Protocol API

DeviceNet Master

V2.4.0

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC080301API11EN | Revision 11 | English | 2016-06 | Released | Public

Table of Contents

1 Introduction

1.1 Abstract

This manual describes the application interface of the DeviceNet-Master Stack. The aim of this manual is to support the developer during the integration process of the given stack into a user application.

This protocol stack is based on the Hilscher Task Layer Reference Programming Model. It is a description of how to program a task in general, which is defined as a combination of appropriate functions belonging to the same type of protocol layer. It further more defines how different tasks have to communicate with each other in order to exchange their layer information in between. The Reference Model is commonly used by all developers at Hilscher and shall be used by you as well when writing your Application Task on top of the Stack.

1.2 List of Revisions

Rev	Date	Name	Revisions
10	2013-09-24	RG, TD	Firmware/ stack version V 2.3.13.x Reference to netX Dual-Port Memory Interface Manual Revision 12. Added description of new request/confirmation packet in AP task: Get LED State Service - DEVNET_AP_CMD_GET_LED_STATE_REQ/CNF Added description of new indication/response packet in FAL task: DEVNET_FAL_CMD_SET_LED_IND/RES – Set LED Indication
11	2016-05-11	TD, RG	Firmware/stack version V2.4.0 Reference to netX Dual-Port Memory Interface Manual Revision 12.
			Change in section *Technical Data:* Enlargement of maximum size of input/output data from 3584 to 5736/5760 bytes.
			New source code example in section *Download Configuration Request*. See *Source Code Example: Complete Configuration Procedure* and *Source Code Example: Download Configuration Data* .
			Section *The Application Interface* partly restructured.
			Some sequence diagrams added.

Table 1: List of Revisions

1.3 Intended Audience

This manual is suitable for software developers with the following background:

- Knowledge of the programming language C
- Knowledge of the use of the real-time operating system rcX

1.4 System Requirements

This software package has the following system requirements:

- netX-Chip as CPU hardware platform
- Operating system for task scheduling required
- Operating system rcX

1.5 Specifications

1.5.1 Technical Data

The data below applies to DeviceNet Master firmware and stack version V2.4.0.

Feature	Value
Maximum number of cyclic input data	5736 bytes (status information is managed separately)
Maximum number of cyclic output data	5760 bytes
Maximum number of cyclic input data	255 bytes/connection
Maximum number of cyclic output data	255 bytes/connection
Maximum number of supported slaves	63
Maximum Configuration data	1000 bytes/slave
Acyclic communication	Explicit connection All service codes are supported
Baud rates	125 kBits/s, 250 kBit/s, 500 kBit/s Auto-detection mode is not supported.
Data transport layer	CAN frames
Connections	Bit Strobe Change of State Cyclic Poll Explicit Peer-to-Peer Messaging
Fragmentation Explicit and I/O	supported
UCMM	supported
Common and extended diagnostic	Firmware supports common and extended diagnostic in the dual-port-memory for loadable firmware.
Objects	Identity Object (Class Code 0x01) Message Router Object (Class Code 0x02) DeviceNet Object (Class Code 0x03) Connection Object (Class Code 0x05) Acknowledge Handler Object (Class Code 0x2B)
Firmware/stack available for netX	
netX 50, netX51, netX52	no
netX 100, netX 500	yes
PCI	
DMA Support for PCI targets	yes
Slot Number	
Slot number supported for	CIFX 50-DN
Configuration	
Configuration by tool SYCON.net (Download or exported configuration file named config.nxd).	
Configuration by packets to transfer bus and node parameters.	

Table 2: Technical Data DeviceNet Master Protocol Stack

1.6 Terms, Abbreviations and Definitions

Term	Description
AP	Application on top of the Stack
API	Application Programmer Interface
AREP	Application Reference End Point
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
CIP	Common Industrial Protocol
COS	Change of State
DL	Data Link (Layer)
DN	DeviceNet
DNM	DeviceNet Master
DNS	DeviceNet Slave
DN_FAL	DeviceNet Fieldbus Application Layer (abstract designation for the DeviceNet Stack)
DPM	Dual Port Memory
FAL	Fieldbus Application Layer (a.k.a. DeviceNet Stack, abstract designation for a Fieldbus layer)
LSB	Least Significant Byte
MAC ID	Media Access Control Identifier
MSB	Most Significant Byte
ODVA	Open DeviceNet Vendors Association
SDU	Service Data Unit
UCMM	Unconnected Message Manager

Table 3: Terms, abbreviations and definitions

All variables, parameters and data used in this manual have basically the LSB/MSB (“Intel”) data representation. This corresponds to the convention of the Microsoft C Compiler.

1.7 References

This document is based on the following specifications respectively documents:

[1] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual - netX based products. Revision 12, English, 2012

[2] ODVA: The CIP Networks Library, Volume 1, “Common Industrial Protocol (CIP™)”, Edition 3.10, April 2011

[3] ODVA: The CIP Networks Library, Volume 3, “DeviceNet Adaptation of CIP”, Edition 1.11, April 2011

[4] Operating Instruction Manual - DTM for Hilscher DeviceNet Master Devices, Revision 12, September 2013.

1.8 Legal Notes

1.8.1 Copyright

1.8.2 Important Notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.8.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.8.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Fundamentals

2.1 General Access Mechanisms on netX Systems

This chapter explains the possible ways to access a Protocol Stack running on a netX system:

1. By accessing the Dual Port Memory Interface directly or via a driver.
2. By accessing the Dual Port Memory Interface via a shared memory.
3. By interfacing with the Stack Task of the Protocol Stack.

The picture below visualizes these three ways:

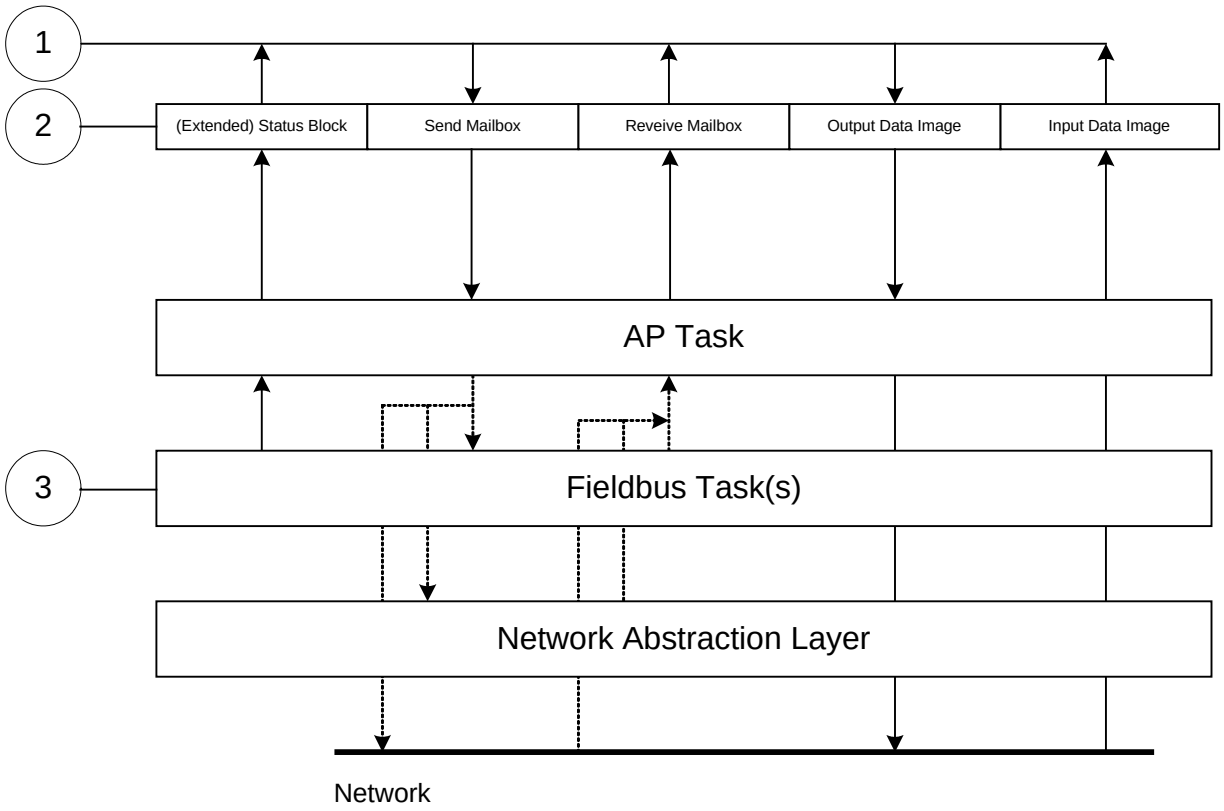

Figure 1*: The three different Ways to access a Protocol Stack running on a netX System*

This chapter explains how to program the stack (alternative 3) correctly while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the shared DPM). Finally, chapter 5 describes the entire interface to the protocol stack in detail.

Depending on whether you choose the stack-oriented approach or the Dual Port Memory-based approach, you will need either the information given in this chapter or those of the next chapter to be able to work with the set of packet interfaces described in chapter “5 The Application Interface”. All of those functions use the four parameters `ulDest, ulSrc, ulDestId and ulSrcId.` This chapter and the next one inform about how to work with these important parameters.

2.2 Accessing the Protocol Stack by Programming the AP Task's Queue

In general, programming the AP task or the stack has to be performed according to the rules of the Hilscher Task Layer Model. There you can also find more information about the variables discussed in the following section.

2.2.1 Getting the Receiver Task Handle of the Process Queue

To get the handle of the process queue of the DeviceNetAP-Task or the DeviceNet FAL -Task the Macro TLR_QUE_IDENTIFY() needs to be used. This macro delivers a pointer to the handle of the intended queue to be accessed (which is returned within the third parameter, phQue), if you provide it with the name of the queue (and an instance of your own task). The correct ASCII-queue names for accessing the DeviceNetAP -Task or the DeviceNet FAL -Task, which you have to use as current value for the first parameter (pszIdn), are

ASCII Queue name	Description
"QUE_DEVNET_AP"	Name of the DeviceNet Application-Task process queue
"DEVNET_FAL_QUE"	Name of the DeviceNet Field Application Layer-Task process queue

Table 4: Names of Queues in Device Net Firmware

The returned handle has to be used as value ulDest in all initiator packets the AP-Task intends to send to the DeviceNet FAL-Task. This handle is the same handle that has to be used in conjunction with the macros like TLR_QUE_SENDPACKET_FIFO/LIFO() for sending a packet to the respective task.

2.2.2 Meaning of Source- and Destination-related Parameters

The meaning of the source- and destination-related parameters is explained in the following table:

Variable	Meaning
ulDest	Application mailbox used for confirmation
ulSrc	Queue handle returned by TLR_QUE_IDENTIFY() as described above.
ulSrcId	Used for addressing at a lower level

Table 5: Meaning of Source- and Destination-related Parameters.

2.3 Accessing the Protocol Stack via the Dual Port Memory Interface

This chapter defines the application interface of the DeviceNet Master Stack.

2.3.1 Communication via Mailboxes

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer to and from the netX.

Send Mailbox Packet transfer from host system to netX firmware

Receive Mailbox Packet transfer from netX firmware to host system

For more details about acyclic data transfer via mailboxes, see section 3.2. Acyclic Data (Mailboxes) in this context, is described in detail in section 3.2.1 “General Structure of Messages or Packets for Non-Cyclic Data Exchange” while the possible codes that may appear are listed in section 3.2.2. “Status & Error Codes”.

However, this section concentrates on correct addressing the mailboxes.

2.3.2 Using Source and Destination Variables correctly

2.3.2.1 How to use `ulDest` for Addressing rcX and the netX Protocol Stack by the System and Channel Mailbox

The preferred way to address the netX operating system rcX is through the system mailbox; the preferred way to address a protocol stack is through its channel mailbox. All mailboxes, however, have a mechanism to route packets to a communication channel or the system channel, respectively. Therefore, the destination identifier *ulDest* in a packet header has to be filled in according to the targeted receiver. See the following example:

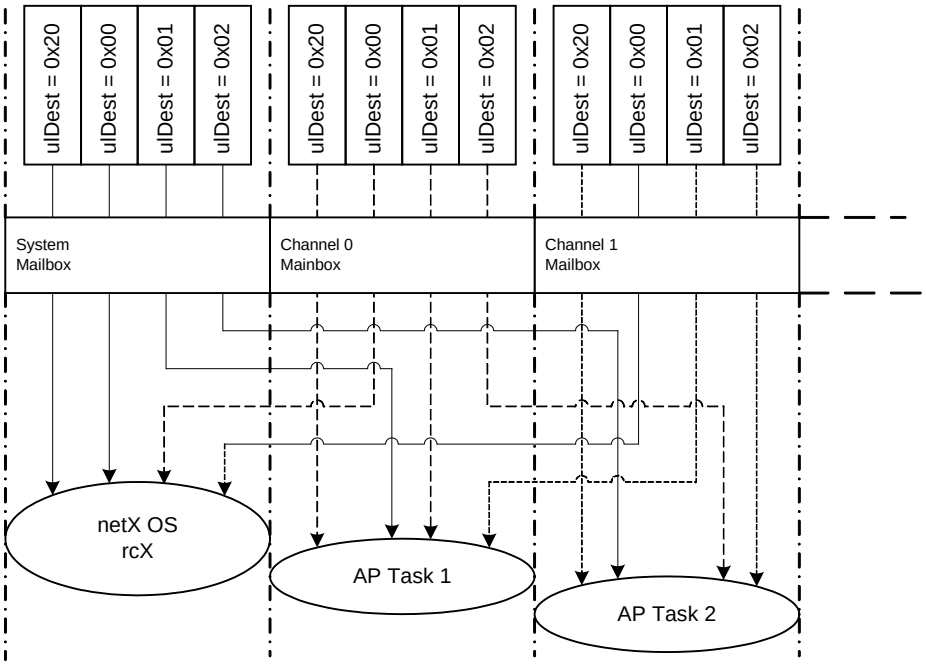

Figure 2: Use of `ulDest` in Channel and System Mailbox

For use in the destination queue handle, the tasks have been assigned to hexadecimal numerical values as described in the following table:

ulDest	Description
`0x00000000`	Packet is passed to the netX operating system rcX
`0x00000001`	Packet is passed to communication channel 0
`0x00000002`	Packet is passed to communication channel 1
`0x00000003`	Packet is passed to communication channel 2
`0x00000004`	Packet is passed to communication channel 3
`0x00000020`	Packet is passed to communication channel of the mailbox
else	Reserved, do not use

Table 6: Meaning of Destination-Parameter ulDest.Parameters.

The figure and the table above both show the use of the destination identifier *ulDest*.

A remark on the special channel identifier `0x00000020` (= *Channel Token*). The Channel Token is valid for any mailbox. That way the application uses the same identifier for all packets without actually knowing which mailbox or communication channel is applied. The packet stays 'local'. The

system mailbox is a little bit different, because it is used to communicate to the netX operating system rcX. The rcX has its own range of valid commands codes and differs from a communication channel.

Unless there is a reply packet, the netX operating system returns it to the same mailbox the request packet went through. Consequently, the host application has to return its reply packet to the mailbox the request was received from.

2.3.2.2 How to use `ulSrc` and `ulSrcId`

Generally, a netX protocol stack can be addressed through its communication channel mailbox. The example below shows how a host application addresses a protocol stack running in the context of a netX chip. The application is identified by a number (#444 in this example). The application consists of three processes identified by the numbers #11, #22 and #33. These processes communicate through the channel mailbox with the AP task of the protocol stack. Have a look at the following figure:

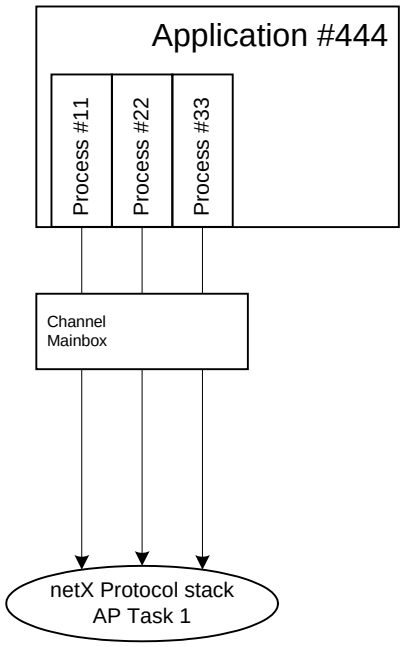

Figure 3: Using `ulSrc` and `ulSrcId`

Example:

This example applies to command messages initiated by a process in the context of the host application. If the process #22 sends a packet through the channel mailbox to the AP task, the packet header has to be filled in as follows:

Object	Variable Name	Numeric Value	Explanation
Destination Queue Handle	`ulDest`	`= 32 (0x00000020)`	This value needs always to be set to 0x00000020 (the channel token) when accessing the protocol stack via the local communication channel mailbox.
Source Queue Handle	`ulSrc`	`= 444`	Denotes the host application (#444).
Destination Identifier	`ulDestId`	`= 0`	In this example, it is not necessary to use the destination identifier.
Source Identifier	`ulSrcId`	`= 22`	Denotes the process number of the process within the host application and needs therefore to be supplied by the programmer of the host application.

Table 7: Example for correct Use of Source- and Destination-related Parameters.:

For packets through the channel mailbox, the application uses 32 (= 0x20, *Channel Token*) for the destination queue handler *ulDest*. The source queue handler *ulSrc* and the source identifier *ulSrcId* are used to identify the originator of a packet. The destination identifier *ulDestId* can be used to address certain resources in the protocol stack. It is not used in this example. The source queue handler *ulSrc* has to be filled in. Therefore, its use is mandatory; the use of *ulSrcId* is optional.

The netX operating system passes the request packet to the protocol stack's AP task. The protocol stack then builds a reply to the packet and returns it to the mailbox. The application has to make sure that the packet finds its way back to the originator (process #22 in the example).

How to Route rcX Packets

To route an rcX packet the source identifier *ulSrcId* and the source queues handler *ulSrc* in the packet header hold the identification of the originating process. The router saves the original handle from *ulSrcId* and *ulSrc*. The router uses a handle of its own choices for *ulSrcId* and *ulSrc* before it sends the packet to the receiving process. That way the router can identify the corresponding reply packet and matches the handle from that packet with the one stored earlier. Now the router replaces its handles with the original handles and returns the packet to the originating process.

2.3.3 Obtaining useful Information about the Communication Channel

A communication channel represents a part of the Dual Port Memory and usually consists of the following elements:

- Output Data Image

  is used to transfer cyclic process data to the network (normal or high-priority)

- Input Data Image

  is used to transfer cyclic process data from the network (normal or high-priority)

- Send Mailbox

  is used to transfer non-cyclic data to the netX

- Receive Mailbox

  is used to transfer non-cyclic data from the netX

- Control Block

  allows the host system to control certain channel functions

- Common Status Block

  holds information common to all protocol stacks

- Extended Status Block

  holds protocol specific network status information

This section describes a procedure how to obtain useful information for accessing the communication channel(s) of your netX device and to check if it is ready for correct operation.

Proceed as follows:

1) Start with reading the channel information block within the system channel (usually starting at address `0x0030`).

2) Then you should check the hardware assembly options of your netX device. They are located within the system information block following offset `0x0010` and stored as data type `UINT16`. The following table explains the relationship between the offsets and the corresponding xC Ports of the netX device:

`0x0010`	Hardware Assembly Options for xC Port[0]
`0x0012`	Hardware Assembly Options for xC Port[1]
`0x0014`	Hardware Assembly Options for xC Port[2]
`0x0016`	Hardware Assembly Options for xC Port[3]

Table 8: Address Assignment of Hardware Assembly Options

Check each of the hardware assembly options whether its value has been set to `RCX_HW_ASSEMBLY_DEVICENET = 0x0040`. If true, this denotes that this xCPort is suitable for running the DeviceNet Master protocol stack. Otherwise, this port is designed for another communication protocol. In most cases, xC Port[2] will be used for field bus systems, while xC Port[0] and xC Port[1] are normally used for Ethernet communication.

3) You can find information about the corresponding communication channel (0…3) under the following addresses:

0x0050	Communication Channel 0
0x0060	Communication Channel 1
0x0070	Communication Channel 2
0x0080	Communication Channel 3

Table 9: Addressing Communication Channel 0-3

In devices which support only one communication system which is usually the case (either a single field bus system or a single standard for Industrial-Ethernet communication), always communication channel 0 will be used. In devices supporting more than one communication system you should also check the other communication channels.

4) There you can find such information as the ID (containing channel number and port number) of the communication channel, the size and the location of the handshake cells, the overall number of blocks within the communication channel and the size of the channel in bytes. Evaluate this information precisely in order to access the communication channel correctly.

The information is delivered as follows:

Size of Channel in Bytes

Address	Data Type	Description
0x0050	UINT8	Channel Type = COMMUNICATION (must have the fixed value define RCX_CHANNEL_TYPE_COMMUNICATION = 0x05)
0x0051	UINT8	ID (Channel Number, Port Number)
0x0052	UINT8	Size / Position Of Handshake Cells
0x0053	UINT8	Total Number Of Blocks Of This Channel
0x0054	UINT32	Size Of Channel In Bytes
0x0058	UINT8[8]	Reserved (set to zero)

Table 10: Address Assignment of Communication Channels demonstrated at Communication Channel 0

These addresses correspond to communication channel 0, for communication channels 1, 2 and 3 you have to add an offset of 0x0010, 0x0020 or 0x0030 to the address values, respectively.

2.4 Client/Server Mechanism

2.4.1 Application as Client

The host application may send request packets to the netX firmware at any time (transition 1 ⇨ 2). Depending on the protocol stack running on the netX, parallel packets are not permitted (see protocol specific manual for details). The netX firmware sends a confirmation packet in return, signaling success or failure (transition 3 ⇨ 4) while processing the request.

The host application has to register with the netX firmware in order to receive indication packets (transition 5 ⇨ 6). This can be done using the RCX_REGISTER_APP_REQ packet. For more information how to use this packet for registration, see the DPM manual (reference [1]), chapter 4.18 “Register / Unregister an Application. Depending on the command code of the indication packet, a response packet to the netX firmware may or may not be required (transition 7 ⇨ 8).

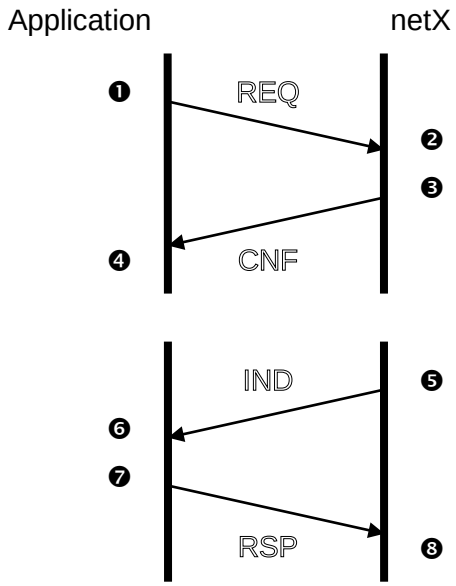

Figure 4: Transition Chart Application as Client

❶ ❷ The host application sends request packets to the netX firmware.

❸ ❹ The netX firmware sends a confirmation packet in return.

❺ ❻ The host application receives indication packets from the netX firmware.

❼ ❽ The host application sends response packet to the netX firmware (may not be required).

REQ	Request	CNF	Confirmation
IND	Indication	RSP	Response

2.4.2 Application as Server

The host application has to register with the netX firmware in order to receive indication packets. Depending on the protocol stack, this is done either implicit (if application opens a TCP/UDP socket) or explicit (if application wants to receive unsolicited DPV1 packets).

When an appropriate event occurs and the host application is registered to receive such a notification, the netX firmware passes an indication packet through the mailbox (transition 1 ⇨ 2). The host application is expected to send a response packet back to the netX firmware (transition 3 ⇨ 4).

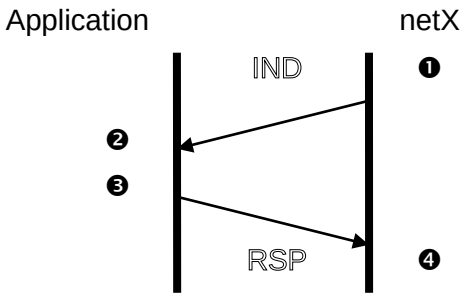

Figure 5: Transition Chart Application as Server

❶ ❷ The netX firmware passes an indication packet through the mailbox.

❸ ❹ The host application sends response packet to the netX firmware.

IND Indication RSP Response

3 Dual-Port Memory

All data in the dual-port memory is structured in blocks. According to their functions, these blocks use different data transfer mechanisms. For example, data transfer through mailboxes uses a synchronized handshake mechanism between host system and netX firmware. The same is true for IO data images, when a buffered handshake mode is configured. Other blocks, like the status block, are read by the host application and use no synchronization mechanism.

Types of blocks in the dual-port memory are outlined below:

Mailbox	transfer non-cyclic messages or packages with a header for routing information
Data Area	holds the process image for cyclic I/O data or user defined data structures
Control Block	is used to signal application related state to the netX firmware
Status Block	holds information regarding the current network state
Change of State	collection of flags that initiate execution of certain commands or signal a change of state

3.1 Cyclic Data (Input/Output Data)

The input block holds the process data image received **from** the network whereas the output block holds data sent **to** the network

For the controlled / buffered mode, the protocol stack updates the process data in the internal input buffer for each valid bus cycle. Each IO block uses handshake bits for access synchronization. Input and output data block handshake operates independently from each other. When the application toggles the input handshake bit, the protocol stack copies the data from the internal buffer into the input data image of the dual-port memory. Now the application can copy data from the dual-port memory and then give control back to the protocol stack by toggling the appropriate input handshake bit. When the application / driver toggles the output handshake bit, the protocol stack copies the data from the output data image of the dual-port memory into the internal buffer. From there the data is transferred to the network. The protocol stack toggles the handshake bits back, indicating to the application that the transfer is finished and a new data exchange cycle may start. This mode guarantees data consistency over both input and output area.

3.1.1 Input Process Data

The input data block is used by field bus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The input data image is used to receive cyclic data **from** the network.

The default size of the input data image is 5760 byte. However, not all available space is actually used by the protocol stack. Depending on the specific protocol, the area actually available for user data might be much smaller than 5760 byte. An input data block may or may not be available in the dual-port memory. It is always available in the default memory map (see the *netX Dual-Port Memory Manual [1]*).

Note: 24 byte are used for status information (8 byte for the list of configured slaves, 8 byte for the list of activated slaves and 8 byte for the list of slaves with faults or errors). Therefore the maximum amount of really usable input data is 5736 byte.

The contents of these 24 byte is identical to the contents of the second part of the Extended Status Block beginning at address 0x0100, see *Table 32: Extended Status Block for DeviceNet-Master – Second part (State Field Definition Block)* of this document.

Input Data Image			
Offset	**Type**	**Name**	**Description**
0x2680	UINT8	`abPd0Input[5760]`	Input Data Image Cyclic Data From The Network

Table 11: Input Data Image

3.1.2 Output Process Data

The output data block is used by field bus or industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The output data Image is used to send cyclic data from the host **to** the network.

The default size of the output data image is 5760 byte. However, not all available space is actually used by the protocol stack. Depending on the specific protocol, the area actually available for user data might be much smaller than 5760 byte. An output data block may or may not be available in the dual-port memory. It is always available in the default memory map (see *netX DPM Manual [1]*).

Output Data Image			
Offset	**Type**	**Name**	**Description**
0x1000	UINT8	`abPd0Output[5760]`	Output Data Image Cyclic Data To The Network

Table 12: Output Data Image

3.2 Acyclic Data (Mailboxes)

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer to and from the netX processor.

Send Mailbox	Packet transfer from host system to firmware
Receive Mailbox	Packet transfer from firmware to host system

The send and receive mailbox areas are used by field bus and industrial Ethernet protocols providing a non-cyclic data exchange mechanism. Another use of the mailbox system is to allow access to the firmware running on the netX chip itself for diagnostic and identification purposes.

The send mailbox is used to transfer acyclic data **to** the network or **to** the firmware. The receive mailbox is used to transfer acyclic data **from** the network or **from** the firmware.

A send/receive mailbox may or may not be available in the communication channel. It depends on the function of the firmware whether or not a mailbox is needed. The location of the system mailbox and the channel mailbox is described in the *netX DPM Interface Manual [1]*.

Note: Each mailbox can hold one packet at a time. The netX firmware stores packets that are not retrieved by the host application in a packet queue. This queue has limited space and may fill up so new packets maybe lost. To avoid these data loss situations, it is strongly recommended to empty the mailbox frequently, even if packets are not expected by the host application. Unexpected command packets should be returned to the sender with an Unknown Command in the status field; unexpected reply messages can be discarded.

3.2.1 General Structure of Messages or Packets for Non-Cyclic Data Exchange

The non-cyclic packets through the netX mailbox have the following structure:

Structure Information				
Area	**Variable**	**Type**	**Value / Range**	**Description**
Head	Structure Information			
	`ulDest`	UINT32		Destination Queue Handle
	`ulSrc`	UINT32		Source Queue Handle
	`ulDestId`	UINT32		Destination Queue Reference
	`ulSrcId`	UINT32		Source Queue Reference
	`ulLen`	UINT32		Packet Data Length (In Bytes)
	`ulId`	UINT32		Packet Identification As Unique Number
	`ulSta`	UINT32		Status / Error Code
	`ulCmd`	UINT32		Command / Response
	`ulExt`	UINT32		Extension Flags
	`ulRout`	UINT32		Routing Information
Data	Structure Information			
	…	…		User Data Specific To The Command

Table 13: General Structure of Packets for non-cyclic Data Exchange.

Some of the fields are mandatory; some are conditional; others are optional. However, the size of a packet is always at least 10 double-words or 40 bytes. Depending on the command, a packet may or may not have a data field. If present, the content of the data field is specific to the command, respectively the reply.

Destination Queue Handle

The *ulDest* field identifies a task queue in the context of the netX firmware. The task queue represents the final receiver of the packet and is assigned to a protocol stack. The *ulDest* field has to be filled out in any case. Otherwise, the netX operating system cannot route the packet. This field is mandatory.

Source Queue Handle

The *ulSrc* field identifies the sender of the packet. In the context of the netX firmware (inter-task communication) this field holds the identifier of the sending task. Usually, a driver uses this field for its own handle, but it can hold any handle of the sending process. Using this field is mandatory. The receiving task does not evaluate this field and passes it back unchanged to the originator of the packet.

Destination Identifier

The *ulDestId* field identifies the destination of an unsolicited packet from the netX firmware to the host system. It can hold any handle that helps to identify the receiver. Therefore, its use is mandatory for unsolicited packets. The receiver of unsolicited packets has to register for this.

Source Identifier

The *ulSrcId* field identifies the originator of a packet. This field is used by a host application, which passes a packet from an external process to an internal netX task. The *ulSrcId* field holds the handle of the external process. When netX operating system returns the packet, the application can identify the packet and returns it to the originating process. The receiving task on the netX does not evaluate this field and passes it back unchanged. For inter-task communication, this field is not used.

Length of Data Field

The *ulLen* field holds the size of the data field in bytes. It defines the total size of the packet's payload that follows the packet's header. The size of the header is not included in *ulLen*. So the total size of a packet is the size from *ulLen* plus the size of packet's header. Depending on the command, a data field may or may not be present in a packet. If no data field is included, the length field is set to zero.

Identifier

The *ulId* field is used to identify a specific packet among others of the same kind. That way the application or driver can match a specific reply or confirmation packet to a previous request packet. The receiving task does not change this field and passes it back to the originator of the packet. Its use is optional in most of the cases. However, it is mandatory for sequenced packets.

Example: Downloading big amounts of data that does not fit into a single packet. For a sequence of packets the identifier field is incremented by one for every new packet.

Status / Error Code

The *ulSta* field is used in response or confirmation packets. It informs the originator of the packet about success or failure of the execution of the command. The field may be also used to hold status information in a request packet.

Command / Response

The *ulCmd* field holds the command code or the response code, respectively. The command/response is specific to the receiving task. If a task is not able to execute certain commands, it will return the packet with an error indication. A command is always even (the least significant bit is zero). In the response packet, the command code is incremented by one indicating a confirmation to the request packet.

Extension Flags

The extension field *ulExt* is used for controlling packets that are sent in a sequenced manner. The extension field indicates the first, last or a packet of a sequence. If sequencing is not required, the extension field is not used and set to zero.

Routing Information

The *ulRout* field is used internally by the netX firmware only. It has no meaning to a driver type application and therefore set to zero.

User Data Field

This field contains data related to the command specified in *ulCmd* field. Depending on the command, a packet may or may not have a data field. The length of the data field is given in the *ulLen* field.

3.2.2 Status & Error Codes

The following status and error codes can be returned in `ulSta`: For list of error codes please see manual named *netX Dual-Port Memory Interface [1]*.

3.2.3 Differences between System and Channel Mailboxes

The mailbox system on netX provides a non-cyclic data transfer channel for field bus and industrial Ethernet protocols. Another use of the mailbox is allowing access to the firmware running on the netX chip itself for diagnostic purposes. There is always a send and a receive mailbox. Send and receive mailboxes utilize handshake bits to synchronize these data or diagnostic packages through the mailbox. There is a pair of handshake bits for both the send and receive mailbox.

The netX operating system rcX only uses the system mailbox.

- The *system mailbox*, used to send packet to rcX operating system, however, has a mechanism to route packets to a communication channel (protocol stack).
- A *channel mailbox* passes packets to its own protocol stack only.

3.2.4 Send Mailbox

The send mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **send** mailbox is used to transfer non-cyclic data **to** the network or **to** the protocol stack.

The size is 1596 bytes for the send mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of packages that can be accepted.

3.2.5 Receive Mailbox

The receive mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **receive** mailbox is used to transfer non-cyclic data **from** the network or **from** the protocol stack.

The size is 1596 bytes for the receive mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of waiting packages (for the receive mailbox).

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

Master Status			
Offset	**Type**	**Name**	**Description**
0x0200	UINT16	usPackagesAccepted	Packages Accepted Number of Packages that can be Accepted
0x0202	UINT16	usReserved	Reserved Set to 0
0x0204	UINT8	abSendMbx[1596]	Send Mailbox Non Cyclic Data To The Network or to the Protocol Stack
0x0840	UINT16	usWaitingPackages	Packages waiting Counter of packages that are waiting to be processed
0x0842	UINT16	usReserved	Reserved Set to 0
0x0844	UINT8	abRecvMbx[1596]	Receive Mailbox Non Cyclic Data **from** the network or **from** the protocol stack

Table 14: Channel Mailboxes.

Channel Mailboxes Structure

```
typedef struct tagNETX_SEND_MAILBOX_BLOCK
{
UINT16 usPackagesAccepted;
UINT16 usReserved;
UINT8 abSendMbx[ 1596 ];
} NETX_SEND_MAILBOX_BLOCK;
typedef struct tagNETX_RECV_MAILBOX_BLOCK
{
UINT16 usWaitingPackages;
UINT16 usReserved;
UINT8 abRecvMbx[ 1596 ];
} NETX_RECV_MAILBOX_BLOCK;
```

3.3 Status

A status block is present within the communication channel. It contains information about network and task related issues. In some respects, status and control block are used together in order to exchange information between host application and netX firmware. The application reads a status block whereas the control block is written by the application. Both status and control block have registers that use the *Change of State* mechanism (see also section 2.2.1 of the *netX Dual-Port-Memory Manual [1]*).

3.3.1 Common Status

The Common Status Block contains information that is the same for all communication channels. The start offset of this block depends on the size and location of the preceding blocks. The status block is always present in the dual-port memory.

