name has lower priority.

7.3.5.3 int8_t j_sendPGN (const dataSendInfo_t * *sendInfos*)

Description:

Similar to j_sendPGNByCallback(). But here the data to send is given via pointer in dataSendInfo_t structure. All data must be available there.
Therefore no callback is needed to get data, but 'finish callback' (see j_setCallback_PGNSend()) for multipacket messages (>8 bytes) is called anyway.

Warning:

Make sure data does not change while sending multipacket messages.

See also:

- j_sendPGNByCallback()
- Example: Send PGN without callback

7.3.5.4 int8_t j_sendPGNByCallback (const uint8_t *destAddr*, const uint32_t *pgn*, const uint8_t *priority*, const uint16_t *dataLen*)

Description:

Sends a PGN while automatically deciding whether to use transport protocol or not. Data to send is taken from application via callback set in j_setCallback_PGNSend().

Parameters:

destAddr Destination address or `ADR_GLOBAL`.
pgn PGN.
priority Priority from 0 to 7, where 0 is highest.
dataLen Length of PGN data. (0 to 1785)

Return values:

0 On success.
else Error.

Remarks:

If `dataLen <= 8` and PGN implies PDU2 Format, it is automatically broadcast.

See also:

- j_sendPGN()
- Example: Send PGN with callback

7.3.5.5 int8_t j_sendRequest (const uint8_t *destAddr*, const uint32_t *pgn*)

Description:

Sends a request for PGN message.

Parameters:

destAddr Destination address or ADR_GLOBAL .
pgn Requested PGN.

Return values:

0 When successfully copied to sendqueue.
else Error.

Remarks:

To answer own global requests as well, the callback given to j_setCallback_RequestReceived() is automatically called when destAddr == ADR_GLOBAL .

7.3.5.6 int8_t j_sendRequestForAddressClaimed (const uint8_t *destAddr*)

Description:

Sends 'Request for Address Claimed' message to the given Address. Receiver then broadcasts its name and address, which fires the callback set with j_setCallback_ClaimEvent().

Parameters:

destAddr Address to get claimed message from. (Or `ADR_GLOBAL` to get infos from all devices)

Remarks:

This Request is the only one which is allowed without valid source address. It's automatically set to `ADR_NULL` when current source address is not valid. `destAddr == ADR_GLOBAL` then the own 'Address Claimed' (or 'Cannot Claim') message is sent automatically.

7.3.5.7 int8_t j_addAutoBroadcast (const jtimer_t *interval*, const uint32_t *pgn*, const uint8_t *priority*, const uint16_t *dataLen*)

Description:

Adds an item to the autobroadcast list. Data to send is taken from application via callback set in j_setCallback_PGNSend().

Parameters:

interval Interval between the broadcasts in timer ticks . Max with 16 bit jtimer_t is 32768 (2^15), max for 32 bit jtimer_t is 2147483648 (2^31).
pgn PGN.
priority Priority from 0 to 7, where 0 is highest.
dataLen Data length.

Return values:

<0 Error.
>=0 Id. Needed for j_removeAutoBroadcast().

See also:

- j_removeAutoBroadcast()
- J_TP_USE_PRIORITY_FOR_BAM_QUEUE

Warning:

All autobroadcast items are deleted when source address is lost. Therefore all items should be added when successfully claimed an address, see j_setCallback_ClaimEvent().

Remarks:

Minimum interval for multipacket messages is not checked by the stack. (Bear in mind that there have to be at least 50 ms between the packets in multipacket messages)

7.3.5.8 void j_removeAutoBroadcast (const uint8_t *id*)

Description:

Removes an item from the autobroadcast list.

Parameters:

id Id of item to remove.

See also:

- j_addAutoBroadcast()

7.3.5.9 void j_pauseAutoBroadcasts (const jtimer_t *duration*)

Description:

Pauses all items in the autobroadcast list.

Parameters:

duration Duration in *timer ticks* .

Remarks:

- An active broadcast is not aborted/paused.
- A re-calling of this function within a pause will not append the new duration to the existing. It just sets a new end time for the pause.
- The next broadcast time for every item in the list will be the end of the pause + its interval.

7.3.5.10 void j_unPauseAutoBroadcasts ()

Description:

Used to continue auto broadcasts prior timeout set with j_pauseAutoBroadcasts().

