User Manual

J1939 Protocol Stack

V 3.4

Version history

Version	Changes	Date	Editor
1.0	First version	14.09.2014	boe
1.2	Add Proprietary Protocol	24.11.2015	boe
1.6	Adapted to 1.6	16.06.2017	phi
1.8	Adapted to 1.8	08.08.2018	boe
3.0	Adapted to common CAN layer 3.0 Added managed SPNs	1.11.2018	ged
3.2	Diagnostic Messages added	05.06.2019	boe
3.3	New Version	10.03.2020	boe
3.4	Add Diagnose Messages	19.03.2020	hil

Table of Contents

References

J1939-21 Data Link Layer
J1939-71 Application Layer
J1939-73 Application Layer Diagnostic

1 Overview

The J1939 protocol stack provides basic communication mechanisms for a SAE J1939 compliant communication of devices. Thus it is possible to integrate J1939 communication services in a fast and easy way. The provided services can be used by a user friendly Application Programming Interface (API).

The protocol stack is separated into a hardware-independent part and a hardware-dependent part with a defined interface between them. So it is easy to port it to new targets.

The configuration, parametrization and scaling is realized by compiler defines and a graphical tool to ensure optimal use of the resources of the target.

2 Characteristics

- separation between hardware-dependent part and generic part with defined interface
- ANSI-C compliant
- cyclic transmission and reception of messages (PGNs)
- monitoring of received PGNs
- transport protocols (TP) BAM and CMDT
- node Id is configurable and can be set using Address Claiming
- configurable and scalable
- flexible user interface

3 J1939 Protocol Stack concept

- all services and functions can be (de)activated by #defines
- all TX and RX PGNs are defined in a central table with their attributes
- strict encapsulation of data, access is only realized by functions via different modules (no global variables in stack)
- each service provides its own initialization function

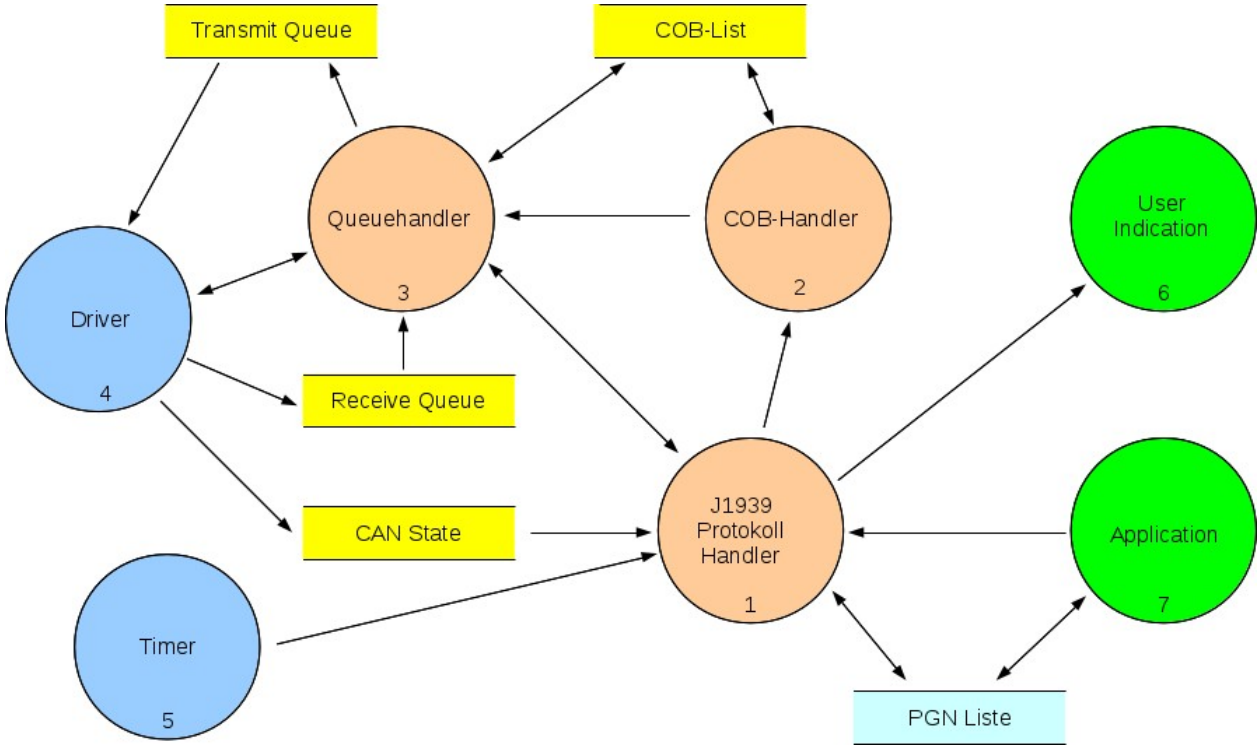


Illustration 1: Overview over Stack

The function blocks (FB)

- J1939 Protocol handler FB 1)
- COB handler (FB 2)
- Queue handler (FB 3)
- driver (FB 4)

are called from the main service function *j1939CommTask()*, which calls the J1939 functions.

This main service function *j1939CommTask()* has to be called after the following events:

- a new CAN message is available in RX queue
- the timer has been expired
- the CAN communication state has changed (e.g. error passive or bus off).

With an operating system, this can be realized by using signals. Without operating system it is sufficient to poll the function cyclically.

All function for J1939 services return the execution state as data type RET_T. When requesting a PGN this return value is not the response from the query, but the state of the request. The response of the requested node is signaled by an indication function. All indication functions have to be registered as callback functions. (see section 4).

4 Indication functions (Call back functions)

Internal events in the J1939 protocol stack can be connected with an indication function. The application must provide a call back function for this, which is called at the corresponding event. Events may be registered with the following function.

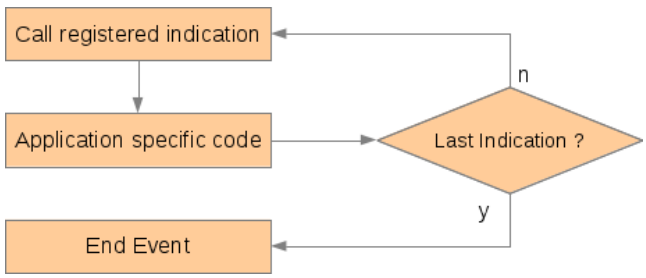
j1939EventRegister_<EVENT_TYPE>(*functionName*[1]);

For each event multiple indication functions may be registered. These functions will be called after another. The number of indication functions has to be defined in J1939 DeviceDesigner[2] or gen_define.h. The data type for *functionName-Pointer* depends on the service.