3.3.1.1 All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Structure Definition

Common Status			
Offset	**Type**	**Name**	**Description**
0x0010	UINT32	`ulCommunicationCOS`	Communication Change of State READY, RUN, RESET REQUIRED, NEW, CONFIG AVAILABLE, CONFIG LOCKED
0x0014	UINT32	`ulCommunicationState`	Communication State NOT CONFIGURED, STOP, IDLE, OPERATE
0x0018	UINT32	`ulCommunicationError`	Communication Error Unique Error Number According to Protocol Stack
0x001C	UINT16	`usVersion`	Version Version Number of this Diagnosis Structure
0x001E	UINT16	`usWatchdogTime`	Watchdog Timeout Configured Watchdog Time
0x0020	UINT16	`usHandshakeMode`	Handshake Mode Process Data Transfer Mode (see netX DPM Interface Manual)
0x0022	UINT16	`usReserved`	Reserved. Set to 0
0x0024	UINT32	`ulHostWatchdog`	Host Watchdog Joint Supervision Mechanism Protocol Stack Writes, Host System Reads
0x0028	UINT32	`ulErrorCount`	Error Count Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	`ulErrorLoglnd`	Error Log Indicator Total Number Of Entries In The Error Log Structure (not supported yet)
0x0030	UINT32	`ulReserved[2]`	Reserved. Set to 0

Table 15: Common Status Structure Definition

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
  UINT32  ulCommunicationCOS;
  UINT32  ulCommunicationState;
  UINT32  ulCommunicationError;
  UINT16  usVersion;
  UINT16  usWatchdogTime;
  UINT16  ausReserved[2];
  UINT32  ulHostWatchdog;
  UINT32  ulErrorCount;
  UINT32  ulErrorLogInd;
  UINT32  ulReserved[2];
  union
  {
    NETX_MASTER_STATUS_T  tMasterStatus;    /* for master implementation */
    UINT32                aulReserved[6];   /* otherwise reserved        */
  } unStackDepended;
} NETX_COMMON_STATUS_BLOCK_T;
```

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
  UINT32  ulCommunicationCOS;
  UINT32  ulCommunicationState;
  UINT32  ulCommunicationError;
  UINT16  usVersion;
  UINT16  usWatchdogTime;
  UINT16  ausReserved[2];
  UINT32  ulHostWatchdog;
  UINT32  ulErrorCount;
  UINT32  ulErrorLogInd;
  UINT32  ulReserved[2];
  union
  {
    NETX_MASTER_STATUS_T  tMasterStatus;    /* for master implementation */
    UINT32                aulReserved[6];   /* otherwise reserved        */
  } unStackDepended;
} NETX_COMMON_STATUS_BLOCK_T;
```

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the n*etX Change of State Command* flag in the netX communication flags register (see section 3.2.2.1 of the netX DPM Interface Manual [1]). The application then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state (see section 3.2.2.2 of the netX DPM Interface Manual [1]).

ulCommunicationCOS - netX writes, Host reads		
Bit	**Short name**	**Name**
D31..D7	unused, set to zero	
D6	Restart Required Enable	RCX_COMM_COS_RESTART_REQUIRED_ENABLE
D5	Restart Required	RCX_COMM_COS_RESTART_REQUIRED
D4	Configuration New	RCX_COMM_COS_CONFIG_NEW
D3	Configuration Locked	RCX_COMM_COS_CONFIG_LOCKED
D2	Bus On	RCX_COMM_COS_BUS_ON
D1	Running	RCX_COMM_COS_RUN
D0	Ready	RCX_COMM_COS_READY

Table 16: Communication State of Change

Communication Change of State Flags (netX System ⇨ Application)

Bit	Definition / Description
0	Ready (RCX_COMM_COS_READY) 0 - … 1 - The *Ready* flag is set as soon as the protocol stack is started properly. Then the protocol stack is awaiting a configuration. As soon as the protocol stack is configured properly, the *Running* flag is set, too.
1	Running (RCX_COMM_COS_RUN) 0 - … 1 -The *Running* flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the *Ready* flag and the *Running* flag are set.
2	Bus On (RCX_COMM_COS_BUS_ON) 0 - … 1 -The *Bus On* flag is set to indicate to the host system whether or not the protocol stack has the permission to open network connections. If set, the protocol stack has the permission to communicate on the network; if cleared, the permission was denied and the protocol stack will not open network connections.
3	Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - … 1 -The *Configuration Locked* flag is set, if the communication channel firmware has locked the configuration database against being overwritten. Re-initializing the channel is not allowed in this state. To unlock the database, the application has to clear the *Lock Configuration* flag in the control block (see page 48).
4	Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - … 1 -The *Configuration New* flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated. This flag may be set together with the *Restart Required* flag.
5	Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - … 1 -The *Restart Required* flag is set when the channel firmware requests to be restarted. This flag is used together with the *Restart Required Enable* flag below. Restarting the channel firmware may become necessary, if a new configuration was downloaded from the host application or if a configuration upload via the network took place.
6	Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - … 1 - The *Restart Required Enable* flag is used together with the *Restart Required* flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the *Enable* mechanism see section 2.3.2 of the netX DPM Interface Manual [1])).
7 … 31	Reserved, set to 0

Table 17: Meaning of Communication Change of State Flags

Communication State (All Implementations)

The communication state field contains information regarding the current network status of the communication channel. Depending on the implementation, all or a subset of the definitions below is supported.

- UNKNOWN #define RCX_COMM_STATE_UNKNOWN 0x00000000
- NOT_CONFIGURED #define RCX_COMM_STATE_NOT_CONFIGURED 0x00000001
- STOP #define RCX_COMM_STATE_STOP 0x00000002
- IDLE #define RCX_COMM_STATE_IDLE 0x00000003
- OPERATE #define RCX_COMM_STATE_OPERATE 0x00000004

Communication Channel Error (All Implementations)

This field holds the current error code of the communication channel. If the cause of error is resolved, the communication error field is set to zero (= RCX_SYS_SUCCESS) again. Not all of the error codes are supported in every implementation. Protocol stacks may use a subset of the error codes below.

- SUCCESS #define RCX_SYS_SUCCESS 0x00000000

Runtime Failures

- WATCHDOG TIMEOUT #define RCX_E_WATCHDOG_TIMEOUT 0xC000000C

Initialization Failures

- (General) INITIALIZATION FAULT
  `#define RCX_E_INIT_FAULT 0xC0000100`
- DATABASE ACCESS FAILED `#define RCX_E_DATABASE_ACCESS_FAILED 0xC0000101`

Configuration Failures

- NOT CONFIGURED #define RCX_E_NOT_CONFIGURED 0xC0000119
- (General) CONFIGURATION FAULT
  #define RCX_E_CONFIGURATION_FAULT 0xC0000120
- INCONSISTENT DATA SET #define RCX_E_INCONSISTENT_DATA_SET 0xC0000121
- DATA SET MISMATCH #define RCX_E_DATA_SET_MISMATCH 0xC0000122
- INSUFFICIENT LICENSE #define RCX_E_INSUFFICIENT_LICENSE 0xC0000123
- PARAMETER ERROR #define RCX_E_PARAMETER_ERROR 0xC0000124
- INVALID NETWORK ADDRESS #define RCX_E_INVALID_NETWORK_ADDRESS 0xC0000125
- NO SECURITY MEMORY #define RCX_E_NO_SECURITY_MEMORY 0xC0000126

Network Failures

- (General) NETWORK FAULT #define RCX_COMM_NETWORK_FAULT 0xC0000140
- CONNECTION CLOSED #define RCX_COMM_CONNECTION_CLOSED 0xC0000141
- CONNECTION TIMED OUT #define RCX_COMM_CONNECTION_TIMEOUT 0xC0000142
- LONELY NETWORK #define RCX_COMM_LONELY_NETWORK 0xC0000143
- DUPLICATE NODE #define RCX_COMM_DUPLICATE_NODE 0xC0000144
- CABLE DISCONNECT #define RCX_COMM_CABLE_DISCONNECT 0xC0000145

Version (All Implementations)

The version field holds version of this structure. It starts with one; zero is not defined.

- STRUCTURE VERSION #define RCX_STATUS_BLOCK_VERSION 0x0001

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see section 4.13 of the netX DPM Interface Manual [1].

Host Watchdog (All Implementations)

The protocol stack supervises the host system using the watchdog function. If the application fails to copy the value from the device watchdog location (section 3.2.5 of the netX DPM Interface Manual [1]) to the host watchdog location (section 3.2.4 of the netX DPM Interface Manual [1]), the protocol stack assumes that the host system has some sort of problem and shuts down all network connections. For details on the watchdog function, refer to section 4.13 of the netX DPM Interface Manual [1].

Error Count (All Implementations)

This field holds the total number of errors detected since power-up, respectively after reset. The protocol stack counts all sorts of errors in this field no matter if they were network related or caused internally.

Error Log Indicator (All Implementations)

Not supported yet: The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero.

3.3.1.2 Master Implementation

In addition to the common status block as outlined in the previous section, a master firmware maintains the following structure.

Master Status Structure Definition

```
typedef struct NETX_MASTER_STATUS_Ttag
{
  UINT32 ulSlaveState;
  UINT32 ulSlaveErrLogInd;
  UINT32 ulNumOfConfigSlaves;
  UINT32 ulNumOfActiveSlaves;
  UINT32 ulNumOfDiagSlaves;
  UINT32 ulReserved;
} NETX_MASTER_STATUS_T;
```

Master Status			
Offset	**Type**	**Name**	**Description**
0x0010	Structure	See common structure in table *Common Status Block*	
0x0038	UINT32	`ulSlaveState`	Slave State OK, FAILED (At Least One Slave)
0x003C	UINT32	`ulSlaveErrLogInd`	Slave Error Log Indicator Slave Diagnosis Data Available: EMPTY, AVAILABLE
0x0040	UINT32	`ulNumOfConfigSlaves`	Configured Slaves Number of Configured Slaves On The Network
0x0044	UINT32	`ulNumOfActiveSlaves`	Active Slaves Number of Slaves Running Without Problems
0x0048	UINT32	`ulNumOfDiagSlaves`	Faulted Slaves Number of Slaves Reporting Diagnostic Issues
0x004C	UINT32	`ulReserved`	Reserved Set to 0

Table 18: Master Status Structure Definition

Slave State

The slave state field is available for master implementations only. It indicates whether the master is in cyclic data exchange to all configured slaves. In case there is at least one slave missing or if the slave has a diagnostic request pending, the status is set to *FAILED*. For protocols that support non-cyclic communication only, the slave state is set to *OK* as soon as a valid configuration is found.

Status and Error Codes		
Code (Symbolic Constant)	**Numerical Value**	**Meaning**
RCX_SLAVE_STATE_UNDEFINED	0x00000000	UNDEFINED
RCX_SLAVE_STATE_OK	0x00000001	OK
RCX_SLAVE_STATE_FAILED	0x00000002	FAILED (at least one slave)
Others are reserved		

Table 19: Status and Error Codes

Slave Error Log Indicator

The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero.

Note: This function is not yet supported.

Number of Configured Slaves

The firmware maintains a list of slaves to which the master has to open a connection. This list is derived from the configuration database created by SYCON.net or from configuration using DEVNET_FAL_CMD_DOWNLOAD_REQ, please see section 5.4.1. This field holds the number of configured slaves.

Number of Active Slaves

The firmware maintains a list of slaves to which the master has successfully opened a connection.

Ideally, the number of active slaves is equal to the number of configured slaves. For certain fieldbus systems it could be possible that the slave is shown as activated, but still has a problem in terms of a diagnostic issue. This field holds the number of active slaves.

Number of Faulted Slaves

If a slave encounters a problem, it can provide an indication of the new situation to the master in certain fieldbus systems. As long as those indications are pending and not serviced, the field holds a value unequal zero. If no more diagnostic information is pending, the field is set to zero.

3.3.1.3 Slave Implementation

The slave firmware uses only the common structure as outlined in section 3.2.5.1 of the Hilscher netX Dual-Port-Memory Manual [1].

3.3.2 Extended Status

The content of the channel specific extended status block is specific to the implementation. Depending on the protocol, a status area may or may not be present in the dual-port memory. It is always available in the default memory map (see section 3.2.1 of *netX Dual-Port Memory Manual [1]*).

Note: Have in mind, that all offsets mentioned in this section are relative to the beginning of the common status block, as the start offset of this block depends on the size and location of the preceding blocks.

netX Extended Status Field Definition Structure

```
typedef struct NETX_EXTENDED_STATE_FIELD_DEFINITION_Ttag
{
  UINT8    abExtendedStatus[172];    /* Default, protocol specific inform. area */
  NETX_EXTENDED_STATE_FIELD_T tExtStateField;  /* Extended status structures */
} NETX_EXTENDED_STATE_FIELD_DEFINITION_T;
```

Extended Status Block			
Offset	**Type**	**Name**	**Description**
0x0050	`UINT8[]` (the first 64 bytes correspond to `DN_FAL_EXT_DIAG_T`)	`abExtendedStatus[172]`	Area containing DeviceNet-related information. See *Table 21: Extended Status Block for DeviceNet-Master* below (first 64 bytes of `abExtendedStatus`)
0x0090	`UINT8[]`		Reserved area, currently unused, (rest of `abExtendedStatus`)
0x00FC	`Structure` `NETX_EXTENDED_STATE_FIELD_T`	`tExtStateField`	Structure to define Status Fields and their Properties. Status type and properties are specific to protocol implementation

Table 20: Extended Status Block

Note: Each offset is always related to the beginning of correspondent channel start.

The definition of the first structure remains specific to correspondent protocol and contains additional information about network status (i.e. flags, error counters, events etc.).

The second structure begins at offset 0x00FC and provides the definition of the up to 32 independent State Fields. These state fields can be defined to represent a kind of bit-list, byte-list etc. with up to 65535 entities. In this way a common access mechanism for different state definitions and quantities can be provided independent of protocol implementation.

For the DeviceNet Master protocol implementation, the first structure of the extended status area is structured as follows:

```
typedef struct DN_FAL_EXT_DIAG_Ttag
{
  TLR_UINT8      bGlobalBits;        /* Collective global status bits         */
  #define DN_FAL_GLB_BIT_CTRL 0x01   /* Faulty parameter                      */
  #define DN_FAL_GLB_BIT_ACLR 0x02   /* Auto clear error                      */
  #define DN_FAL_GLB_BIT_NEXC 0x04   /* At least one slave not exchange data  */
  #define DN_FAL_GLB_BIT_FAT  0x08   /* Fatal error                           */
  #define DN_FAL_GLB_BIT_EVE  0x10   /* Busevent error                        */
  #define DN_FAL_GLB_BIT_NRDY 0x20   /* Application is not ready              */
  #define DN_FAL_GLB_BIT_DMAC 0x40   /* Duplicate MAC ID detected             */
  #define DN_FAL_GLB_BIT_PDUP 0x80   /* Check duplicate MAC ID in process     */

  TLR_UINT8     bDNM_State;          /* Operate mode                          */
                                     /* DN_FAL_MODE_OFFLINE (0x00)            */
                                     /* DN_FAL_MODE_STOP    (0x40)            */
                                     /* DN_FAL_MODE_IDLE    (0x80)            */
                                     /* DN_FAL_MODE_RUN     (0xC0)            */

  struct {/* Pending device error                 */
    TLR_UINT8   bDevAddr;            /* Error device address                  */
    TLR_UINT8   bErrEvent;           /* Error number                          */
  } tError;

  TLR_UINT16    usBus_Err_Cnt;       /* Bus event counter                     */
  TLR_UINT16    usBus_Off_Cnt;       /* Bus timeout counter                   */
                                     /* Reserved bytes                        */
  TLR_UINT32    ulDNM_ExtraErr;      /* Extra error code for DNM              */
  TLR_UINT8     abReserved[4];

                                     /* Bit wise operate /diag state per slave*/
  TLR_UINT8     abDv_cfg_active[8];  /* Configured slave declared as active   */
  TLR_UINT8     abDv_cfg_inactive[8];/* Configured slave declared as inactive */
  TLR_UINT8     abDv_state_expl[8];  /* Established explicit connection       */
  TLR_UINT8     abDv_state_io[8];    /* Established I/O connection            */
  TLR_UINT8     abDvDiag[8];         /* Slaves with diagnostic                */
  TLR_UINT8     abDv_reserved[8];    /* Reserved field                        */
```

The meaning of these parameters is:

Extended Status Block for DeviceNet-Master			
Address	**Type**	**Name**	**Description**
0x50	unsigned char	bGlobalBits	Global bits (Bit field to show bus and master main errors)
0x51	unsigned char	bDNM_state	Master main state of the device system (see below)
0x52	unsigned char	bDevAddr	Faulty device address (0-63)
0x53	unsigned char	bErrEvent	Corresponding error specification
0x54	unsigned short	usBus_Error_Cnt	Number of detected low transmission qualities
0x56	unsigned short	usBus_Off_Cnt	Number of cancelled transmissions and CAN-chip reinitializations
0x58	unsigned char	ulDNM_ExtraErr	Extra error code for DeviceNet Master Contains current status of the server connection
0x5C	unsigned char	abReserved[4]	Reserved area
0x60			Bit-Ready, Cfg-Ready and diagnostic display of slave devices:
0x60	unsigned char	abDv_cfg_active[8]	Bit field which slave is declared as configured and is managed active on the bus (see the table below)
0x68	unsigned char	abDv_cfg_inactive[8]	Bit field which slave is declared as configured and is not active on the bus (see the table below)
0x70	unsigned char	abDv_state_expl[8]	Bit field which slave is active, means where an explicit connection is established(see table below)
0x78	unsigned char	abDv_State_io[8]	Bit field which slave is active, means where an IO connection is established (see the table below)
0x80	unsigned char	abDv_Diag[8]	See the table below
0x88	unsigned char	abDv_Reserved[8]	Reserved bit field

Table 21: Extended Status Block for DeviceNet-Master

The single items of the Extended Status Block for DeviceNet Master have the following meaning

- bGlobalBits/ Global Bits

  The global bits byte bGlobalBits is a bit field to indicate bus and master main errors.

  This bit field serves as a collective display of global notifications. Notified errors can either occur at the device itself or at the slaves. In order to distinguish the different errors the variable bDevAddr contains the error location (i.e. bus address), while the variable bErrEvent specifies the corresponding error number. If more than one error is determined, the error location will always show the lowest faulty bus address.

  In detail, these bits have the following meaning:

Bit	Error type	Explanation
0	DN_FAL_GLB_BIT_CTRL	CONTROL-ERROR: This error is caused by incorrect parameterization.
1	DN_FAL_GLB_BIT_ACLR	AUTO-CLEAR-ERROR: The device stopped the communication to all slaves and reached the auto-clear end state, i.e. the DeviceNet Master stopped the

		communication to all other devices too, if configured to proceed in such a manner. Please see 0.
2	DN_FAL_GLB_BIT_NEXC	NON-EXCHANGE-ERROR: At least one slave has not reached the data exchange state and no process data are exchanged with it.
3	DN_FAL_GLB_BIT_FAT	FATAL-ERROR: Because of a severe bus error, no bus communication is possible any more
4	DN_FAL_GLB_BIT_EVE	Bus EVENT ERROR: A bus error occurred.
5	DN_FAL_GLB_BIT_NRDY	Host-NOT-READY-NOTIFICATION: Indicates whether the host program has set its state to operative or not. If the bit is set the host program is not ready to communicate
6	DN_FAL_GLB_BIT_DMAC	Duplicate MAC ID A duplicate MAC ID has been detected
7	DN_FAL_GLB_BIT_PDUP	Process DUPlicate MAC ID Check duplicate MAC ID in process

Table 22: Error Types in Global Bits

- bDNM_state/ DeviceNet Master main state

  The master main state represents the operation mode of the DeviceNet Master stack. This operation mode is defined in section 5.5.2 "Set Operation Mode Service DEVNET_FAL_CMD_SET_MODE_REQ/CNF" of this document.

  Allowed operation modes are:

Operation mode	Value
DN_FAL_MODE_OFFLINE	0x00
DN_FAL_MODE_STOP	0x40
DN_FAL_MODE_IDLE	0x80
DN_FAL_MODE_RUN	0xC0

Table 23: Operation Modes of the DeviceNet Master and their Values

  If you want to change this state, you can accomplish this by sending a Set Operation Mode Service DEVNET_FAL_CMD_SET_MODE_REQ/CNF packet to the DevNet FAL - task, for more information about this topic see section 5.5.2 of this document.

- bDevAddr / Location of error

  The bDevAddr parameter together with the bErrEvent parameter constitutes the error description.

  bDevAddr contains the station address of the station, where an error occurred within the network. When an error occurs on master, the bDevAddr field will contain the MAC ID of the master.

- bErrEvent/ Error code

The other component of the error description, the variable bErrEvent delivers the corresponding error number next to the error source. All possible numbers are listed below at the end of this subsection.

- usBus_Error_Cnt/ Counter for the bus error events

  This variable represents a counter for detected low transmission qualities and other severe bus errors such as short circuits on the bus, etc…

- usBus_Off_Cnt/ Counter for bus timeouts

  This variable represents a counter for the number of rejected DeviceNet telegrams because of severe errors (Number of cancelled transmissions and CAN-chip reinitializations).

- ulDNM_ExtraErr

  This variable contains extra error information, i.e. the last occurred error code of the DeviceNet Master protocol stack. Most common values for are TLR_E_CONNECTION_TIMEOUT, TLR_E_WATCHDOG_TIMEOUT, TLR_E_APPLICATION_NOT_READY, TLR_E_DUPLICATE_NODE, TLR_E_APPLICATION_FALSE and TLR_E_BUS_OFF.

- abReserved[4]/ Reserved area

  This data block is reserved.

- abDv_cfg_active[8]/ Bit field for configured and active slaves (configuration area)

  This variable is a field of 8 bytes and contains the information which slave devices are currently configured and active, too. The following table shows the assignment of the bits to the corresponding slave station addresses (MAC ID):

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
0x60	7	6	5	4	3	2	1	0
0x61	15	14	13	12	11	10	9	8
0x62	23	22	21	20	19	18	17	16
…								
0x67	63	62	61	60	59	58	57	56

Table 24: Relationship between Slave Device MAC ID and the corresponding abDv_cfg_active[8] Bit

If the abDv_cfg_active bit of the corresponding slave is logically

'1', the slave is configured in the master, and currently active.

'0', the slave is not configured in the master or currently not active.

- abDv_cfg_inactive[8]/ Bit field for configured and inactive slaves (configuration area)

  This variable is a field of 8 bytes and contains the information which slave devices are currently configured, but also inactive. The following table shows the assignment of the bits to the corresponding slave station addresses (MAC ID):

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
0x68	7	6	5	4	3	2	1	0

0x69	15	14	13	12	11	10	9	8
0x6A	23	22	21	20	19	18	17	16
…								
0x6F	63	62	61	60	59	58	57	56

Table 25: Relationship between Slave Device MAC ID and the corresponding abDv_cfg_inactive[8] Bit

If the abDv_cfg_inactive bit of the corresponding slave is logically

'1', the slave is configured in the master, and currently inactive.

'0', the slave is not configured in the master or currently active

- abDv_state_expl[8]/ Bit field for active slave devices where an explicit connection is established

  This variable is a field of 8 bytes and contains the information which slave devices are currently active and have established an explicit connection, too. The following table shows, which bit is related to which slave device MAC ID:

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
0x70	7	6	5	4	3	2	1	0
0x71	15	14	13	12	11	10	9	8
0x72	23	22	21	20	19	18	17	16
…								
0x77	63	62	61	60	59	58	57	56

Table 26: Relationship between Slave Device MAC ID and the corresponding abDv_state_expl[8] Bit

If the abDv_state_expl bit of the corresponding slave is logically

'1', the slave has established an explicit connection.

'0', the slave has not established an explicit connection.

- abDv_State_io[8]/ Bit field for active slave devices where an IO connection is established

  This variable is a field of 8 bytes and contains the information which slave devices are currently active and have established an IO connection, too. The following table shows the assignment of the bits to the corresponding slave station addresses:

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
0x78	7	6	5	4	3	2	1	0
0x79	15	14	13	12	11	10	9	8
0x7A	23	22	21	20	19	18	17	16
…								
0x7F	63	62	61	60	59	58	57	56

Table 27: Relationship between Slave Device MAC ID and the corresponding abDv_State_io[8] Bit

If the abDv_State_io bit of the corresponding slave is logically

'1', the slave has established an explicit connection.

'0', the slave has not established an explicit connection.

- abDv_diag[8] / Slave diagnostic area

This variable is a field of 8 bytes containing the diagnostic bit of each slave. The following table shows the relationship between the device address (MAC ID) of the slave and the corresponding bit in the variable abDv_diag.

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
0x80	7	6	5	4	3	2	1	0
0x81	15	14	13	12	11	10	9	8
0x82	23	22	21	20	19	18	17	16
…								
0x87	63	62	61	60	59	58	57	56

Table 28: Relationship between Slave Device MAC ID and the corresponding abDv_diag Bit

If the abDv_diag[8] bit of the corresponding slave is logically

'1', newly received diagnostic values are available in the internal diagnostic buffer or one of the diagnostics bit of the device has changed. This data can be read out by the HOST with a message which is described in the chapter 'The message interface' in this manual.

'0', since the last diagnostic buffer read access of the host, no values have been changed in this buffer.

The values in variable abDv_diag are only valid, if the device runs in the main state *OPERATE*.

	abDv_State_io = 0	abDv_State_io = 1
abDv_diag = 0	▪ The device is not operative, no process data exchange between Hilscher master device and slave takes place. ▪ Perhaps this slave is not configured or not able to respond.	▪ Slave device is present on the bus, device guarding is active. ▪ DataIOExchange between Hilscher master device and slave device happens as configured
abDv_diag = 1	▪ The slave device is not operating, device guarding failed or a configuration fault has been detected.. ▪ The Hilscher master device provides new diagnostic data in the internal diagnostic buffer to be read out by the host.	▪ The slave device is present on the bus. Device guarding is active and the Hilscher master device and the corresponding slave device exchange their I/O data. ▪ The Hilscher master device provides new diagnostic data in the internal diagnostic buffer to be read out by the host.

Table 29: Relationship between abDv_State_IO bit and abDv_Diag bit

- abDv_reserved[8] / Reserved field

  This field is reserved. It has a length of 8-bits.

The following error numbers are valid for `bErr_Event`, if the error occurred at the DeviceNet Master:

Possible Network Errors

`bErr_Event`	signification	error source
52	Unknown process data handshake mode configured	Configuration
53	Baudrate out of range	Configuration
54	DEVICE MAC-ID address out of range	Configuration
57	Duplicate DEVICE MAC-ID address detected in the network	Configuration or network
58	No device entry found in the current configuration database	Download error in the current data base, please contact technical support
210	No database found on the system	Configuration not downloaded, DEVICE is not configured by SyCon.
212	Failure in reading the database	Contact technical support.
220	User watchdog failed	Application
221	No data acknowledge from user	Application
223	DeviceNet Master has stopped bus communication because of CAN based bus off error.	Network error
226	Master firmware downloaded to slave device	User Error

Table 30: Errors which may occur in the Network (`bErr_device_adr` is equal to 255)

The following error numbers are valid for bErr_Event, if the error occurred at a DeviceNet Slave:

bErr_Event	signification	error source	Remarks/help
0	No error		
1	Missing node	Device	Device guarding failed, after device was operational. Check if device is still running
2	Device has been stopped	Device	Try to restart the device
30	Device access timeout	Device	The slave device does not respond, check the baudrate and its MAC-ID
(31)	No response from device	Device	The slave device does not response at all.
32	Device rejects access with unknown error code	Device	Use single device diagnostic to get reject code
35	Device response in allocation phase with connection error	Device	Use single device diagnostic to get additional reject code
36	Produced connection (process data input length in the view of the Hilscher master device) is different to the configured one	Device / configuration	Use single device diagnostic to get real produced connection size
37	Consumed connection (process data output length in the view of the Hilscher master) size is different to the configured one	Device / configuration	Use single device diagnostic to get real consumed connection size.
38	Device service response telegram unknown and not handled	Device / Hilscher master device	Use single device diagnostic to get real consumed connection size
39	Connection already in request	Device	The connection will be released automatically.
40	Number of CAN-message data bytes in read produced or consumed connection size response unequal 4	Device	Device does not have operability with Hilscher master device or it conflicts with the DeviceNet specification.
41	Predefined master slave connection already exists	Device / Hilscher master device	The connection will be released automatically
42	Length in polling device response unequal produced connection size	Device	
43	Sequence error in device polling response	Device	First two segments in multiplexed transfer were received
44	Fragment error in device polling response	Device	Fragmentation counter while multiplexed transfer differs from the awaited one
45	Sequence error in device polling response	Device	Middle or last segment was received before the first segment
46	Length in bit strobe device response unequal produced connection size	Device	

47	Length in COS or cyclic device response unequal produced connection size	Device	
48	Sequence error in device COS or cyclic response	Device	Middle or last segment was received before the first segment
49	Fragment error in device COS or cyclic response	Device	Fragmentation counter while multiplexed transfer differs from the awaited one
50	Sequence error in device COS or cyclic response	Device	First two segments in multiplexed transfer were received
51	UCMM group not supported	Device	Change the UCMM group
52	Device Keying failed: Vendor ID mismatch	Device / configuration	Check configured Vendor ID against devices Vendor ID
53	Device Keying failed: Device Type mismatch	Device / configuration	Check configured Device Type against devices Device Type
54	Device Keying failed: Product Code mismatch	Device / configuration	Check configured Product Code against devices Product Code
55	Device Keying failed: Revision mismatch	Device / configuration	Check configured Revision against devices Revision
59	Double device address configured in actual configuration	Configuration	Each device at DeviceNet must have its own MAC-ID
60	Whole size indicator of one device data set is corrupt	Configuration	Download error in the current data base, contact technical support
61	Size of the additional table for predefined master slave connections is corrupt	Configuration	Download error in the current data base, contact technical support
62	Size of predefined master slave I/O configuration table is corrupt	Configuration	Download error in the current data base, contact technical support
63	Predefined master slave I/O configuration does not correspond to the additional table	Configuration	Number of I/O modules and the number of configured offset addresses are different
64	Size indicator of parameter data table corrupt	Configuration	Value of size indicator to small
65	Number of inputs declared in the additional table does not correspond to the number in the I/O configuration table	Configuration	Each entry in the I/O configuration must have only one entry in the additional table
66	Number of outputs declared in the additional table does not correspond to the number in the I/O configuration table	Configuration	Each entry in the I/O configuration must have only one entry in the additional table
67	Unknown data type in I/O configuration detected	Configuration	Support BOOLEAN,BYTE,WO RD,DWORD and STRING only
68	Data type of a defined I/O module in a connection does not correspond with the defined data size	Configuration	Following type and size are valid BOOLEAN = 1 byte UINT8 = 1 byte UINT16 = 2byte UINT32 = 4byte
69	Configured output address of one module oversteps the	Configuration	The process data image is limited to 3584 bytes

	possible address range of 3584 bytes		
70	Configured input address of one module oversteps the possible address range of 3536 bytes	Configuration	The process data image is limited to 3536 bytes
71	One predefined connection type is unknown	Configuration	Support of cyclic, polled, change of state, bit strobed only
72	Multiple connections defined in parallel	Configuration	Supporting only one type of connection to one device
73	The configured `Exp_Packet_Rate` value of one connection is less than the `Prod_Inhibit_Time` value	Configuration	Expected packet rate must be larger than the production inhibit time

Table 31: Errors which may occur in the DeviceNet Master Device (`bErr_Rem_Adr` is not equal to 255)

Besides the global bits, master main state, error informations and counters, additional status bit lists of slaves are defined in this structure (namely slave configuration area, slave state information area, slave diagnostic area). These bit lists contain the current state information of all slave devices the master communicates with (i.e. 8 bytes = 64 devices). Despite the fact that the implementation of extended status block is protocol specific, the place and definition of these bit lists are to a greater or lesser extent similar for all Hilscher Fieldbus Master protocol stacks. The layout of this block is still maintained with actual specification and will be supported further. The example below shows a generic way to define the corresponding location of the bit lists located at the offsets 0x60, 0x68, 0x70, 0x78 and 0x80 (see table above). Three state structures are needed to be defined to locate such bit lists i.e. inside of input data block.

Extended Status Block for DeviceNet-Master – Second part (State Field Definition Block)			
Offset	Type	Name	Description/Value
0x00FC	`unsigned char`	`bReserved[3]`	Reserved. Do not use.
0x00FF	`unsigned char`	`bNumStateStructs`	Number of State Structures defined below = 3
↓	**`NETX_EXTENDED_STATE_STRUCT_T`**	**`atStateStruct[0]`**	**Structure to define State field properties**
0x0100	`unsigned char`	`bStateArea`	=0. State field is located in standard input area of channel 0
	`unsigned char`	`bStateTypeID`	=1. Corresponds to a bit list (one bit per node) of configured nodes
	`unsigned short`	`usNumOfStateEntries`	=64. Corresponds to 64 bits, each representing a slave
	`unsigned long`	`ulStateOffset`	Contains an offset pointer to a state field inside input data area 0, which contains the bit field for configured and active slaves. See description of the bit field for configured and active slaves above.
↓	**`NETX_EXTENDED_STATE_STRUCT_T`**	**`atStateStruct[1]`**	**Structure to define State field properties**
0x0108	`unsigned char`	`bStateArea`	=0. State field is located in standard input area of channel 0
	`unsigned char`	`bStateTypeID`	=2. Corresponds to a bit list (one bit per node) of active nodes
	`unsigned short`	`usNumOfStateEntries`	=64. Corresponds to 64 bits, each representing a slave

	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the bit field for configured and inactive slaves. See description of the bit field for configured and inactive slaves above.
↓	**NETX_EXTENDED_STATE_STRUCT_T**	**atStateStruct[2]**	**Structure to define State field properties**
0x0110	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=3. Corresponds to a bit list (one bit per node) of diagnostic nodes
	unsigned short	usNumOfStateEntries	=63. Corresponds to 63 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the bit field for active slave devices where an explicit connection is established. See description of the bit field for active slave devices where an explicit connection is established above
↓	**NETX_EXTENDED_STATE_STRUCT_T**	**atStateStruct[3]**	**Structure to define State field properties**
0x0118	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=4. Corresponds to a bit list (one bit per node) of configured nodes
	unsigned short	usNumOfStateEntries	=63. Corresponds to 63 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the bit field for active slave devices where an IO connection is established See description of the bit field for active slave devices where an IO connection is established above.
↓	**NETX_EXTENDED_STATE_STRUCT_T**	**atStateStruct[4]**	**Structure to define State field properties**
0x0120	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=5. Corresponds to a bit list (one bit per node) of configured nodes
	unsigned short	usNumOfStateEntries	=63. Corresponds to 63 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the slave diagnostic area. See description of the slave diagnostic area above

Table 32: Extended Status Block for DeviceNet-Master – Second part (State Field Definition Block)

If the location of the state fields is defined to be inside of input data area 0 block (as it is shown in generic example above), the corresponding bitlists will be updated by the stack consistently to the data in this area. Moreover, the data and corresponding state fields can be read out by the host application as one data block i.e. with DMA support.

3.4 Control Block

A control block is always present within the communication channel. In some respects, control and status block are used together in order to exchange information between host application and netX firmware. The control block is written by the application, whereas the application reads a status block. Both control and status block have registers that use the *Change of State* mechanism (see also section 2.2.1 of the *netX Dual-Port-Memory manual, reference* [1]).

The following gives an example of the use of control and status block. The host application wishes to lock the configuration settings of a communication channel to protect them against changes. The application sets the *Lock Configuration* flag in the control block to the communication channel firmware. As a result, the channel firmware sets the *Configuration Locked* flag in the status block (see below), indicating that the current configuration settings cannot be deleted, altered, overwritten or otherwise changed.

The control block of a dual-port memory features a watchdog function to allow the operating system running on the netX supervise the host application and vice versa. The control area is always present in the dual-port memory.

Control Block			
Offset	**Type**	**Name**	**Description**
0x0008	UINT32	`ulApplicationCOS`	Application Change Of State State Of The Application Program INITIALIZATION, LOCK CONFIGURATION
0x000C	UINT32	`ulDeviceWatchdog`	Device Watchdog Host System Writes, Protocol Stack Reads

Table 33: Communication Control Block

Communication Control Block Structure

```
typedef struct NETX_CONTROL_BLOCK_Ttag
{
UINT32 ulApplicationCOS;
UINT32 ulDeviceWatchdog;
} NETX_CONTROL_BLOCK_T;
```

For more information concerning the Control Block please refer to the netX DPM Interface Manual [1].

4 Getting started / Configuration

This section explains some essential information you should know when starting to work with the DeviceNet Master Protocol API.

4.1 Overview about Essential Functionality

You can find the most commonly used functionality of the DeviceNet Master Protocol API within the following sections of this document:

Topic	Section Number	Section Name
Configuration	5.4.1	Download Configuration DEVNET_FAL_CMD_DOWNLOAD_REQ/CNF
	5.4.2	Clear Configuration Service DEVNET_FAL_CMD_CLR_CONFIG_REQ/CNF
Controlling/Monitoring/Diagnosis of the Stack Task	5.5.1	Stack Initialization Service DEVNET_FAL_CMD_INIT_REQ/CNF
	5.5.2	Set Operation Mode Service DEVNET_FAL_CMD_SET_MODE_REQ/CNF
	5.5.3	Parameter Upload Service DEVNET_FAL_CMD_UPLOAD_REQ/CNF
	5.5.4	Device Diagnosis Service DEVNET_FAL_CMD_DEV_DIAG_REQ/CNF
	5.5.5	DEVNET_FAL_CMD_FAULT_IND/RES – Indication of a Fault
	5.5.6	Operation Mode Indication DEVNET_FAL_CMD_SET_MODE_IND/RES
	5.5.7	DEVNET_FAL_CMD_SET_LED_IND/RES – Set LED Indication
Input/Output Data Services	5.6.1	Acyclic Bit-Strobing Service DEVNET_FAL_CMD_ACYC_BTS_REQ/CNF
	5.6.2	Acyclic Poll Service DEVNET_FAL_CMD_ACYC_POLL_REQ/CNF
	5.6.3	New Output Indication DEVNET_FAL_CMD_NEW_OUTPUT_IND/RES
Explicit Message Services	5.7.1	Get Attribute Service DEVNET_FAL_CMD_GET_ATT_REQ/CNF
	5.7.2	Set Attribute Service DEVNET_FAL_CMD_SET_ATT_REQ/CNF
	5.7.3	DEVNET_FAL_CMD_REMOTE_SERVICE_REQ/CNF – Remote Service
	5.7.4	Local Service DEVNET_FAL_CMD_LOCAL_SERVICE_REQ/CNF
Raw CAN Frame Service	5.8.1	CAN Registered Service DEVNET_FAL_CMD_CAN_FWD_REG_REQ/CNF
	5.8.2	CAN Forward Service DEVNET_FAL_CMD_CAN_FWD_IND/RES
	5.8.3	DEVNET_FAL_CMD_CAN_DATA_REQ/CNF - CAN Data Request
Bus Diagnosis Service	5.9.1	Get Lifelist Service DEVNET_FAL_CMD_LIFELIST_REQ/CNF
Application Register Service	5.10.1	Register Application Service DEVNET_FAL_CMD_AP_REGISTER_REQ/CNF

Table 34: Overview about Essential Functionality (Cyclic and acyclic Data Transfer and Alarm Handling).

4.2 Object Modeling

The device is modeled as a collection of objects. Object modeling organizes related data and procedures into one entity: the object. An object is a collection of related services and attributes. Services are procedures an object performs. Attributes are characteristics of objects represented by values or variables. Typically, attributes provide status information or govern the operation of an object. An object's behavior is an indication of how the object responds to particular events.

The following objects are present in the device and available from the link. The application is free to define device specific objects and register them with the message router.

For details refer to the DeviceNet specification.