7.3.5.11 int8_t j_init (const uint8_t *canNetNum*, const jcanbaud_t *canBaudRate*, const uint8_t *autoClaimAddr*, const uint8_t * *ownName*)

Description:

Initializes the stack and calls can_init(). Only after this is done (and the callbacks are set) all other functions can be used. Returns 0 on success.

Might be called multiple times to change device name, but this will also:

- Abort all active multipacket transfers.
- Clear the autobroadcast list.
- Reset PGN receive source filter (if defined).

Parameters:

canNetNum CAN net number. See also J_CAN_NET_COUNT.
canBaudRate CAN baudrate, usually J_CAN_BAUD_DEFAULT. The predefined enum values (J_CAN_BAUD_10 to J_CAN_BAUD_1000) are only used for the esd ntcan library. Other values may be defined to your j1939CAN.c implementation.
autoClaimAddr See Remarks.
ownName Pointer to own name. Data is copied (8 bytes).

See also:

- Example: Complete Application

Remarks:

If an autoClaimAddr other than ADR_NULL is given then the stack automatically claims that address. If that fails (due to low priority for example) or finished successfully the callback given with j_setCallback_ClaimEvent() is called.

The AAC bit in the name is ignored by the stack: it's up to the application to act in compliance with that bit. (i.e. don't use the auto-claiming-another-address feature when the name indicates that the device is not capable of that, or vice versa)

Warning:

Set callbacks in advance. In particular the mandatory callbacks: j_setCallback_PGNSend() and j_setCallback_PGNReceived().

7.3.5.12 void j_finish ()

Description:

Mainly for closing can handle (if can_finish() is implemented). Also active transfers are stopped etc.

7.3.5.13 void j_processData (const jtimer_t *ticksDelta*)

Description:

Main stack function. New CAN messages are processed here, pending messages are sent, and so on.

Parameters:

ticksDelta Timer ticks since last call.

Prerequisites:

- j_init().
- j_setCallback_PGNReceived() or j_setCallback_SpecialPGNReceived().
- j_setCallback_PGNSend().

See also:

- Example: Complete Application

Remarks:

Accuracy of all timed events (such as 50 ms waiting to send next packet) depend on the fast regularly calling of this function.

7.3.5.14 void j_startAddrClaim (const uint8_t *addr*)

Description:

Sends 'Address Claimed Message' for the given address. All address contention is done automatically. Result will be available to the callback set in j_setCallback_ClaimEvent().

Use with `addr = ADR_NULL` to give up own address. The 'Cannot Claim Address' message is sent automatically then. (With pseudo random delay as described in J1939-81)

Parameters:

addr Address to claim.

See also:

- j_setCallback_ClaimEvent()

Remarks:

Not necessary with autoClaimAddr address given in j_init().
'Request for Address Claimed' message is not used before claiming, that has to be done manually with the j_sendRequestForAddressClaimed() function.

7.3.5.15 void j_setCallback_ClaimEvent (cb_ClaimEvent_t * *function*)

Description:

Sets the callback function for claim events.

Parameters:

function The function to call.

See also:

- Example: j_setCallback_ClaimEvent

7.3.5.16 void j_setCallback_RequestReceived (cb_RequestReceived_t * *function*)

Description:

Sets the callback function for incoming requests.

Parameters:

function The function to call.

Remarks:

Not called for 'Address Claimed' requests, these are answered automatically. Also not called when currently no address claimed.

See also:

- Example: j_setCallback_RequestReceived

7.3.5.17 void j_setCallback_PGNAnnounce (cb_PGNAnnounce_t * *function*)

Description:

Sets the callback function for incoming BAM and CTS/RTS messages. There is decided whether to accept that multipacket message or not.

Parameters:

function The function to call.

See also:

- Example: j_setCallback_PGNAnnounce

7.3.5.18 void j_setCallback_PGNSend (cb_PGNSend_t * *function*)

Description:

Sets the callback function when the stack needs new data. That is when sending with j_sendPGNByCallback() or j_addAutoBroadcast().

Parameters:

function The function to call.

See also:

- j_setCallback_PGNSend

Remarks:

To notice when stack has finished collecting data (for example to change/free that data) the callback is also called then, with the callback struct's `nBytes == 0`.
(Also called 'finish callback')

Warning:

The 'finish callback' is called for every single transfer. When a transfer is started while another is still active then **two** finish callbacks will be fired anyway.
Therefore you have to count the finish callbacks if you want to change the data there.