The following events can be handled:

EVENT_TYPE	Event	Parameter	Return value
COMM_EVENT	Communication state changed (e.g. TX queue full)	CAN/Comm state	
CAN_STATE	CAN state change (e.g. CAN error passive)	CAN state	
CLAIM_ADDRESS	Address claim conflict	Old claim address valid claim address	New claim address
COMMANDED_ADDRESS	New Address commanded	New Address	
REC_ALL_CLAIM_ADDRESS	any address claim received	Node address, Device NAME	
RECEIVE_PGN	PGN received SPN data already updated	PGN number Source address Receive state	
RECEIVE_CHECK_PN	PGN received SPN data **not yet** updated	PGN number Source address	Reception allowed
TRANSMIT_PGN	PGN should be transmitted	PGN number	Transmission allowed
REQUESTED_PGN	PGN request received	Requested PGN number	

If an event happens, all registered indication functions for this event are called:



1 usually the name of a function or any function pointer
2 or CAN DeviceDesigner for CAN-MultiProtocol-Stack

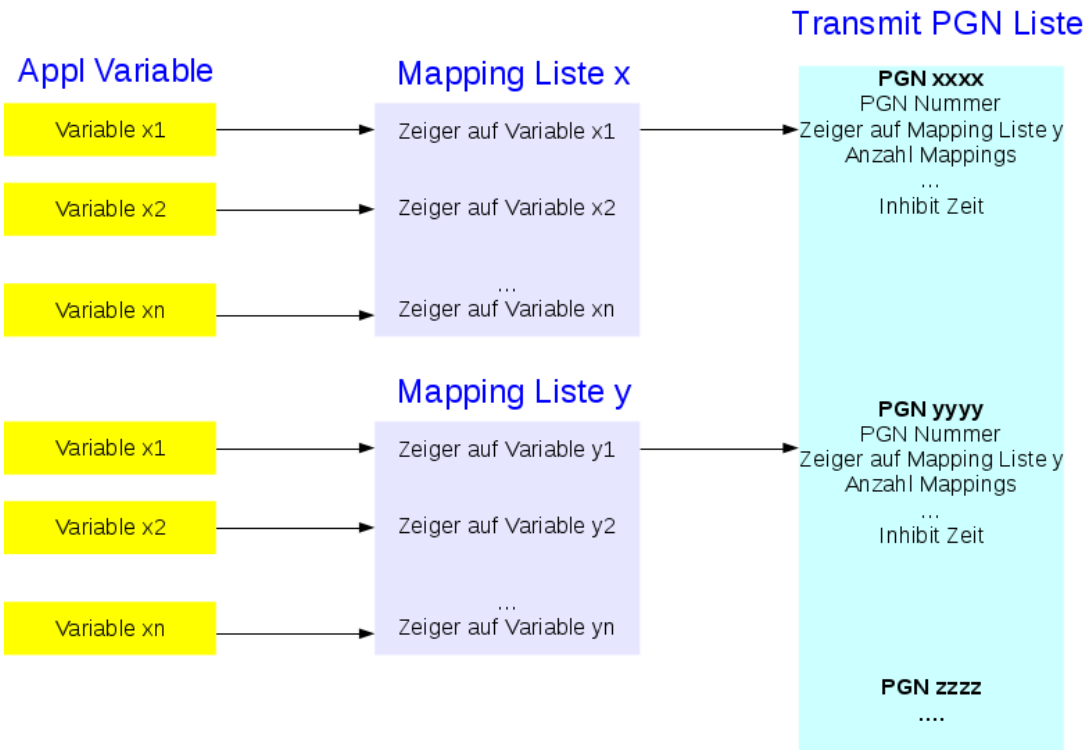
5 PGN Configuration

The configuration of TX PGNs and RX PGNs is realized by 2 lists that are passed to the stack:
- Transmit PGN list
- Receive PGN list

The transmission and reception is handled according to the attributes as specified for each PGN.
Each entry in these lists is a complete description of all attributes of this PGN:

- PGN number for reference
- PGN priority
- cycle time for transmission or monitoring of reception
- inhibit time for TX PGNs
- pointers to application variable that are assigned to a PGN
- number of assigned application variables.

The assignment of application variables is realized by using mapping tables. These contain a pointer to the application variable and the lengths of the variable in bits for transmission or reception for each entry. The internal memory format of the variable may be larger. For example a 1-bit variable may be stored as unsigned char, but for transmission or reception only the defined length in bits is used.
By using the mapping tables with pointers to application variables, the application may work directly with the application variables without the need for additional copy operations at reception or transmission. Additional so called managed variables may be used with are handled by the stack internally. Access to managed variables is only possible by API functions (see next section).
The mapping tables may not contain gaps. For unused entries dummy mapping entries have to be added for the unused bits until the next 8/16/32 bit value.



All Variables can either be defined as C variable in the application code, or as a stack managed variable. All C variables can be directly accessed, managed variables have to be accessed by the corresponding get or put function (also see next chapter).

5.1 Access functions

The access to all application variables in SPNs can be done by API functions irrespective, if the variable already exists in the application or if it is a managed variable, that is managed by the stack or if it is a dynamic SPN that has been created at run-time. In order to identify the follow parameters have to be specified:

- data direction (transmission or reception)
- PGN
- SPN.

The following access functions provide read access:

- j1939SpnGet_u8()
- j1939SpnGet_u16()
- j1939SpnGet_u32().

For access to variables with different bit sizes always the next bit size has to be used. For example when reading a 3 bit variable, j1939SpnGet_u8() shall be used.

The following access functions provide write access to SPN variables:

- j1939SpnPut_u8()
- j1939SpnPut_u16()
- j1939SpnPut_u32().

For access to variables with different bit sizes always the next bit size has to be used. For example when writing a 21 bit variable, j1939SpnPut_u32() shall be used.