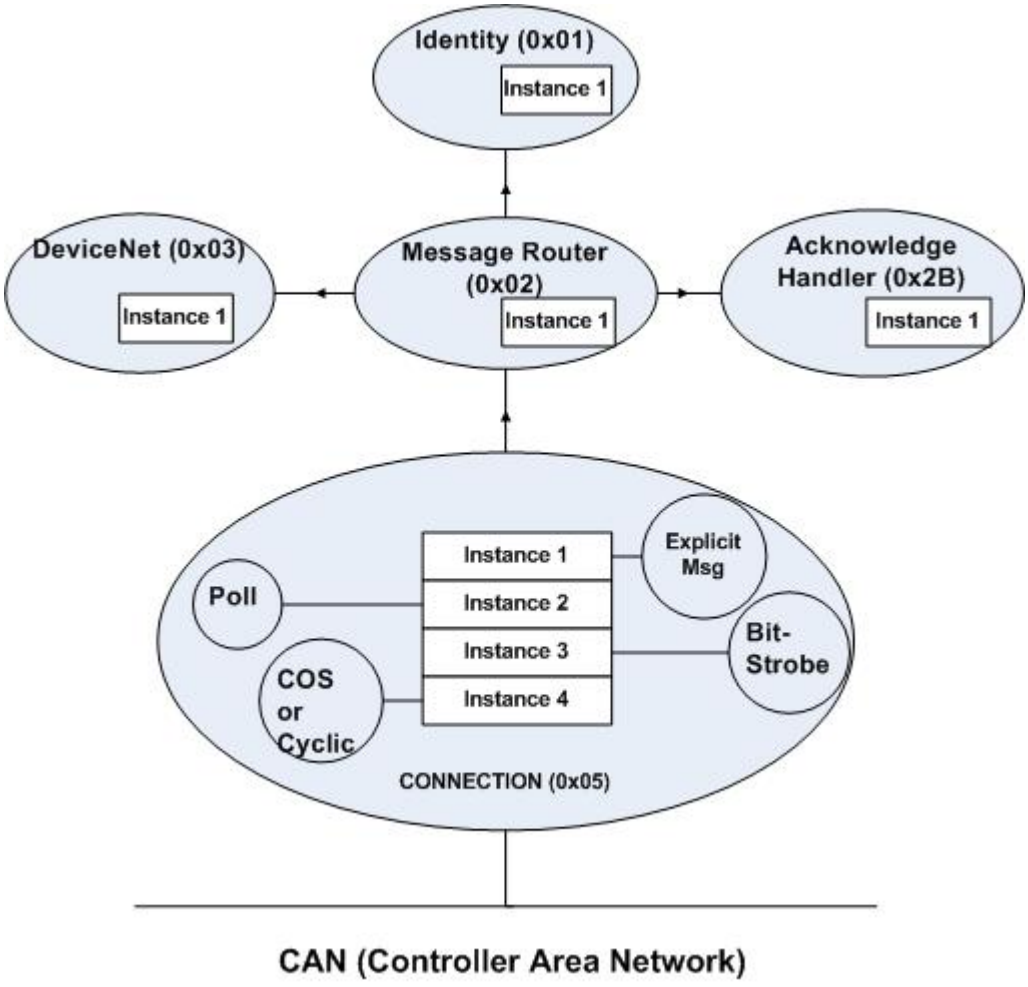

Figure 6: Objects Model of Hilscher DeviceNet Master stack

4.2.1 Identity Object (Class Code: 0x01)

The Identity Object provides identification and general information about the device. The first instance identifies the whole device. It is used for electronic keying and by applications wishing to determine what nodes are on the network.

Supported Features

Instance	Name	Attribute ID	Name	Supported Service
0	Class			Not Supported
1	Instance Attributes	1	Vendor ID	Get Attribute Single
		2	Device Type	Get Attribute All Reset
		3	Product Code	
		4	Major Revision	
			Minor Revision	
		5	Status	
		6	Serial Number	
		7	Product Name	

Table 35: Identity Object Supported Features

4.2.2 Message Router Object (Class Code: 0x02)

The Message Router Object provides a messaging connection point through which a client may address a service to any object class or instance residing in the physical device.

Supported Features

There are no services supported by the Message Router Object.

4.2.3 DeviceNet Object (Class Code: 0x03)

The DeviceNet Object contains information about the configured DeviceNet Slave. Examples of this information include MAC ID, Baud rate, etc. as shown below. This object only supports Get Attribute Single and Set Attribute Single.

Supported Features

Instance	Name	Attribute ID	Name	Supported Service
0	Class	1	Revision	Get Attribute Single
1	Instance Attributes	1	MAC ID	Get Attribute Single Set Attribute Single
		2	Baudrate	
		3	Bus Off Interrupts	
		4	Bus Off Counter	
		5	Object Allocation Information	

Table 36: DeviceNet Object Supported Features

4.2.4 Connection Object (Class Code: 0x05)

The Connection Object consists of multiple objects which provide information about the current connection status. Each instance relates to a different connection type, for example Explicit Messaging connection, Bit Strobe, Polled, and Change of State, Cyclic. Please refer to Figure 6: Objects Model of Hilscher DeviceNet Master stack.

Supported Features

Instance	Name	Attribute ID	Name	Supported Service
0	Class	1	Revision	Not Supported
1	Instance Attributes Explicit Messaging Connection	1	Connection State	Get Attribute Single Set Attribute Single
		2	Connection Type	
		3	Transport Type	
		4	Produced Connection ID	
		5	Consumed Connection ID	
		6	Initial Com Characteristics	
		7	Produced Connection Size	
		8	Consumed Connection Size	
		9	Expected Packet Rate	
		12	Timeout Action	
		13	Produced Path Length	
		14	Produced Connection Path ID	
		15	Consumed Path Length	
		16	Consumed Connection Path ID	
		17	Inhibit Time	
2	Polled Connection	1- 17	As Above.	
3	Bit Strobe Connection	1 - 17	As Above.	
4	Change of State Connection	1 - 17	As Above.	

Table 37: Connection Object Supported Features

4.2.5 Acknowledge Handler Object (Class Code: 0x2B)

The Acknowledge Handler Object is responsible for handling acknowledge response messages. This object supports both Get and Set Attribute Single. Attributes supported are shown below.

Supported Features

Instance	Name	Attribute ID	Name	Supported Service
0	Class			Not Supported
1	Instance Attributes	1	Acknowledge Timer	Get Attribute Single Set Attribute Single
		2	Acknowledge Handler Retry Limit	
		3	COS Produced ID	

Table 38: Acknowledge Handler Object Supported Features

4.3 Configuration of Bus, Slave, Server and Device Parameters

This section explains how to configure the bus, the master and the slaves correctly. This can be done either by sending packets writing the parameters via dual-port memory mailbox or by using Hilscher's configuration tool SYCON.net. The section introduces both methods and contains a detailed description of all parameters.

4.3.1 Write Access to the Dual-Port Memory

In order to change the configuration parameters for the master in the dual-port memory, a `DN_FAL_PACKET_DOWNLOAD_REQ` packet has to be sent to the protocol stack. For more information how to accomplish this, please refer to section 5.4.1 of this manual

Similarly, in order to check the currently active parameters, the transfer of parameters to the application task can also be performed. Please see section 5.5.3 "Parameter Upload Service DEVNET_FAL_CMD_UPLOAD_REQ/CNF" for more information about the `DN_FAL_PACKET_UPLOAD_REQ` packet.

4.3.2 Using the configuration tool SYCON.net

The easiest way to configure the DeviceNet Master is using Hilscher's configuration tool SYCON.net.

The configuration tool is described in a separate manual [1]. See there for more information.

- First, you need to create a project in SYCON.net. This is described in detail in the SYCON.net documentation [1].
- Configure the bus and master parameters as described in the SYCON.net documentation.
- After you completed your project, you can right-click on the icon of the DeviceNet Master and select "*Connect*".
- You will see that the name of the DeviceNet Master will get a green background. Now right-click on the icon again and select "*Download*".
- This will download the configuration file to the device. It is stored in a RAM Disk in a channel dependent directory (`"PORT_0"` for channel 0, `"PORT_1"` for channel 1, etc.).
- After the download is finished, the driver requests the DeviceNet Master firmware to perform a Channel-Init. All current connections will be shut down by the firmware and a restart will be performed.
- During this restart, the configuration that has been downloaded previously will be evaluated and used.

4.4 Configuration Using the Packet API

In section 4.2 “*Object Modeling*” the default Hilscher CIP Object Model is described. This section explains how these objects can be configured using the Packet API of the DeviceNet Master stack.

In order to determine what packets you should use first you need to select one of the following scenarios the DeviceNet Master Protocol stack can be run with.

- **Scenario: Loadable Firmware (LFW)**

  The host application and the DeviceNet Master Protocol Stack run on different processors. While the host application runs on a separate CPU the DeviceNet Master Protocol Stack runs on the netX processor together with a connecting software layer, the AP task.

  The connection of host application and Protocol Stack is accomplished via a driver (Hilscher cifX Driver, Hilscher netX Driver) as software layer on the host side and the AP task as software layer on the netX side. Both communicate via a dual port memory. This situation corresponds to alternative 1 and 2 in the introduction of section „General Access Mechanisms on netX Systems“. For alternative 1 this situation is illustrated in *Figure 7: Loadable Firmware* Scenario:

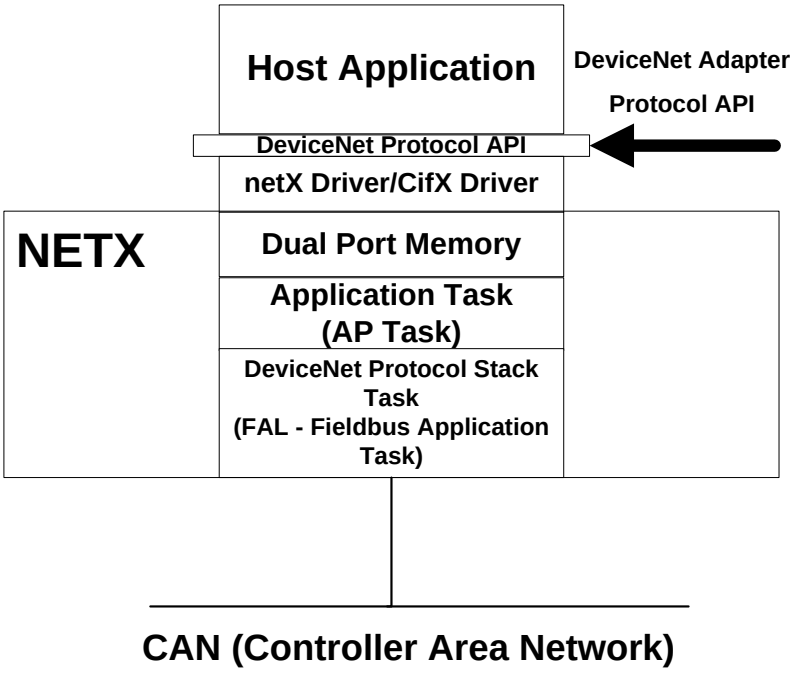

Figure 7: Loadable Firmware Scenario

- **Scenario: Linkable Object Module (LOM)**

  Both the host application and the DeviceNet Master Protocol Stack run on the same processor, the netX. There is no need for drivers or a stack-specific AP task. Application and Protocol Stack are statically linked. This situation corresponds to alternative 3 in the introduction of section 2.1 *“General Access Mechanisms on netX Systems* ”. It is illustrated in Figure 8:

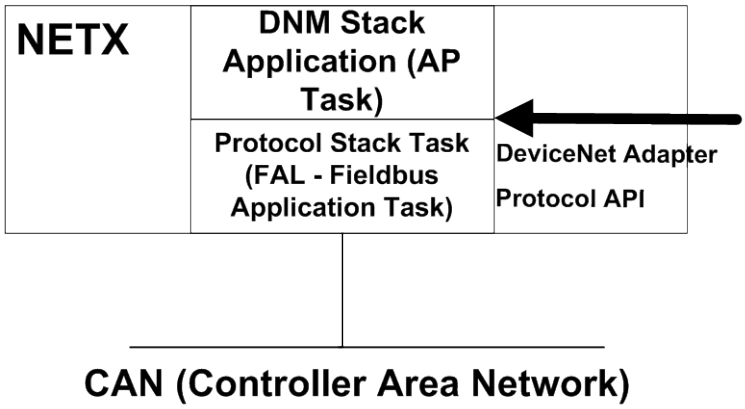

Figure 8: Linkable Object Modules Scenario

After making the scenario decision there are some Packet Sets available. The Packet Set must be chosen depending on the requirements for the device you want to develop and on the CIP Object Model you want the device to have.

Table 39: Packet Sets shows the available sets and describes the general functionalities that come with the corresponding set.

Scenario	Name of Packet Set	Description
Loadable Firmware	**Basic** (Loadable Firmware)	This set provides a basic functionality. The user should use this type of configuration if the application to be developed is a host application. The stack application runs in the context of netX.
Linkable object module	**Stack** (see section 4.4.2 “Extended Packet Set” for a detailed packet list)	This Configuration Set corresponds basically to the Extended Configuration Set of the Loadable Firmware. There are only some differences in the packet handling, which in turn depends of the configuration.

Table 39: Packet Sets

4.4.1 Basic Packet Set

To configure the DeviceNet Master Stack the following packets are necessary:

No. of section	Packet Name	Command Code (REQ/CNF)
5.4.1	Download Configuration DEVNET_FAL_CMD_DOWNLOAD_REQ/CNF	0x3802/ 0x3803
	RCX_START_STOP_COMM_REQ – Start or stop communication. (see reference [1] “DPM Manual” for more information)	0x2F30/ 0x2F31

Table 40: Basic Packet Set – Configuration Packets

The packets of packet set “Basic” should be sent in sequence (the next packet is sent after the confirmation of the previous packet received) as showed in *Figure 9: Configuration Sequence Using the Basic Packet* Set.

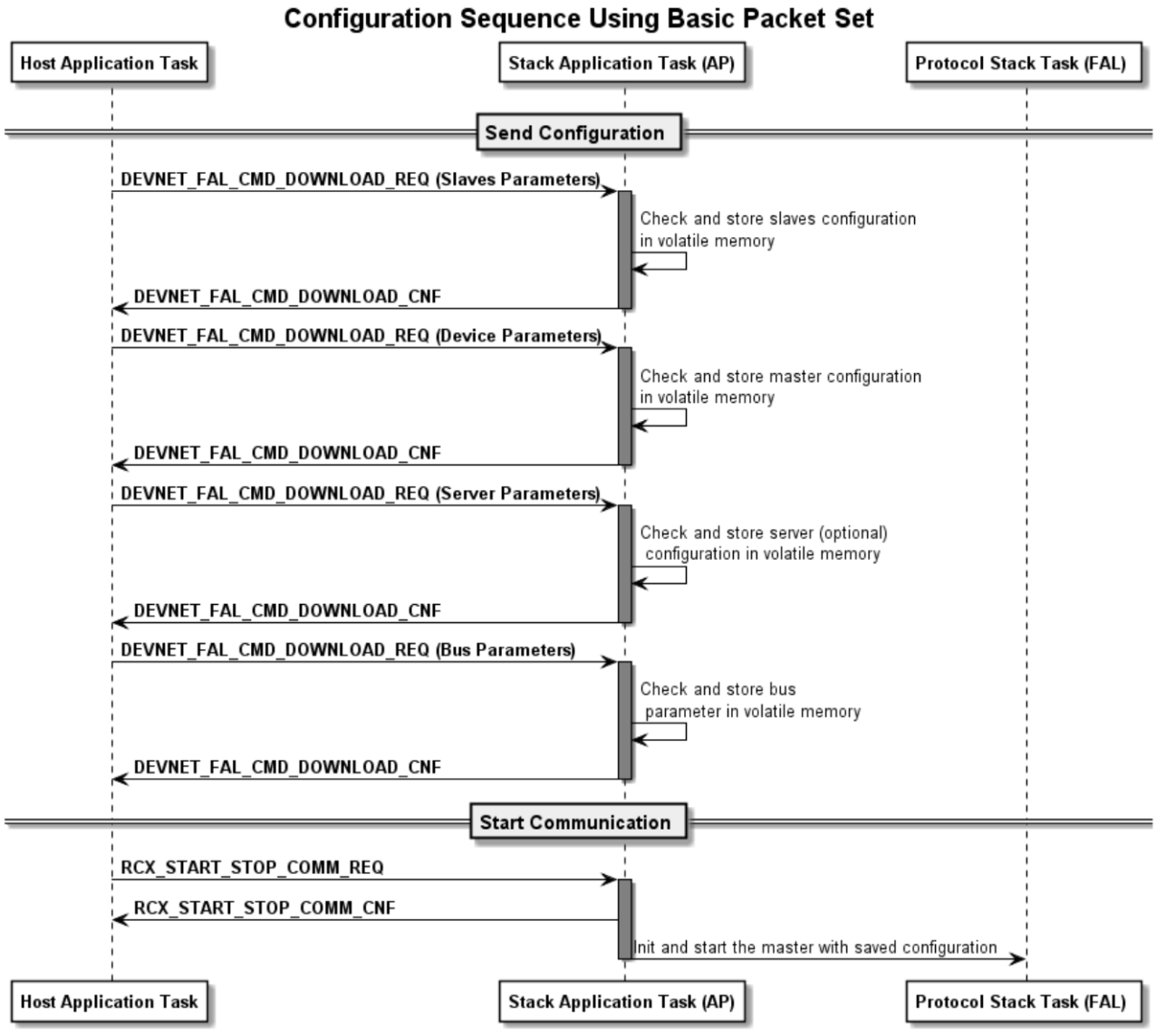

Figure 9: Configuration Sequence Using the Basic Packet Set

4.4.2 Extended Packet Set

To configure the DeviceNet Master Stack the following packets are necessary:

No. of section	Packet Name	Command Code (REQ/CNF)
5.5.2	Set Operation Mode Service DEVNET_FAL_CMD_SET_MODE_REQ/CNF	0x3804/ 0x3805
5.5.6	Operation Mode Indication DEVNET_FAL_CMD_SET_MODE_IND/RES	0x3818/ 0x3819
5.4.1	Download Configuration DEVNET_FAL_CMD_DOWNLOAD_REQ/CNF	0x3802/ 0x3803

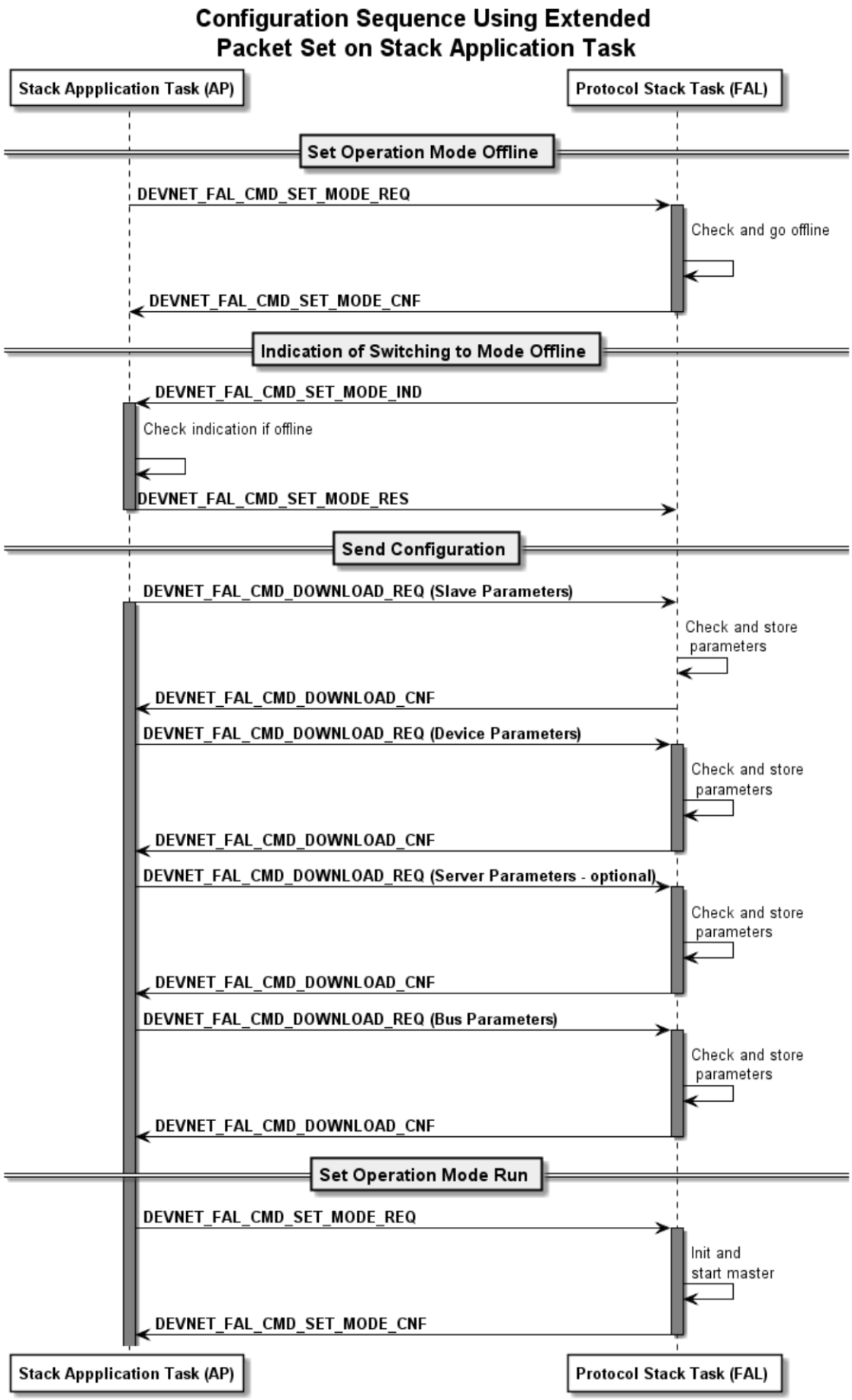

Figure 10: Configuration Sequence Using the Extended Packet Set

4.5 Detailed Description of DeviceNet Configuration Parameters

The configuration parameters of DeviceNet consist of four different Areas, containing the data for

- Bus parameters,
- Device Parameters,
- Slave Parameters
- Server Parameter (in case the master acts as server to other client node on bus).

4.5.1 Detailed Description of Bus Parameters

The following table informs about the relevant slave parameters for the DeviceNet Master firmware such as an explanation of the meaning of the parameter and ranges of allowed values:

Bus Parameters, Meanings and Ranges

Parameter	Meaning	Range of Values
Communication Auto Release	The start of the device can be performed either application controlled or automatically: ▪ *Automatic*: Network connections are opened automatically without taking care of the state of the host application. ▪ *Application controlled:* The channel firmware is forced to wait for the host application to set the Application Ready flag in the communication change of state register (see section 3.2.5.1 of the *netX DPM Interface Manual [1]*). For more information concerning this topic see section 4.4.1 “Controlled or Automatic Start” of the *netX DPM Interface Manual [1]*.	Application controlled, Automatic
I/O Status **(not yet supported)**	This parameter is represented by bits 1 and 2 of the system flags. Using this parameter you can set the status of the input or the output data. For each input and output date the following status information (in byte) is memorized in the dual-port memory. The bits have the following meaning: Bit 1 (I/O Status Enable): 0 = Status disabled 1 = Status enabled **(not yet supported)** Bit 2 (I/O Status 8/32Bit): 0 = 1 Byte mode **(not yet supported)** 1 = 4 Byte mode **(not yet supported)**	
Watchdog Time [ms]	Watchdog time within which the device watchdog must be retriggered from the application program while the application program monitoring is activated. When the watchdog time value is equal to 0 respectively the application program monitoring is deactivated.	[0 … 65535] ms, default = 1000 ms, 0 = Off
MAC ID	This parameter defines the DeviceNet Master-Task address of the node (Node ID or MAC ID) within the network.	Allowed values: 0 – 63
Baudrate	Baud rate of DeviceNet connection. Coding see Table 42 below!	Allowed values: all baud rates offered in list. Default value: 0

ConfigFlags	Configuration flags	Valid values: 0 – 31 Default value: 0
EnableFlags	Enable Flags. Not used.	Set to 0.
Reserved	There is a reserved area for future extensions	40 Bytes

Table 41: Slave Parameters, their Meanings and their Ranges of allowed Values

Watchdog Time

The watchdog time parameter specifies the time in multiples of 1 msec. the device has to supervise the host program if it has started the host-watchdog functionality once

Own MAC ID Parameter

The Own MAC ID parameter defines the master's DeviceNet address within the network. Valid choices are in the range from 0 to 63, other values are not allowed by the DeviceNet specification and therefore not supported by the device.

Baudrate

The applicable baud rates can be coded according to the values specified in the following table:

Symbolic Name	Baud rate	Value
DN_FAL_BAUDRATE_500	500 kBit/s	1
DN_FAL_BAUDRATE_250	250 kBit/s	2
DN_FAL_BAUDRATE_125	125 kBit/s	3

Table 42: Available Baud Rate Values

Bus Parameter Configuration Flags

The following table shows the meaning of the bus parameter configuration flags byte:

D7	D6	D5	D4	D3	D2	D1	D0	Description
							x	**DN_FAL_MSK_BUS_PRM_CFG_FLAG_AUTOCLEAR**
						x		**Reserved (not used)**
					x			**DN_FAL_MSK_BUS_PRM_CFG_FLAG_CONTINUE_ON_BOI**

Table 43: Meaning of Bus Parameter Configuration Flags Byte

If these bits are set to 1, the appropriate item will be activated, otherwise it will be deactivated.

The single bits of the flags variable have the following meaning:

Bit 0 Auto-Clear on/off

This bit defines the behavior of the DeviceNet Master if one slave device classified as active has been disconnected.

Value	Meaning
1	Active
0	Inactive

Table 44: Meaning of Auto Clear Bit

- Active means the DeviceNet Master stops the communication to all other devices too,
- Inactive means the DeviceNet Master tries to restart the disconnected device and keeps running.

Bit 1 This bit is reserved and currently unused. Set to 0.

Bit 2 Continue on Bus-off (`DN_FAL_MSK_BUS_PRM_CFG_FLAG_CONTINUE_ON_BOI`):

This flag defines the behavior of the DeviceNet Master in case a *'bus off'* indication from the CAN controller occurs. If the `DN_FAL_MSK_BUS_PRM_CFG_FLAG_CONTINUE_ON_BOI` flag is set to the value 1, the DeviceNet Master will continue its operation. Otherwise, the DeviceNet Master will stop running.

EnableFlags

The 'EnableFlags' of bus parameter is currently not used.

4.5.2 Detailed Description of Slave Parameters

Additionally to the configuration of the bus and the master, also the slaves connected with the master need to be configured.

For each of the up to 63 allowed slaves a data structure is held which allows the administration of each slave.

The details of this complex structure are discussed later in this document in section 5.4.1.5

4.5.3 Detailed Description of Device Parameters

Parameter	Meaning	Range of Values
ulConfigFlags	Configuration flags. Not used.	Allowed values: 0 – 0xFFFFFFFF Default value: 0
ulEnableFlags	Enable Flags	Allowed values: 0x00000000 - 0x0000007F
usVendorId	Vendor ID	Default 283 but changeable.
usProductType	Product Type	Default 12 but changeable.
usProductCode	Product Code	Allowed values: 0x0000 - 0xFFFF Default value: 0
bMinorRev	Minor Version	Allowed values: 0-255 Default value: 1
bMajorRev	Major Version	Allowed values: 0-255 Default value: 1
ulSerialNumber	Serial Number	Allowed values: 0x00000000 - 0xFFFFFFFF
abReserved[3]	Reserved	
bProductNameLen	Product Name length	Allowed values: 1-32
abProductName[32]	Product Name	ASCII Characters
abReserved2[8]	Reserved	

Table 45: Device Parameters

Device Parameter ConfigFlags

This flag word is currently reserved and has a fixed value of 0.

Device Parameter EnableFlags

The EnableFlags byte decides whether the following identity variables

- ProductType,
- ProductCode,
- MinorRev,
- MajorRev
- and the ProductName

are filled up with valid data or not.

The following table shows the meaning of the ‘EnableFlags’ byte:

D7	D6	D5	D4	D3	D2	D1	D0	Description
							x	DN_FAL_MSK_DEV_PRM_ENABLE_VENDORID
						x		DN_FAL_MSK_DEV_PRM_ENABLE_PRODUCTTYPE
					x			DN_FAL_MSK_DEV_PRM_ENABLE_PRODUCTCODE
				x				DN_FAL_MSK_DEV_PRM_ENABLE_MAJORMINORREV
			x					DN_FAL_MSK_DEV_PRM_ENABLE_SERIALNR
		x						DN_FAL_MSK_DEV_PRM_ENABLE_PRODUCTNAME
	x							DN_FAL_MSK_DEV_PRM_ENABLE_QUICK_CONNECT

Table 46: Meaning of EnableFlags Byte

If the corresponding bit is set to 1, the variable contains valid data and is enabled, i.e. the appropriate item (Vendor ID, product type, product code, minor and major revision together, serial number and product name) can be set by the user, otherwise these parameters will be filled with default value from protocol stack.

VendorId

The `VendorId` device parameter is a DeviceNet specific unique number which is fixed by the ODVA for each DeviceNet manufacturer. Please refer to the ODVA Licensed Vendor List on www.odva.org for further details. The DeviceNet Master-Task itself uses this ID during the Duplicate MAC-ID check phase and within each sent Duplicate MAC-Id check response. The value range of this variable is not limited. The Hilscher ID is 283 (decimal).

This value is related to the *Vendor ID Attribute* of the Identity Object, see section 4.2.1.

ProductType

The ProductType device parameter is the indication of the general type of product. The Hilscher standard value for this is 12 which represents a Communications Adapter.

This value is related to the *Device Type Attribute* of the Identity Object, see section 4.2.1.

ProductCode

The ProductCode device parameter is the identification of a particular product within a defined device type. This code is device specific and can be defined by the manufacturer.

This value is related to the *Product Code Attribute* of the Identity Object, see section 4.2.1

MinorRev

The MinorRev device parameter is one part of the revision which identifies the revision of the DNM device. The revision attribute consists of Major and Minor Revisions and they are typically displayed as major.minor.

This value is related to the *Minor Revision Attribute* of the Identity Object, see section 4.2.1

MajorRev

The MajorRev device parameter is the second part of the revision. The Major Revision attribute is limited to 7 bits. The eighth bit is reserved by DeviceNet and must have a default value of zero.

This value is related to the *Major Revision Attribute* of the Identity Object, see section 4.2.1.

SerialNumber

The SerialNumber device parameter is the eight digit serial number of the user created device. It should be unique.

This value is related to the *Serial Number Attribute* of the Identity Object, see section 4.2.1.

ProductNameLen

The ProductNameLen device parameter is the length of ProdName. The maximum number of characters in this string is 32 and would be never zero.

ProductName[32]

The ProdName device parameter is a text string that should represent a short description of the product/product family. The maximum number of characters in this string is 32.

The number of characters must be set in parameter ProdNameLen.

This value is related to the *Product Name Attribute* of the Identity Object, see section 4.2.1.

4.5.4 Detailed Description of Server Parameters

Additionally to its client functionality, a DeviceNet Master also offers server functionality for master-to-master communication. This server functionality also only needed to be configured if the user wants to access the server functionality of the master. It can be activated by setting at least one of the two parameters ***SrvConsConnSize*** and ***SrvProdConnSize*** to a non-zero value. Vice versa, setting both values to 0 inhibits the master-to-master communication.

The following table informs about the relevant server parameters for the DeviceNet Master firmware such as an explanation of the meaning of the parameter and ranges of allowed values:

Slave Parameters, Meanings and Ranges

Parameter	Meaning	Range of Values
SrvConsConnSize	Consumed I/O connection size as server	0 - 255
ConsOffset	Offset address in input area for server input data	0 – 5736
SrvProdConnSize	Produced I/O connection size as server	0 – 255
ProdOffset	Offset address in output area for server output data	0 - 5760

Table 47: Slave Parameters, their Meanings and their Ranges of allowed Values

SrvConsConnSize

This parameter specifies the length of the data area within the input buffer to be used for the master-to-master information exchange.

It relates to the *Consumed Connection Size* attribute of the Connection object, see section 4.2.4 for more information.

ConsOffset

If a non-zero value is specified for *SrvConsConnSize* information is required where in the IO area to store the input data. This parameter specifies where (i.e. at which offset) in the input buffer the data area for the master-to-master information exchange begins.

SrvProdConnSize

This parameter specifies the length of the data area within the output buffer to be used for the master-to-master information exchange.

This parameter relates to the *Produced Connection Size* attribute of the Connection object, see section 4.2.4 for more information.

ProdOffset

If a non-zero value is specified for *SrvConsConnSize* information is required where in the IO area to store the output data. This parameter specifies where (i.e. at which offset) in the output buffer the data area for the master-to-master information exchange begins.

4.6 Diagnosis

4.6.1 Diagnosis with Packet DEVNET_FAL_CMD_DEV_DIAG_REQ/CNF

Diagnosis can be accomplished using packet DEVNET_FAL_CMD_DEV_DIAG_REQ/CNF, see section 5.5.4 ”118” on page 118.

The structure DN_FAL_DEV_DIAG_DATALOAD_T contained in the tDataLoad parameter of the confirmation packet supplies the following information:

Structure DN_FAL_DEV_DIAG_DATALOAD_T		
Structure element name	**Type**	**Meaning**
DN_FAL_DEV_DIAG_DATA_T	UINT8	Main part of Diagnostic Data Structure
abData [DN_FAL_MAX_DATA_SIZE-6]	UINT8[]	Device specific part of Diagnostic Data Structure, does not contain any user data

Table 48: DN_FAL_DEV_DIAG_DATALOAD_T - Diagnostic Data Structure

The structure DN_FAL_DEV_DIAG_DATA_T within DN_FAL_DEV_DIAG_DATALOAD_T contains the following information:

Structure DN_FAL_DEV_DIAG_DATA_T		
Structure element name	**Type**	**Meaning**
bNodeExtraDiag	UINT8	Additional diagnosis flags for the specific node addressed by parameter ubDevMacId. See *Table 50: Flags within bNodeExtraDiag*
bDevMainState	UINT8	Device Main State (Main state of device handler). See *Table 51: Meaning of bDevMainState byte of structure DN_FAL_DEV_DIAG_DATA_T* Only present for debugging purposes.
bOnlineError	UINT8	Online error
bGeneralErrorCode	UINT8	General Error Code according to DeviceNet specification. See *Generic Error Codes.*
bAdditionalCode	UINT8	Additional Error Code, see documentation of respective slave device
usHrtBeatTimeout	UINT16	Heartbeat Timeout Value

Table 49: DN_FAL_DEV_DIAG_DATA_T - Diagnostic Data Structure

The bits of the bNodeExtraDiag byte of structure DN_FAL_DEV_DIAG_DATA_T have the following meaning:

Bit #	Flag	Flag Name	Meaning/Description
D0	0x01	DN_FAL_NODE_DIAG_NOT_CONFIGURED	Not configured In the current configuration, the device is not active and therefore it is not handled.
D1	0x02	-	Reserved
D2	0x04	-	Reserved
D3	0x08	DN_FAL_NODE_DIAG_UCMM_SUPPORT	The slave is UCMM-capable (present only in version 2.3.10 and later)
D4	0x10	DN_FAL_NODE_DIAG_CFG_FAULT	Configuration fault The differences between device produced and consumed connection size to the resulting configured ones
D5	0x20	DN_FAL_NODE_DIAG_PRM_FAULT	Parameter fault The device had denied the access to at least one configured customized attribute in writing during connection establishment phase.
D6	0x40	-	Reserved
D7	0x80	DN_FAL_NODE_DIAG_NO_RES	No response from the slave.

Table 50: Flags within bNodeExtraDiag

The bDevMainState byte of structure DN_FAL_DEV_DIAG_DATA_T has the following meaning:

For each slave device there is a state machine handler active, which specifically handles the startup procedure, the process data transmission and the error handling of the device depending on its behavior and its configuration. Each state is represented by a number in bDevMainState.