7.3.5.19 void j_setCallback_PGNReceived (cb_PGNReceived_t * *function*)

Description:

Sets the callback function for received PGN data.

Parameters:

function The function to call.

See also:

- j_setCallback_PGNAnnounce()
- Example: j_setCallback_PGNReceived

7.3.5.20 int8_t j_setCallback_SpecialPGNReceived (uint32_t *pgn*, cb_PGNReceived_t * *function*)

Description:

This function is similar to j_setCallback_PGNReceived().
But here the given callback is only called for a particular PGN. The maximum number of that callbacks is defined with .
To remove a PGN: call this function with the function parameter set to `NULL` .

Parameters:

function The function to call.
pgn The PGN upon which receipt the function should be called.

Return values:

0 On success.
else Error (List full).

See also:

- j_setCallback_PGNReceived()

Remarks:

- This function is only available when J_RECEIVE_ONLY_LISTED_PGN is defined greater than 0.
- It's optimized for adding all needed PGNs at system start. When many add/remove operations are necessary while active it might need some tuning.

7.3.5.21 void j_setPGNSourceAddressFilter (const uint8_t *addr*, const uint8_t *allow*)

Description:

Sets whether to receive PGNs from a particular address or not. Only available when J_RECEIVE_PGN_SOURCE_SPECIFIC is defined. Default: allow all addresses.

Parameters:

addr Address to filter or `ADR_GLOBAL` to set filter for all addresses.
allow **1** : receive PGNs from `addr` , **0** : don't receive PGNs from `addr` .

See also:

- J_RECEIVE_PGN_SOURCE_SPECIFIC
- J_RECEIVE_ONLY_LISTED_PGN

Remarks:

Has to be called after j_init().

7.3.5.22 void j_setCallback_DMReceived (cb_PGNReceived_t * *function*)

Description:

Identical to j_setCallback_PGNReceived() but only called for the Diagnostic Message (DM) PGNs.

Parameters:

function The function to call.

Remarks:

- Only available when J_SPECIAL_CALLBACK_FOR_DM defined as 1.
- Callback set in j_setCallback_PGNReceived() is not called for DM then.
- Ignored when J_RECEIVE_ONLY_LISTED_PGN > 0.

See also:

- j_setCallback_PGNReceived()

Warning:

If J_SPECIAL_CALLBACK_FOR_DM is defined as 1 this callback is mandatory.

8. Library Versions

The header file **j1939.h** is used to access the library functions. Basically this file offers the same Functions/Macros/Defines as described under “7.3 j1939stack.h File Reference“ and “7.2 j1939defs.h File Reference“

8.1 Defines in Library Versions

The .dll / .so file was compiled with the following values:

J_NUM_DEVICES = 8
J_RECEIVE_ONLY_LISTED_PGN = 32
J_RECEIVE_PGN_SOURCE_SPECIFIC = 1
J_SPECIAL_CALLBACK_FOR_DM = 1
J_TP_MAX_RCV_SOCKETS = 16
J_TP_MAX_SEND_SOCKETS = 16
J_TP_BAM_QUEUE_SIZE = 8
J_TP_MAX_AUTO_BROADCASTS = 16
J_TP_USE_PRIORITY_FOR_BAM_QUEUE = 1
J_TIMER_SIZE_32_BIT = 1
J_TIMER_TICKS_PER_MS = 1
J_CAN_NET_COUNT = 4

As J_NUM_DEVICES is defined greater than 1 almost all functions have an additional “device number” parameter:

The `j_sendPGN` function for example is described as:

```
j_sendPGN(const dataSendInfo_t* sendInfos)
```

But with the additional parameter it will be:

```
j_sendPGN(const uint8_t devNum, const dataSendInfo_t* sendInfos)
```