All functions return the status of the access operation, so that the application is informed about the result. So it is strictly recommanded to check the return value in the application to know if a variable whas sucessfully read or written.

5.2 Dynamic PGNs

In order to create or delete PGNs and SPNS at run-time, the stack module j1939_dynpgn.c needs to be added to the project. It provides functions to create/delete/enable/disable PGNs dynamically and to change the mapping of the SPNs in the PGN.

Application variables can be used for mapped SPNs as well. In contrast to automatically created variables (using malloc) the pointer to the C variable has to be passed and the C variable will be automatically used to store the value of the SPN.

j1939DynPgnInit()	Grundlegende Initialisierung
j1939DynPgnCreatePgn(...)	Erzeugung einer J1939 PGN inkl. notwendiger Eigenschaften
j1939DynPgnCreateSpn(...)	Erzeugung und Mappen einer J1939 SPN

...

To access the content of the dynamic SPNs the normal access functions like j1939SpnPut_xx() or j1939SpnGet_xx() can be used. The #define J1939_DYNAMIC_PGN needs to be set to support dynamic PGNs.

To dynamically create PGNs malloc() and free() are used. If other target specific functions shall be used for the memory handling, the #defines J1939malloc and J1939free need to be redefined.

6 J1939 Protocol Stack Services

6.1 Initialization functions

Before the stack may be used, the following functions have to be called:

j1939HardwareInit()	Basic Hardware initialization
j1939drvCanInit()	CAN Cantroller initialization
j1939drvTimerInit()	Timer initialization
j1939CanEnable()	Start of CAN Controllers
j1939StackInit()	initialization of J1939 services

6.2 Address Claiming

The node address according to J1939 depends on the device functionality and can be specified at compile time. The device will start with a specified address as defined using the #define J1939_ADDRESS_START. If this address is already used, it has to be check with devices has the higher priority. This is based on the Device Name (see section 6.2.1). The device with the lower Device Name may keep its address, the one with the higher device name needs to use a new one. The lower the Device Name, the higher its priority.

If a device has lost the address claiming it has to:

- switch to reception mode, if the address can not be changed

- or it may try to claim a different address

- or it may wait to get a new address from a configuration tool.

The J1939 stack handles these options automatically. If the start address is already in use and the modification of the address is allowed (see Arbitrary Address Capable Bit), an indication function is called, which is registered by j1939EventRegister_ADDRESS_CLAIM(). The application may now use a different address or keep the device in reception mode.

Anyway, a node id which is set by a configuration tool overwrites the address set by the application.

The application is also informed about a successful address claiming by the ADDRESS_CLAIM indication function.

6.2.1 Device Name

Each J1939 device has a unique Device Name, which is used for address claiming and it defines the priority of the device. It is composed of the following components:

Name	Length in Bit
Identity Number	21
Manufacturer Code	11
ECU Instance	3

Name	Length in Bit
Function Instance	5
Function	8
Vehicle System	7
Vehicle System Instance	4
Industry Group	3
Arbitrary Address Capable (node Id change allowed)	1

These values must be provided by the application and may passed to the stack via the structure J1939_DEVICE_NAME_T during initialization.

6.3 PGN Transmission

All PGNs that are defined in the PGN transmit list can be sent any time as long as the node has a valid node Id. Even PGNs with a cycle time may be sent asynchronous. If the transmission is not possible, the send function return a return value which is not RET_OK.

PGNs up to a size of 8 bytes may be send by the function *j1939SendPgn()*. The stack composes the data according to the mapping and starts its transmission. (The message is copied into the TX queue and the driver will transmit it, when it is possible).

If the PGN is longer than 8 bytes, one of the transport protocols BAM or CMDT have to be used. The functions *j1939SendCmdt()* resp. *j1939SendBam()* can be used for it (see section 6.6.1 and 6.6.2).

The variables of the cyclic PGNs may be updated directly before the automatic transmission. This can be done in the indication function which is registered by *j1939EventRegister_TRANSMIT_PGN()*.

6.4 PGN Reception

All PGNs that are defined in the PGN reception list are received automatically by the stack and the application variables are updated automatically. The reception does not depend on the node Id of the producer. After all variables have been updated, the stack calls the indication function which has been registered by *j1939EventRegister_RECEIVE_PGN()* in order to inform the application about it.

In order to reject a PGN reception, another callback function needs to be register using *j1939EventRegister_RECEIVE_CHECK_PGN()*.

It is also possible to monitor the reception interval. To do so set the *cycleTime* for the PGN to a value > 0 and the stack will inform the application if the PGN is not received within the specific time.

6.5 PGN Requests

All PGNs that are defined in the PGN reception may be requested as well. This can be done by the function *j1939RequestPgn()*. The reception is done in the same way as with any other PGN.

6.6 Transport Services

In order to transfer PGNs whose length exceeds 8 bytes there are 2 services available that can transfer data up to 1785 bytes:

BAM	Broadcast Communication	No control data flow/Handshake
CMDT	Point to Point	Handshake and abort protocol defined

PGNs may be transmitted or received using these protocols. The configuration of PGNs is the same as with the normal PGNs up to 8 bytes. That means that these PGNs have to be added to the TX or RX list as well.

6.6.1 BAM

BAM messages are send as broadcast and use a cyclic communication without handshake or data flow control. The configuration is done as with all other messages in the TX PGN list resp. RX PGN list.

Transmission can be triggered by *j1939SendBam()* and the reception is the same as with other PGNs.

6.6.2 CMDT

CMDT messages are exchanged between exactly 2 nodes. To ensure that the data are acknowledged, a special handshake communication and an abort message are defined.

The configuration is done as with all other messages in the TX PGN list resp. RX PGN list.

Transmission can be started by *j1939SendCmdt().* As it is a point-to-point communication the destination address has to specified as well.

The reception is the same as with other PGNs.

6.7 Proprietary Protocol

There are 2 messages defined to transfer proprietary PGNs:

Protocol	**PGN**	**Receiver**	**Protocol for messages > 8 Bytes**
Proprietary A	oxFEBF	fixed	CMDT

Proprietary B	0xFF00..0xFFFF	Broadcast	BAM

Depending on the length of the message, it is send as standard message or using a transport protocol. The stack determines by itself the required TP protocol. The 2 functions *j1939ProprietaryTransmit_A()* and *j1939ProprietaryTransmit_B()* can be used for transmission.