Value	Definition	Explanation
0	DV_ENTER	State machine enter
1	DV_INACTIVE	Device inactive, not handled
2	DV_MANAGE_SERVER	Own MAC ID, state waiting for all incoming messages destinated to the master as server.
3	DV_INIT_PRED_MSTSL	initialize internal predefined master slaves structures
4	DV_ALLOC_PRED_MSTSL	allocated predefined master slave connection set request
5	DV_WAIT_FOR_ALLOC	wait for predefined master slave allocation connection response
6	DV_RELEASE_MSTSL_CONN	release predefined master slave connection set request
7	DV_WAIT_FOR_RELEASE	wait for predefined master slave release connection response
8	DV_INIT_IO_CONF	initialize internal I/O configured structures
9	DV_ALLOC_IO_CONN	allocate configured I/O connection request
10	DV_WAIT_FOR_IO_ALLOC	wait for I/O allocation response
11	DV_RELEASE_IO_CONN	release I/O connection request
12	DV_WAIT_FOR_IO_RELEASE	wait for I/O connection release response
13	DV_READ_CONSUMED_SIZE	read consumed connection size
14	DV_WAIT_READ_CONS	wait for read consumed connection size response
15	DV_CHECK_CONS_SIZE	compare consumed connection size with internal configured one
16	DV_READ_PRODUCED_SIZE	read produced connection size

Value	Definition	Explanation
17	DV_WAIT_READ_PROD	wait for read produced connection size response
18	DV_CHECK_PROD_SIZE	compare produced connection size with internal configured one
19	DV_ANNOUNCE_CONNECTION	configure the I/O connection structures and register it
20	DV_SET_EXPCT_RATE	set expected packet rate
21	DV_WAIT_EXPCT_RATE	wait for set expected packet rate response
22	DV_END_IO_POLL	I/O poll request first time
23	DV_WAIT_IO_POLL	wait for I/O poll response
24	DV_END_IO_POLL_2	I/O poll request second time
25	DV_WAIT_IO_POLL_2	wait for I/O poll response
26	DV_END_IO_POLL_3	I/O poll request third time
27	DV_WAIT_IO_POLL_3	wait for I/O poll response
28	DV_HRTB_TIME_OUT	heartbeat timeout to the device
29	DV_PRX_SRV_REQ	PRX server request
30	DV_OPEN_EXPLICIT_CONN	open unconnected explicit connection request first time
31	DV_OPEN_EXPLICIT_RES	wait for unconnected explicit connection response
32	DV_OPEN_EXPLICIT_CONN_2	open unconnected explicit connection request second time
33	DV_OPEN_EXPLICIT_RES_2	wait for unconnected explicit connection response
34	DV_CHECK_CLOSE_CON	close unconnected connection request
35	DV_WAIT_CLOSE_RES	wait for close unconnected connection response
36	DV_RELEASE_CONNECTION	release all established connections request
37	DV_WAIT_RELEASE_ALL	ALL wait for connection release response
38	DV_USR_EXPL_CONN	open user unconnected explicit connection request
39	DV_USR_EXPLICIT_RES	wait for user explicit connection response
40	DV_USR_PRED_MSL_CONN	user predefined master slave allocate connection request
41	DV_WAIT_FOR_USER_ALLOC	ALLOC wait for user allocation response
42	DV_CHECK_USR_CLOSE_CON	user close unconnected connection request
43	DV_WAIT_USR_CLOSE_RES	with for user close unconnected response
44	DV_QUERY_SERVICE	get or set user defined attribute request
45	DV_WAIT_ACCESS	wait for user defined get or set attribute response
46	DV_CHECK_GETSET_ASW	send or wait
47	DV_SET_EXPL_EPR	set explicit EPR
48	DV_WAIT_EXPL_EPR	wait for set explicit EPR
49	DV_WAIT_BIT_STROBE_RES	wait for bit-strobe response
50	DV_WAIT_COS_RES	wait for change-of-state response
51	DV_END_IO_COS	end IO change-of-state
52	DV_SET_EXPCT_RATE_2	set expected rate 2
53	DV_WAIT_EXPCT_RATE_2	wait for expected rate 2
54	DV_CHECK_EXPCT_2	check expected rate 2
55	DV_RELEASE_STOP	release stop
56	DV_WAIT_REL_ALL_STOP	wait for release stop
57	DV_STOPPED	stopped
58	DV_SET_WDOG_TIMEOUT	set watchdog timeout

Value	Definition	Explanation
59	DV_WAIT_WATCH_TIMEOUT	wait for watchdog timeout
60	DV_SET_WDOG_TIMEOUT_2	set watchdog timeout 2
61	DV_WAIT_WDOG_TIMEOUT_2	wait for watchdog timeout 2
62	DV_SET_PRODUCT_INHIBIT	set product inhibit
63	DV_WAIT_SET_PRODUCT	wait for set product inhibit
64	DV_WAIT_CYCLIC_RES	wait for cyclic response
65	DV_GET_WDOG_TIMEOUT	get watchdog timeout
66	DV_WAIT_GET_WDOG_TMOUT	wait for get watchdog timeout
67	DV_CHECK_WATCH_TIMEOUT	check watchdog timeout
68	DV_GET_WDOG_TIMEOUT2	get watchdog timeout 2
69	DV_WAIT_GETWDOG_TMOUT2	wait for get watchdog timeout 2
70	DV_CHECK_WATCH_TIMEOUT2	check watchdog timeout 2
71	DV_SUSPENDED	suspended
72	DV_SET_OPEN_EXPL_ZERO	set open explicit 0
73	DV_WAIT_OPEN_EXPL_ZERO	wait for set open explicit 0
74	DV_RESET_DV	reset DV
75	DV_WAIT_RESET	wait for reset DV
76	DV_DO_NEXT_PRM	do next parameter
77	DV_WAIT_PRM	wait for do next parameter
78	DV_GET_VENDOR_ID	get Vendor ID
79	DV_WAIT_GET_VENDOR_ID	wait for get Vendor ID
80	DV_CHECK_VENDOR_ID	check Vendor ID
81	DV_GET_DEVICE_TYPE	get Device Type
82	DV_WAIT_GET_DEV_TYPE	wait for get Device Type
83	DV_CHECK_DEVICE_TYPE	check Device Type
84	DV_GET_PRODUCT_CODE	get Product Code
85	DV_WAIT_GET_PROD_CODE	wait for get Product Code
86	DV_CHECK_PRODUCT_CODE	check Product Code
87	DV_GET_REVISION	get Revision
88	DV_WAIT_GET_REVISION	wait for get Revision
89	DV_CHECK_REVISION	check Revision
90	DV_USR_UNCONN_EXPLICIT	user unconnect explicit
91	DV_WAIT_UNCONN_ACCESS	wait for unconnect access
92	DV_CHECK_UNCONN_ANSWER	check unconnect answer
93	DV_USR_CLOSE_UCMM	user close UCMM
94	DV_WAIT_USER_CLOSE	wait for User close
95	DV_END_USER_CLOSE	end User close
96	DV_DEFERRED_DELETE	deferred delete
97	DV_WAIT_DEFERRED_DEL	wait for deferred delete
98	DV_SET_EXPL_EPR_2500	set explicit EPR 2500
99	DV_WAIT_EXPL_EPR_2500	wait for set explicit EPR 2500
100	DV_WAIT_EXPL_RESET	wait for explicit reset

Value	Definition	Explanation
101	DV_WAIT_EXPL_HEARTBEAT	wait for explicit heartbeat
102	DV_DUP_MAC_ID_CHECK	duplicate MAC ID check
103	DV_QX_CNXN	Quick connection
104	DV_QX_CNXN_1ST_TRY	Quick Connection, first try
105	DV_QX_CNXN_2ND_TRY	Quick Connection, second try
106	DV_QXCNXN_2TRY_WAIT	wait for Quick connection, second try
107	DV_PRX_TRY_OPEN_EXZ	Proxy Service, Try Open Explicit Connection
108	DV_PRX_WAIT_OPEN_EXZ	Proxy Service, Wait Open Explicit Connection
109	DV_PRX_SRV_QUERY	Proxy Service, Service Query
110	DV_PRX_SRV_QUERY_WAIT	Proxy Service, Wait For Service Query
111	DV_PRX_SRV_RES	Proxy Service, Service Response
112	DV_PRX_SRV_DONE	Proxy Service, Service Done

Table 51: Meaning of bDevMainState byte of structure DN_FAL_DEV_DIAG_DATA_T

The bOnlineError byte of structure DN_FAL_DEV_DIAG_DATA_T has the following meaning:

In this byte the current online error of the device station is stored if there is any. See the Error Event table of the global bus status field for possible entries.

The bGeneralErrorCode byte of structure DN_FAL_DEV_DIAG_DATA_T informs about the common errors.

The Generic Error Codes appearing in the context of the confirmation packet have the following meaning:

bGeneral ErrorCode	Error Code/ Signification
0	DNM_CIP_GEN_E_SUCCESS Service was successfully performed by the object specified.
2	DNM_CIP_GEN_E_RESOURCE_UNAVAILABLE Resources needed for the object to perform the service were unavailable
8	DNM_CIP_GEN_E_SERVICE_NOT_SUPPORTED The requested service has not implemented / defined for this Class / Instance.
9	DNM_CIP_GEN_E_INVALID_ATTRIBUTE_VALUE Invalid attribute data detected.
11	DNM_CIP_GEN_E_ALREADY_IN_REQUESTED_STATE The object is already in the mode/state being requested by the service.
12	DNM_CIP_GEN_E_OBJECT_STATE_CONFLICT The object cannot perform the requested service in its current mode/state.
14	DNM_CIP_GEN_E_ATTRIBUTE_NOT_SETTABLE A request to modify a non-modifiable attribute has been received.
15	DNM_CIP_GEN_E_PRIVILEGE_VIOLATION Privilege violation (a permission check failed)
16	DNM_CIP_GEN_E_DEVICE_STATE_CONFLICT Device state conflict (The device's current mode/state prohibits the execution of the service)
17	DNM_CIP_GEN_E_REPLY_DATA_TOO_LARGE The data to be transmitted in the response buffer is larger than the allocated response buffer.
19	DNM_CIP_GEN_E_NOT_ENOUGH_DATA The service did not supply enough data to perform the specified operation.
20	DNM_CIP_GEN_E_ATTRIBUTE_NOT_SUPPORTED The attribute specified in the request is not supported

bGeneral ErrorCode	Error Code/ Signification
21	DNM_CIP_GEN_E_TOO_MUCH_DATA Too much data. The service supplied more data than was expected.
22	DNM_CIP_GEN_E_OBJECT_DOES_NOT_EXIST The object specified does not exist in the device.

Table 52: Generic Error Codes in the Context of the Confirmation Packet

For a full list of all Generic Error Codes, see section 6.4 "*Generic Error Codes*".

The `bAdditionalCode` byte of structure `DN_FAL_DEV_DIAG_DATA_T` contains device-specific information in case of `bGeneralErrorCode` is equal to `DNM_CIP_GEN_E_VENDOR_SPECIFIC_ERROR` (0x1F = 31 decimal). Refer to the documentation of the respective slave device.

The `usHrtBeatTimeout` byte of structure `DN_FAL_DEV_DIAG_DATA_T` is a timer. If a device is supervised by the expected packet rate of a connection and times out then, the timer `usHrtBeatTimeout` will be incremented. So the actual value gives an overview how good the transmission quality to this device is and how often a timeout has happened. After a device times out the protocol stack always tries to reestablish the connection immediately.

4.6.2 Get Diagnostic Information Remotely From Connected Slaves

4.6.2.1 Using the Get Attribute Service for obtaining Diagnostic Information

In order to obtain more diagnostic information, you can use the *Explicit Service* as described in section *Explicit Message Services*. The following assumption is made:

We assume your device has the MAC-ID 3. Otherwise you need to adapt the `bDeviceAddr` parameter (see below) accordingly.

Example: Obtaining the name of the device

You can then set the following values, for instance, in order to retrieve the name of the device:

Structure `DN_FAL_PACKET_GETSET_ATT_REQ_T`			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure `TLR_PACKET_HEADER_T`			
ulDest	UINT32	0x20	Destination Queue-Handle
ulSrc	UINT32	0	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier.
ulLen	UINT32	12	DN_FAL_GETSET_ATT_REQ_SIZE
ulId	UINT32		Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Set to zero
ulCmd	UINT32	0x380A	`DEVNET_FAL_CMD_GET_ATT_REQ` - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing, set to zero
tData - Structure `DN_FAL_SDU_GETSET_ATT_REQ_T`			
bDeviceAddr	UINT8	3	Device address, MAC-ID of device to be addressed
abReserved[3]	UINT8[]	0,0,0	Reserved zero
usClass	UINT16	1	Class ID of identity object
usInstance	UINT16	1	Instance ID of identity object. This class has only 1 instance.
usAttribute	UINT16	7	Attribute ID of product name.
usReserved	UINT16	0	Set to zero.

Table 53: Obtaining the Name of the Device using the Get Attribute Service

Obtaining the configured number of data (for produced and consumed connections)

You can then set the following values, for instance, in order to retrieve the name of the device (use `usAttribute=7` for producer and `usAttribute=8` for consumer connections):

Structure DN_FAL_PACKET_GETSET_ATT_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0x20	Destination Queue-Handle
ulSrc	UINT32	0	Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier.
ulLen	UINT32	12	DN_FAL_GETSET_ATT_REQ_SIZE
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Set to zero
ulCmd	UINT32	0x380A	`DEVNET_FAL_CMD_GET_ATT_REQ` - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing, set to zero
tData - Structure DN_FAL_SDU_GETSET_ATT_REQ_T			
bDeviceAddr	UINT8	3	Device address, MAC-ID of device to be addressed
abReserved[3]	UINT8[]	0,0,0	Reserved zero
usClass	UINT16	5	Class ID of connection object
usInstance	UINT16	 2 3 4	Instance ID, specifies the type of connection Polling connection object Bit strobe connection object Cyclic/change-of-state connection object.
usAttribute	UINT16	 7 8	Attribute ID Produced Size Consumed Size
usReserved	UINT16	0	Set to zero

Table 54: Obtaining the configured Number of Data (for produced and consumed Connections) using the Get Attribute Service

4.6.2.2 Using the Get Slave Connection Information Request

An application which is based on Hilscher's DPM for netX can obtain diagnostic and status information about all slaves connected to and administered by this master from the master firmware as described in general in the netX DPM Manual (reference [1]). (This information only concerns cyclic data transfer.) The DeviceNet Master firmware supports this feature.

Note: The operating system rcX of the netX uses handles in order to access slaves. This is done in a possibly unexpected way as these handles are **not equal to the MAC ID**.

The diagnostic information consists of three lists for the administration of:

- the configured slaves
- the activated slaves
- the slaves with a known fault

Retrieving the diagnostic information is a two-step-process as you first retrieve the handle using the *Get Slave Handle* request and subsequently you retrieve the diagnostic information using the handle.

1. Retrieve the handle by the *Get Slave Handle* request (`RCX_GET_SLAVE_HANDLE_REQ`, Command code `0x2F08`) which is described in the netX DPM Manual (reference [1]), chapter 5.2.2.1. In order to do so, you need to choose which kind of list of the above mentioned slave lists you want to obtain. The confirmation packet you will receive (`RCX_GET_SLAVE_HANDLE_CNF`, Command code `0x2F09`) will then deliver an array of handles to the elements of the selected list.
2. This allows obtaining of a diagnosis structure for the specific slave by the *Get Slave Connection Information* request (`RCX_GET_SLAVE_CONN_INFO_REQ`, Command code `0x2F0A`). This packet requires the handle of the specific slave taken from the array of handles to the elements of the selected list obtained in the first step. You will then receive the confirmation packet (`RCX_GET_SLAVE_CONN_INFO_CNF`, Command code `0x2F0B`) delivering – besides others – a structure `tState` containing the following structure with information about the selected slave from the master firmware (see reference 1 for more details). For a DeviceNet Master this structure looks as following:

```
typedef struct DN_DEV_PRM_HEADER_Ttag
{
  TLR_UINT16      usDevParaLen;     /* length of whole parameter data set     */

  TLR_UINT8       bDvFlag;          /* device related flags                   */
  #define DN_DV_FLAG_UCMM_SUPP     0x01/* UCMM manager supported by device    */
  #define DN_DV_FLAG_QUICK_CONNECT 0x02/* key check Quick Connect enabled     */
  #define DN_DV_FLAG_KEY_VENDOR    0x04/* key check Vendor ID on/off          */
  #define DN_DV_FLAG_KEY_DTYPE     0x08/* key check Device Type on/off        */
  #define DN_DV_FLAG_KEY_PCODE     0x10/* key check Product Code on/off       */
  #define DN_DV_FLAG_KEY_REV       0x20/* key check for Revision on/off       */
  #define DN_DV_FLAG_ACTIVE        0x80/* set device access active            */
  #define DN_DV_FLAG_RESERVED      0x42/* Reserved flags                      */

  TLR_UINT8       bUcmmGroup;       /* dynamic UCMM connection group          */
  #define DNM_GROUP_1 0
  #define DNM_GROUP_2 1
  #define DNM_GROUP_3 3
  TLR_UINT16      usRecFragTimer;   /* timeout value for reconnection and for */
                                    /* fragmented transfer also usFragTimeout */
  TLR_UINT16      usVendorID;       /* Vendor ID                              */
  TLR_UINT16      usDeviceType;     /* Device Type                            */
  TLR_UINT16      usProductCode;    /* Product code                           */
  TLR_UINT8       bMajorRevision;   /* Revision major digit                   */
  TLR_UINT8       bMinorRevision;   /* Revision minor digit                   */

  TLR_UINT8       bOctetString[2];  /* reserved field                         */
} DN_DEV_PRM_HEADER_T;
```

The meaning of the various variables is explained in the respective comments.

5 The Application Interface

This chapter defines the user application interface of the DeviceNet Stack. There are two different levels the user can access the DeviceNet stack.

- The Communication Channel Interface
- The DeviceNet FAL - Task Interface

The Communication Channel Interface is the Hilscher's Dual port Memory Interface for field buses or other communication stacks. A typical application is using PC cards or COM modules with a discrete DPM and accessing the DeviceNet via Driver API. This interface is also available, when a user develops his own application within the netX system and works with the virtual DPM system intern.

The second level is when a user implements its own application task direct over the DevNet FAL Task. Then a user has to take care of the configuration process, mapping I/O data and status information by his own. Typically when a user works with its own dual port memory layout, the application itself has to be developed then as a task according to the Hilscher's Task Layer Reference Model.

Note: All packets described in this chapter can be used both when working with loadable firmware and with linkable object modules.

5.1 The Dual Port Memory Interface

This chapter defines the user interface functions available when using the Dual Port Memory to interface to the DeviceNet Master Stack.

The user application developed for the Dual Port Memory interface should follow the details outlined in the "netX DPM Interface Manual" (Reference [1]).

The following chapter defines the packets which may be sent or received by the user using the Dual Port Memory Interface method.

5.2 The DevNet AP – Task

Within the DeviceNet Stack the DevNet AP - Task is an application layer on top of the DeviceNet FAL- Task. It is responsible for transferring the I/O, diagnostic and acyclic data from and to the DeviceNet FAL Task on the one hand and the Dual port Interface on the other hand. Furthermore, the DeviceNetAP Task is responsible for all user application interactions and represents the only counterpart of the user within the current DeviceNet-Stack implementation.

To get the handle of the process queue of the DevNetAP -Task the Macro `TLR_QUE_IDENTIFY()` has to be used in conjunction with the following ASCII-Queue name

ASCII Queue name	Description
`"QUE_DEVNET_AP"`	Name of the DevNet AP -Task process queue

Table 55: DevNetAP-Task Process Queue

The returned handle has to be used as value `ulDest` in all initiator packets the user intends to send to the DevNet AP -Task. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDPACKET_FIFO/LIFO()` for sending a packet to the DevNet AP -Task.

5.2.1 Get LED State Service - DEVNET_AP_CMD_GET_LED_STATE_REQ/CNF

This packet can be used to inquire the current LED state, LED mode and LED color from the stack (see respective parameters of confirmation packet).

Currently this packet only supports LED Type NS (`ulLedType =1`)

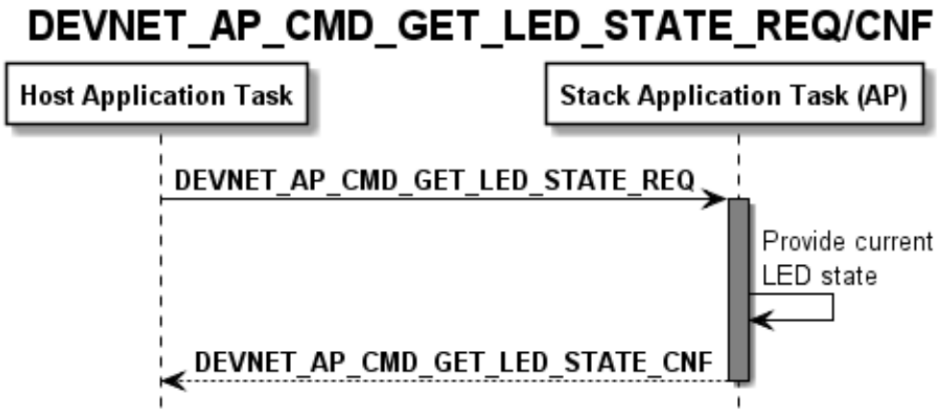

Figure 11 Sequence diagram for DEVNET_AP_CMD_GET_LED_STATE_REQ packet

5.2.1.1 Get LED State Request

The structure of the request packet is described as follows:

Packet Structure Reference

```
/*****************************************************************************/
/*                      DNM_AP_PACKET_GET_LED_REQ_T Structure                  */
/*****************************************************************************/

#define DEVNET_AP_CMD_GET_LED_STATE_REQ   0x00003904
#define DNM_AP_LED_TYPE_NS                                                  1

typedef struct DNM_AP_LED_STATE_REQ_Ttag
{
  TLR_UINT32 ulLedType;  /* MNS, NS, MS */
  //TLR_UINT32 ulLedMode;  /* ON, OFF, FLASH */
  //TLR_UINT32 ulLedColor; /* RED, GRN, OFF */
} DNM_AP_LED_STATE_REQ_T;

#define DNM_AP_LED_STATE_REQ_SIZE (sizeof(DNM_AP_LED_STATE_REQ_T))

typedef struct DN_AP_PACKET_GET_LED_REQ_Ttag
{
  TLR_PACKET_HEADER_T     tHead;
  DNM_AP_LED_STATE_REQ_T  tData;
} DNM_AP_PACKET_GET_LED_REQ_T;
```

Packet Description

Structure DNM_AP_PACKET_GET_LED_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	See section *Status/Error Codes DevNet AP – Task*
ulCmd	UINT32	0x3904	DEVNET_AP_CMD_GET_LED_STATE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DNM_AP_LED_STATE_REQ_T			
ulLedType	UINT32	1	LED Type (MNS, NS, MS): 1: NS (Currently only NS supported)

Table 56: DNM_AP_PACKET_GET_LED_REQ_T - Get LED State Request

5.2.1.2 Get LED State Confirmation

The confirmation packet provides the current state of the LED.

Packet Structure Reference

```
/*****************************************************************************/
/*                        DNM_AP_PACKET_GET_LED_CNF_T Structure                */
/*****************************************************************************/

#define DEVNET_AP_CMD_GET_LED_STATE_CNF   0x00003905

typedef struct DNM_AP_LED_STATE_CNF_Ttag
{
  /*DNM_LED_TYPE_NS = 1 */
  TLR_UINT32 ulLedType;  /* MNS, NS, MS */

  /* DNM_LED_MODE_STATIC = 0, DNM_LED_MODE_FLASH = 1 */
  TLR_UINT32 ulLedMode;  /* ON, OFF, FLASH */

  /*DNM_LED_COLR_OFF = 0,DNM_LED_COLR_GRN = 1, DNM_LED_COLR_RED = 2 */
  TLR_UINT32 ulLedColor; /* RED, GRN, OFF */
}
DNM_AP_LED_STATE_CNF_T;

#define DNM_AP_LED_STATE_CNF_SIZE (sizeof(DNM_AP_LED_STATE_CNF_T))

typedef struct DNM_AP_PACKET_GET_LED_CNF_Ttag
{
  TLR_PACKET_HEADER_T    tHead;
  DNM_AP_LED_STATE_CNF_T tData;
}
DNM_AP_PACKET_GET_LED_CNF_T;
```

Packet Description

Structure DNM_AP_LED_STATE_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet AP – Task*
ulCmd	UINT32	0x3905	DEVNET_AP_CMD_GET_LED_STATE_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DNM_AP_LED_STATE_CNF_T			
ulLedType	UINT32	1	LED Type: 1: NS LED
ulLedMode	UINT32	0..1	LED Mode 0: Static (DNM_LED_MODE_STATIC) 1: Flash (DNM_LED_MODE_FLASH)
ulLedColor	UINT8	0..2	LED Color 0: Off (DNM_LED_COLR_OFF) 1:Green (DNM_LED_COLR_GRN) 2: DNM_LED_COLR_RED

Table 57: DNM_AP_LED_STATE_CNF_T - Confirmation of Get LED State Request

5.2.2 IO Scan Service DEVNET_AP_CMD_IO_SCAN_REQ/CNF

The AP task uses this command to start a new cycle of I/O data scanning. It sends this packet to its own queue. The AP task does not accept this command from other tasks (Host or Stack task).

5.2.3 Handled Commands

The DevNetAP - Task itself has no own commands besides the LED packets described above. But the DevNetAP Task knows all the commands from the underlying DeviceNet Stack and the Common Global task commands. The AP Task will route the commands to the appropriate Task. The AP task will not route the full command set of the underlying DeviceNet stack and in every state. The reason is that the AP-Task handles some control commands of the underlying stack by its own automatic, and that execution of these commands is not allowed for the user. Also it depends on the current implementation status of the FW.

The following list explains which commands are currently handled by the protocol stack and which ones will be rejected in the current version:

Commands	Handled	Description
Common Global Stack Commands		
DIAG_INFO_GET_COMMON_STATE_REQ	yes	Returns the common status information that is the same for all communication channels
DIAG_INFO_GET_EXTENDED_STATE_REQ	yes	Returns the DeviceNet specific status information of the extended status block
CONFIGURATION_RELOAD_REQ	yes	Indication when new configuration is available, cause a reset of the DeviceNet stack,
DeviceNetFAL Task specific Commands		
DEVNET_FAL_CMD_INIT_REQ	yes (Routed)	Cause a reset/initialization of the DeviceNet stack. The packet will be forwarded to the underlying FAL task.
DEVNET_FAL_CMD_DOWNLOAD_REQ	yes	Download configuration data
DEVNET_FAL_CMD_SET_MODE_REQ	yes (Routed)	The packet will be forwarded to the underlying FAL task.
DEVNET_FAL_CMD_CLR_CONFIG_REQ	yes	Clear configuration Service
DEVNET_FAL_CMD_NEW_OUTPUT_IND	yes	Indication of new output from FAL task.
DEVNET_FAL_CMD_GET_ATT_REQ	yes	DeviceNet Get Attribute Service
DEVNET_FAL_CMD_SET_ATT_REQ	yes	DeviceNet Set Attribute Service
DEVNET_FAL_CMD_AP_REGISTER_REQ	yes	Application Register Service
DEVNET_FAL_CMD_FAULT_IND	yes	The fault indication comes from underlying FAL task.
DEVNET_FAL_CMD_ACYC_BTS_REQ	yes	Acyclic Bit-strobe
DEVNET_FAL_CMD_LIFELIST_REQ	yes	Generate Lifelist
DEVNET_FAL_CMD_UPLOAD_REQ	yes	Parameter Upload
DEVNET_FAL_CMD_SET_MODE_IND	yes	The set mode (failed or succeed) indication from FAL task.
DEVNET_FAL_CMD_DEV_DIAG_REQ	Yes (Routed)	The packet will be forwarded to the underlying FAL task.

Commands	Handled	Description
DEVNET_FAL_CMD_REMOTE_SERVICE_REQ	yes	Send service request to a slave.
DEVNET_FAL_CMD_LOCAL_SERVICE_REQ	yes	Send service request to the master itself.
DEVNET_FAL_CMD_ACYC_POLL_REQ	yes	Perform a poll request to a slave at any time.
DEVNET_FAL_CMD_CAN_FWD_REG_REQ	yes	Register a CAN ID for forwarding to host task.
DEVNET_FAL_CMD_CAN_FWD_IND	yes	Indication of CAN ID comes from bus.
DEVNET_FAL_CMD_CAN_DATA_REQ	yes	Send a CAN frame on bus at any time.
DEVNET_FAL_CMD_SET_LED_IND	yes	Set LEDs indication from FAL task.
CAN DL Task specific Commands		
CAN_DL_CMD_ …	no (Rejected)	All CAN commands are rejected from the AP task. The user has no access to the CAN layer.

Table 58: DeviceNet Master - Handled Commands

5.2.4 Extended Status Information

The DeviceNet stack uses the extended status field within the communication channel to transfer the status information described below. The AP task is responsible to map this information into the extended status block of the communication channel. To obtain this information the user can send the global command DIAG_INFO_GET_EXTENDED_STATE_REQ to the AP task or call the appropriate cifX DriverAPI function when accessing to the communication channel via driver. In its current implementation the extended status field DEVNET_AP_EXT_STATUS_T contains one structure "tGlobalStateField". In future it can maybe enlarged by adding additional diagnostic structures if necessary.

Structure Reference

You can find a detailed structure reference of structure "DN_FAL_EXT_DIAG_T tGlobalStateField" in section 3.3.2 "*Extended Status*".

The established structure "DN_FAL_EXT_DIAG_T tGlobalStateField" informs about global bus states as well as individual states of the managed devices. To hold the information preferably compact, the device specific information is stated in bit fields.

For a detailed discussion of the meaning of all these variables and bit fields also refer to section 3.3.2.

5.3 The DevNet FAL - Task

Within the DeviceNet Stack the DevNet FAL-Task coordinates the underlying slave state machines used for processing of the various services.

Furthermore, it is responsible for all application interactions and represents the counterpart of the AP-Task within the current DeviceNet Stack implementation.

To get the handle of the process queue of the DevNetFAL-Task the Macro `TLR_QUE_IDENTIFY()` has to be used in conjunction with the following ASCII-Queue name

ASCII Queue name	Description
"DEVNET_FAL_QUE"	Name of the DevNetFAL -Task process queue

Table 59: DevNetFAL-Task Process Queue

The returned handle has to be used as value `ulDest` in all initiator packets the AP-Task intends to send to the DevNet FAL -Task. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDPACKET_FIFO/LIFO()` for sending a packet to the DevNetFAL -Task.

In detail, the following functionality is provided by the DevNetFAL -Task:

Overview over Packets of the DevNetFAL -Task			
No. of section	**Packet**	**Command code (REQ/CNF or IND/RES)**	**Page**
5.5.1	`Stack` Initialization Service DEVNET_FAL_CMD_INIT_REQ/CNF	0x3800/ 0x3801	109
5.5.2	`Set` Operation Mode Service DEVNET_FAL_CMD_SET_MODE_REQ/CNF	0x3804/ 0x3805	112
5.4.1	`Download` Configuration DEVNET_FAL_CMD_DOWNLOAD_REQ/CNF	0x3802/ 0x3803	87
5.7.1	Get Attribute Service `DEVNET_FAL_CMD_GET_ATT_REQ/CNF`	0x380A/ 0x380B	126
5.7.2	`Set` Attribute Service DEVNET_FAL_CMD_SET_ATT_REQ/CNF	0x380C/ 0x380D	141
5.6.1	`Acyclic` Bit-Strobing Service DEVNET_FAL_CMD_ACYC_BTS_REQ/CNF	0x3812/ 0x3813	126
5.9.1	Get Lifelist Service `DEVNET_FAL_CMD_LIFELIST_REQ/CNF`	0x3814/ 0x3815	153
5.5.3	`Parameter` Upload Service DEVNET_FAL_CMD_UPLOAD_REQ/CNF	0x3816/ 0x3817	115
5.5.5	`DEVNET_FAL_CMD_FAULT_IND/RES` – Indication of a Fault	0x3810/ 0x3811	121
5.4.2	Clear Configuration Service DEVNET_FAL_CMD_CLR_CONFIG_REQ/CNF	0x3806/ 0x3807	107
5.6.3	New Output Indication `DEVNET_FAL_CMD_NEW_OUTPUT_IND/RES`	0x3808/ 0x3809	134
5.10.1	Register Application Service DEVNET_FAL_CMD_AP_REGISTER_REQ/CNF	0x380E/ 0x380F	165
5.7.3	DEVNET_FAL_CMD_REMOTE_SERVICE_REQ/CNF – Remote Service	0x3822/ 0x3823	145

5.7.4	Local Service DEVNET_FAL_CMD_LOCAL_SERVICE_REQ/CNF	0x3824/ 0x3825	150
5.5.4	Device Diagnosis Service DEVNET_FAL_CMD_DEV_DIAG_REQ/CNF	0x3820/ 0x3821	118
5.5.6	Operation Mode Indication DEVNET_FAL_CMD_SET_MODE_IND/RES	0x3818/ 0x3819	123
5.6.2	Acyclic Poll Service DEVNET_FAL_CMD_ACYC_POLL_REQ/CNF	0x38E0/ 0x38E1	130
5.8.1	CAN Registered Service DEVNET_FAL_CMD_CAN_FWD_REG_REQ/CNF	0x38E2/ 0x38E3	153
5.8.2	CAN Forward Service DEVNET_FAL_CMD_CAN_FWD_IND/RES	0x38E4/ 0x38E5	156
5.8.3	DEVNET_FAL_CMD_CAN_DATA_REQ/CNF - CAN Data Request	0x38E6/ 0x38E7	159
5.5.7	DEVNET_FAL_CMD_SET_LED_IND/RES – Set LED Indication	0x38E8/ 0x38E9	125

Table 60: Overview over the Packets of the DevNetFAL -Task of the DeviceNet Master Protocol Stack

5.4 Configuration Services

5.4.1 Download Configuration DEVNET_FAL_CMD_DOWNLOAD_REQ/CNF

The command DEVNET_FAL_CMD_DOWNLOAD_REQ is used to download a configuration to the DeviceNet stack. The download is divided into so called areas like slave parameter sets or the bus parameter set.

The download needs to be performed in following order:

- DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER (corresponds to the device parameter structure DN_FAL_DEV_PRM_T, see subsections 5.4.1.3"*Coding of the Device Parameter Structure*" and 4.5.3"*Detailed Description of Device Parameters*")
- DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER (optional, corresponds to the server parameter structure DN_FAL_SRV_PRM_T, see subsections 5.4.1.4"*Coding of the Server Parameter Structure*" and 4.5.4"*Detailed Description of Server Parameters*")
- DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER (corresponds to the slave parameter structure, see subsections 5.4.1.5"*Coding of the Slave Parameter Structure*" and 4.5.2"*Detailed Description of Slave Parameters*")
- DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER (corresponds to the bus parameter structure DN_FAL_BUS_PRM_T, see subsections 5.4.1.6"*Coding of the Bus Parameter Structure*" and 4.5.1"*Detailed Description of Bus Parameters*")

The download of the bus parameters finishes the configuration process. This packet could be sent from host application task or stack application task as illustrated in *Figure 9: Configuration Sequence Using the Basic Packet* Set and *Figure 10: Configuration Sequence Using the Extended Packet* Set

5.4.1.1 Download Configuration Request

The data structure of the download configuration request looks as follows:

Packet Structure Reference

```
#define DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER      (0)
#define DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER    (1)
#define DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER   (2)
#define DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER   (3)

typedef struct DN_FAL_SDU_DOWNLOAD_REQ_Ttag {
  TLR_UINT16  usArea;
  TLR_UINT16  usSubArea;
  TLR_UINT8   abData[1000];
} DN_FAL_SDU_DOWNLOAD_REQ_T;

typedef struct DN_FAL_PACKET_DOWNLOAD_REQ_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_DOWNLOAD_REQ_T tData;
} DN_FAL_PACKET_DOWNLOAD_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_DOWNLOAD_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier set to zero
ulSrcId	UINT32	0 ... 2^{32}-1	AP Process Source End Point Identifier
ulLen	UINT32	4 + n	Packet Data Length in bytes DN_FAL_DOWNLOAD_REQ_SIZE + n = number of download data in bytes in abData[…]
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Packet Status/Error set to zero
ulCmd	UINT32	0x3802	DEVNET_FAL_CMD_DOWNLOAD_REQ - Command
ulExt	UINT32	x	Sequence information. TLR_PACKET_SEQ_NONE TLR_PACKET_SEQ_FIRST TLR_PACKET_SEQ_MIDDLE TLR_PACKET_SEQ_LAST
ulRout	UINT32	0	Routing, set to zero
Structure DN_FAL_SDU_DOWNLOAD_REQ_T			
usArea	UINT16	0..3	Area code of where the data is to be downloaded. DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER
usSubArea	UINT16	0..n	depends on usArea …_AREA_BUS_PARAMETER 0 …_AREA_SLAVE_PARAMETER 0..63 (Mac ID) …_AREA_SERVER_PARAMETER 0 …_AREA_DEVICE_PARAMETER 0
abData [0..1000]	UINT8[]	xxx	Download data contents depends on usArea

Table 61: DEVNET_FAL_CMD_DOWNLOAD_REQ – Request Command for Configuration Download

Source Code Example: Complete Configuration Procedure

```
TLR_RESULT DnUser_Download_Configuration(DN_USER_RSC_T FAR* ptRsc)
{
  TLR_RESULT eRslt = TLR_S_OK;
  TLR_UINT8  *pabDownData = NULL;
  DN_FAL_DEV_PRM_T tDevPrm;
  DN_FAL_SRV_PRM_T tSrvPrm;
  DN_FAL_BUS_PRM_T tBusPrm;
  TLR_UINT8       *pabSlvPrm;
  TLR_UINT32       ulSlvPrmLen;

  InitDeviceParameter(&tDevPrm);                       //See source example below
  InitServerParameter(&tSrvPrm);                       //See source example below
  InitBusParameter(&tBusPrm);                          //See source example below
  InitGetSlaveParameter(2,&pabSlvPrm,&ulSlvPrmLem);    //See source example below

  for( ulDownPrm = 0; ulDownPrm < 4; ulDownPrm++ )
  {
    switch( ulDownPrm )
    {
      case 0: //Download Device Parameter
      {
        pabDownData = (TLR_UINT8*)&tDevPrm;
        ulDownLen = sizeof(DN_FAL_DEV_PRM_T);
        usDownArea = DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER;
        usDownSubArea = 0;
      }
      break;
      case 1: //Download Server Parameter
      {
        pabDownData = (TLR_UINT8*)&tSrvPrm;
        ulDownLen = sizeof(DN_FAL_DEV_PRM_T);
        usDownArea = DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER;
        usDownSubArea = 0;
        break;
      }
      case 2: //Download one Slave Parameter set
      {
        pabDownData = (TLR_UINT8*)&abSlvPrm;
        ulDownLen = sizeof(abSlvPrm);
        usDownArea = DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER;
        usDownSubArea = 2; // Device address 2
        break;
      }
      case 3: //Download Bus Parameter
      {
        pabDownData = (TLR_UINT8*)&tBusPrm;
        ulDownLen = sizeof(DN_FAL_BUS_PRM_T);
        usDownArea = DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER;
        usDownSubArea = 0;
        break;
      }
    }
  }
  //See source example below for this function
  eRslt = DnUser_Download( ptRsc, pabDownData, ulDownLen,
                           usDownArea, usDownSubArea );
  if(eRslt != TLR_S_OK ) {
    break; //for
  }
  return eRslt;
}
```

Source Code Example: Download Configuration Data

```
TLR_RESULT  DnUser_Download( DN_USER_RSC_T FAR* ptRsc,
                             TLR_UINT8 *pabData, TLR_UINT32 ulLen,
                             TLR_UINT16 usArea, TLR_UINT16 usSubArea   )
{
  TLR_RESULT eRslt = TLR_S_OK;
  DN_FAL_PACKET_DOWNLOAD_REQ_T *ptDownReq = NULL;

  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptDownReq );
  if( eRslt == TLR_S_OK )
  {
     TLR_QUE_LINK_SET_PACKET_SRC(ptDownReq,ptRsc->tLoc.tQueSrcApp);
     ptDownReq->tHead.ulCmd = DEVNET_FAL_CMD_DOWNLOAD_REQ;
     ptDownReq->tHead.ulLen = 4; //Min length for download

     /* set area and sub area */
     ptDownReq->tData.usArea = usArea;
     ptDownReq->tData.usSubArea = usSubArea;

     if( ulLen <= sizeof(ptDownReq->tData.abData )
     {
       TLR_MEMCPY( &ptDownReq->tData.abData[0], pabData, ulLen );
       ptDownReq->tHead.ulLen += ulLen;
       eRslt=TLR_QUE_SENDPACKET_FIFO(ptRsc->tLoc.tDstQue,ptSetModeReq,TLR_FINITE);
     }
     else
      {
       eRslt = TLR_E_FAIL;
     }

     if( eRslt != TLR_S_OK )
     {
       TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptInitReq);
     }
  }
  return eRslt;
}
```

5.4.1.2 Download Configuration Confirmation

The confirmation packet is simply a copy of request packet with extra information stored in variable `ulSta`, which indicates whether the request has been successful or not.

In case of error, the variable `ulSta` is filled with a error code (see *Table 128: Status/Error Codes DevNet FAL - Task*).

Packet Structure Reference

```
typedef struct DN_FAL_SDU_DOWNLOAD_CNF_Ttag {
  TLR_UINT16 usArea;
  TLR_UINT16 usSubArea;
}DN_FAL_SDU_DOWNLOAD_CNF_T;


typedef struct DN_FAL_PACKET_CFG_DOWN_CNF_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_DOWNLOAD_CNF_T tData;
} DN_FAL_PACKET_CFG_DOWN_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_CFG_DOWN_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32		Source End Point Identifier, untouched
ulLen	UINT32	4	`sizeof(DN_FAL_SDU_DOWNLOAD_CNF_T)`
ulId	UINT32	0 ... 2^{32}-1	Packet Identification, untouched
ulSta	UINT32	x	*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3803	`DEVNET_FAL_CMD_DOWNLOAD_CNF` - Command
ulExt	UINT32	x	Extension, untouched
ulRout	UINT32	×	Routing, do not touch
tData - Structure DN_FAL_SDU_DOWNLOAD_CNF_T			

usArea	UINT16	0..3	Area code of where the data is to be downloaded.
			`DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER`
			`DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER`
			`DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER`
			`DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER`

usSubArea	UINT16	0..n	depends on usArea
			`…_AREA_BUS_PARAMETER 0`
			`…_AREA_SLAVE_PARAMETER 0..63 ( Mac ID )`
			`…_AREA_SERVER_PARAMETER 0 …_AREA_DEVICE_PARAMETER 0`

Table 62: DEVNET_FAL_CMD_DOWNLOAD_CNF – Confirmation of Request Command for Configuration Download

5.4.1.3 Coding of the Device Parameter Structure

The structure DN_FAL_DEV_PRM_T contains the configurable device parameters of the stack. Each parameter that wants to be set must be enabled with a corresponding bit in the ulEnableFlags. The device parameter is transferred in the data portion within the DN_FAL_DOWNLOAD_REQ packet.