8.2 Differences between library and source code version

- Although J_RECEIVE_ONLY_LISTED_PGN is defined, **all** PGNs are received. If you want to receive only listed PGNs, just use j_setCallback_SpecialPGNReceived() to set them. After first usage of that function J_RECEIVE_ONLY_LISTED_PGN becomes valid.
- Although J_SPECIAL_CALLBACK_FOR_DM is defined, a callback set with j_setCallback_DMReceived is not mandatory. With the library version this is optional and the default callback (set with j_setCallback_PGNReceived()) is called then.
- **Library versions usually not run under realtime OS**. So times are not guaranteed in any way. An example is the address claiming procedure: if a device claims an address and receives no answer for 250 ms it uses this address. If your windows device is using this address and should react within 250 ms you can't even tell if your application will ever run within these 250 ms. (But increasing process priority usually helps)

Additional functions:

- `char* j_getDLLVersion(void)` returns pointer to a short version string
- `char* j_getErrorString(int8_t errorCode)` returns pointer to a short error description string

9. Index

Index

addr
  cb_ClaimEventInfo_t ... 32
Application Layer (J1939/7x) ... 16
cb_ClaimEventInfo_t ... 32
  addr ... 32
  deviceName ... 32
  deviceNum ... 32
  evtType ... 32
cb_DataReceivedInfo_t ... 33
  customData ... 33
  data ... 34
  dataLen ... 34
  destAddr ... 33
  deviceNum ... 34
  fromAddr ... 33
  offset ... 33
  pgn ... 33
  status ... 34
cb_DataSendInfo_t ... 35
  data ... 35
  destAddr ... 35
  deviceNum ... 35
  nBytes ... 35
  offset ... 35
  pgn ... 35
cb_PGNAnnounceInfo_t ... 36
  customData ... 36
  destAddr ... 36
  deviceNum ... 36
  fromAddr ... 36
  msgSize ... 36
  pgn ... 36
cb_RequestReceivedInfo_t ... 37
  deviceNum ... 37
  fromAddr ... 37
  pgn ... 37
  toGlobal ... 37
customData
  cb_DataReceivedInfo_t ... 33
  cb_PGNAnnounceInfo_t ... 36
data
  cb_DataReceivedInfo_t ... 34
  cb_DataSendInfo_t ... 35
  dataSendInfo_t ... 38
dataLen
  cb_DataReceivedInfo_t ... 34
  dataSendInfo_t ... 38
dataSendInfo_t ... 38
  data ... 38

- dataLen....38
- destAddr....38
- pgn....38
- priority....38

destAddr....
- cb_DataReceivedInfo_t....33
- cb_DataSendInfo_t....35
- cb_PGNAnnounceInfo_t....36
- dataSendInfo_t....38

Device ADDRESS....11

Device NAME....10

deviceName....
- cb_ClaimEventInfo_t....32

deviceNum....
- cb_ClaimEventInfo_t....32
- cb_DataReceivedInfo_t....34
- cb_DataSendInfo_t....35
- cb_PGNAnnounceInfo_t....36
- cb_RequestReceivedInfo_t....37

evtType....
- cb_ClaimEventInfo_t....32

fromAddr....
- cb_DataReceivedInfo_t....33
- cb_PGNAnnounceInfo_t....36
- cb_RequestReceivedInfo_t....37

j_addAutoBroadcast....
- j1939stack.h....49

J_CAN_NET_COUNT....
- j1939defs.h....42

j_compareNameTo....
- j1939stack.h....46

J_DEBUG_OUTPUT....
- j1939defs.h....40

j_finish....
- j1939stack.h....52

j_init....
- j1939stack.h....51

J_NUM_DEVICES....
- j1939defs.h....40

j_pauseAutoBroadcasts....
- j1939stack.h....50

j_processData....
- j1939stack.h....52

J_RECEIVE_ONLY_LISTED_PGN....
- j1939defs.h....41

J_RECEIVE_PGN_SOURCE_SPECIFIC....
- j1939defs.h....41

j_removeAutoBroadcast....
- j1939stack.h....50

j_sendACK....
- j1939stack.h....44

j_sendAcknowledgment....
- j1939stack.h....44

j_sendNACK....