6.8 Diagnostic Extension

There are several message types available for diagnostics that differ in length and error type. The definition of these messages can be created in the same way as the normal PGNs, or being used with the Diagnostic Extension Module directly. There are several functions in this module to create, compose, send, and receive diagnostic messages.

The arrangement of the diagnostic messages can be defined using the corresponding diagnostic function.

Usually Diagnostic Messages are requested by a service tool. If there is a request pending, the stack can call the callback witch is registered by *j1939RegisterEvent_REQUESTED_PGN()*. *The application can then decide to send the requested Diagnostic Message.*

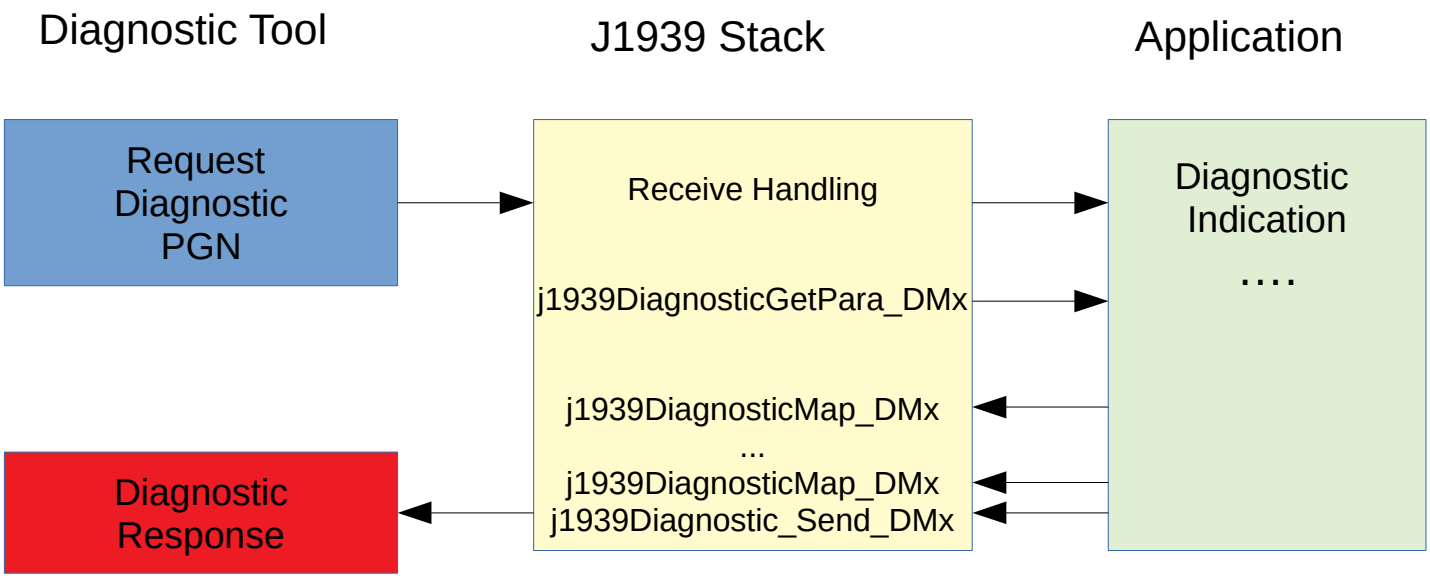
Example:

Tool requested PGN 65227 (DM2)

callback function gets called with parameter PGN 65227

Application calls mapping function *1939DiagnosticMap_DMx()* and send function *j1939DiagnosticSend_DMx()*.

Parameter of a received message can be decoded with the corresponding function like: *j1939DiagnosticGetPara_Dmx()*



All the data is saved in an internal buffer, the size of the buffer can be adjusted with the configuration tool JDD. This size is the maximum size for a Diagnostic Message.

The buffer is only valid the callback function of *j1939RegisterEvent_REQUESTED_PGN()* and can be overwritten by new messages.

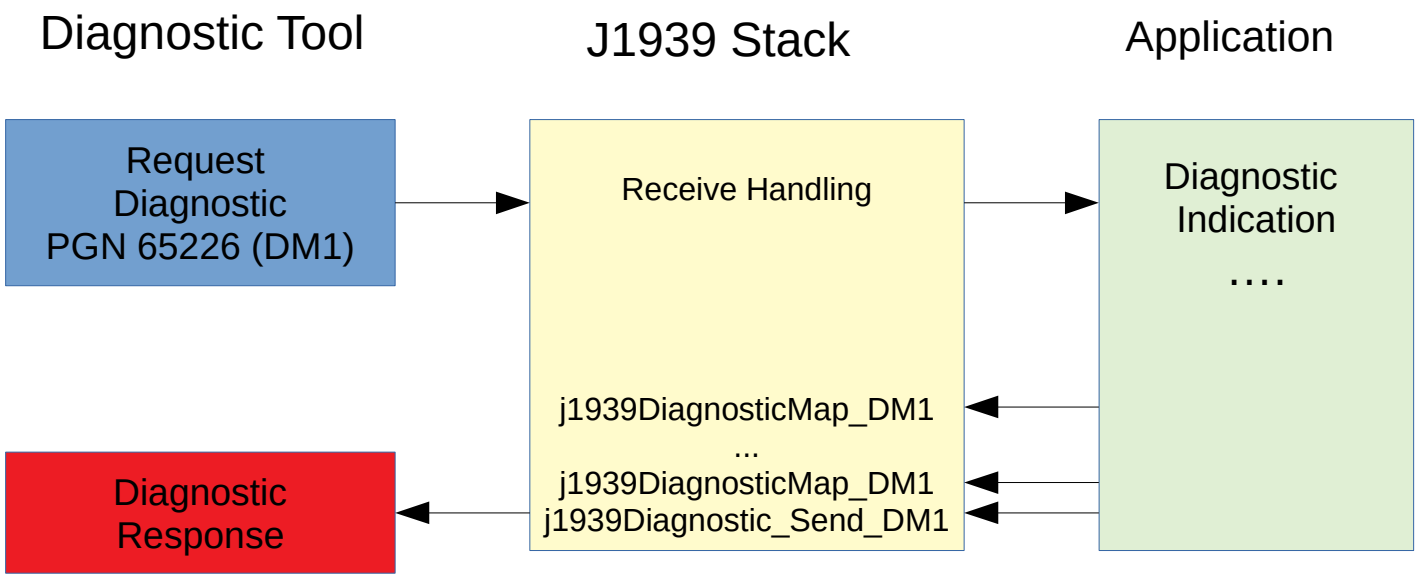
Attention! Some Diagnostic Message need the support for BAM/CMDT.