Structure Reference

```
typedef struct DN_FAL_DEV_PRM_Ttag {

  TLR_UINT32   ulConfigFlags;     /* Configuration flags */
  TLR_UINT32   ulEnableFlags;     /* Enable Flags        */
  TLR_UINT16   usVendorId;        /* Vendor ID           */
  TLR_UINT16   usProductType;     /* Product Type        */
  TLR_UINT16   usProductCode;     /* Product Code        */
  TLR_UINT8    bMinorRev;         /* Minor Version       */
  TLR_UINT8    bMajorRev;         /* Major Version       */
  TLR_UINT32   ulSerialNumber;    /* Serial Number       */
  TLR_UINT8    abReserved[3];     /* Reserved            */
  TLR_UINT8    bProductNameLen;   /* Product Name length */
  TLR_UINT8    abProductName[32]; /* Product Name        */
  TLR_UINT8    abReserved2[8];    /* Reserved            */

} DN_FAL_DEV_PRM_T;
```

Variable	Type	Value	Description
ulConfigFlags	UINT32	0	Reserved, set to zero
ulEnableFlags	UINT32	0x0000000 - 0x0000003F	Flags to declare valid each of the following parameter
usVendorId	UINT16	1 - 65535	Vendor ID
usProductType	UINT16	0 - 65535	Product Type
usProductCode	UINT16	0 - 65535	Product Code
bMinorRev	UINT8	1 - 255	Minor Version
bMajorRev	UINT8	1 - 255	Major Version
ulSerialNumber	UINT32	0x0000000 - 0xFFFFFFFF	Serial number
abReserved[3]	UINT8[]	0x00	Reserved set to zero
bProductNameLen	UINT8	0 … 32	Number of character in abProductName
abProductName[32]	UINT8[]	ASCII	Product Name String
abReserved2[8]	UINT8[]	0x00	Reserved set to zero

Table 63: Device Parameter Structure

Flag	Flag Name DN_FAL_MSK_DEV_PRM_ENABLE …	Description
0x00000001	_VENDORID	Validate the vendor ID
0x00000002	_PRODUCTTYPE	Validate the product type
0x00000004	_PRODUCTCODE	Validate the product code
0x00000008	_MAJORMINORREV	Validate the minor/ major revision
0x00000010	_SERIALNR	Validate the serial number
0x00000020	_PRODUCTNAME	Validate the product name
0xFFFFFC0	_RESERVED	Reserved Flags always set to zero

Table 64: Flags within `ulEnableFlags`

Source Code Example

```
TLR_VOID InitDeviceParameter( DN_FAL_DEV_PRM_T *ptDevPrm )
{
  TLR_MEMSET( ptDevPrm,0x00,sizeof(DN_FAL_DEV_PRM_T));

  ptDevPrm->ulEnableFlags    = (DN_FAL_MSK_DEV_PRM_ENABLE_VENDORID|
                                DN_FAL_MSK_DEV_PRM_ENABLE_PRODUCTTYPE|
                                DN_FAL_MSK_DEV_PRM_ENABLE_PRODUCTCODE|
                                DN_FAL_MSK_DEV_PRM_ENABLE_SERIALNR|
                                DN_FAL_MSK_DEV_PRM_ENABLE_PRODUCTNAME);

  ptDevPrm->usVendorId        = 283;        // Hilscher ODVA vendor ID
  ptDevPrm->usProductType     = 12;         // Communication adapter
  ptDevPrm->usProductCode     = 1;          // My unique product code
  ptDevPrm->ulSerialNumber    = 0x12345678; // My unique device serial number
  ptDevPrm->bMajorRev         = 0;          // Not enabled by EnableFlags
  ptDevPrm->bMinorRev         = 0;          // Not enabled by EnableFlags
  ptDevPrm->bProductNameLen   = 14;         // 14 significant character
  TLR_MEMCPY( &tDevPrm.abProductName[0],“My_Device_Name”,14);
}
```

5.4.1.4 Coding of the Server Parameter Structure

The structure DN_FAL_SRV_PRM_T contains the configurable server parameter (slave) part of the stack. The device parameters are transferred in the data portion within the DN_FAL_DOWNLOAD_REQ packet.

Structure Reference

```
typedef struct DN_FAL_SRV_PRM_Ttag {

  TLR_UINT8  bSrvConsConnSize; /* consumed I/O connection size as server       */
  TLR_UINT16 usConsOffset;     /* offset addr in input data area               */
  TLR_UINT8  bSrvProdConnSize; /* produced I/O connection size as server       */
  TLR_UINT16 usProdOffset;     /* offset addr in output area output data       */
  TLR_UINT8  abReserved[58];   /* reserved                                     */
} DN_FAL_SRV_PRM_T;
```

Variable	Type	Value	Description
bSrvConsConnSize	UINT8	0 - 255	consumed I/O connection size as server
usConsOffset	UINT16	0 - 3583	offset addr in input data area
bSrvProdConnSize	UINT8	0 - 255	produced I/O connection size as server
usProdOffset	UINT16	0 - 3583	offset addr in output data area
abReserved[58]	UINT8[]	0x00	Reserved set to zero

Table 65: Server Parameter Structure

Source Code Example

```
TLR_VOID DnApInitServerParameter( DN_FAL_SRV_PRM_T *ptSrvPrm )
{
  TLR_MEMSET( ptSrvPrm,0x00,sizeof(DN_FAL_SRV_PRM_T));

  ptSrvPrm->bSrvConsConnSize  = 8;     // 8 Byte receive data
  ptSrvPrm->usConsOffset      = 512;   // Consumed offset
  ptSrvPrm->bSrvProdConnSize  = 8;     // 8 Byte send data
  ptSrvPrm->usConsOffset     = 512;   // Produce offset
}
```

5.4.1.5 Coding of the Slave Parameter Structure

The parameter structure for one slave parameter set is a combination of different structures. Refer to following overview how this structure is assembled. The slave parameter is transferred in the data portion within the DN_FAL_DOWNLOAD_REQ packet.

Slave Parameter Assembly

```
| - DN_DEV_PRM_HEADER_T
|
+ - DN_PRED_MSTSL_CFG_DATA_T
|   |
|   + - DN_PRED_MSTSL_CONNOBJ_T (1st)
|   |   + - DN_PRED_MSTSL_IO_OBJ_HEADER_T
|   |   + - DN_IO_MODULE (1st)
|   |   + - DN_IO_MODULE (2nd)
|   |   + - …            (n)
|   |
|   + - DN_PRED_MSTSL_CONNOBJ_T (2nd)
|   |   + - DN_PRED_MSTSL_IO_OBJ_HEADER_T
|   |   + - DN_IO_MODULE (1st)
|   |   + - DN_IO_MODULE (2nd)
|   |   + - …            (n)
|   |
|   + - … (n)
|
| - DN_PRED_MSTSL_ADD_TAB_T
|
+ - DN_EXPL_SET_ATTR_DATA_T
|   |
|   + - DN_SET_ATTR_DATA_T (1st)
|   + - DN_SET_ATTR_DATA_T (2nd)
|   + - …                  (n)
|
+ - DN_UCMM_CONN_OBJ_CFG_DATA (not supported)
+ - DN_UCMM_CONN_OBJ_ADD_TAB  (not supported)
```

Header Structure	Variable name	Type	Explanation
DN_DEV_PRM_HEADER	usDevParaLen	word	Length of whole data set inclusive the length (2 bytes) of the size indicator
	bDvFlag	byte	Bit field, to activate the parameter data set and to indicate which kind of connections are supported, enable disable device keying
	bUcmmGroup	byte	If device supported Dynamic Connections(UCMM) the connection group must specified here
	usRecFragTimer	word	Timer value for in multiples of millisecond: - slave device times out and DeviceNet-Master tries the reconnection - maximum timeout duration in fragmentation protocol between two fragments.
	usVendorID	word	Identification of vendor, managed by ODVA
	usDeviceType	word	Indication of general type of product. List is available in DeviceNet spec
	usProductCode	word	Identification of particular product
	bMajorRevision	byte	Major revision of device
	bMinorRevision	byte	Minor revision of device
	bOctetString	2 byte	Reserved bytes
DN_PRED_MSTSL_CFG_DATA_T	usPredMstSlCfg DataLen	word	Length of the following predefined master slave connection field inclusive the length (2 bytes) of the size indicator
	DN_PRED_MSTSL_ CONNOBJ_T	structure	Predefined connections which shall be established to this device see corresponding HEADER file for structure
DN_PRED_MSTSL_ADD_TAB_T	usAddTabLen	word	Length of the following additional process data offset table data inclusive the length (2 bytes) of the size indicator
	bInputCount	byte	Number of input offset addresses in the offset table
	bOutputCount	byte	Number of output offset addresses in the offset table
	ausIOOffsets	word array	Offset address table for I/O data in the dual-port memory I/O area
DN_EXPL_SET_ATTR _DATA_T	usAttrDataLen	word	Length of the following set attribute data field inclusive the length (2 bytes) of the size indicator
	DN_SET_ATTR_ DATA_T	structure	Attributes which shall be changed via explicit messaging before doing I/O messaging see corresponding HEADER file for structure
DN_UCMM_CONN_OBJ_CFG_DATA_T	usCfgDataLen	word	Length of the UCMM connection field inclusive the length (2 bytes) of the size indicator. Structure currently not supported. So length must be set to 2.
DN_UCMM_CONN_OBJ_ADD_TAB_T	usAddTabLen	word	Length of the following additional process data offset table data inclusive the length (2 bytes) of the size indicator. Structure currently not supported, so the length value must be set to 4 fixed.
	bInputCount	byte	Number of input offset addresses in the offset table
	bOutputCount	byte	Number of output offset addresses in the offset table

Header Structure	Variable name	Type	Explanation
	ausIOOffsets	word array	Offset address table for I/O data in the dual-port memory I/O area. Not supported

Table 66: Slave Parameter Structure

The main length indicator usDevParaLen fixes the length of the whole data block including the length of the size indicator itself. The length can be calculated with the formula:

Length Calculation of Slave Parameter

```
usDevParaLen = 2                          // size indicator itself
               + 14                       // header bytes
               + usPredMstSlCfgDataLen    // Predefined MstSl Config data
               + usAddTabLen              // Address table length
               + usAttDataLen             // ExplSetAttrData length
               + 2                        // usCfgDataLen fixed
               + 4                        // usAddTabLen fixed
…
```

This variable is followed by a special bit field called bDvFlag, declaring the parameter data set either as active or inactive. Only if the 'ACTIVE' bit (D7) is set, the DeviceNet-Master will actually activate the network access for this device.

bDvFlag

D7	D6	D5	D4	D3	D2	D1	D0
ACTIVE	RES	KEY_ REV	KEY_ PCODE	KEY_ DTYPE	KEY_ VENDOR	QUICK CONNECT	UCMM_SUPP

Table 67: bDvFlag

The bit UCMM_SUPP indicates whether the device supports the dynamic establishment of explicit messaging connections. Depending on this bit value, the DeviceNet Master will try to establish the explicit connection via unconnected services twice (1 second duration time between each try) before it will start to try it via predefined master slave connection services. Setting this bit for a slave device which does not support UCMM has the disadvantage that at least 2 seconds are lost until the devices connection can be established. The bits 'KEY_XXX' are enabling or disabling the electronic keying of a device. If enabled, the master checks the corresponding values usVendorID, usDeviceType, usProductCode, bMajorRevision and bMinorRevision from the device data set with the current physical values in the Identity Object of the device during its initialization. If one of these values differs, the connection to the device is denied because of safety reasons. The bit 'ACTIVE' activates or inactivates the data set. If the DeviceNet Master is in mode OPERATE and the bit is cleared, then the device is not handled and remains in unconnected state for the specific device. If the bit 'QUICK CONNECT' is set, this indicates that the corresponding slave is marked within the master configuration as a quick connect node. If there is only one single quick connect node present, then the quick connect feature of the master is also activated. When a node has been marked as quick connect node, the master will concurrently try to send UCMM and Alloc Master/Slave requests. Depending on the response, an explicit connection will be established via UCMM or Group 2 Only Explicit Connection Port. For further details, please refer to the CIP Networks Library, vol. 3, section 2.3.4, 3.15-1 (reference [3]).

The parameter bUcmmGroup defines the message group across which messages associated with this connection are to be exchanged if the device is UCMM capable. The following values are defined:

```
#define DN_GROUP_1   0
#define DN_GROUP_2   1
#define DN_GROUP_3   3
```

Remember that the device itself selects the message group later across which the connection shall take place when the connection will be established. If the device cannot satisfy the intended bUcmmGroup selection, then it rejects the request and returns an error response. The value usRecFragTimer defines the "wait for acknowledgement" timer value which supervises after each transferred explicit request message the incoming corresponding response message. Furthermore the timer value defines the duration timer value between 2 requests that is used by the DeviceNet Master to retry the establishment of the devices connection after the connection was lost. The value range goes from 1 to 65535. The DN_PRED_MSTSL_CFG_DATA_T structure informs the DeviceNet Master about the I/O connection that shall be established to the device and about the amount of produced or consumed I/O bytes that is transferred. The explicit message channel is established to each device anyway and must not be configured. So if it is desired, only the explicit message channel shall be established by the DeviceNet Master, then the DN_PRED_MSTSL_CFG_DATA_T structure must be left empty. That means the parameter usPredMstSlCfgDataLen must be set to the value 2.

Header Structure	variable name	type	explanation
DN_PRED_MSTSL_CONNOBJ	bConnectionType	byte	type of connection that shall be established
	bWatchdogTimeout action	byte	behavior of the device if watchdog timeout expires through this connection
	usProdInhibitTime	word	minimum delay between new data production in multiples of 1ms
	usExpPacketRate	word	watchdog timer value of this connection
	bNumOfIOModules	byte	number of following defined process data modules
	DN_IO_MODULE_T	structure	process data module definition
	... for multiple connections	...	...
DN_PRED_MSTSL_CONNOBJ	bConnectionType	byte	type of connection that shall be established
	bWatchdogTimeout action	byte	behavior of the device if watchdog timeout expires through this connection
	usProdInhibitTime	word	minimum delay between new data production in multiples of 1ms
	usExpPacketRate	word	watchdog timer value of this connection
	bNumOfIOModules	byte	number of following defined process data modules
	DN_IO_MODULE_T	structure	process data module definition

Table 68: Additional Header Structures

It is possible to configure multiple I/O connections to a device, like the structure above shows, but this is not usual to have in DeviceNet. Usually, there is only one single connection to a device.

See the structures

- DN_PRED_MSTSL_CONNOBJ_T
- DN_PRED_MSTSL_IO_OBJ_HEADER_T
- DN_I_O_MODULE_T

in the header file.

The bConnectionType defines which type of I/O connection shall be established to this device. The following values are defined:

```
#define DN_TYPE_CYCLIC 0x08
#define DN_TYPE_CHG_OF_STATE 0x04
#define DN_TYPE_BIT_STROBED 0x02
#define DN_TYPE_POLLED 0x01
#define DN_TYPE_CHG_OF_STATE_ACK 0x10
#define DN_TYPE_CYLIC_ACK 0x20
```

The bWatchdogTimeoutAction defines the behavior if the watchdog timer of the I/O connection expires. The following values are defined and their functionality is closer described in the DeviceNet specification.

```
#define DN_TRANSMISSION_TO_TIMEOUT 0x00
#define DN_AUTO_DELETE 0x01
#define DN_AUTO_RESET 0x02
```

The parameter usProdInhibitTime, one for each connection, configures the minimum delay time between new data production in multiples of a millisecond. The timer is reloaded each time new data production through the established connection occurs. While the timer is running, the DeviceNet Master suppresses new data production until the timer has expired. This method prevents that the device is overloaded with requests coming in too fast. The value 0 defines no inhibit time and data production can and will be done as fast as possible. If in polled mode, for example, a usProdInhibitTime of 1000 (decimal) is configured, then the poll request message to the device will be sent every second. The usExpPacketRate, one for each connection, is always transferred to the device before starting the I/O transfer. The value is used by the device later to reload its 'Transmission Trigger' and 'Watchdog Timer'. The 'Transmission Trigger Timer' is used in a 'cyclic' I/O connection to control the time when the data shall be produced. Expiration of this timer then is an indication that the associated connection must transmit the corresponding I/O message. In 'change of state' connections the timer is used to avoid the watchdog timeout in this connection, when a production has not occurred since the timer was activated or reloaded. Remark: the usProdInhibitTime is verified against the usExpPacketRate. If the usExpPacketRate value is unequal zero but less than the usProdInhibitTime value, then an error is returned.

The bNumOfIOModules value defines how many so called Input / Output modules are defined in the directly following structure DN_IO_MODULE_T. The table DN_IO_MODULE_T defines which type of process data and in sum how many data shall be transferred through this connection. The structure DN_IO_MODULE_T itself has a size of two bytes per defined I or O module. The first byte of each structure contains the bDataType of the I/O module. Following values are defined:

```
#define DN_DT_BOOLEAN  1   /* bit module */
#define DN_DT_UINT8         2   /* byte module */
#define DN_DT_UINT16       3   /* word module */
#define DN_DT_UINT32       4   /* long module */
#define DN_DT_STRING     10  /* byte array module */
```

To distinguish whether the module is either an output (consumed data in the view of the device) or an input one (produced data in the view of the device) in the view of the DeviceNet Master, the upper bit in the bDataType decides the data direction. If the bit is set then the module is defined as an output module. #define DN_OUTPUT 0x80. The bDataType is followed by the bDataSize indicator of the module. Except for the bDataType DN_DT_STRING, all other bDataTypes have fixed bDataSize values. The DeviceNet Master checks whether the type and its size correspond.

```
DN_DT_BOOLEAN -> bDataSize = 1
DN_DT_UINT8 -> bDataSize = 1
DN_DT_UINT16 -> bDataSize = 2
DN_DT_UINT32 -> bDataSize = 4
```

In case of DN_DT_STRING the bDataSize value can have a value range from 1 to 255. There are no limitations how many modules can be configured in the table. Also the DeviceNet Master does not care for the order of the modules' definition especially not if input and output modules are defined in a mixed way.

One entry in the DN_IO_MODULE_T table must result in a corresponding entry in the DN_PRED_MSTSL_ADD_TAB_T structure which contains the dual-port memory offset address where to store the module data in case of input and where the read out the module data in case of output. The sizes of all configured modules are added up separately for input and output and the sum represents the 'Produced_' and 'Consumed_Connection_Size' of this I/O connection.

Both values are compared while the startup procedure of the device with its real 'Produced_' and 'Consumed_Connection_Size' by reading it with the 'Get_Attribute' DeviceNet command. If one of the summarized values is not equal to the read real present value, then the access to this device is denied totallly. If a device for example has a Produced_Connection_Size of 8 bytes and a Consumed_Connection_Size of 3 bytes then the IO_Modules table could look like this:

variable name	contents
Num_Of_IO_Modules	02 hex = 2 modules
bDataType	0A hex = DN_DT_STRING, input
bDataSize	08 hex = 8 bytes
bDataType	8A hex = DN_DT_STRING, output
bDataSize	03 hex = 3 bytes

Table 69: Example for IO_Modules table

If the device data shall be handled transparently as byte strings in the process data image areas of the DeviceNet Master in both directions, the table DN_PRED_MSTSL_ADD_TAB_T must contain only 2 process data offset addresses. But it is possible to divide the process data into multiple modules. This makes sense if the device itself is for example a pluggable I/O rack, which can contain modular I/O submodules like analog or digital ones. In mixed constellations, the analog values can be placed in a different location within the process image than the digital values.

Then the table could have the following entries:

variable name	contents
Num_Of_IO_Modules	06 hex = 6 modules
bDataType	06 hex = DN_DT_UINT16, input
bDataSize	02 hex = 2 byte
bDataType	05 hex = DN_DT_UINT8, input
bDataSize	01 hex = 1 byte
bDataType	86 hex = DN_DT_UINT16, output
bDataSize	02 hex = 2 byte
bDataType	05 hex = DN_DT_UINT8, input
bDataSize	01 hex = 1 byte
bDataType	85 hex = DN_DT_UINT8, output
bDataSize	01 hex = 1 byte
bDataType	07 hex = DN_DT_UINT32, input
bDataSize	04 hex = 4 byte

Table 70: Another Example for IO_Modules table

In this example the table DN_PRED_MSTSL_ADD_TAB_T must contain 6 process data offset addresses, namely one for each module. See structure DN_IO_MODULE_T in the header file. One entry in the DN_IO_MODULE_T table must result a corresponding entry in the DN_PRED_MSTSL_ADD_TAB_T. In this table the offset address in the dual-port memory of each process data is stored, where the DeviceNet-Master later has to start the reading of the data as outputs and sending it to the device or starts to write it into as inputs when an input message has been received. See the following structure:

variable name	type	contents
usAddTabLen	word	Length of the following additional process data offset table data inclusive the length (2 bytes) of the size indicator.
bInputCount	byte	number of inputs following in the IO_Offset table
bOutputCount	byte	number of outputs following in the IO_Offset table
ausIOOffsets[...]	word array	IO_Offsets in the order: first all input offsets then all output offsets

Table 71: Structure of DN_PRED_MSTSL_ADD_TAB_T

Also see structure DN_PRE_MSTSL_ADD_TAB_T in the header file. The ausIOOffsets have to be placed in order to each configured I/O module in the table DN_IO_MODULE_T so that the DeviceNet Master can establish a relationship between both tables and can associate them together later when performing the I/O exchange. For an output process data module a corresponding output offset results, for an input process data module a corresponding input offset results. If inputs and outputs are configured at the same time, the offset table must contain first all input offsets and then all output offsets. All offsets must be configured as byte offsets, except an offset for a single bit DN_DT_BOOLEAN process data. There an offset must be set up like the following figure illustrates:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Bit offset			Byte offset												
			byte offset in the dual port memory (0 - 3583 decimal)												
Corresponding bit offset in the byte 0 -7dec															

Table 72: Structure of IO_Offsets

To complete the last example above, here is a related DN_PRE_MSTSL_ADD_TAB_T example: The 4 input modules data shall be located at byte 4-5dec,0,20dec,3000-3003dec. The word output module shall be located at byte 10dec-11dec and the byte output module at byte 2dec.

variable name	contents
usAddTabLen	0010 hex = 16 bytes in length, inclusive the length indicator
bInputCount	04 hex = 4 input offsets following
bOutputCount	02 hex = 2 output offsets following
ausIOOffsets[0]	0004 hex = word input module, byte 4-5
ausIOOffsets[1]	0000 hex = byte input module, byte 0
ausIOOffsets[2]	0014 hex = byte input module, byte 20
ausIOOffsets[3]	0BB8 hex = long input module, byte 3000-3003
ausIOOffsets[4]	000A hex = word output module, byte 10-11
ausIOOffsets[5]	0002 hex = byte output module, byte 2

Table 73: Structure of DN_PRE_MSTSL_ADD_TAB_T

The DN_PRED_MSTSL_CFG_DATA_T structure represents the parameter data containing the structure of each device. The DeviceNet Master takes all configured attributes with their configured values from this table and compares the contents during the device's startup with the physically present attribute values of the device. If these values do not match, the DeviceNet Master sends the configured new value to the device with the DeviceNet *'Set_Attribute'* write command. DNM devices normally store such parameter data into an electronic erasable PROM, which has limitations in its write procedures. This mechanism *'write if different*' guarantees that the parameter data is not transferred useless and the PROM would not be programmed in vain to hold its life expectancy high. If a write command would be denied by the device, the DeviceNet Master does not stop the parameterization and finalizes it by taking the next configured entry. A denied write request will only be logged in the internal device specific diagnostic structure, which can be read out with the DNM_Device_Diag message.

Header Structure	variable name	type	explanation
DN_SET_ATTR_DATA_T	usClass ID	word	class which shall be addressed, 0- 65535
	usInstanceID	word	instance in this class which shall be addressed 0-65535
	bAttributeID	byte	attribute which shall be addressed, 0- 255
	bDataCnt	byte	length of following data 0-255
	abData[...]	octet string	data field
	... if more attributes shall be changed	...	...
DN_SET_ATTR_DATA_T	usClass ID	word	class which shall be addressed, 0- 65535
	usInstanceID	word	instance in this class which shall be addressed 0-65535
	bAttributeID	byte	attribute which shall be addressed, 0- 255
	bDataCnt	byte	length of following data 0-255
	abData[...]	octet string	data field

Table 74: Structure of DN_SET_ATTR_DATA_T

See structure DN_SET_ATTR_DATA_T in the header file. There are no limitations how many attributes can be configured in the table. Each attribute is addressed in DeviceNet and in each entry by its usClassID, usInstanceID location and its bAttributeID. The number of data bytes which shall be compared and written if unequal is fixed in the variable bDataCnt. If no attributes shall be changed while the device's network startup procedure is executed, then the leading length indicator usAttrDataLen must contain the value 2 which represents the byte length of the indicator itself then.

If for example the attribute 1 = 'MAC_ID' in DeviceNet object class 3 instance 1 shall be changed to the value 10 when the device is started up later, the DN_EXPL_SET_ATTR_DATA_T entry should have the following data entries:

Variable name	Contents
usAttrDataLen	0009 hex
usClassID	0003 hex
usInstanceID	0001 hex
bAttributeID	01 hex
bDataCnt	01 hex
abData[0]	10 hex

Table 75: Example for DN_EXPL_SET_ATTR_DATA_T

The next table DN_UCMM_CONN_OBJ_CFG_DATA_T is used for further extension of the firmware of the DeviceNet Master. Because of this, only the size of the structure usCfgDataLen is defined at the moment in a device data set. The value in there must be set up to 2 to be compatible in future with incoming firmware versions. Based on the same reason, the corresponding DN_UCMM_CONN_OBJ_ADD_TAB_T and its length must be reserved also in the device data set. usAddTabLen must be set up to a length of 4 fixed.

Example of Message Device Parameter Data Sets Hexadecimal

```
TLR_RESULT InitGetSlaveParameter( TLR_UINT32 ulSlvNum, TLR_UINT8** ppabSlvPrm,
                                  TLR_UINT32 *pulPrmLen )
{
  //  - No Input and output process data, explicit only connection
  STATIC TLR_UINT8 abSlvSet1{} =
  {
    1E,00,80,00,e8,03,00,00,00,00,00,00,00,00,00, //DN_DEV_PRM_HEADER_T
    02,00, //DN_PRED_MSTSL_CFG_DATA_T, empty, only length itself = 2
    04,00,00,00, //DN_PRED_MSTSL_ADD_TAB_T no input and output offset
    02,00, //DN_SET_ATTIBUTE_DATA_T, empty table, only length itself = 2
    02,00, //DN_UCMM_CONN_OBJ_CFG_DATA_T, empty table, only length itself = 2
    04,00,00,00 //DN_UCMM_CONN_OBJ_ADD_TAB_T no offsets
  }
  //  - Polling connection, 7 byte input , 8 bytes output
  STATIC TLR_UINT8 abSlvSet2{} =
  {
    2D,00,80,00,40,06,00,00,00,00,00,00,00,00,00, //DN_DEV_PRM_HEADER_T
    0D,00,01,00,0A,00,00,00,02,0A,07,8A,08, //DN_PRED_MSTSL_CFG_DATA_T
    08,00,01,01,00,00,00,00, //DN_PRED_MSTSL_ADD_TAB_T offset = 0
    02,00, //DN_SET_ATTIBUTE_DATA_T, empty table, only length itself = 2
    02,00, //DN_UCMM_CONN_OBJ_CFG_DATA_T, empty table, only length itself = 2
    04,00,00,00 //DN_UCMM_CONN_OBJ_ADD_TAB_T no offsets
  }
  //  - Polling connection UCMM group 3 capable device, 7 byte input 8 bytes output
  TLR_UINT8 abSlvSet3{} =
  {
    2D,00,81,03,40,06,00,00,00,00,00,00,00,00,00, //DN_DEV_PRM_HEADER_T
    0D,00,01,00,14,00,1E,00,02,0A,07,8A,08, //DN_PRED_MSTSL_CFG_DATA_T
    08,00,01,01,00,00,00,00, //DN_PRED_MSTSL_ADD_TAB_T offset = 0
    02,00, //DN_SET_ATTIBUTE_DATA, empty table, only length itself = 2
    02,00, //DN_UCMM_CONN_OBJ_CFG_DATA_T, empty table, only length itself = 2
    04,00,00,00 //DN_UCMM_CONN_OBJ_ADD_TAB_T no offsets
  }
  //  - Bit-Strobe connection, 8 bytes input
  TLR_UINT8 abSlvSet3{} =
  {
    29,00,80,00,40,06,00,00,00,00,00,00,00,00,00, //DN_DEV_PRM_HEADE_T R
    0B,00,02,00,14,00,1E,00,01,0A,08, //DN_PRED_MSTSL_CFG_DATA_T
    06,00,01,00,00, //DN_PRED_MSTSL_ADD_TAB_T input offset = 0
    02,00, //DN_SET_ATTIBUTE_DATA, empty table, only length itself = 2
    02,00, //DN_UCMM_CONN_OBJ_CFG_DATA_T, empty table, only length itself = 2
    04,00,00,00 //DN_UCMM_CONN_OBJ_ADD_TAB_T no offsets
  }
  switch(ulSlvNum ) {
   case 1:  {*ppabSlvPrm = &abSlvSet1[0]; *pulPrmLen = sizeof(abSlvSet1);} break;
   case 2:  {*ppabSlvPrm = &abSlvSet2[0]; *pulPrmLen = sizeof(abSlvSet2);} break;
   case 3:  {*ppabSlvPrm = &abSlvSet3[0]; *pulPrmLen = sizeof(abSlvSet3);} break;
   case 4:  {*ppabSlvPrm = &abSlvSet4[0]; *pulPrmLen = sizeof(abSlvSet4);} break;
   default: {*ppabSlvPrm =NULL; *pulPrmLen = 0;} break;
  }
}
```

5.4.1.6 Coding of the Bus Parameter Structure

The structure DN_FAL_BUS_PRM_T contains the bus parameters of the stack which are available to be set. The bus parameters are transferred into the data portion within the DN_FAL_DOWNLOAD_REQ packet. After sending the bus parameters the configuration process is finished. It is not possible to send any further slave, server or device parameters afterwards.

Structure Reference

```
typedef struct DN_FAL_BUS_PRM_Ttag
{
  TLR_UINT32   ulSystemFlags;     /* System flags                          */
  TLR_UINT32   ulWdgTime;         /* Watchdog                              */
  TLR_UINT32   ulOwnMacId;        /* Device MAC ID                         */
  TLR_UINT32   ulBaudrate;        /* Device Baudrate                       */
  TLR_UINT32   ulConfigFlags;     /* Configuration flags                   */
  TLR_UINT32   ulEnableFlags;     /* Enable Flags                          */
  TLR_UINT8    abReserved[40];    /* Reserved                              */
} DN_FAL_BUS_PRM_T;
```

Variable	Type	Value	Description
ulSystemFlags	UINT32	0x00000000 - 0x00000001	System Flags
ulWdgTime	UINT32	0x00000014 - 0x0000FFFF 0x00000000	Watchdog Time. Deactivated if set to 0.
ulOwnMacId	UINT32	0 .. 63	Own MAC ID
ulBaudrate	UINT32	1 .. 3	Baud rate 1 = DN_FAL_BAUDRATE_500 2 = DN_FAL_BAUDRATE_250 3 = DN_FAL_BAUDRATE_125
ulConfigFlags	UINT32	0x00000000 - 0x00000003	Configuration Flags
ulEnableFlags	UINT32	0x00000000	Reserved, set to zero
abReserved[..]	UINT8[]	0x00	Reserved, set to zero

Table 76: Bus Parameter Structure

Flag	Flag Name DN_FAL_MSK_BUS_PRM_SYS_FLAG	Description
0x00000001	_CTRL_RELEASE	If this bit is set, the communication starts when application is ready If not set, it starts automatically to communicate
0xFFFFFFFE	_FLAG_RESERVED	Reserved Flags always set to zero

Table 77: System Flags

Flag	Flag Name DN_FAL_MSK_BUS_PRM_CFG_FLAG	Description
0x00000001	_AUTOCLEAR	The flag defines the system behavior if one device classified as active has been disconnected. If AutoClear mode is active, then the DeviceNet-Master stops the communication to all other devices too, else it keeps running and tries to restart the missed device.
0xFFFFFFFE	_FLAG_RESERVED	Reserved Flags, always set to zero

Table 78: Configuration Flags

Source Code Example

```
TLR_VOID InitBusParameter( DN_FAL_BUS_PRM_T *ptBusPrm )
{
  TLR_MEMSET( ptBusPrm,0x00,sizeof(DN_FAL_BUS_PRM_T));
  ptBusPrm->ulOwnMacId        = 1;                    // Own MAC ID
  ptBusPrm->usProductCode     = DN_FAL_BAUDRATE_125; // Baudrate
}
```

5.4.2 Clear Configuration Service DEVNET_FAL_CMD_CLR_CONFIG_REQ/CNF

This packet clears the configuration area. This packet has the same effect as the packet RCX_DELETE_CONFIG_REQ. Please refer to reference [1] for further information.

Packet Structure Reference

```
/** SDU: Clear Config req/cnf ***********************************************/
typedef struct DN_FAL_SDU_CLR_CONFIG_REQ_Ttag
{
  TLR_UINT32 ulCfgArea;
}DN_FAL_SDU_CLR_CONFIG_REQ_T;

/** PACKET: Clear configuration req/con *************************************/
typedef struct DN_FAL_PACKET_CLEAR_CONFIG_REQ_Ttag
{
  TLR_PACKET_HEADER_T           tHead;                  /* Packet Header     */
  DN_FAL_SDU_CLR_CONFIG_REQ_T   tData;                  /* Packet data       */
} DN_FAL_PACKET_CLEAR_CONFIG_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_CLEAR_CONFIG_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3806	DEVNET_FAL_CMD_CLR_CONFIG_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_CLR_CONFIG_REQ_T			
ulCfgArea	UINT32		Configuration area. Not used. Set to 0.

Table 79: DN_FAL_PACKET_CLEAR_CONFIG_REQ_T – Clear Configuration Request

Packet Structure Reference

```
typedef struct DN_FAL_PACKET_CLEAR_CONFIG_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                /* Packet Header     */
} DN_FAL_PACKET_CLEAR_CONFIG_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_CLEAR_CONFIG_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3807	DEVNET_FAL_CMD_CLR_CONFIG_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 80: DN_FAL_PACKET_CLEAR_CONFIG_CNF_T – Confirmation of Clear Configuration

5.5 Controlling/Monitoring/Diagnosis of the Stack Task

5.5.1 Stack Initialization Service DEVNET_FAL_CMD_INIT_REQ/CNF

The initialization request is used to force the DeviceNet protocol stack into a determined state, e.g. to reset the stack. The mode of the init command determines the method and initialization action.