  j1939stack.h....44
j_sendPDU....
  j1939stack.h....45
j_sendPGN....
  j1939stack.h....46
j_sendRequest....
  j1939stack.h....48
j_sendRequestForAddressClaimed....
  j1939stack.h....48
j_setCallback_ClaimEvent....
  j1939stack.h....53
j_setCallback_DMReceived....
  j1939stack.h....58
j_setCallback_PGNAnnounce....
  j1939stack.h....54
j_setCallback_PGNReceived....
  j1939stack.h....55
j_setCallback_PGNSend....
  j1939stack.h....55
j_setCallback_RequestReceived....
  j1939stack.h....54
j_setCallback_SpecialPGNReceived....
  j1939stack.h....56
j_setPGNSourceAddressFilter....
  j1939stack.h....57
J_SPECIAL_CALLBACK_FOR_DM....
  j1939defs.h....41
j_startAddrClaim....
  j1939stack.h....53
J_TEST_FOR_MESSAGES_FROM_SELF....
  j1939defs.h....42
J_TIMER_SIZE_32_BIT....
  j1939defs.h....42
J_TIMER_TICKS_PER_MS....
  j1939defs.h....42
J_TP_BAM_QUEUE_SIZE....
  j1939defs.h....41
J_TP_MAX_AUTO_BROADCASTS....
  j1939defs.h....41
J_TP_MAX_RCV_SOCKETS....
  j1939defs.h....41
J_TP_MAX_SEND_SOCKETS....
  j1939defs.h....41
J_TP_USE_PRIORITY_FOR_BAM_QUEUE....
  j1939defs.h....42
j_unPauseAutoBroadcasts....
  j1939stack.h....50
J_USE_NTCAN_H....
  j1939defs.h....42
j1939defs.h....
  J_CAN_NET_COUNT....42
  J_DEBUG_OUTPUT....40
  J_NUM_DEVICES....40
  J_RECEIVE_ONLY_LISTED_PGN....41

  J_RECEIVE_PGN_SOURCE_SPECIFIC....41
  J_SPECIAL_CALLBACK_FOR_DM....41
  J_TEST_FOR_MESSAGES_FROM_SELF....42
  J_TIMER_SIZE_32_BIT....42
  J_TIMER_TICKS_PER_MS....42
  J_TP_BAM_QUEUE_SIZE....41
  J_TP_MAX_AUTO_BROADCASTS....41
  J_TP_MAX_RCV_SOCKETS....41
  J_TP_MAX_SEND_SOCKETS....41
  J_TP_USE_PRIORITY_FOR_BAM_QUEUE....42
  J_USE_NTCAN_H....42
j1939stack.h....43
  j_addAutoBroadcast....49
  j_compareNameTo....46
  j_finish....52
  j_init....51
  j_pauseAutoBroadcasts....50
  j_processData....52
  j_removeAutoBroadcast....50
  j_sendACK....44
  j_sendAcknowledgment....44
  j_sendNACK....44
  j_sendPDU....45
  j_sendPGN....46
  j_sendPGNByCallback....47
  j_sendRequest....48
  j_sendRequestForAddressClaimed....48
  j_setCallback_ClaimEvent....53
  j_setCallback_DMReceived....58
  j_setCallback_PGNAnnounce....54
  j_setCallback_PGNReceived....55
  j_setCallback_PGNSend....55
  j_setCallback_RequestReceived....54
  j_setCallback_SpecialPGNReceived....56
  j_setPGNSourceAddressFilter....57
  j_startAddrClaim....53
  j_unPauseAutoBroadcasts....50
  jtimer_t....45
jtimer_t....
  j1939stack.h....45
msgSize....
  cb_PGNAnnounceInfo_t....36
nBytes....
  cb_DataSendInfo_t....35
offset....
  cb_DataReceivedInfo_t....33
  cb_DataSendInfo_t....35
Parameter Group Number (PGN)....13
pgn....
  cb_DataReceivedInfo_t....33
  cb_DataSendInfo_t....35
  cb_PGNAnnounceInfo_t....36
  cb_RequestReceivedInfo_t....37
  dataSendInfo_t....38

Physical Layer (J1939/11)....8
priority....
  dataSendInfo_t....38
Protocol Data Unit (PDU)....14
status....
  cb_DataReceivedInfo_t....34
toGlobal....
  cb_RequestReceivedInfo_t....37

10. Reference

1: SAE International, J1939 - Recommended Practices, 2000
2: ISO 11898, Road Vehicles - Controller Area Network (CAN), 1999
3: SAE International, J1939/11 - Physical Layer, 1999
4: SAE International, J1939/21 - Data Link Layer, 1998
5: SAE International, J1939/31 - Network Layer, 1997
6: SAE International, J1939/71 - Vehicle Application Layer, 2002
7: SAE International, J1939/81 - Network Management, 1997