6.8.1 Diagnostic DM1

There are 2 special functions to simplify the mapping for diagnostic type DM1 messages.

The function *j1939DiagnosticMap_DM1()* can be used to update the mapping of the data for the DM1 PGN, because the number of DTC (Diagnostic Trouble Code) depends on the current system configuration.

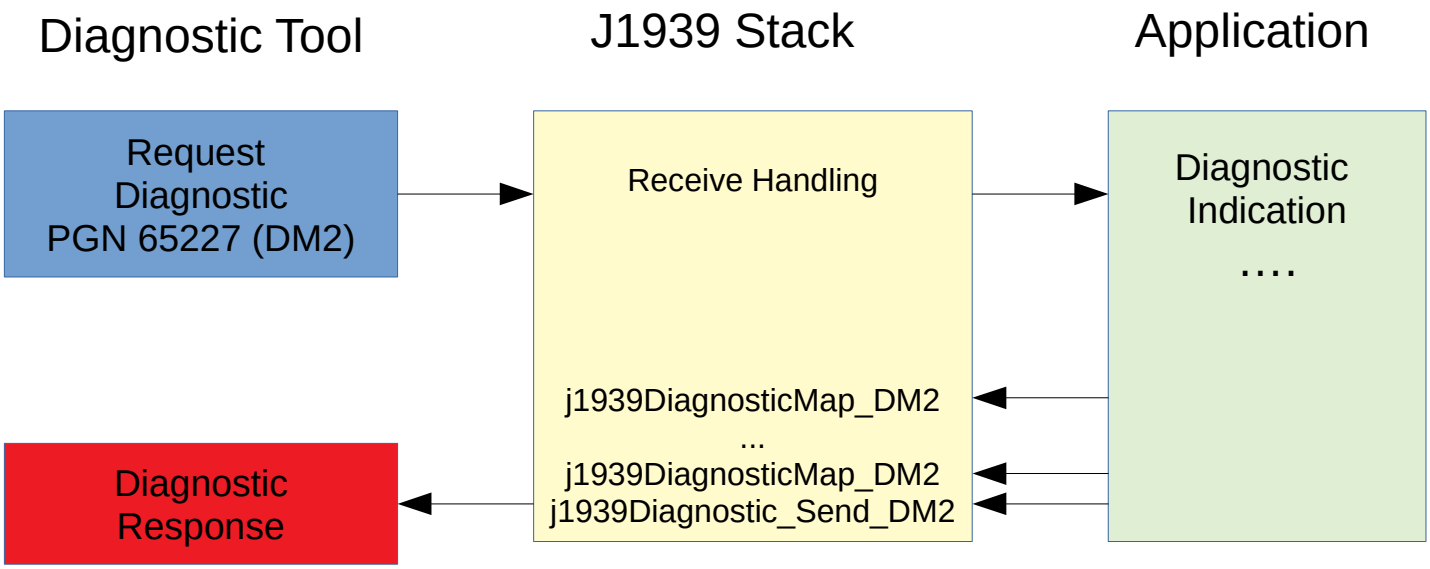
Transmission can be started by *j1939DiagnosticTransmit_DM1().*



Diagnostic data of type DM1 are transmitted cyclic. The corresponding request takes place in the indication function that was registered with *j1939RegisterEvent_DIAGNOSTIC_REQUESTED_PGN()*.

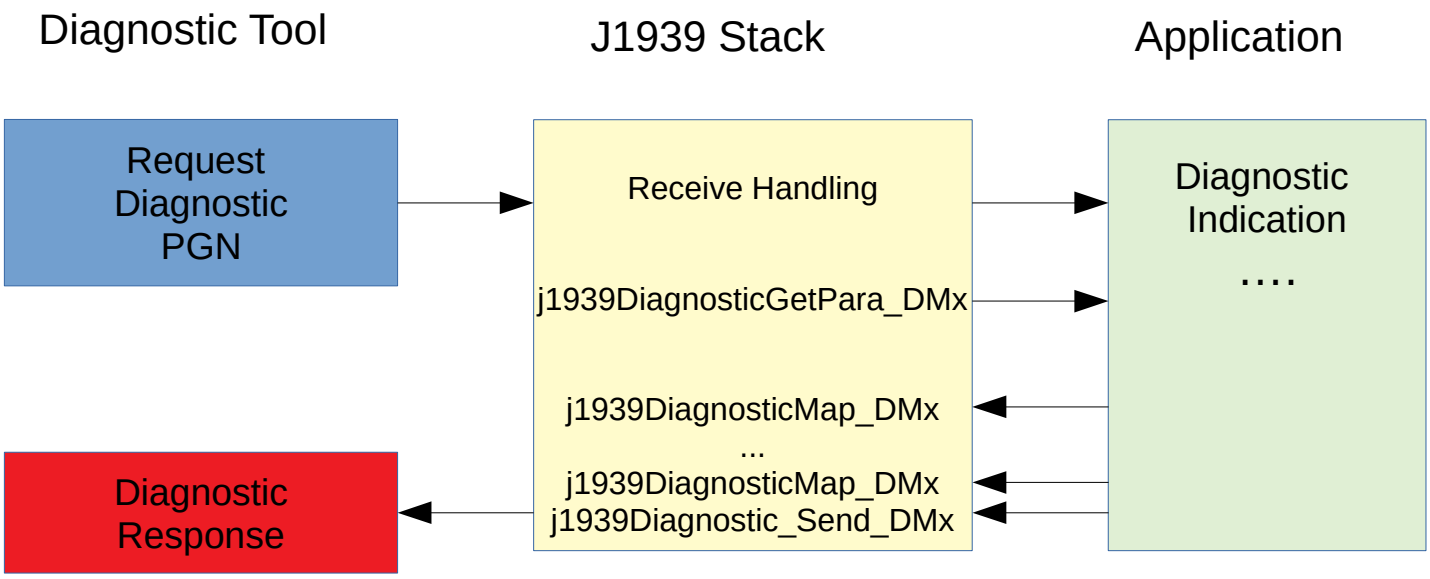
6.8.2 Diagnostic DM2

Requests for Diagnostic Message DM2 are indicated with the indication function registered with *j1939RegisterEvent_DIAGNOSTIC_REQUESTED_PGN()*. The mapping is done by calling *1939DiagnosticMap_Dm2()* the desired number of times. The sending is then triggered by calling *j1939DiagnosticSend_Dm2()*.