Currently only the value 0 (*"Reset same as after a power on"*) is supported. See also Configuration Services.

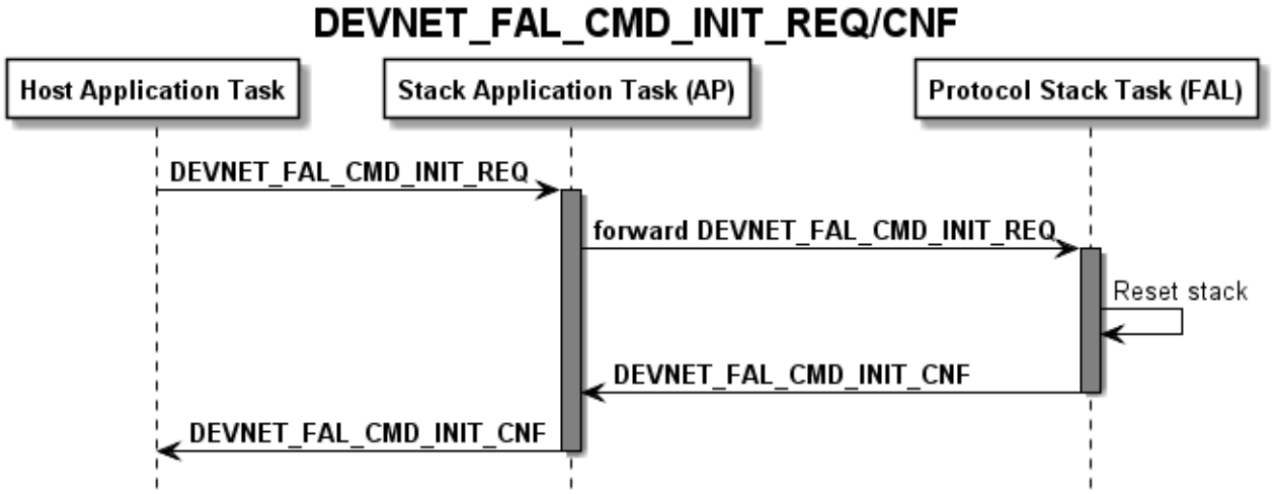

Figure 12: Sequence diagram of the DEVNET_FAL_CMD_INIT_REQ

Packet Structure Reference

```
#define DN_FAL_INIT_MODE_RESET        0x00000000

typedef struct DN_FAL_SDU_INIT_REQ_Ttag {
  TLR_UINT32  ulMode;
} DN_FAL_SDU_INIT_REQ_T;

typedef struct DN_FAL_PACKET_INIT_REQ_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_INIT_REQ_T tData;
} DN_FAL_PACKET_INIT_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_INIT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	`sizeof(DN_FAL_SDU_INIT_REQ_T)`
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3800	`DEVNET_FAL_CMD_INIT_REQ` - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing, set to zero
tData - Structure DN_FAL_SDU_INIT_REQ_T			
ulMode	UINT32	0	`DN_FAL_INIT_MODE_RESET`

Table 81: `DN_FAL_CMD_INIT_REQ` – Request Command for DN Init

Source Code Example

```
TLR_RESULT DnUser_StackReset_Req(DN_USER_RSC_T FAR* ptRsc)
{
  TLR_RESULT eRslt;
  DN_FAL_PACKET_INIT_REQ_T  *ptInitReq;

  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, &ptInitReq);
  if( eRslt == TLR_S_OK )
  {
    TLR_QUE_LINK_SET_PACKET_SRC(ptRegAppReq,ptRsc->tLoc.tQueSrcDnApp);
    ptInitReq->tHead.ulCmd = DEVNET_FAL_CMD_INIT_REQ;
    ptInitReq->tHead.ulLen = sizeof(ptInitReq->tData);

    /* Set the reset mode */
    ptRegAppReq->tData.ulMode = DN_FAL_INIT_MODE_RESET;

    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tLoc.tDnFalQue,ptInit,TLR_FINITE);
    if( eRslt != TLR_S_OK )
    {
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptInitReq);
    }
  }
  return eRslt;
}
```

Packet Structure Reference

```
typedef struct DN_FAL_SDU_INIT_CNF_Ttag {
  TLR_UINT32  ulMode;
} DN_FAL_SDU_INIT_CNF_T;

typedef struct DN_FAL_PACKET_INIT_CNF_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_INIT_CNF_T tData;
} DN_FAL_PACKET_INIT_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_INIT_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, untouched
ulSrc	UINT32		Source Queue-Handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, untouched
ulLen	UINT32	4	`sizeof(DN_FAL_SDU_INIT_CNF_T)`
ulId	UINT32	0 ... 2^{32}-1	Packet Identification, untouched
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3801	`DEVNET_FAL_CMD_INIT_CNF` - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	×	Routing, do not touch
tData - Structure DN_FAL_SDU_INIT_CNF_T			
ulMode	UINT32	x	returned, untouched

Table 82: `DEVNET_FAL_CMD_INIT_CNF` – Confirmation of Init Request

5.5.2 Set Operation Mode Service DEVNET_FAL_CMD_SET_MODE_REQ/CNF

The command DEVNET_FAL_CMD_SET_MODE_REQ is used to set the operational mode of the DevNet FAL -Task.

Valid modes are:

- OFFLINE
  The DeviceNet Master is not active on the network and is not accessible for other devices.
- STOP
  The DeviceNet Master performs the Duplicate-MAC Id check procedure, but does not start any I/O communication. The master is accessible for other devices with explicit messaging
- IDLE
  Currently the same as RUN
- RUN
  The DeviceNet Master starts I/O communication to all configured slaves.

See also *Configuration Services*.

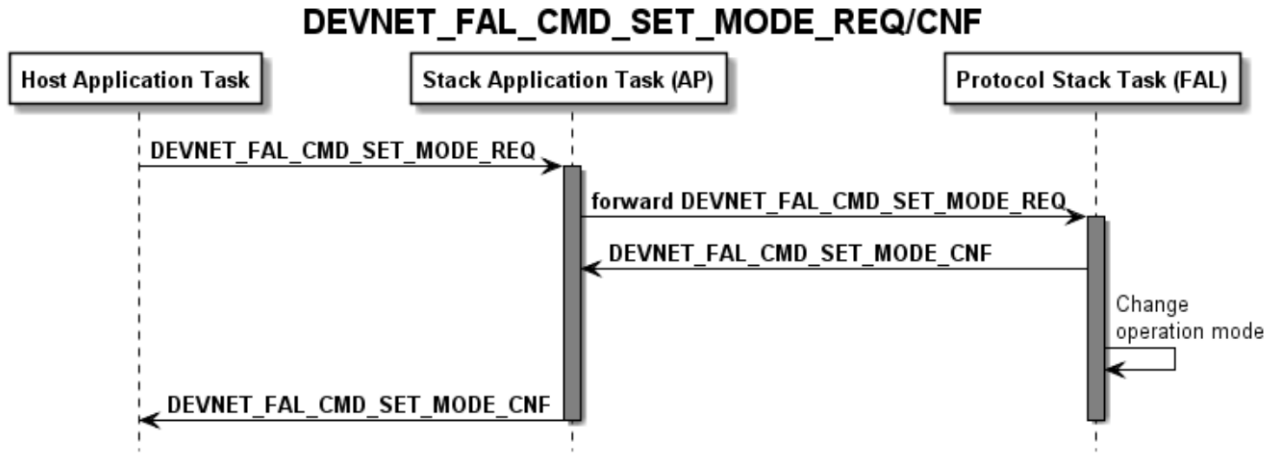

Figure 13: Sequence diagram of the DEVNET_FAL_CMD_SET_MODE_REQ packet

Packet Structure Reference

```
#define DN_FAL_MODE_OFFLINE   (0x00)
#define DN_FAL_MODE_STOP      (0x40)
#define DN_FAL_MODE_IDLE      (0x80)
#define DN_FAL_MODE_RUN       (0xC0)

typedef struct DN_FAL_SDU_SET_MODE_REQ_Ttag {
  TLR_UINT32  ulMode;
  TLR_UINT8   ubExtErrCode;
} DN_FAL_SDU_SET_MODE_REQ_T;

typedef struct DN_FAL_PACKET_SET_MODE_REQ_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SET_MODE_REQ_T tData;
} DN_FAL_PACKET_SET_MODE_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_SET_MODE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, returned untouched
ulSrcId	UINT32	0 ... 2^{32}-1	AP Process Source End Point Identifier
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Packet Status/Error set to zero
ulCmd	UINT32	0x3804	DEVNET_FAL_CMD_SET_MODE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing set to zero
tData - Structure DN_FAL_SDU_SET_MODE_REQ_T			
ulMode	UINT32	 0x00 0x40 0x80 0xC0	Mode of operation. DN_FAL_MODE_OFFLINE DN_FAL_MODE_STOP DN_FAL_MODE_IDLE DN_FAL_MODE_RUN
ubExtErrCode	UINT8		Extended Error Code. Used internally by AP task.

Table 83: DEVNET_FAL_CMD_SET_MODE_REQ – Request Command for setting the DevNet Operation Mode

Source Code Example

```
TLR_RESULT DnUser_Stop_Req(DN_USER_RSC_T FAR* ptRsc)
{
  TLR_RESULT eRslt = TLR_S_OK;
  DN_FAL_PACKET_SET_MODE_REQ_T   *ptSetModeReq = NULL;

  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool, & ptSetModeReq);
  if( eRslt == TLR_S_OK )
  {
    TLR_QUE_LINK_SET_PACKET_SRC(ptSetModeReq,ptRsc->tLoc.tQueSrcApp);
    ptSetModeReq->tHead.ulCmd = DEVNET_FAL_CMD_SET_MODE_REQ;
    ptSetModeReq->tHead.ulLen = sizeof(DN_FAL_SDU_SET_MODE_REQ_T);

    /* Set the mode to stop */
    ptSetModeReq->tData.ulMode = DN_FAL_MODE_STOP;

    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tLoc.tDstQue,ptSetModeReq,TLR_FINITE);
    if( eRslt != TLR_S_OK )
    {
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptInitReq);
    }
  }
  return eRslt;
}
```

Packet Structure Reference

```
typedef struct DN_FAL_SDU_SET_MODE_CNF_Ttag {
  TLR_UINT32 ulMode;
} DN_FAL_SDU_SET_MODE_CNF_T;

typedef struct DN_FAL_PACKET_SET_MODE_CNF_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_SET_MODE_CNF_T tData;
} DN_FAL_PACKET_SET_MODE_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_SET_MODE_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle, returned untouched
ulSrc	UINT32		Source Queue-Handle of AP Process Queue, returned untouched
ulDestId	UINT32	0	Destination End Point Identifier, returned untouched
ulSrcId	UINT32	0 ... 2^{32}-1	AP Process Source End Point Identifier, returned untouched
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification, untouched
ulSta	UINT32	0 ... 2^{32}-1	*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3805	DEVENT_FAL_CMD_SET_MODE_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	×	Routing, do not touch
tData - Structure DN_FAL_SET_MODE_CNF_T			
ulMode	UINT32	0 ... 3	Confirmation of mode set. Reflexion of ulMode in request packet. DN_FAL_MODE_OFFLINE DN_FAL_MODE_STOP DN_FAL_MODE_IDLE DN_FAL_MODE_RUN

Table 84: DEVNET_FAL_CMD_SET_MODE_CNF – Confirmation of DN Set Mode Command

5.5.3 Parameter Upload Service DEVNET_FAL_CMD_UPLOAD_REQ/CNF

The command DEVNET_FAL_CMD_UPLOAD_REQ is used to upload a configuration (i.e. a complete set of parameters) from the DeviceNet stack to the application. The upload is divided into so called areas like bus, device, slave or server parameter sets.

The same area definitions apply as in the context of the Download Configuration DEVNET_FAL_CMD_DOWNLOAD_REQ/CNF command; see below in the description of the parameters.

The data delivered in the abData[1000] array of the confirmation packet is structured exactly as described in sections 5.4.1.3 to 5.4.1.6 of this document:

- DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER (corresponds to the device parameter structure DN_FAL_DEV_PRM_T, see subsections 5.4.1.3"*Coding of the Device Parameter Structure*" and 4.5.3"*Detailed Description of Device Parameters*")
- DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER (optional, corresponds to the server parameter structure DN_FAL_SRV_PRM_T, see subsections 5.4.1.4"*Coding of the Server Parameter Structure*" and 4.5.4"*Detailed Description of Server Parameters*")
- DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER (corresponds to the slave parameter structure, see subsections 5.4.1.5"*Coding of the Slave Parameter Structure*" and 4.5.2"*Detailed Description of Slave Parameters*")
- DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER (corresponds to the bus parameter structure DN_FAL_BUS_PRM_T, see subsections 5.4.1.6"*Coding of the Bus Parameter Structure*" and 4.5.1"*Detailed Description of Bus Parameters*")

The usArea parameter decides which option of those four will be applied.

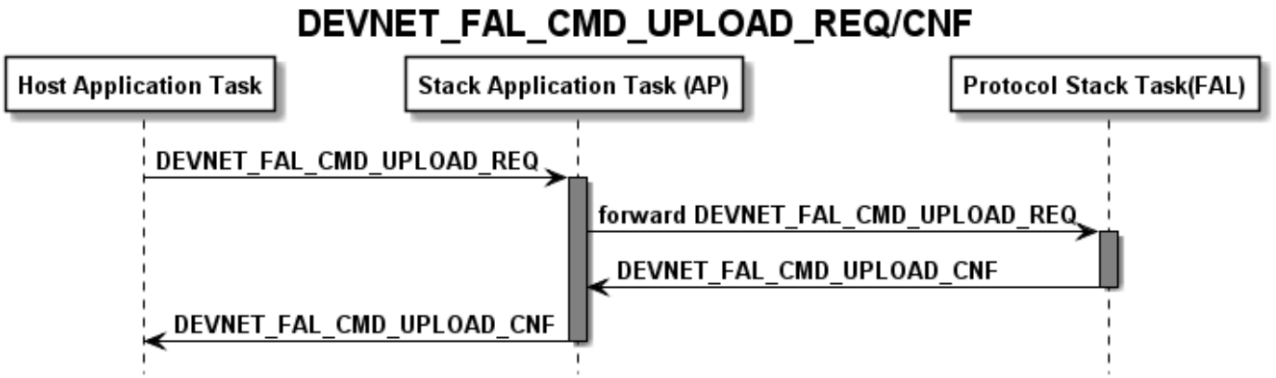

Figure 14: Sequence diagram of the DEVNET_FAL_CMD_UPLOAD_REQ packet

Packet Structure Reference

```
typedef struct DN_FAL_SDU_UPLOAD_REQ_Ttag
{
  TLR_UINT16 usArea;
  TLR_UINT16 usSubArea;
} DN_FAL_SDU_UPLOAD_REQ_T;

typedef struct DN_FAL_PACKET_UPLOAD_REQ_Ttag
{
  TLR_PACKET_HEADER_T         tHead;                          /* Packet header */
  DN_FAL_SDU_UPLOAD_REQ_T     tData;                          /* Packet data   */
} DN_FAL_PACKET_UPLOAD_REQ_T;
```

Packet Description

structure DN_FAL_PACKET_UPLOAD_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3816	DEVNET_FAL_CMD_UPLOAD_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - structure DN_FAL_SDU_UPLOAD_REQ_T			
usArea	UINT16	0 ... 3	Area code of where the data is to be downloaded. DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER
usSubArea	UINT16	0..63	depends on usArea …_AREA_BUS_PARAMETER 0 …_AREA_SLAVE_PARAMETER 0..63 (Mac ID) …_AREA_SERVER_PARAMETER 0 …_AREA_DEVICE_PARAMETER 0

Table 85: DEVNET_FAL_CMD_UPLOAD_REQ - Parameter Upload

Packet Structure Reference

```
typedef struct DN_FAL_SDU_UPLOAD_CNF_Ttag
{
  TLR_UINT16 usArea;
  TLR_UINT16 usSubArea;
  TLR_UINT8  abData[1000];
}DN_FAL_SDU_UPLOAD_CNF_T;

#define DN_FAL_UPLOAD_CNF_SIZE sizeof(DN_FAL_SDU_UPLOAD_CNF_T)
typedef struct DN_FAL_PACKET_UPLOAD_CNF_Ttag
{
  TLR_PACKET_HEADER_T         tHead;                          /* Packet header    */
  DN_FAL_SDU_UPLOAD_CNF_T     tData;                          /* Packet data      */
}DN_FAL_PACKET_UPLOAD_CNF_T;
```

structure DN_FAL_PACKET_UPLOAD_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	1004	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3817	DEVNET_FAL_CMD_UPLOAD_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - structure DN_FAL_SDU_UPLOAD_CNF_T			
usArea	UINT16	0 ... 3	Area code of where the data is to be downloaded. DN_FAL_DOWNLOAD_AREA_BUS_PARAMETER DN_FAL_DOWNLOAD_AREA_SLAVE_PARAMETER DN_FAL_DOWNLOAD_AREA_SERVER_PARAMETER DN_FAL_DOWNLOAD_AREA_DEVICE_PARAMETER
usSubArea	UINT16	0..63	depends on usArea …_AREA_BUS_PARAMETER 0 …_AREA_SLAVE_PARAMETER 0..63 (Mac ID) …_AREA_SERVER_PARAMETER 0 …_AREA_DEVICE_PARAMETER 0
abData[1000]	UINT8[]		Download data contents depending on usArea

Table 86: DEVNET_FAL_CMD_UPLOAD_CNF – Confirmation of Upload

5.5.4 Device Diagnosis Service DEVNET_FAL_CMD_DEV_DIAG_REQ/CNF

This packet allows obtaining diagnostic data concerning the device. From which device to take the diagnostic data is decided by the ubDevMacId MAC ID parameter of the request packet.

You can also decide using the fGetOnly parameter of the request packet whether to reset (=0) or not to reset (=1) the corresponding diagnosis bit for the chosen slave within the Slave diagnostic area (Field abDv_diag[8], see page 42).

The diagnostic data are delivered as part of the confirmation packet (data structure tDataLoad). This data structure looks as follows:

The structure DN_FAL_DEV_DIAG_DATALOAD_T contained in the tDataLoad parameter of the confirmation packet contains the following information:

Structure CANOPEN_MASTER_NODE_DIAG_T		
Structure element name	**Type**	**Meaning**
DN_FAL_DEV_DIAG_DATA_T	UINT8	Main part of Diagnostic Data Structure
abData [DN_FAL_MAX_DATA_SIZE-6]	UINT8[]	Device specific part of Diagnostic Data Structure

Table 87: DN_FAL_DEV_DIAG_DATALOAD_T - Diagnostic Data Structure

For the details of structure DN_FAL_DEV_DIAG_DATA_T contained in DN_FAL_DEV_DIAG_DATALOAD_T, see section 4.6.1 "*Diagnosis*".

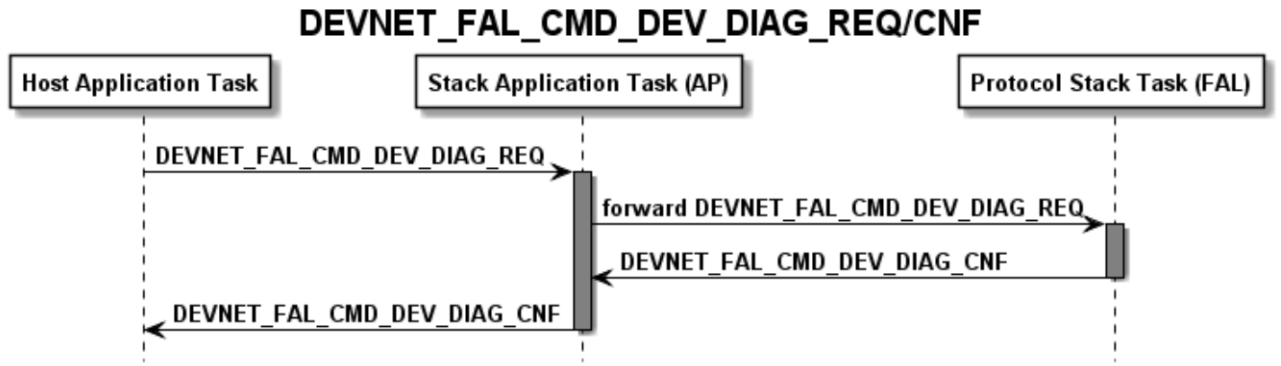

Figure 15: Sequence diagram of the DEVNET_FAL_CMD_DEV_DIAG_REQ packet

Packet Structure Reference

```
typedef struct DN_FAL_DEV_DIAG_REQ_DATA_Ttag{

  TLR_UINT8    ubDevMacId;
  TLR_BOOLEAN  fGetOnly;   /* 1 = do not reset corresponding bit in abDv_diag */
  TLR_UINT8    ubReserved;
}DN_FAL_DEV_DIAG_REQ_DATA_T;

#define DN_FAL_DEV_DIAG_REQ_DATA_SIZE sizeof(DN_FAL_DEV_DIAG_REQ_DATA_T)

typedef struct DN_FAL_PACKET_DEV_DIAG_REQ_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  DN_FAL_DEV_DIAG_REQ_DATA_T      tData;
}DN_FAL_PACKET_DEV_DIAG_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_DEV_DIAG_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	6	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3820	DEVNET_FAL_CMD_DEV_DIAG_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_DEV_DIAG_REQ_DATA_T			
ubDevMacId	UINT8	0-63	MAC ID of device
fGetOnly	BOOLEAN32	0,1	1 = do not reset corresponding bit in abDv_diag
ubReserved	UINT8	0	Reserved, always 0

Table 88: DN_FAL_PACKET_DEV_DIAG_REQ_T - Device Diagnostics Request

Packet Structure Reference

```
typedef struct DN_FAL_DEV_DIAG_DATA_Ttag
{
#define DN_FAL_NODE_DIAG_NO_RES            0x80
#define DN_FAL_NODE_DIAG_PRM_FAULT         0x20
#define DN_FAL_NODE_DIAG_CFG_FAULT         0x10
#define DN_FAL_NODE_DIAG_UCMM_SUPPORT      0x08
#define DN_FAL_NODE_DIAG_NOT_CONFIGURED    0x01

  TLR_UINT8  bNodeExtraDiag;
  TLR_UINT8  bDevMainState;
  TLR_UINT8  bOnlineError;
  TLR_UINT8  bGeneralErrorCode;
  TLR_UINT8  bAdditionalCode;
  TLR_UINT16 usHrtBeatTimeout;

} DN_FAL_DEV_DIAG_DATA_T;

typedef union DN_FAL_DEV_DIAG_DATALOAD_Ttag{
  DN_FAL_DEV_DIAG_DATA_T tDiagData;
  TLR_UINT8              abData[DN_FAL_MAX_DATA_SIZE-6];
                              /*Size of ubDevMacId+udDataLen+ubDiagType = 6*/
}DN_FAL_DEV_DIAG_DATALOAD_T;

typedef struct DN_FAL_DEV_DIAG_CNF_DATA_Ttag{

  TLR_UINT8    ubDevMacId;
  TLR_UINT32   ulDatLen;
  TLR_UINT8    ubDiagType;                          /*Preserved. Do no used */
  DN_FAL_DEV_DIAG_DATALOAD_T tDataLoad;
}DN_FAL_DEV_DIAG_CNF_DATA_T;
```

Packet Description

Structure DN_FAL_PACKET_DEV_DIAG_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	<=1556	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3821	DEVNET_FAL_CMD_DEV_DIAG_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_DEV_DIAG_DATALOAD_T			
ubDevMacId	UINT8	0-63	MAC ID of device
ulDatLen	UINT32		Data length (of structure DN_FAL_DEV_DIAG_DATA_T)
ubDiagType	UINT8	0-255	Preserved, do not use
tDataLoad	DN_FAL_DEV_DIAG_DATALOAD_T		Structure containing diagnosis data

Table 89: DN_FAL_PACKET_DEV_DIAG_CNF_T - Confirmation of Device Diagnostics Request

5.5.5 DEVNET_FAL_CMD_FAULT_IND/RES – Indication of a Fault

This indication packet is sent from the DeviceNet Master protocol stack task to the stack application (AP task) in order to indicate a fault. The reason why a fault occurred is coded in the `ulFault` variable.

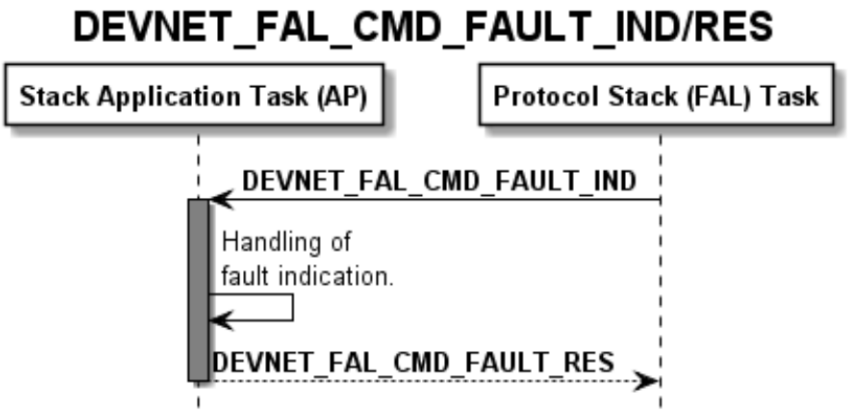

Figure 16: Sequence diagram of the DEVNET_FAL_CMD_FAULT_IND packet

Packet Structure Reference

```
typedef struct DN_FAL_SDU_FAULT_IND_Ttag
{
  TLR_UINT32 ulFault;
} DN_FAL_SDU_FAULT_IND_T;

typedef struct DN_FAL_PACKET_FAULT_IND_Ttag
{
  TLR_PACKET_HEADER_T           tHead;                    /* Packet Header     */
  DN_FAL_SDU_FAULT_IND_T        tData;                    /* Packet data       */
} DN_FAL_PACKET_FAULT_IND_T;

#define DN_FAL_FAULT_IND_SIZE (sizeof(DN_FAL_SDU_FAULT_IND_T))
```

Packet Description

Structure DN_FAL_PACKET_FAULT_IND_T			**Type: Indication**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3810	DEVNET_FAL_CMD_FAULT_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Structure DN_FAL_PACKET_FAULT_IND_T			Type: Indication
tData - Structure DN_FAL_SDU_FAULT_IND_T			
ulFault	UINT32		Code indicating which fault has occurred. See Table 128: Status/Error Codes DevNet FAL - Task

Table 90: DEVNET_FAL_CMD_FAULT_IND - Indication of a Fault

Packet Structure Reference

```
typedef struct DN_FAL_PACKET_FAULT_RES_Ttag
{
  TLR_PACKET_HEADER_T           tHead;                      /* Packet Header     */
}DN_FAL_PACKET_FAULT_RES_T;

#define DN_FAL_FAULT_RES_SIZE (0)
```

Packet Description

Structure DN_FAL_PACKET_FAULT_RES_T			Type: Response
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3811	DEVNET_FAL_CMD_FAULT_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 91: DEVNET_FAL_CMD_FAULT_RES – Response to Indication of a Fault

5.5.6 Operation Mode Indication DEVNET_FAL_CMD_SET_MODE_IND/RES

This indication is issued each time when the DeviceNet operation mode (supplied in parameter `ulMode`) changes.

Valid modes are:

- OFFLINE (0x00)
  - The DeviceNet Master is not active on network, and is not accessible for other devices.
- STOP (0x40)
  - The DeviceNet Master performs the Duplicate-MAC Id check procedure, but does not start any I/O communication. The master is accessible for other devices with explicit. messaging
- IDLE (0x80)
  - Currently same as `RUN`
- RUN (0xC0)
  - The DeviceNet Master starts I/O communication to all configured slaves.

The reason can be taken from the reason code, see table below:

Reason code	Value	Meaning
`DUP_MAC_ID_ERROR`	1	Duplicate MAC ID
`SUPPLY_VOLTAGE_ERROR`	2	Wrong supply voltage

Table 92: Reason codes of Set DeviceNet Operation Mode Indication

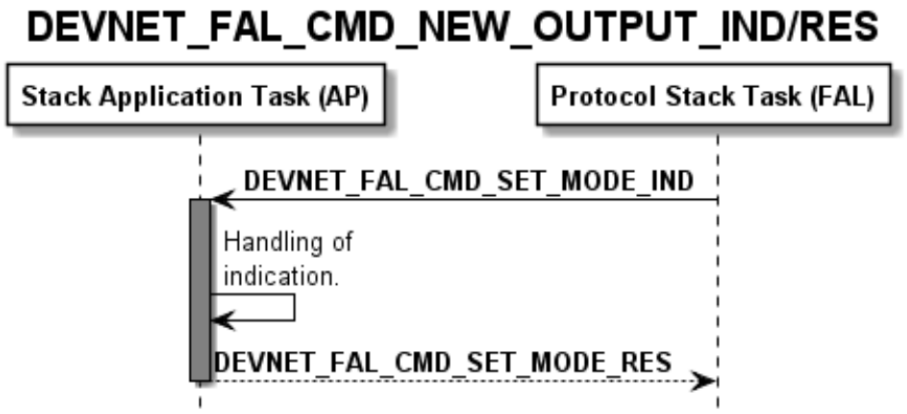

Figure 17: Sequence diagram of the DEVNET_FAL_CMD_SET_MODE_IND packet

Packet Structure Reference

```
#define DN_FAL_MODE_OFFLINE   (0x00)
#define DN_FAL_MODE_STOP      (0x40)
#define DN_FAL_MODE_IDLE      (0x80)
#define DN_FAL_MODE_RUN       (0xC0)


/** SDU: Set Mode ind/res ***************************************************/
typedef struct DN_FAL_SDU_SET_MODE_IND_Ttag
{
  #define DUP_MAC_ID_ERROR 0x00000001
  #define SUPPLY_VOLTAGE_ERROR 0x00000002
  TLR_UINT32 ulMode;
  TLR_UINT32 ulReason;
} DN_FAL_SDU_SET_MODE_IND_T;

/** PACKET: Set mode ind/res ************************************************/
typedef struct DN_FAL_PACKET_SET_MODE_IND_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                  /* Packet Header    */
  DN_FAL_SDU_SET_MODE_IND_T       tData;                  /* Packet data      */
} DN_FAL_PACKET_SET_MODE_IND_T;
```

Packet Description

Structure DN_FAL_PACKET_SET_MODE_IND_T			**Type: Indication**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	8	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3818	DEVNET_FAL_CMD_SET_MODE_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_SET_MODE_IND_T			
ulMode	UINT32	0, 0x40, 0x80,0xC0	Mode (see above)
ulReason	UINT32	1,2	Reason code (see above)

Table 93: DN_FAL_PACKET_SET_MODE_IND_T - Set DeviceNet Operation Mode Indication

Packet Structure Reference

```
typedef struct DN_FAL_SDU_SET_MODE_RES_Ttag
{
  TLR_UINT32 ulMode;
} DN_FAL_SDU_SET_MODE_RES_T

typedef struct DN_FAL_PACKET_SET_MODE_RES_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                  /* Packet Header    */
  DN_FAL_SDU_SET_MODE_RES_T       tData;                  /* Packet data      */
} DN_FAL_PACKET_SET_MODE_RES_T;
```

Packet Description

Structure DN_FAL_PACKET_SET_MODE_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3819	DEVNET_FAL_CMD_SET_MODE_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_SET_MODE_RES_T			
ulMode	UINT32	0, 0x40, 0x80,0xC0	Mode (see above)

Table 94: DN_FAL_PACKET_SET_MODE_RES_T - Response to Set DeviceNet Operation Mode Indication

5.5.7 DEVNET_FAL_CMD_SET_LED_IND/RES – Set LED Indication

This command is currently not used.

5.6 Input/Output Data Services

5.6.1 Acyclic Bit-Strobing Service DEVNET_FAL_CMD_ACYC_BTS_REQ/CNF

This packet has been designed to accomplish acyclic bit-strobing. This is a special method to synchronize the host application and the DeviceNet master device and to plan synchronous actions of master and (single or multiple) slaves.

Bit strobing is initiated by the DeviceNet Master by sending a special I/O message, i.e. the bit strobe command message. This bit strobe command message is a multi-cast message allowing multiple slaves to react synchronously to the same message as they should receive it at the same time. Within a DeviceNet Slave, the bit strobe command and its associated response are received and transmitted by one single connection object.

The reaction to the bit strobe command is application-specific and known at the DeviceNet Master.

In an 8-byte telegram each of 64 available bits represents one of the 64 possible participants of the DeviceNet network as the number of MAC IDs is limited to 64. This telegram contents is also denominated as the “master scan list”. For the structure of the master scan list, the following rules apply:

- The lowest MAC ID, i.e. 0, is assigned to the low-order bit D0 in the lowest-order byte (0) of the telegram.
- The highest MAC ID, i.e. 63, is assigned to the high-order bit D7 in the highest-order byte (7) of the telegram.

These rules result in the assignment table which is represented by *Table 95: Assignment Table for Bits in Master Scan List and associated MAC IDs of DeviceNet Slaves*.

Each of these slaves receiving the bit strobing command message has the possibility to decide between the following alternatives on reception of a bit strobe command from the master:

- Ignore the bit strobe command, for instance a polled device would behave in this way.
- Consume both the bit strobe command and its data
- Consume the bit strobe command only, but ignore its data

The applied telegrams are unconfirmed. Instead of a confirmation a time-out mechanism has been established in the following manner: After a previously determined time (the time-out value), the DeviceNet Master will collect and evaluate the responses of the DeviceNet slaves, and provide the results to the application.

In order to perform bit-strobing practically, the following information needs to be provided:

- A list of the slave devices in the DeviceNet network needs to be bit-strobed (`ullStrobeMsk`). This is done by providing the bit mask of a width of 64-bits. The application sets the corresponding bit to query data from remote node via the bit-strobing mechanism. Although with a bit-strobe request from the master, all bit-strobe capable devices can respond but only responses from nodes whose bits are set in variable `ullStrobeMsk` are packed into the confirmation packet. See *Table 96: DEVNET_FAL_CMD_ACYC_BTS_REQ – Acyclic Bit-Strobing*.
- Strobe-Command which will be sent on the bus (`ullStrobeCmd`). This is done by providing the bit mask of a width of 64-bits. See *Table 96: DEVNET_FAL_CMD_ACYC_BTS_REQ – Acyclic Bit-Strobing*.
- Strobe Time-out (`ulTimeout`)

The time-out value just defines the duration how long the DeviceNet Master has to wait. If time-out occurs, FAL Tasks will pack all available responses in confirmation packet and send it back to application. If the time-out value is too small, then it may happen that some responses from some node will be missed within the confirmation packet. If all nodes needed (specified in `ullStrobeMsk`) respond within time-out, all responses will be packed in the confirmation packet.

Bit / Byte	D7	D6	D5	D4	D3	D2	D1	D0
0 (LSB)	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
…								
7 (MSB)	63	62	61	60	59	58	57	56

Table 95: Assignment Table for Bits in Master Scan List and associated MAC IDs of DeviceNet Slaves

The results provided by the confirmation packet contain

- A status list indicating which devices reacted to the bit strobing. This status list is structured similarly to the device list. If the `ulStatusList` bit of the corresponding slave is logically

  '1', the slave responded to bit-strobing request

  '0', the slave did not response to bit-strobing request.

- An additional data array `atBtsData[64]` containing information about each device within the DeviceNet network.

Note: If a `DEVNET_FAL_CMD_ACYC_BTS_REQ` packet has already been sent earlier and is still not completely processed, the status of the confirmation packet will be set to `TLR_E_DEVNET_FAL_BTS_IN_PROGRESS` indicating that processing of the acyclic bit-strobing request is still in progress.

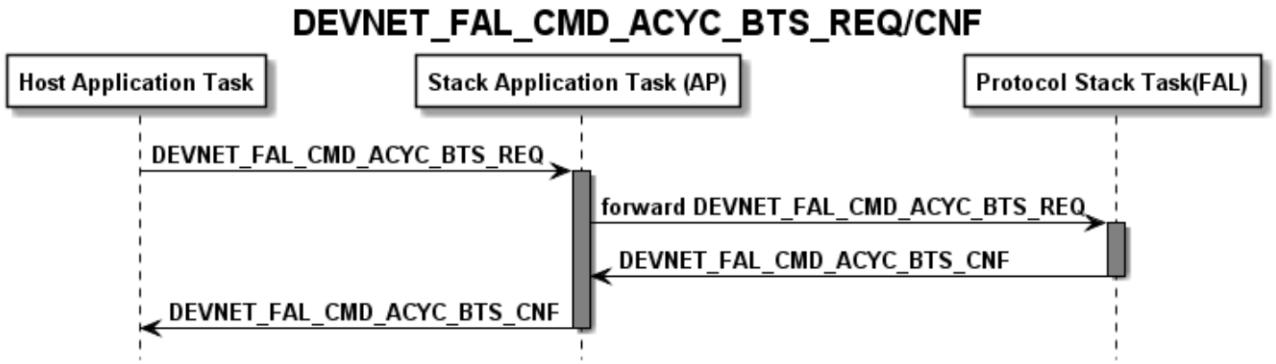

Figure 18: Sequence diagram of the DEVNET_FAL_CMD_ACYC_BTS_REQ packet

Packet Structure Reference

```
typedef struct DN_FAL_BTS_REQ_Ttag
{
  TLR_UINT64 ullStrobeMsk;                                  /* Strobe Mask    */
  TLR_UINT64 ullStrobeCmd;                                  /* Strobe Command */
  TLR_UINT32 ulTimeOut;
}DN_FAL_BTS_REQ_T;

typedef struct DN_FAL_PACKET_BTS_REQ_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  DN_FAL_BTS_REQ_T                tData;
} DN_FAL_PACKET_BTS_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_BTS_REQ_T			**Type: Request**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	20	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3812	DEVNET_FAL_CMD_ACYC_BTS_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_BTS_REQ_T			
ullStrobeMsk;	UINT64	0x00000000-0xFFFFFFFF	List of devices needed to be strobed. (1 bit per device according to assignment as explained in Table 95 above)
ullStrobeCmd	UINT64	0x00000000-0xFFFFFFFF	Bit-Strobe Command which will be sent on the bus (1 bit per device according to assignment as explained in Table 95 above)
ulTimeOut	UINT32	0x00000000-0xFFFFFFFF	Time-out value specified in milliseconds.