6.8.3 Diagnose DM3 .. DM64

Requests for Diagnose data of type DM3 are indicated by the indication function registered by *j1939RegisterEvent_DIAGNOSTIC_REQUESTED_PGN()* The single or multiple mapping is done by *j1939DiagnosticMap_DMx()* and sending is done by *j1939DiagnosticSend_DMx()*.



6.8.4 Diagnose Request Messages

Requests for diagnostic data will be indicated by the registered indication function. The single or multiple mapping is done with *j1939DiagnosticMap_DMx()* and sending is done with *j1939DiagnosticSend_DMx()*.

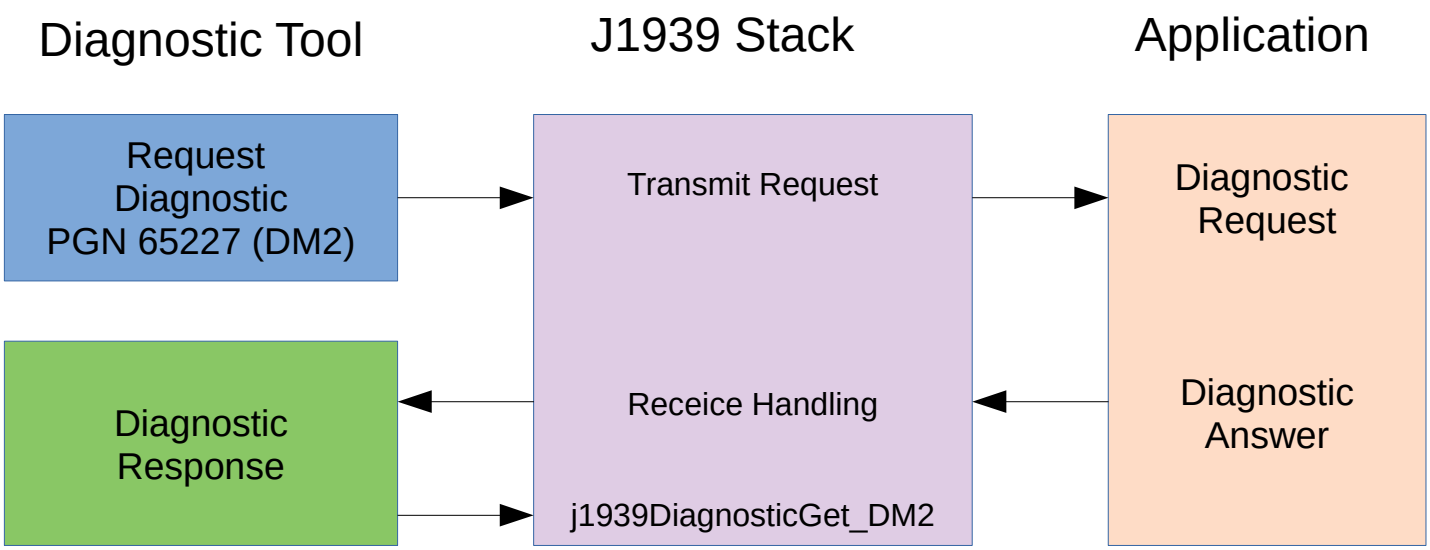
Received parameters for diagnostic messages can be determined by using *j1939DiagnosticGetPara_DMx()*.

6.9 Diagnostic Tool Extension

Usually a Diagnostic Message is requested by a tool. To request Diagnostic Messages from other devices the function *j1939RequestPgn()* can be used.

Received Diagnostic Messages are indicated to the appliction with the callback registered by *j1939Register_DIAGNOSTIC_RECEIVED_PGN()*. In the context of the callback all the received data can be evaluated.

Example: After receiving DM1 → *j1939DiagnosticGet_DM1()*



6.10 Communication state

Changes in the communication state can be triggered by hardware events (Bus-Off, Error Passive, overflow, CAN message received, transmission interrupt) or by a timer (return from bus-off). These state changes are signaled by a registered indication function (see j1939RegisterEvent_COMM_EVENT).

The following table describes which events cause a change of the communication state:

Event/Change of state	New state	Description
Bus-OFF	Bus-OFF	CAN controller is Bus-OFF, no communication possible
Bus-OFF Recovery	Bus-OFF	CAN controller tries to switch from bus-Off to active state
Return from Bus-OFF	Bus-On	CAN controller is ready to communicate and was able to receive or transmit at least 1 message
Error Passive	Bus-on, CAN passive	CAN controller is in error passive state
Error Active	Bus on	CAN controller is in error active state
CAN Controller overrun	-	Messages are lost in the CAN controller. The event is signaled at each loss of a message
REC-Queue full	-	Receive queue is full
REC-Queue overflow	-	Messages are lost because the receive queue is full. This event is signaled at each loss of a message.
TR-Queue full	Bus-Off/On, Tr-Queue full	Transmit Queue is full, the current message is saved, following message will not be saved
TR-Queue overflow	Bus-Off/On, Tr-Queue overflow	Transmit Queue is full, the message was not saved
TR-Queue empty	Bus-On, Tr-Queue ready	Transmit is ready to store messages (at least 50% free)

7 Timer Handling

The Timer handling is based on a cyclic timer. Its interval can be individually defined for each application and the use of an external timer is possible as well. A timer interval is called timer tick and the timer tick is the base for all timed actions in the J1939 Stack.

A new Timer is started by *coTimerStart()* and sorted into the linked timer list, so that all timed actions are sorted in this list. Thus after one timer tick only the first timer has to be checked as the following timers cannot be expired yet.

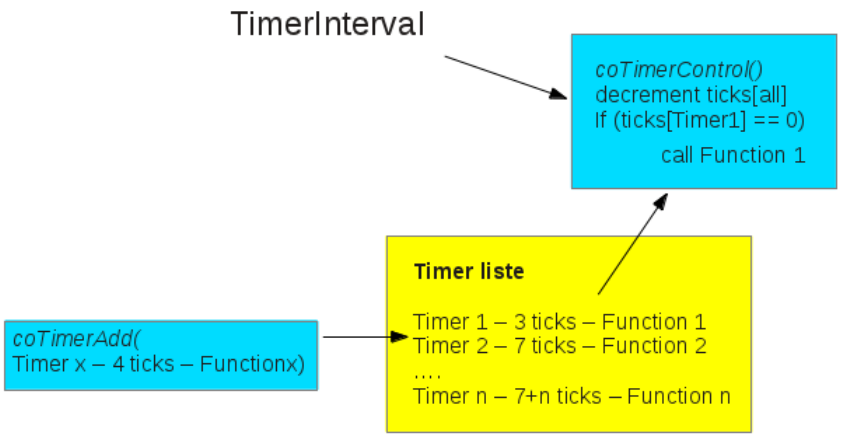


Illustration 2: Timer Handling

The timer structure must be provided by the calling function. This means also that there is no limitation of the number of timers.

It might happen that not all times will be a multiple of a timer tick. In this case it is possible to specify if the timer time shall be rounded up or down. This is done when starting the timer by using the function *coTimerStart().*

8 Driver

The driver consists of a part for the CPU and a part for the CAN controller.

CPU driver

The task of the CPU driver is to provide a constant timer tick. It can be created by a hardware interrupt or derived from another application timer.

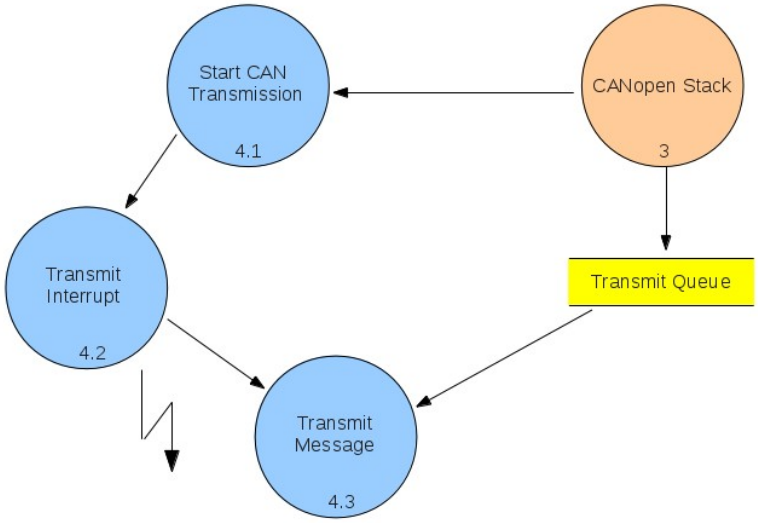
CAN driver

The task of the CAN driver is to handle and to configure the CAN controller, to send and to receive CAN messages and to provide the current state of the CAN. The buffer handling is done by the J1939 Protocol Stack.

8.1 CAN Transmit

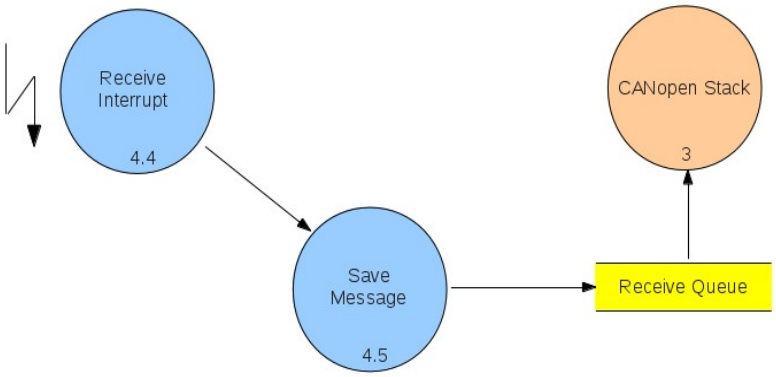
Messages to be transmitted are transferred by the J1939 stack into the transmit queue. Transmission itself is then started by the function *codrvCanStartTransmission().* Transmission of all messages is interrupt driven. The function *codrvCanStartTransmission()* has only to issue or simulate an TX interrupt.

The TX interrupt service function has to use *codrvCanTransmit()* to get the next message from the queue, program the CAN controller and transmit it. This is done until the TX queue is empty.



8.2 CAN Receive

Reception of CAN messages is interrupt driven. The received CAN message is transferred into the RX queue and can be later used by the J1939 stack.



9 Using operation systems

To use the J1939 stack together with a real time operational system (RTOS) there are two possibilities:

1. Use of the J1939 Stack within one task only and cyclically call of the central stack function
2. Separation into multiple tasks

   This requires an inter task communication.

9.1 Separation into multiple tasks

If the J1939 Stack is called from multiple tasks, polling of the central stack function is no longer necessary, but it is this the central function which has to be called at the following events:

- CAN Transmission interrupt
- CAN Receive interrupt
- CAN State interrupt (if supported)
- Timer interrupt (or timer tick signal, timer task)