Table 96: DEVNET_FAL_CMD_ACYC_BTS_REQ – Acyclic Bit-Strobing

Packet Structure Reference

```
typedef struct DM_FAL_BTS_MSG_Tag
{
  TLR_UINT8 bNodeAddr;
  TLR_UINT8 bLen;
  TLR_UINT8 bRsrv;
  TLR_UINT8 abData[8];
}DM_FAL_BTS_MSG_T;

typedef struct DN_FAL_BTS_CNF_Ttag
{
  TLR_UINT64 ullStatusList;
  DM_FAL_BTS_MSG_T atBtsData[64];
}DN_FAL_BTS_CNF_T;

typedef struct DN_FAL_PACKET_BTS_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  DN_FAL_BTS_CNF_T                tData;
}DN_FAL_PACKET_BTS_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_BTS_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	712	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3813	DEVNET_FAL_CMD_ACYC_BTS_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_BTS_CNF_T			
ullStatusList	UINT64	0x00000000-0xFFFFFFFF	A status list indicating which devices reacted to the bit strobing. Bits are assigned as shown in Table 95.
atBtsData[64]	DN_FAL_BTS_MSG_T	Array	Data. 64 maximal responses from remote nodes. An element is only valid if corresponding bit in ulStatusList is set.

Table 97: DEVNET_FAL_CMD_ACYC_BTS_CNF – Confirmation of Acyclic Bit-Strobing

5.6.2 Acyclic Poll Service DEVNET_FAL_CMD_ACYC_POLL_REQ/CNF

The user can use this command to send an arbitrary acyclic poll request to a MAC ID (even if the slave with this MAC ID is not available on the bus). When there is already a poll connection between the master and slave, this command will send an extra poll request to the cyclic poll request of the master to the slave. In case of I/O connection, the length is greater than maximal data load of CAN frame, 8 bytes, the poll request is fragmented according to the DeviceNet specification (references [2] and [3]). Because the asynchronous nature of this service, the data in the confirmation packet is just a copy of the request data.

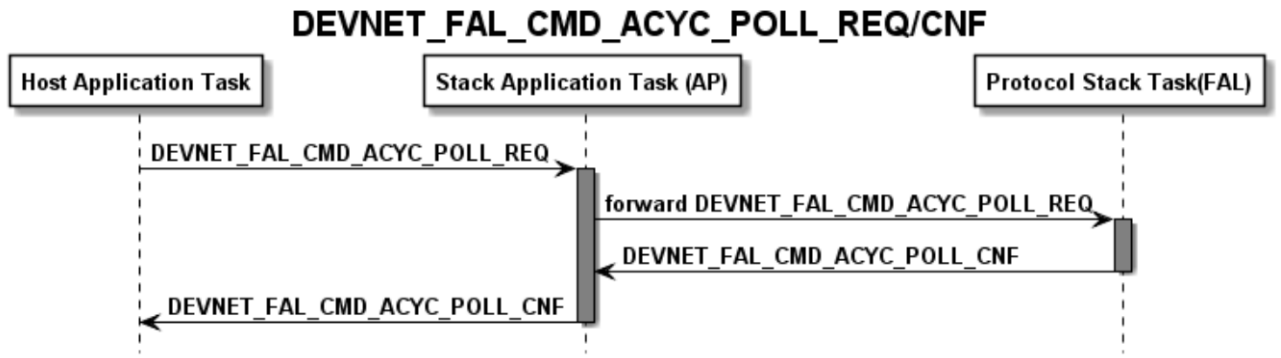

Figure 19: Sequence diagram of the DEVNET_FAL_CMD_ACYC_POLL_REQ packet

Packet Structure Reference

```
#define DN_FAL_SDU_ACYC_IO_POLL_MAX_DATA  (512)
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_SDU_ACYC_IO_POLL_REQ_Ttag
{
  TLR_UINT8  bDevMacId;           /**< MAC ID of the slave. Max 64            */

  TLR_UINT16 usPollDatLen;        /**< Effective Length of Data in abSrvData  */

  TLR_UINT8  abPollData[DN_FAL_SDU_ACYC_IO_POLL_MAX_DATA];          /**< Data */
}
DN_FAL_SDU_ACYC_IO_POLL_REQ_T;


#define DN_FAL_ACYC_IO_POLL_REQ_SIZE \
         (sizeof(DN_FAL_PACKET_ACYC_IO_POLL_REQ_T)- sizeof(TLR_PACKET_HEADER_T))
/*! Sending of  a poll request to slave at any time using
 * #DEVNET_FAL_CMD_ACYC_POLL_REQ */
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_PACKET_ACYC_IO_POLL_REQ_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                  /**< Packet Header  */

  DN_FAL_SDU_ACYC_IO_POLL_REQ_T   tData;                  /**< Data           */
}
DN_FAL_PACKET_ACYC_IO_POLL_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_ACYC_IO_POLL_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	Length of this header (40 bytes) + length of bDevMacId (1 bytes) + usPollDatLen
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Set to zero
ulCmd	UINT32	0x380A	DEVNET_FAL_CMD_ACYC_POLL_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing, set to zero
tData - Structure DN_FAL_SDU_ACYC_POLL_REQ_T			
bDevMacId	UINT8	0 ... 63	Device address, MAC-ID of device to be addressed
usPollDatLen	UINT16	0 ... 512	Length of data of the poll request
abPollData[0 .. 511]	UINT8[]	0	Data of the poll request

Table 98: DEVNET_FAL_CMD_ACYC_POLL_REQ – Acyclic Poll Request

Source Code Example

```
TLR_RESULT DnUser_AcycPollReq( DN_USER_RSC_T FAR* ptRsc )
{
  TLR_RESULT eRslt = TLR_S_OK;
  DN_FAL_PACKET_ACYC_IO_POLL_REQ_T   *ptPollReq = NULL;

  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,& ptPollReq);
  if( eRslt == TLR_S_OK )
  { //Zero packet memory
    TLR_MEMSET(&ptPollReq ->tHead,0x00,sizeof(ptPollReq ->tHead));
    TLR_MEMSET(&ptPollReq ->tData,0x00, DN_FAL_ACYC_IO_POLL_REQ_SIZE);

    //Init packet header
    TLR_QUE_LINK_SET_PACKET_SRC(ptPollReq,ptRsc->tLoc.tQueSrcApp);
    ptGetAttReq->tHead.ulCmd = DEVNET_FAL_CMD_ACYC_POLL_RQ_REQ;
    ptGetAttReq->tHead.ulLen = 51;

    //Init packet data
    ptPollReq->tData.bDevMacId = 31;     //DeviceAddress
    ptPollReq->tData.usPollDatLen = 8;   //Poll Length

    ptPollReq->tData.abPollData[0] = 0x00
    ptPollReq->tData.abPollData[1] = 0x11
    ptPollReq->tData.abPollData[2] = 0x22
    ptPollReq->tData.abPollData[3] = 0x33
    ptPollReq->tData.abPollData[4] = 0x44
    ptPollReq->tData.abPollData[5] = 0x55
    ptPollReq->tData.abPollData[6] = 0x66
    ptPollReq->tData.abPollData[7] = 0x77

   //Send packet
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tLoc.tDstQue, ptPollReq,TLR_FINITE);
    if( eRslt != TLR_S_OK ) {
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptPollReq);
    }
  }
  return eRslt;
}
```

Packet Structure Reference

```
#define DN_FAL_SDU_ACYC_IO_POLL_MAX_DATA  (512)
typedef struct DN_FAL_SDU_ACYC_IO_POLL_CNF_Ttag
{
  TLR_UINT8  bDevMacId;

  TLR_UINT16 usPollDatLen;

  TLR_UINT8  abPollData[DN_FAL_SDU_ACYC_IO_POLL_MAX_DATA];          }
DN_FAL_SDU_ACYC_IO_POLL_CNF_T;

#define DN_FAL_ACYC_IO_POLL_REQ_SIZE \
         (sizeof(DN_FAL_PACKET_ACYC_IO_POLL_REQ_T)- sizeof(TLR_PACKET_HEADER_T))
typedef struct DN_FAL_PACKET_ACYC_IO_POLL_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;

  DN_FAL_SDU_ACYC_IO_POLL_CNF_T   tData;
}
DN_FAL_PACKET_ACYC_IO_POLL_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_ACYC_IO_POLL_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	Length of this header (40 bytes) + length of bDevMacId (1 bytes) + usPollDatLen
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Set to zero
ulCmd	UINT32	0x380B	DEVNET_FAL_CMD_ACYC_POLL_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing, set to zero
tData - Structure DN_FAL_SDU_ACYC_POLL_CNF_T			
bDevMacId	UINT8	0-63	Device address, MAC-ID of device to be addressed
usPollDatLen	UINT16	0 ... 512	Length of data of the poll request
abPollData[0 .. 511]	UINT8[]	0	Data of the poll response from slave.

Table 99: DEVNET_FAL_CMD_ACYC_POLL_CNF – Acyclic Poll Confirmation

5.6.3 New Output Indication DEVNET_FAL_CMD_NEW_OUTPUT_IND/RES

This indication will be issued when new output data are available.

Neither the indication packet nor the response packet has any parameters.

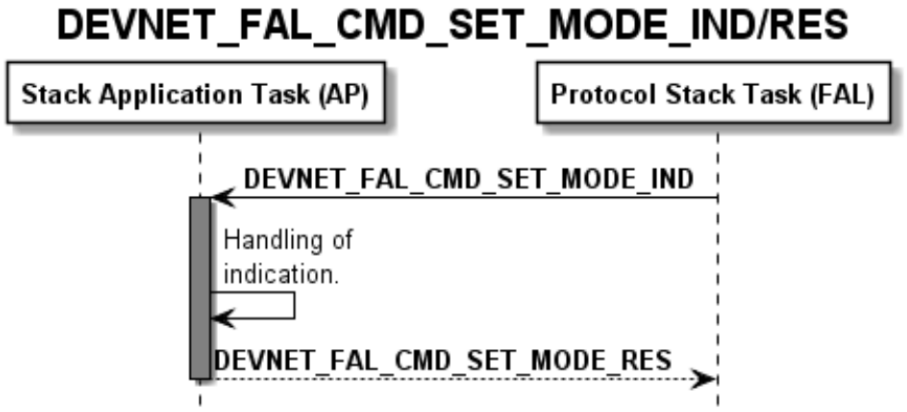

Figure 20: Sequence diagram of the DEVNET_FAL_CMD_NEW_OUTPUT_IND packet

Packet Structure Reference

```
/** PACKET: New output indication ***********************************************/
typedef struct DN_FAL_PACKET_NEW_OUTPUT_IND_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                  /* Packet Header    */
} DN_FAL_PACKET_NEW_OUTPUT_IND_T;
```

Packet Description

Structure DN_FAL_PACKET_NEW_OUTPUT_IND_T			Type: Indication
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3808	DEVNET_FAL_CMD_NEW_OUTPUT_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 100: DN_FAL_PACKET_NEW_OUTPUT_IND_T - New Output Indication

Packet Structure Reference

```
typedef struct DN_FAL_PACKET_NEW_OUTPUT_RES_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                  /* Packet Header    */
} DN_FAL_PACKET_NEW_OUTPUT_RES_T;
```

Packet Description

Structure DN_FAL_PACKET_NEW_OUTPUT_RES_T			Type: Response
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3809	DEVNET_FAL_CMD_NEW_OUTPUT_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 101: DN_FAL_PACKET_NEW_OUTPUT_RES_T - Response to New Output Indication

5.7 Explicit Message Services

5.7.1 Get Attribute Service DEVNET_FAL_CMD_GET_ATT_REQ/CNF

The command DEVNET_FAL_GET_ATT_REQ is used by a user application to request the value of an attribute from a device/slave according the DeviceNet Object model. For a brief description of the pre-defined objects see section 4.2 “*Object Modeling*”.

An explanation of how this command is used is shown as following diagram.

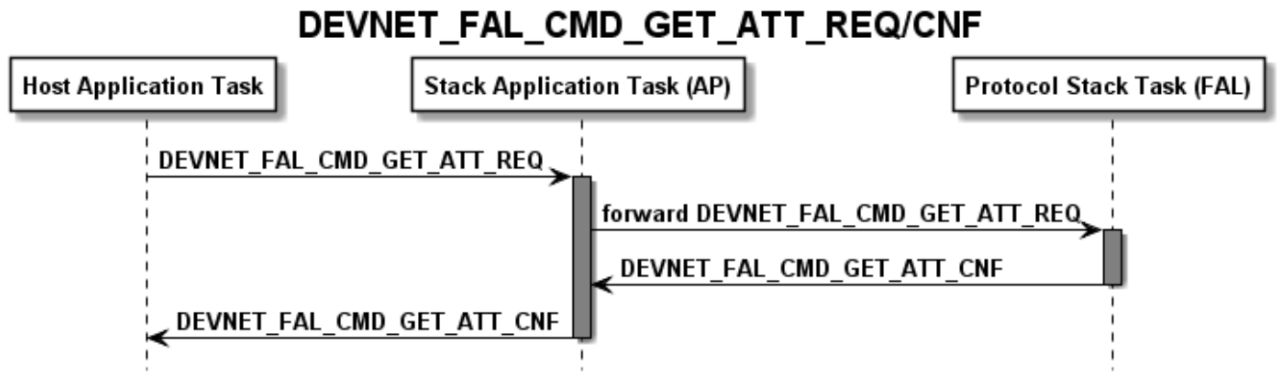

Figure 21: Sequence diagram of the DEVNET_FAL_CMD_GET_ATT_REQ packet

Packet Structure Reference

```
#define DN_FAL_SDU_GETSET_ATT_MAX_DATA  (512)
typedef struct DN_FAL_SDU_GETSET_ATT_REQ_Ttag {
  TLR_UINT8  bDeviceAddr;
  TLR_UINT8  abReserved[3];
  TLR_UINT16 usClass;
  TLR_UINT16 usInstance;
  TLR_UINT16 usAttribute;
  TLR_UINT16 usReserved;
  TLR_UINT8  abAttData[DN_FAL_SDU_GETSET_ATT_MAX_DATA];
}DN_FAL_SDU_GETSET_ATT_REQ_T;


typedef struct DN_FAL_PACKET_GET_ATT_REQ_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_GETSET_ATT_REQ_T tData;
} DN_FAL_PACKET_GET_ATT_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_GET_ATT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	DN_FAL_GETSET_ATT_REQ_SIZE
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Set to zero
ulCmd	UINT32	0x380A	DEVNET_FAL_CMD_GET_ATT_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing, set to zero
tData - Structure DN_FAL_SDU_GETSET_ATT_REQ_T			
bDeviceAddr	UINT8	0-63	Device address, MAC-ID of device to be addressed
abReserved[3]	UINT8[]	0	Reserved zero
usClass	UINT16	0 ... 2^{16}-1	CIP Class ID of the DeviceNet slave. (For pre-defined classes please refer to *"The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1" of the reference document [2])*
usInstance	UINT16	0 ... 2^{16}-1	Instance ID of the object with class ID above. . The instance ID in combination with class ID will form a unique identification of an object.
usAttribute	UINT16	0 ... 2^{16}-1	Attribute ID within the object. Pre-defined attributes of an object is described in detail in object profiles of reference [2] and [3].
usReserved	UINT16	0	Set to zero
abAttData[0 .. 511]	UINT8[]	0	Attribute data not used in request

Table 102: DEVNET_FAL_CMD_GET_ATT_REQ - Get Attribute Request

Source Code Example

```
TLR_RESULT DnUser_GetAttReq( DN_USER_RSC_T FAR* ptRsc )
{
  TLR_RESULT eRslt = TLR_S_OK;
  DN_FAL_PACKET_GET_ATT_REQ_T   *ptGetAttReq = NULL;

  eRslt = TLR_POOL_PACKET_GET(ptRsc->tLoc.hPool,&ptGetAttReq);
  if( eRslt == TLR_S_OK )
  { //Zero packet memory
    TLR_MEMSET(&ptGetAttReq->tHead,0x00,sizeof(ptGetAttReq->tHead));
    TLR_MEMSET(&ptGetAttReq->tData,0x00,DN_FAL_GETSET_ATT_REQ_SIZE);

    //Init packet header
    TLR_QUE_LINK_SET_PACKET_SRC(ptGetAttReq,ptRsc->tLoc.tQueSrcApp);
    ptGetAttReq->tHead.ulCmd = DEVNET_FAL_CMD_GET_ATT_REQ;
    ptGetAttReq->tHead.ulLen = DN_FAL_GETSET_ATT_REQ_SIZE;

    //Init packet data
    ptSetModeReq->tData.bDeviceAddr = 2; //DeviceAddress
    ptSetModeReq->tData.usClass     = 1; //IdentityObject
    ptSetModeReq->tData.usInstance  = 1; //Instance
    ptSetModeReq->tData.usAttribute = 7; //ProductName

    //Send packet
    eRslt = TLR_QUE_SENDPACKET_FIFO(ptRsc->tLoc.tDstQue,ptSetModeReq,TLR_FINITE);
    if( eRslt != TLR_S_OK ) {
      TLR_POOL_PACKET_RELEASE(ptRsc->tLoc.hPool, ptInitReq);
    }
  }
  return eRslt;
}
```

Packet Structure Reference

```
#define DN_FAL_SDU_GETSET_ATT_MAX_DATA  (512)
typedef struct DN_FAL_SDU_GETSET_ATT_CNF_Ttag {
  TLR_UINT8  bDeviceAddr;
  TLR_UINT8  abReserved[3];
  TLR_UINT16 usClass;
  TLR_UINT16 usInstance;
  TLR_UINT16 usAttribute;
  TLR_UINT8  bGenErr;
  TLR_UINT8  bAddErr;
  TLR_UINT8  abAttData[DN_FAL_SDU_GETSET_ATT_MAX_DATA];
}DN_FAL_SDU_GETSET_ATT_CNF_T;


typedef struct DN_FAL_PACKET_GET_ATT_CNF_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_GETSET_ATT_CNF_T tData;
} DN_FAL_PACKET_GET_ATT_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_GET_ATT_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12 + n	DN_FAL_GETSET_ATT_CNF_SIZE + number of attribute data received
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x380B	DEVNET_FAL_CMD_GET_ATT_CNF - Command
ulExt	UINT32	0	Extension not in use
ulRout	UINT32	x	Routing
tData - Structure DN_FAL_SDU_GETSET_ATT_CNF_T			
bDeviceAddr	UINT8	0-63	MAC-ID of the device that was addressed
abReserved[3]	UINT8	0	Reserved zero
usClass	UINT16	0 ... 2^{16}-1	Class ID, returned. See Table 102.
usInstance	UINT16	0 ... 2^{16}-1	Instance ID, returned. See Table 102.
usAttribute	UINT16	0 ... 2^{16}-1	Attribute ID, returned. See Table 102.
bGenErr	UINT8	0 … 255	See Table 104: Generic Error (Variable bGenErr)
bAddErr	UINT8	0 … 255	See Table 105: Additional Error (Variable bAddErr)
abAttData[0 .. 512]	UINT8 array		Attribute Data received

Table 103: DEVNET_FAL_CMD_GET_ATT_CNF - Confirmation of Get Attribute Request

bGenErr

bGenError	Signification
0	No error
2	Resources unavailable
8	Service not available
9	Invalid attribute value
11	Already in request mode
12	Object state conflict
14	Attribute not settable
15	A permission check failed
16	State conflict, device state prohibits the command execution
19	Not enough data received
20	Attribute not supported
21	Too much data received
22	Object does not exist
23	Reply data too large, internal buffer to small

Table 104: Generic Error (Variable bGenErr)

bAddErr

bAddError	signification
xxx	The additional error is network device specific, refer to the manual of the device and references [2] and [3].

Table 105: Additional Error (Variable bAddErr)

5.7.2 Set Attribute Service DEVNET_FAL_CMD_SET_ATT_REQ/CNF

The command DEVNET_FAL_SET_ATT_REQ is used by a user application to set an attribute to a device/slave according the DeviceNet Object model. For a brief description of the available objects see section 4.2 “*Object Modeling*”.

An explanation of how this command is used follows.

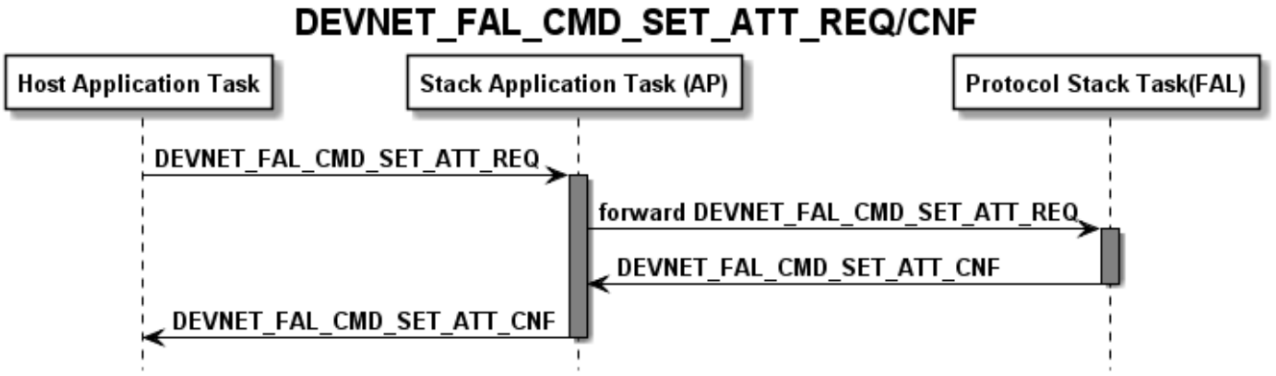

Figure 22: Sequence diagram of the DEVNET_FAL_CMD_SET_ATT_REQ packet

Packet Structure Reference

```
#define DN_FAL_SDU_GETSET_ATT_MAX_DATA  (512)
typedef struct DN_FAL_SDU_GETSET_ATT_REQ_Ttag {
  TLR_UINT8  bDeviceAddr;
  TLR_UINT8  abReserved[3];
  TLR_UINT16 usClass;
  TLR_UINT16 usInstance;
  TLR_UINT16 usAttribute;
  TLR_UINT16 usReserved;
  TLR_UINT8  abAttData[DN_FAL_SDU_GETSET_ATT_MAX_DATA];
}DN_FAL_SDU_GETSET_ATT_REQ_T;


typedef struct DN_FAL_PACKET_SET_ATT_REQ_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_GETSET_ATT_REQ_T tData;
} DN_FAL_PACKET_SET_ATT_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_SET_ATT_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12 + n	sizeof(DN_FAL_SDU_GETSET_ATT_REQ_T) + number of attribute data to set
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Set to zero
ulCmd	UINT32	0x380C	DEVNET_FAL_CMD_SET_ATT_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing, set to zero
tData - Structure DN_FAL_SDU_GETSET_ATT_REQ_T			
bDeviceAddr	UINT8	0-63	Device address, MAC-ID of device to be accessed.
abReserved[3]	UINT8	0	Reserved zero
usClass	UINT16	0 ... 2^{16}-1	CIP Class ID of the DeviceNet slave. (For pre-defined classes please refer to *"The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1" of the reference document [2])*
usInstance	UINT16	0 ... 2^{16}-1	Instance ID of the object with class ID above. . The instance ID in combination with class ID will form a unique identification of an object.
usAttribute	UINT16	0 ... 2^{16}-1	Attribute ID within the object. Pre-defined attributes of an object is described in details in object profiles of references [2] and [3].
usReserved	UINT16	0	Set to zero
abAttData[0 .. 511]	UINT8 array	x	Attribute data to be sent.

Table 106: DEVNET_FAL_CMD_SET_ATT_REQ - Set Attribute Request

Packet Structure Reference

```
#define DN_FAL_SDU_GETSET_ATT_MAX_DATA  (512)
typedef struct DN_FAL_SDU_GETSET_ATT_CNF_Ttag {
  TLR_UINT8  bDeviceAddr;
  TLR_UINT8  abReserved[3];
  TLR_UINT16 usClass;
  TLR_UINT16 usInstance;
  TLR_UINT16 usAttribute;
  TLR_UINT8  bGenErr;
  TLR_UINT8  bAddErr;
  TLR_UINT8  abAttData[DN_FAL_SDU_GETSET_ATT_MAX_DATA];
}DN_FAL_SDU_GETSET_ATT_CNF_T;


typedef struct DN_FAL_PACKET_SET_ATT_CNF_Ttag {
  TLR_PACKET_HEADER_T tHead;
  DN_FAL_SDU_GETSET_ATT_CNF_T tData;
} DN_FAL_PACKET_SET_ATT_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_SET_ATT_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle of AP-Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	DN_FAL_GETSET_ATT_CNF_SIZE
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	x	*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x380D	DEVNET_FAL_CMD_SET_ATT_CNF - Command
ulExt	UINT32	0	Extension not in use
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_GETSET_ATT_REQ_T			
bDeviceAddr	UINT8	0-63	Device address, MAC-ID of device that was addresses
abReserved[3]	UINT8	0	Reserved zero
usClass	UINT16	0 ... 2^{16}-1	Class ID, returned. See Table 106: DEVNET_FAL_CMD_SET_ATT_REQ - Set Attribute Request.
usInstance	UINT16	0 ... 2^{16}-1	Instance ID, returned. Table 106: DEVNET_FAL_CMD_SET_ATT_REQ - Set Attribute Request.
usAttribute	UINT16	0 ... 2^{16}-1	Attribute ID, returned. Table 106: DEVNET_FAL_CMD_SET_ATT_REQ - Set Attribute Request.
bGenErr	UINT8	0 … 255	See Table 104: Generic Error (Variable bGenErr)
bAddErr	UINT8	0 … 255	See Table 105: Additional Error (Variable bAddErr)
abAttData[0-511]	UINT8[]	0	Data of the response from slave.

Table 107: DEVNET_FAL_CMD_SET_ATT_REQ - Confirmation of Set Attribute Request

5.7.3 DEVNET_FAL_CMD_REMOTE_SERVICE_REQ/CNF – Remote Service

This service allows requesting any service from a DeviceNet slave as shown in following table.

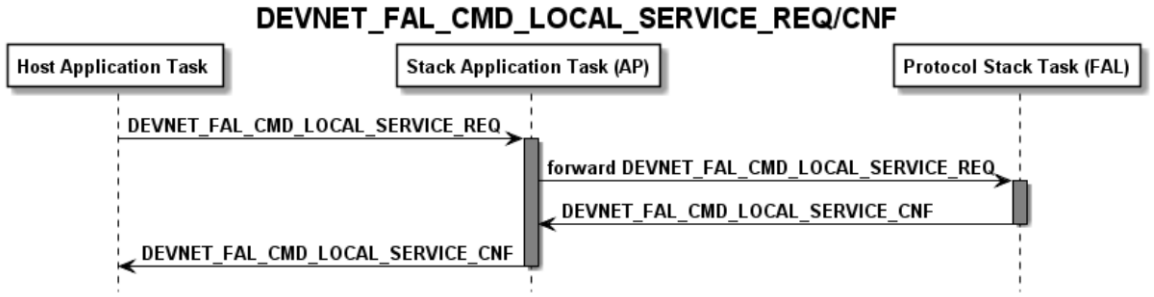

Figure 23: Sequence diagram of the DEVNET_FAL_CMD_REMOTE_SERVICE_REQ packet

Numeric value of ubServiceCode	Service to be executed
00	Reserved
01	Get_Attributes_All
02	Set_Attributes_All
03	Get_Attribute_List
04	Set_Attribute_List
05	Reset
06	Start
07	Stop
08	Create
09	Delete
0A	Multiple_Service_Packet
0B	Reserved for future use
0D	Apply_Attributes
0E	Get_Attribute_Single
0F	Reserved for future use
10	Set_Attribute_Single
11	Find_Next_Object_Instance
12-13	Reserved for future use
14	Error Response (used by DevNet only)
15	Restore
16	Save
17	No Operation (NOP)
18	Get_Member
19	Set_Member
1A	Insert_Member
1B	Remove_Member
1C	GroupSync
1D-31	Reserved for additional Common Services

Table 108: Service Codes

This table is taken from the CIP specification ("*Volume 1 Common Industrial Protocol Specification Appendix A, table A-3.1") [2].*

Note: Not every service is available on every object and on every DeviceNet Slave in the network. Using this packet only makes sense if you check that the selected object exists on the addressed slave device and supports the service selected by variable `ubServiceCode`.

The class and the instance of the object to be accessed are selected by the variables `usClass` and `usInstance` of the request packet. If an attribute is affected by the service (services `Get_Attributes_All`, `Set_Attributes_All`, `Get_Attribute_Single` and `Set_Attribute_Single`), this attribute is selected by variable `usAttribute` of the request packet. Set `usAttribute` to 0 when using other services than these.

Specify the address (i.e. the MAC-ID) of the device on which the object to be accessed can be found in variable `bDevMacId` of the request packet.

The result of the requested service is delivered in array `abSrvData[512]` of the confirmation packet.

In case of successful execution, the variables `bGenErrCode` and `bAddErrCode` of the confirmation packet will have the value 0.

In case of an error, the following happens:

- The variable `ubServiceCode` of the confirmation packet will have the value 0x14.
- The variables `bGenErrCode` and `bAddErrCode` of the confirmation packet will have non-zero values.

For lists of the general and additional error codes, see *Table 104: Generic Error (Variable bGenErr)* and *Table 105: Additional Error (Variable bAddErr)*.

Packet Structure Reference

```
#define DN_FAL_SDU_SERVICE_MAX_DATA  (512)
typedef struct DN_FAL_SDU_SERVICE_REQ_Ttag
{
TLR_UINT8  bDevMacId;
TLR_UINT16 usSrvDatLen;           /* Effective Length of Data in abSrvData  */
TLR_UINT8  ubServiceCode;
TLR_UINT16 usClass;

TLR_UINT16 usInstance;
TLR_UINT16 usAttribute;
TLR_UINT8  usReserved[2];
TLR_UINT8  abSrvData[DN_FAL_SDU_SERVICE_MAX_DATA];
}DN_FAL_SDU_SERVICE_REQ_T;

/* PACKET: Remote Service Request                                          */
#define DN_FAL_SERVICE_REQ_SIZE \
             (sizeof(DN_FAL_PACKET_SERVICE_REQ_T)- sizeof(TLR_PACKET_HEADER_T))
typedef struct DN_FAL_PACKET_SERVICE_REQ_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                    /* Packet Header  */
  DN_FAL_SDU_SERVICE_REQ_T        tData;
} DN_FAL_PACKET_SERVICE_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_SERVICE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	524	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3822	DEVNET_FAL_CMD_REMOTE_SERVICE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_SERVICE_REQ_T			
bDevMacId		0-63	MAC ID of device
usSrvDatLen	UINT16	1-512	Effective Length of Data in abSrvData
ubServiceCode	UINT8	0-255	Service Code
usClass	UINT16	1 … 0xFFFF	CIP Class ID of the DeviceNet slave. (For pre-defined classes please refer to *"The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1" of the reference document [2])*
usInstance	UINT16	1 … 0xFFFF	Instance ID of the object with class ID above. . The instance ID in combination with class ID will form a unique identification of an object.
usAttribute	UINT16	1 … 0xFFFF	Attribute ID within the object. Pre-defined attributes of an object is described in details in object profiles of reference [2] and [3].
usSrvTimeOut	UINT16	0-0xFFFF (ms)	Service Time Out. For non-fragment 100 ms is enough.
abSrvData[512]	UINT8[]		Array for Remote Service Data

Table 109: DN_FAL_PACKET_SERVICE_REQ_T - Remote Service Request

Packet Structure Reference

```
typedef struct DN_FAL_SDU_SERVICE_CNF_Ttag
{
TLR_UINT8  bDevMacId;
TLR_UINT16 usSrvDatLen;           /* Effective Length of Data in abSrvData  */
TLR_UINT8  ubServiceCode;
TLR_UINT16 usClass;               /* This should be 1 Bytes variable?       */

TLR_UINT16 usInstance;            /* See specification of individual object */
TLR_UINT16 usAttribute;           /* See specification of individual object */
TLR_UINT8  bGenErr;               /* General error of service inquiry       */
TLR_UINT8  bAddErr;               /* Additional error of service inquiry    */
TLR_UINT8  abSrvData[DN_FAL_SDU_SERVICE_MAX_DATA];
}DN_FAL_SDU_SERVICE_CNF_T;

/* PACKET: Remote Service Confirm                                          */

#define DN_FAL_SERVICE_CNF_SIZE \
               (sizeof(DN_FAL_PACKET_SERVICE_CNF_T)-sizeof(TLR_PACKET_HEADER_T))
typedef struct DN_FAL_PACKET_SERVICE_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                  /* Packet Header   */
  DN_FAL_SDU_SERVICE_CNF_T        tData;
} DN_FAL_PACKET_SERVICE_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_SERVICE_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3823	DEVNET_FAL_CMD_REMOTE_SERVICE_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_SERVICE_CNF_T			
bDevMacId		0-63	MAC ID of device
usSrvDatLen	UINT16	1-512	Effective Length of Data in abSrvData
ubServiceCode	UINT8	0-31	Service Code
usClass	UINT16	1 … 0xFFFF	CIP Class ID of the DeviceNet slave. (For pre-defined classes please refer to *"The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1" of the reference document [2])*
usInstance	UINT16	1 … 0xFFFF	Instance ID of the object with class ID above. . The instance ID in combination with class ID will form a unique identification of an object.
usAttribute	UINT16	1 … 0xFFFF	Attribute ID within the object. Pre-defined attributes of an object is described in details in object profiles of reference [2] and [3].
bGenErr	UINT8		General error of service inquiry, see *Table 104: Generic Error (Variable bGenErr)*
bAddErr	UINT8		Additional error of service inquiry, see *Table 105: Additional Error (Variable bAddErr)*
abSrvData[512]	UINT8[]		Array for Remote Service Data

Table 110: DN_FAL_PACKET_SERVICE_CNF_T - – Confirmation of Remote Service Request

5.7.4 Local Service DEVNET_FAL_CMD_LOCAL_SERVICE_REQ/CNF

This packet allows accessing local services (i.e. services located at the DeviceNet Master itself).

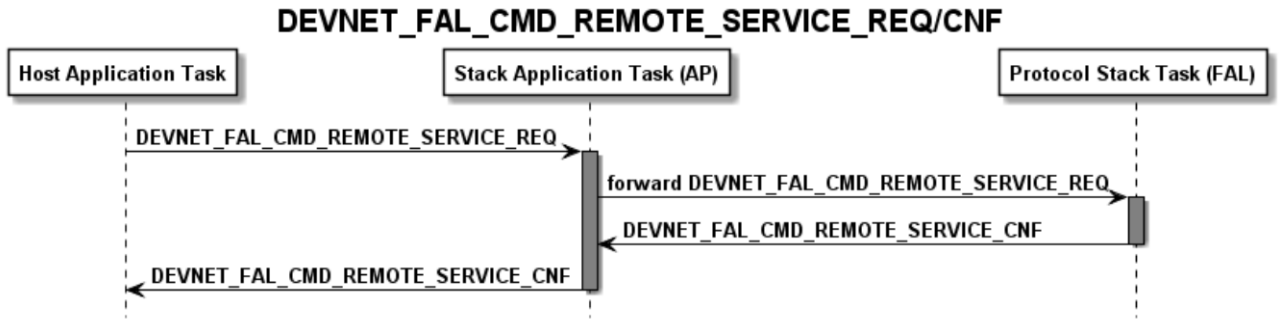

Figure 24: Sequence diagram of the DEVNET_FAL_CMD_LOCAL_SERVICE_REQ packet

The following services can be selected by setting parameter `ubServiceCode` to their respective service code according to the *Table 108: Service Codes*.

Note: However, only the services also mentioned in section 4.2 "*Object Modeling*" of this document, are really supported and accessible!

The class and the instance of the object to be accessed are selected by the variables `usClass` and `usInstance` of the request packet. If an attribute is affected by the service (services `Get_Attributes_All`, `Set_Attributes_All`, `Get_Attribute_Single` and `Set_Attribute_Single`), this attribute is selected by variable `usAttribute` of the request packet. Set `usAttribute` to 0 when using other services than these.

Specify the address (i.e. the MAC-ID) of the device on which the object to be accessed can be found in variable `bDevMacId` of the request packet.

The result of the requested service is delivered in array `abSrvData[512]` of the confirmation packet. How many bytes of array `abSrvData[512]` actually will be used can be specified in variable `usSrvDatLen` of the request packet.

In case of successful execution, the variables `bGenErrCode` and `bAddErrCode` of the confirmation packet will have the value 0.

In case of an error, the following happens:

- The variable `ubServiceCode` of the confirmation packet will have the value 0x14.
- The variables `bGenErrCode` and `bAddErrCode` of the confirmation packet will have non-zero values.

For lists of the general and additional error codes, see *Table 104: Generic Error (Variable bGenErr)* and *Table 105: Additional Error (Variable bAddErr)*.

Packet Structure Reference