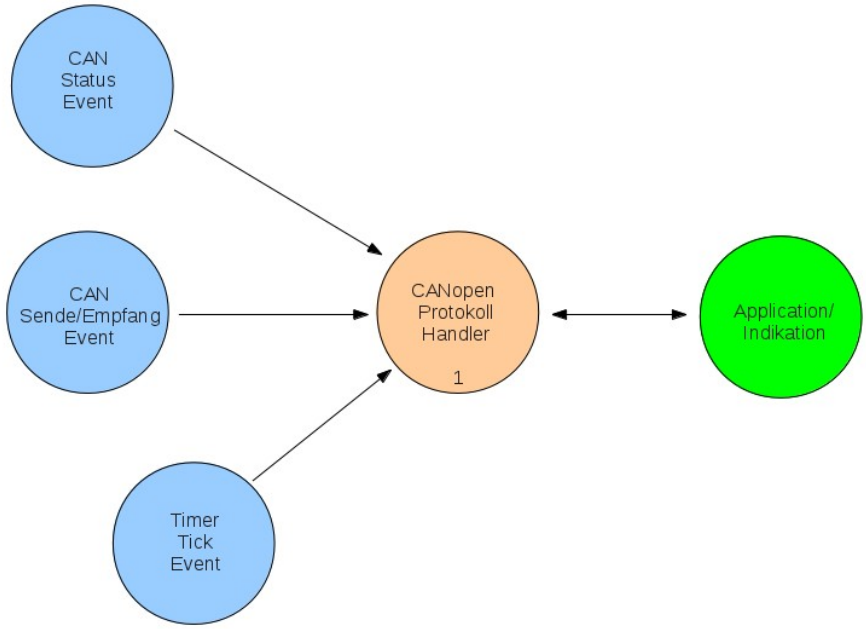
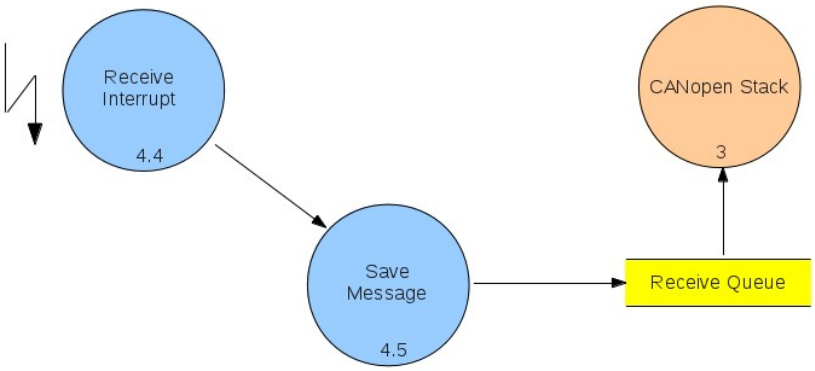


Illustration 3: Process Signal Handling

The implementation of the inter task communication and handling depends of the used operation system.

Macro	Usage	Meaning
CO_OS_SIGNAL_WAIT()	coCommTask()	Waiting for any signal
CO_OS_SIGNAL_TIMER()	Timer handler	Timer Tick
CO_OS_SIGNAL_CAN_STATE()	CAN status interrupt	Changed CAN Status
CO_OS_SIGNAL_CAN_RECEIVE()	CAN Receive Interrupt	New CAN message received
CO_OS_SIGNAL_CAN_TRANSMIT()	CAN Transmit Interrupt	New CAN message transmitted



10 Implementation of an example

The delivery contains several examples. The steps to implement an example depends on the used IDE but the basic steps are identical. Please use the example et1 to start with:

1. go to folder examples_sl/et1
2. configure PGNs in file gen_pgn.c
3. include external declarations in file gen_extern_decl.h
4. adapt configuration in gen_define.h
5. set include paths for the compiler
   - ../../j1939lib_sl/inc
   - ../../colib_sl/inc
   - ../../colib_sl/src
   - ../../codrv_sl/<drivername>
6. build project

Once this example runs, it may be modified to meet your requirements.

Files in example folder:

gen_define.h	Configuration of J1939 stack
gen_pgn.c	PGN configurations
gen_extern_decl.h	external declarations of application variables
main.c	Main file

Makefile	Makefile
<examplename>.cddp	J1939 DeviceDesigner project file

If the J1939 stack package has been purchased with a specific target driver, the will be usually a target specific example such as et1_stm32f1_attolic, or et1_rx63_hew, or et1_samc21 and others with a IDE specific project file.

11 Directory structure

j1939_lib_sl/src	J1939 Protocol Stack Sources and private header
j1939_lib_sl/inc	J1939 Protocol Stack public header files
colib_sl/src	J1939 Protocol Stack adapter layer and queue handling
codrv_sl/	CAN and CPU driver
j1939examples_sl/	example projects
ref_man	reference manual

For the multi line edition of the J1939 stack or the CAN-MultiProtocol-Stack the **_sl**-prefix will be **_ml** instead.

12 Multi-Line Handling

The usage of the Multi-Line stack is the same as with the single-line version. All described features can be used with multiple CAN lines. All data of all lines are handled separately so that all lines can be run independent of each other. The PGN defintions for multi-line applications is created in a single project of the J1939 DeviceDesigner but each line is handled in a separate way.
Each API functions has an additional argument in the beginning which indicates the line as an UNSIGNED8 Value starting at 0. This applies for all stack functions and all indication functions.

Examples for multi-line applications can be found in example_ml/xxx.

13 Changes from older versions to version V3.x

13.1 API function prefixes

The prefix of all API functions has been renamed from

j1939_xxx

to

j1939**X**xx

resp.

j1939drvXxx

Example:

old: j1939_stackInit()

new: j1939**S**tackInit()

13.2 Initialization of the stack

The functions for the initialization of the J1939 stack have been synchronized with our CANopen stack and the CAN-MultiProtocol-Stack. This additionally allows additional parameters and a more flexible usage of the stack.

The initialization sequence should look like this now:

```
/*  Initialize CAN controller with Bitrate 250 Kbit */
if (j1939drvCanInit(250) != RET_OK)  {
      return(1);
}

/* Initialize time */
if (j1939drvTimerSetup(CO_TIMER_INTERVAL) != RET_OK)  {
      return(2);
}

/* Initialize J1939 stack functions */
if (j1939StackInit() != RET_OK)  {
        return(3);
 }
```

13.3 New functionality

13.3.1 Managed variables

In addition to existing application variables now so called managed variables (MV) can be used by the J1939 stack. That means that the memory for an SPN is managed by the stack and can only be accessed by API functions like j1939SpnPut_u16(). For each SPN one can decide between an application variable or a managed variable in the J1939 DeviceDesigner.

13.3.2 Common access functions to SPNs

There is a set of common access functions for all SPNs irrespective of their typ (application variable, managed variable, dynamically created SPN). All SPNs can be accessed by the same functions like e.g. j1939SpnGet_u8() or j1939SpnPut_u32().

Specific access functions for dynamically created SPNs are not required or supported anymore.

13.3.3 Address Claiming indications

The end of the address claiming can now be signalled via the callback function registered with j1939EventRegister_CLAIM_ADDRESS(). So the application can know then a valid address has been claimed.

Additionally the function j1939AddressClaimingStart() has been added to start the address claiming with a new node address.