```
#define DN_FAL_SDU_SERVICE_MAX_DATA  (512)
typedef struct DN_FAL_SDU_SERVICE_REQ_Ttag
{
TLR_UINT8  bDevMacId;
TLR_UINT16 usSrvDatLen;           /* Effective Length of Data in abSrvData  */
TLR_UINT8  ubServiceCode;
TLR_UINT16 usClass;

TLR_UINT16 usInstance;
TLR_UINT16 usAttribute;
TLR_UINT8  usReserved[2];
TLR_UINT8  abSrvData[DN_FAL_SDU_SERVICE_MAX_DATA];
}DN_FAL_SDU_SERVICE_REQ_T;

/* PACKET: Remote Service Request                                          */
#define DN_FAL_SERVICE_REQ_SIZE \
             (sizeof(DN_FAL_PACKET_SERVICE_REQ_T)- sizeof(TLR_PACKET_HEADER_T))
typedef struct DN_FAL_PACKET_SERVICE_REQ_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                    /* Packet Header  */
  DN_FAL_SDU_SERVICE_REQ_T        tData;
} DN_FAL_PACKET_SERVICE_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_SERVICE_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3824	DEVNET_FAL_CMD_LOCAL_SERVICE_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_SERVICE_REQ_T			
bDevMacId		0-63	MAC ID of device Use MAC ID of own device.
usSrvDatLen	UINT16	1-512	Effective Length of Data in abSrvData
ubServiceCode	UINT8	0-31	Service Code as mentioned in *Table 108: Service Codes*.
usClass	UINT16	1 … 0xFFFF	CIP Class ID of the DeviceNet slave. (For pre-defined classes please refer to *"The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1" of the reference document [2])*
usInstance	UINT16	1 … 0xFFFF	Instance ID of the object with class ID above. . The instance ID in combination with class ID will form a unique identification of an object.
usAttribute	UINT16	1 … 0xFFFF	Attribute ID within the object. Pre-defined attributes of an object is described in details in object profiles of reference [2] and [3].
usReserved[2]	UINT8[]		Padding for GenErr and AddErr
abSrvData[512]	UINT8[]		Array for Local Service Data

Table 111: DN_FAL_PACKET_SERVICE_REQ_T – Local Service Request

Packet Structure Reference

```
/* PACKET: Local Service Confirm                                        */

#define DN_FAL_SERVICE_CNF_SIZE \
               (sizeof(DN_FAL_PACKET_SERVICE_CNF_T)-sizeof(TLR_PACKET_HEADER_T))
typedef struct DN_FAL_PACKET_SERVICE_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                  /* Packet Header   */
  DN_FAL_SDU_SERVICE_CNF_T        tData;
} DN_FAL_PACKET_SERVICE_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_SERVICE_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x3825	DEVNET_FAL_CMD_LOCAL_SERVICE_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_SERVICE_CNF_T			
bDevMacId		0-63	MAC ID of device
usSrvDatLen	UINT16	1-512	Effective Length of Data in abSrvData
ubServiceCode	UINT8	0-31	Service Code. The same as in the request packet.
usClass	UINT16	1 … 0xFFFF	CIP Class ID of the DeviceNet slave. (For pre-defined classes please refer to *"The CIP Networks Library, Volume 1 Common Industrial Protocol Specification Chapter 5, Table 5-1.1" of the reference document [2])*
usInstance	UINT16	Valid instance	Instance ID of the object with class ID above. . The instance ID in combination with class ID will form a unique identification of an object.
usAttribute	UINT16	Valid attribute	Attribute ID within the object. Pre-defined attributes of an object is described in details in object profiles of reference [2] and [3].
bGenErr	UINT8		General error of service inquiry, see *Table 104: Generic Error (Variable bGenErr)*
bAddErr	UINT8		Additional error of service inquiry, see *Table 105: Additional Error (Variable bAddErr)*
abSrvData[512]	UINT8[]		Array for Local Service Data

Table 112: DN_FAL_PACKET_SERVICE_CNF_T - – Confirmation of Local Service Request

5.7.5 Get Attributes All Service DEVNET_FAL_CMD_GET_ATT_ALL_REQ/CNF

Not used.

5.8 Raw CAN Frame Service

5.8.1 CAN Registered Service DEVNET_FAL_CMD_CAN_FWD_REG_REQ/CNF

This service is used to register a CAN ID to the stack task. The CAN frame with the registered CAN ID will be forwarded to application task whenever it has been received from bus. The forwarding of CAN ID will be notified with DEVNET_FAL_CMD_CAN_FWD_IND.

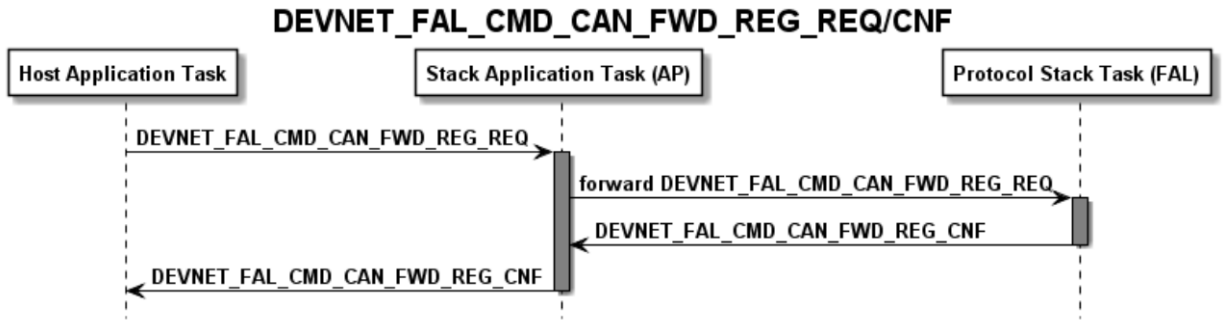

Figure 25: Sequence diagram of the DEVNET_FAL_CMD_CAN_FWD_REQ_REQ packet

Packet Structure Reference

```
typedef struct DN_FAL_SDU_CAN_FWD_REG_REQ_Ttag{
  TLR_UINT16 usCanId;                                  /**< CAN Identifier  */
}DN_FAL_SDU_CAN_FWD_REG_REQ_T;

typedef struct DN_FAL_PACKET_CAN_FORWARD_REQ_Ttag{
  TLR_PACKET_HEADER_T             tHead;                /**< Packet Header  */

  DN_FAL_SDU_CAN_FWD_REG_REQ_T    tData;                /**< Data           */
}DN_FAL_PACKET_CAN_FWD_REG_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_CAN_FWD_REG_REQ_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization.
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x38E2	DEVNET_FAL_CMD_CAN_FWD_REG_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_CAN_FWD_REQ_REQ_T			
usCanId	UINT8	0-2048	CAN ID. This variable is coded as shown in *Table 114: Coding of 16-bit CAN ID variable*.

Table 113: DN_FAL_PACKET_CAN_FWD_REG_REQ_T - Request registering a CAN ID for forwarding

En	Not used				CAN ID										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 114: Coding of 16-bit CAN ID variable.

The Enable Flag allows the user to register (Enable Flag = 1) or unregister (Enable Flag = 0) the CAN-frame with identifier specified in 11-bit CAN-ID field.

Packet Structure Reference

```
#define DN_FAL_CAN_FWD_REG_CNF_SIZE (0)
/*! @see DEVNET_FAL_CMD_CAN_FWD_REG_CNF */
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_PACKET_CAN_FORWARD_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                /**< Packet Header   */
}
DN_FAL_PACKET_CAN_FWD_REG_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_CAN_FWD_REG_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization.
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x38E1	DEVNET_FAL_CMD_CAN_FWD_REG_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 115: DN_FAL_PACKET_CAN_FWD_REG_CNF_T - Confirmation of registering a CAN ID for forwarding

5.8.2 CAN Forward Service DEVNET_FAL_CMD_CAN_FWD_IND/RES

The stack task uses this command to forward a CAN frame that it has received from bus. The CAN ID of the frame is one of the registered CAN IDs that are registered using

CAN Registered Service DEVNET_FAL_CMD_CAN_FWD_REG_REQ/CNF before.

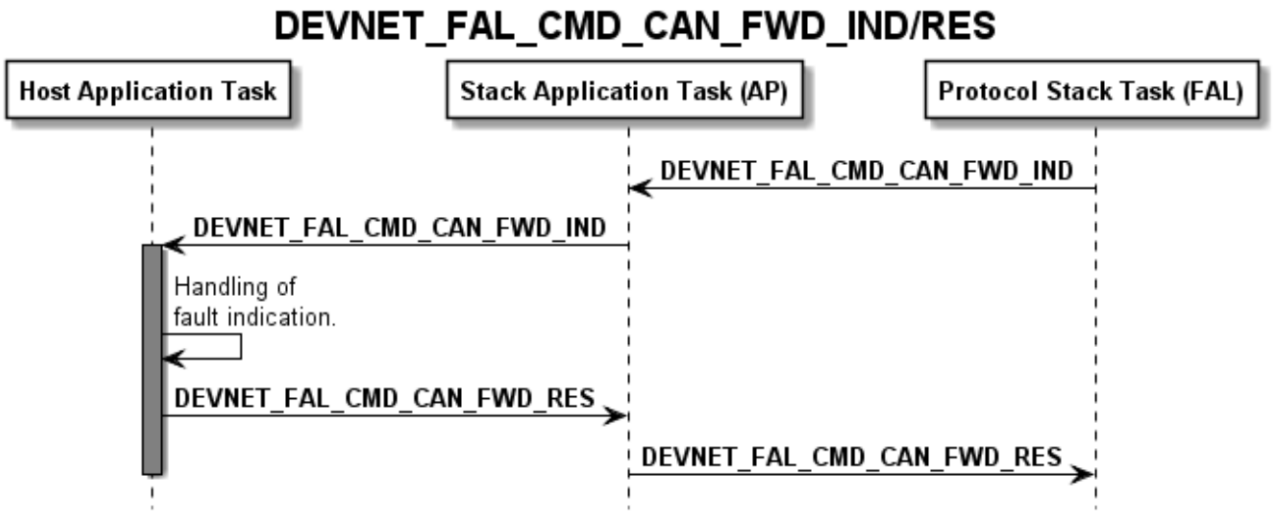

Figure 26: Sequence diagram of the DEVNET_FAL_CMD_CAN_FWD_IND packet

Packet Structure Reference

```
/*! SDU: Can Forward Indication and Response. @see DN_FAL_PACKET_CAN_FWD_IND_T*/
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_SDU_CAN_FRAME_Ttag
{
  /* |x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x| */
  /* .-------.-.---------------------. */
  /*  Length RTR     Identifier       */

  TLR_UINT16 usCanId;                              /**< CAN ID               */

  TLR_UINT8  ubReserved;                           /**< Reserved, set to 0   */

  TLR_UINT8  abDataLoad[8];                        /**< Dataload             */
}
DN_FAL_SDU_CAN_FRAME_T;

/* PACKET: Service Request/Confirm                                           */
#define DN_FAL_CAN_FWD_IND_SIZE (sizeof(DN_FAL_SDU_CAN_FRAME_T))
/*!Indication with #DEVNET_FAL_CMD_CAN_FWD_IND of CAN-ID that is registered
 * before using #DEVNET_FAL_CMD_CAN_FWD_REG_REQ
 */
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_PACKET_CAN_FWD_IND_Ttag
{
  TLR_PACKET_HEADER_T       tHead;                      /**< Packet Header  */

  DN_FAL_SDU_CAN_FRAME_T    tData;
}
DN_FAL_PACKET_CAN_FWD_IND_T;
```

Packet Description

Structure DN_FAL_PACKET_CAN_FWD_IND_T			**Type: Indication**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x38E4	DEVNET_FAL_CMD_CAN_FWD_IND - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_CAN_FRAME_T			
usCanId	UINT16	0-65535	CAN ID. Coding as bit field. See more in *Table 117*.
ubReserved	UINT8	0	Reserved. Set to 0.
abDataLoad[8]	UINT8[]		Dataload of the CAN frame.

Table 116: DN_FAL_PACKET_CAN_FWD_IND_T - Indication of a forwarded CAN frame.

The usCanId is coded as a bit field as following:

Data Load Length				RTR	CAN ID										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 117: - Coding of CAN ID variable

Packet Structure Reference

```
#define DN_FAL_CAN_FWD_RES_SIZE (0)
/*! @see DEVNET_FAL_CMD_CAN_FWD_RES */
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_PACKET_CAN_FWD_RES_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                /**< Packet Header   */
}
DN_FAL_PACKET_CAN_FWD_RES_T;
```

Packet Description

Structure DN_FAL_PACKET_CAN_FWD_RES_T			**Type: Response**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x38E5	DEVNET_FAL_CMD_CAN_FWD_RES - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 118: DN_FAL_PACKET_CAN_FWD_REG_CNF_T - Confirmation of registering a CAN ID for forwarding

5.8.3 DEVNET_FAL_CMD_CAN_DATA_REQ/CNF - CAN Data Request

The command DEVNET_FAL_CMD_CAN_DATA_REQ is used to send a CAN frame at any time on bus.

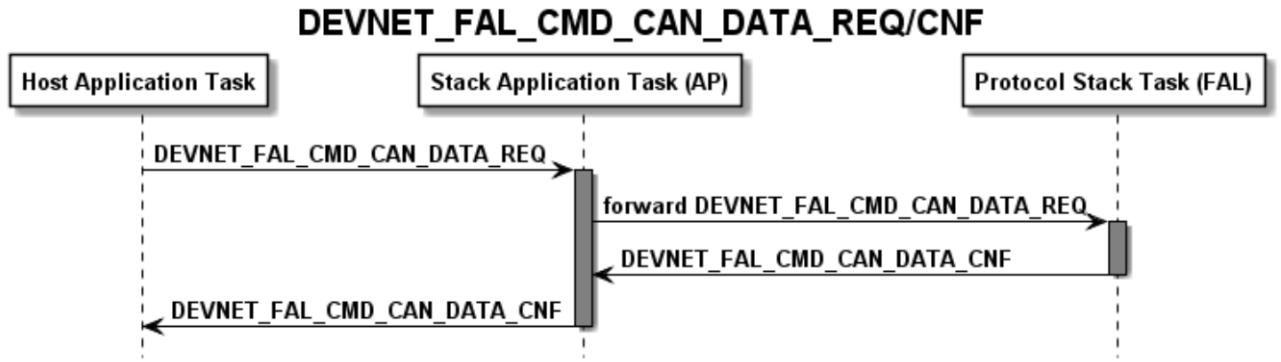

Figure 27: Sequence diagram of the DEVNET_FAL_CMD_CAN_DATA_REQ packet

Packet Structure Reference

```
/*! SDU: Can Forward Indication and Response. @see DN_FAL_PACKET_CAN_FWD_IND_T*/
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_SDU_CAN_FRAME_Ttag
{
  /* |x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x| */
  /* .-------.-.---------------------. */
  /*  Length RTR     Identifier        */

  TLR_UINT16 usCanId;                              /**< CAN ID               */

  TLR_UINT8  ubReserved;                           /**< Reserved, set to 0    */

  TLR_UINT8  abDataLoad[8];                        /**< Data load             */
}
DN_FAL_SDU_CAN_FRAME_T;

#define DN_FAL_CAN_DATA_REQ_SIZE (sizeof(DN_FAL_SDU_CAN_FRAME_T))
/*! Sending of CAN frame on bus at anytime using #DEVNET_FAL_CMD_CAN_DATA_REQ */
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_PACKET_CAN_DATA_REQ_Ttag
{
  TLR_PACKET_HEADER_T       tHead;                       /**< Packet Header  */

  DN_FAL_SDU_CAN_FRAME_T    tData;
}
DN_FAL_PACKET_CAN_DATA_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_CAN_DATA_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x38E6	DEVNET_FAL_CMD_CAN_DATA_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_CAN_FRAME_T			
usCanId	UINT16	0-65535	CAN ID. Coding as bit field. See more in this chapter
ubReserved	UINT8	0	Reserved. Set to 0.
abDataLoad[8]	UINT8[]		Data load of the CAN frame.

Table 119: DN_FAL_PACKET_CAN_DATA_REQ_T - Requesting the sending of a CAN frame on bus..

The usCanId is coded as a bit field as following:

Data Load Length				RTR	CAN ID										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Packet Structure Reference

```
#define DN_FAL_CAN_DATA_CNF_SIZE (0)
/*! @see DEVNET_FAL_CMD_CAN_DATA_CNF */
typedef __PACKED_PRE struct __PACKED_POST DN_FAL_PACKET_CAN_DATA_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;                /**< Packet Header   */
}
DN_FAL_PACKET_CAN_DATA_CNF_T;
```

Packet Description

Structure DN_FAL_PACKET_CAN_DATA_CNF_T			Type: Confirmation
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x38E7	DEVNET_FAL_CMD_CAN_DATA_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 120: DN_FAL_PACKET_CAN_DATA_CNF_T - Confirmation of registering a CAN ID for forwarding

5.9 Bus Diagnosis Services

5.9.1 Get Lifelist Service DEVNET_FAL_CMD_LIFELIST_REQ/CNF

In order to get an overview of all devices physically present in the current network constellation, the DEVNET_FAL_CMD_LIFELIST_REQ command can be used

Note: Only DeviceNet devices which have already finished their auto baud detection phase are taken into account. (This might last some seconds.)

With the request packet, you need to specify a value for the time how long the DeviceNet Master should wait for the response of the slaves.

The confirmation packet contains one byte per single slave indicating its status. The following values are possible:

Symbolic Name	Value	Meaning
DN_FAL_LIFELIST_OWN_MAC	0xFF	Responds with own MAC ID
DN_FAL_LIFELIST_MAC	0x01	Responds with MAC ID
DN_FAL_LIFELIST_NO_MAC	0x00	Does not respond

Table 121: Status Bytes in Life list

If a DEVNET_FAL_CMD_LIFELIST_REQ packet has already been sent earlier and is still not completely processed, the status of the confirmation packet will be set to TLR_E_DEVNET_FAL_LIFELIST_IN_PROGRESS indicating that processing of the life list is still in progress.

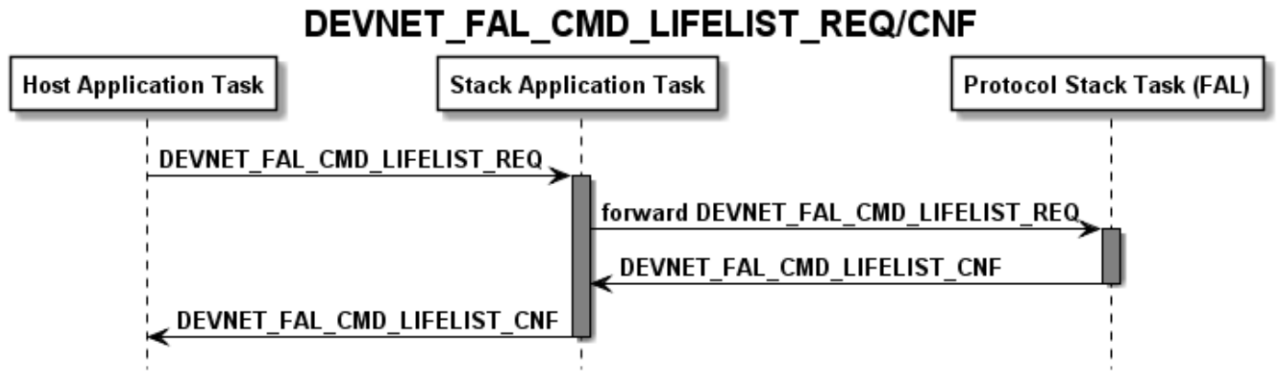

Figure 28: Sequence diagram of the DEVNET_FAL_CMD_LIFELIST_REQ packet

Packet Structure Reference

```
typedef struct DN_FAL_LIFELIST_REQ_Ttag
{
  TLR_UINT32 ulTimeOut;
} DN_FAL_LIFELIST_REQ_T;

typedef struct DN_FAL_PACKET_LIFELIST_REQ_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  DN_FAL_LIFELIST_REQ_T           tData;
} DN_FAL_PACKET_LIFELIST_REQ_T;
```

Packet Description

Structure DN_FAL_PACKET_LIFELIST_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3814	DEVNET_FAL_CMD_LIFELIST_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - Structure DN_FAL_SDU_SET_MODE_REQ_T			
ulTimeOut	UINT32		Time-out value specified in milliseconds

Table 122: DEVNET_FAL_CMD_LIFELIST_REQ – Generate Lifelist

Packet Structure Reference

```
typedef struct DN_FAL_LIFELIST_CNF_Ttag
{
  TLR_UINT8 abDevices[64];
} DN_FAL_LIFELIST_CNF_T;

typedef struct DN_FAL_PACKET_LIFELIST_CNF_Ttag
{
  TLR_PACKET_HEADER_T             tHead;
  DN_FAL_LIFELIST_CNF_T           tData;
} DN_FAL_PACKET_LIFELIST_CNF_T;
```

Packet Description

structure DN_FAL_PACKET_LIFELIST_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	64	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		*See section 6.2* Status/Error Codes DevNet FAL – Task
ulCmd	UINT32	0x3815	DEVNET_FAL_CMD_LIFELIST_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch
tData - structure DN_FAL_LIFELIST_CNF_T			
abDevices[64]	UINT8[]		Lifelist. If the MAC ID is alive on bus, the corresponding byte in this array is not null, e.g., if the device with MAC ID 10 is online on bus, the byte abDevices[10] will be set to 1.

Table 123: DEVNET_FAL_CMD_LIFELIST_CNF – Confirmation of Generate Lifelist

5.10 Register Application Services

5.10.1 Register Application Service DEVNET_FAL_CMD_AP_REGISTER_REQ/CNF

This service is used to register the application at the protocol stack. It is required for correct operation as without calling this service no indications would be sent back to the sending task or queue. This service shall be sent as the very first command before interacting with the stack.

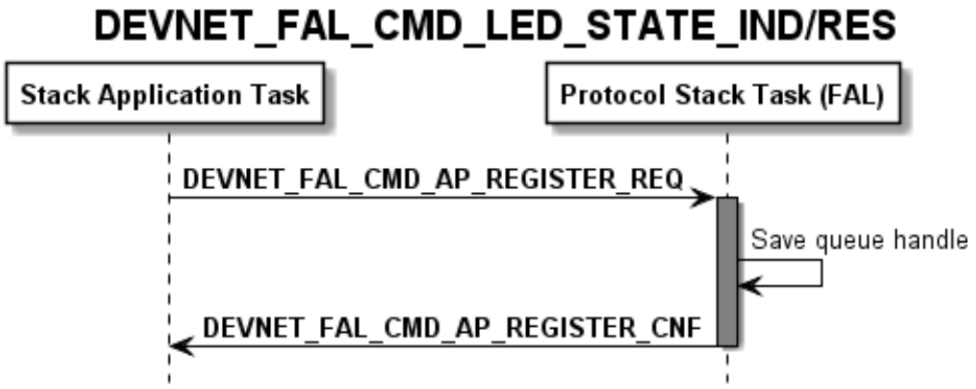

Figure 29: Sequence diagram of the DEVNET_FAL_CMD_AP_REGISTER_REQ packet

Packet Structure Reference

```
/** PACKET: Application register req/con ************************************/
typedef struct DN_FAL_PACKET_AP_REGISTER_REQ_Ttag
{
  TLR_PACKET_HEADER_T           tHead;                  /* Packet Header     */
} DN_FAL_PACKET_AP_REGISTER_REQ_T;
#define DN_FAL_AP_REGISTER_REQ_SIZE (0)
```

Packet Description

Structure DN_FAL_PACKET_AP_REGISTER_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32		Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x380E	DEVNET_FAL_CMD_AP_REGISTER_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 124: DN_FAL_PACKET_AP_REGISTER_REQ_T – Register Application

Packet Structure Reference

```
typedef struct DN_FAL_PACKET_AP_REGISTER_CNF_Ttag
{
  TLR_PACKET_HEADER_T           tHead;                  /* Packet Header     */
}DN_FAL_PACKET_AP_REGISTER_CNF_T;
#define DN_FAL_AP_REGISTER_CNF_SIZE (0)
```

Packet Description

Structure DN_FAL_PACKET_AP_REGISTER_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle
ulSrc	UINT32		Source Queue-Handle
ulDestId	UINT32		Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0 for the Initialization Packet
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	Packet Data Length in bytes
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section *Status/Error Codes DevNet FAL – Task*
ulCmd	UINT32	0x380F	DEVNET_FAL_CMD_AP_REGISTER_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	x	Routing, do not touch

Table 125: DN_FAL_PACKET_AP_REGISTER_CNF_T – Confirmation of Register Application

5.11 The CAN DL - Task

This chapter is for completeness to describe the structure of the DeviceNet stack. The CAN_DL Task provides the data link layer of a DeviceNet stack. The user has no direct access to this layer.

6 Status/Error Codes Overview

Error Codes and returned status are 32 bit values and they are unique! No double error codes will occur. Each Fieldbus/ communication Task has its own status/error range. Every used status/error code is defined in the task related error header coming with the API of the communication stack.

For the DeviceNet Master Stack following error header files are relevant:

File	Description
`TLR_global_error.h`	Definition of global user status/error codes
`CanDl_error.h`	Definition of CAN DL Task status/error codes
`DevNetFAL_error.h`	Definition of DevNetFAL Task status/error codes

Table 126: Header Files containing Status/Error Codes

6.1 Status/Error Codes DevNet AP – Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC05B0001	TLR_E_DEVNET_AP_COMMAND_INVALID Invalid command received.
0xC05B0002	TLR_E_DEVNET_AP_SERVICE_NOT_SUPPORTED Service not supported.
0xC05B0010	TLR_E_DEVNET_AP_NO_DATA_BASE No data base found.
0xC05B0011	TLR_E_DEVNET_AP_ERR_OPEN_DATA_BASE Error while opening data base
0xC05B0012	TLR_E_DEVNET_AP_ERR_READ_DATA_BASE Error while reading data base.
0xC05B0013	TLR_E_DEVNET_AP_TABLE_NOT_FOUND Table not found in data base.
0xC05B0014	TLR_E_DEVNET_AP_INVALID_DNM_DATA_BASE No valid DeviceNet data base.
0xC05B0100	TLR_E_DEVNET_AP_NON_EXCHANGE_SLAVE No data exchange with at least one slave.
0xC05B0101	TLR_E_DEVNET_AP_NON_EXCHANGE_ALL No slave in data exchange.
0xC05B0110	TLR_E_DEVNET_AP_ILLEGAL_PACKET_LENGTH Illegal packet length.
0xC05B0111	TLR_E_DEVNET_AP_WRONG_WD_VALUE Wrong watchdog.

Table 127: Status/Error Codes DevNet AP - Task

6.2 Status/Error Codes DevNet FAL – Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC0470002L	TLR_E_DEVNET_FAL_SERVICE_NOT_SUPPORTED Service not supported.
0xC0470003L	TLR_E_DEVNET_FAL_RESET_IN_REQUEST Reset is in request.
0xC0470004L	TLR_E_DEVNET_FAL_UNRECOVER_RESET_FAULT Unrecoverable reset fault.
0xC0470010L	TLR_E_DEVNET_FAL_SET_MODE_INVALID_MODE Invalid value for 'mode' of command.
0xC0470011L	TLR_E_DEVNET_FAL_SET_MODE_ALREADY_IN_REQUEST Command already in request.
0xC0470020L	TLR_E_DEVNET_FAL_CLR_CONFIG_NOT_ALLOWED_IN_ACTUAL_STATE Not allowed to clear configuration in actual mode.
0xC0470030L	TLR_E_DEVNET_FAL_DOWNLOAD_NOT_ALLOWED_IN_ACTUAL_STATE Download not allowed in actual state.
0xC0470031L	TLR_E_DEVNET_FAL_DOWNLOAD_INVALID_AREA_CODE Invalid value in 'AreaCode' of command.
0xC0470032L	TLR_E_DEVNET_FAL_DOWNLOAD_INVALID_SEQUENCE Sequence error.
0xC0470033L	TLR_E_DEVNET_FAL_DOWNLOAD_TO_MUCH_DATA Too much data.
0xC0470034L	TLR_E_DEVNET_FAL_DOWNLOAD_TO_LESS_DATA Less data.
0xC0470035L	TLR_E_DEVNET_FAL_DOWNLOAD_ALLREADY_CONFIGURED DeviceNet Stack already configured.
0xC0470036L	TLR_E_DEVNET_FAL_DOWNLOAD_FAULTY_CONFIGURATION DeviceNet Stack with a faulty configured loaded.
0xC0470037L	TLR_E_DEVNET_FAL_DPM_BYTE_ALREADY_MAPPED DPM Byte already mapped.
0xC0470038L	TLR_E_DEVNET_FAL_DPM_BYTE_OUT_OF_MEMORY DPM Byte out of DPM Memory.
0xC0470039L	TLR_E_DEVNET_FAL_UPLOAD_INVALID_AREA_CODE Invalid area code at upload
0xC0470100	TLR_E_DEVNET_FAL_BAUDRATE_OUT_OF_RANGE Invalid Baudrate.
0xC0470101	TLR_E_DEVNET_FAL_MAC_ID_OUT_OF_RANGE Invalid MAC ID.
0xC0470102	TLR_E_DEVNET_FAL_ADR_DOUBLE Slave already configured.
0xC0470103	TLR_E_DEVNET_FAL_DATA_SET_FIELD_LEN Invalid length of slave parameter set.
0xC0470104	TLR_E_DEVNET_FAL_PRED_MST_SL_ADD_LEN Invalid length of address table in parameter set.

Hexadecimal Value	Definition Description
0xC0470105	TLR_E_DEVNET_FAL_PRED_MSTSL_CFG_FIELD_LEN Invalid length of predefined master slave config table in parameter set.
0xC0470106	TLR_E_DEVNET_FAL_PRED_MST_SL_ADD_TAB_INCONS Inconsistency between address table and configured connection length.
0xC0470107	TLR_E_DEVNET_FAL_EXPL_PRM_FIELD_LEN Invalid Length of explicit parameter data in slave parameter set.
0xC0470108	TLR_E_DEVNET_FAL_PRED_MSTSL_CFG_ADD_INPUT_INCONS Inconsistency between number of input address offsets and configured input modules.
0xC0470109	TLR_E_DEVNET_FAL_PRED_MSTSL_CFG_ADD_OUTPUT_INCONS Inconsistency between number of output address offsets and configured output modules.
0xC047010A	TLR_E_DEVNET_FAL_UNKNOWN_DATA_TYPE Unknown data type in of the module definition.
0xC047010B	TLR_E_DEVNET_FAL_MODULE_DATA_SIZE Invalid data size in of the module definition.
0xC047010C	TLR_E_DEVNET_FAL_OUTPUT_OFF_RANGE Output address offset exceeds the maximum allowed area.
0xC047010D	TLR_E_DEVNET_FAL_INPUT_OFF_RANGE Input address offset exceeds the maximum allowed area.
0xC047010E	TLR_E_DEVNET_FAL_WRONG_TYPE_OF_CONNECTION Invalid type of connection configured.
0xC047010F	TLR_E_DEVNET_FAL_TYPE_CONNECTION_REDEFINITION Redefinition of connection type.
0xC0470110	TLR_E_DEVNET_FAL_EXP_PACKET_LESS_PROD_INHIBIT Configured 'Production Inhibit Time' is smaller then 'Expected Packet Rate'.
0xC0470111	TLR_E_DEVNET_FAL_PRM_FIELD_LEN_INCONSISTENT Invalid length of parameter field in slave parameter set.
0xC0470112	TLR_E_DEVNET_FAL_SET_BAUDRATE_FAIL Error while setting baudrate.
0xC0470113	TLR_E_DEVNET_FAL_REG_FRAG_TIMEOUT_OUT_OF_RANGE Invalid value of fragmentation timeout.
0xC0470114	TLR_E_DEVNET_FAL_PRM_OUT_MEMORY Out of memory for configuration data.
0xC0470211	TLR_E_DEVNET_FAL_CON_NA No response from device.
0xC0470215	TLR_E_DEVNET_FAL_CON_MDA Too much data received.
0xC0470233	TLR_E_DEVNET_FAL_CON_LE Invalid length of requested service.
0xC0470236	TLR_E_DEVNET_FAL_CON_AD Another service still active.
0xC0470239	TLR_E_DEVNET_FAL_CON_SE Sequence error in response sequence.
0xC0470240	TLR_E_DEVNET_FAL_CON_OC Explicit Message Handler is occupied.

Hexadecimal Value	Definition Description
0xC0470294	TLR_E_DEVNET_FAL_CON_ERR_RES Service Error Response.
0xC0470295	TLR_E_DEVNET_FAL_LIFELIST_IN_PROGRESS Life list request in progress.
0xC0470296	TLR_E_DEVNET_FAL_BTS_IN_PROGRESS Bit-strobe request in progress.
0xC0470297	TLR_E_DEVNET_FAL_BUS_NOT_ONLINE Bus not communicating.
0xC0470298	TLR_E_DEVNET_FAL_24V_NETWORK_POWER_MISSING 24V Network Power Missing
0xC0471000	TLR_E_DEVNET_FAL_SLAVE_NOEXCHANGE Service Error Response.

Table 128: Status/Error Codes DevNet FAL - Task

6.3 Status/Error Codes CAN DL – Task

Hexadecimal Value	Definition Description
0x00000000	TLR_S_OK Status ok
0xC03F0001	TLR_E_CAN_DL_COMMAND_INVALID Invalid command.
0xC03F0002	TLR_E_CAN_DL_CMD_LENGTH_MISMATCH The length code of the command is invalid.
0xC03F0003	TLR_E_CAN_DL_UNKNOWN_PARAMETER_TYPE The parameter type of the command "Set Parameter" is invalid.
0xC03F0004	TLR_E_CAN_DL_SET_MODE_FAILED Within the command "Set Parameter" the function set "CAN Mode" failed.
0xC03F0005	TLR_E_CAN_DL_SET_BAUDRATE_FAILED Within the command "Set Parameter" the function set "Baudrate" failed.
0xC03F0006	TLR_E_CAN_DL_SET_TXABORT_TIME_FAILED Within the command "Set Parameter" the function set "Transmission Abort Timer" failed.
0xC03F0007	TLR_E_CAN_DL_SET_EVENTS_REQUESTED_FAILED Within the command "Set Parameter" the function set "Requested Events" failed.
0xC03F0008	TLR_E_CAN_DL_SET_FILTER_FAILED Within the command "Set Parameter" or "Set Filter the function set "CAN Filter" failed.
0xC03F0009	TLR_E_CAN_DL_SET_ENABLE_DISABLE_RXID_FAILED Within the command Enable or Disable of receive identifiers an error occurred.
0xC03F000A	TLR_E_CAN_DL_TX_FRAME_FAILED At least one CAN frame could not be sent. Normally because the send process was aborted by the transmission abort timer.
0xC03F000BL	TLR_E_CAN_DL_TX_BUFFER_OVERRUN The send request of CAN frames was rejected because the internal buffer for send requests is full.
0xC03F000CL	TLR_E_CAN_DL_UNKNOWN_DIAG_TYPE The diagnostic type of the command "Get Diag" is invalid.
0xC03F000DL	TRL_E_CAN_DL_TX_ABORT_ALREADY_IN_REQUEST The command "Transmission Abort" is already requested.
0xC03F000EL	TRL_E_CAN_DL_TX_ABORT The send process of CAN frames was aborted by "Transmission Abort" command.
0xC03F000FL	TRL_E_CAN_DL_UNKNOWN_APPLICATION The application makes access, is not registered at CAN_DL task.
0xC03F0010L	TLR_E_CAN_DL_AP_ALREADY_REGISTERED The application is already registered.
0xC03F0011L	TLR_E_CAN_DL_CONF_LOCK_FAIL The configuration lock failed.
0xC03F0012L	TLR_E_CAN_DL_CONF_LOCKED The configuration is locked.

Table 129: Status/Error Codes CAN DL - Task

6.4 Generic Error Codes

The following table contains the possible General Error Codes defined within the CIP Standard.

General Status Code (specified hexadecimally)	Status Name	Description
00	Success	The service has successfully been performed by the specified object.
01	Connection failure	A connection-elated service failed. This happened at any location along the connection path.
02	Resource unavailable	Some resources which were required for the object to perform the requested service were not available.
03	Invalid parameter value	See status code 0x20, which is usually applied in this situation.
04	Path segment error	A path segment error has been encountered. Evaluation of the supplied path information failed.
05	Path destination unknown	The path references an unknown object class, instance or structure element causing the abort of path processing.
06	Partial transfer	Only a part of the expected data could be transferred.
07	Connection lost	The connection for messaging has been lost.
08	Service not supported	The requested service has not been implemented or has not been defined for this object class or instance.
09	Invalid attribute value	Detection of invalid attribute data
0A	Attribute list error	An attribute in the Get_Attribute_List or Set_Attribute_List response has a status not equal to 0.
0B	Already in requested mode/state	The object is already in the mode or state which has been requested by the service
0C	Object state conflict	The object is not able to perform the requested service in the current mode or state
0D	Object already exists	It has been tried to create an instance of an object which already exists.
0E	Attribute not settable	It has been tried to change a non-modifiable attribute.
0F	Privilege violation	A check of permissions or privileges failed.
10	Device state conflict	The current mode or state of the device prevents the execution of the requested service.
11	Reply data too large	The data to be transmitted in the response buffer requires more space than the size of the allocated response buffer
12	Fragmentation of a primitive value	The service specified an operation that is going to fragment a primitive data value, i.e. half a REAL data type.
13	Not enough data	The service did not supply all required data to perform the specified operation.
14	Attribute not supported	An unsupported attribute has been specified in the request
15	Too much data	More data than was expected were supplied by the service.
16	Object does not exist	The specified object does not exist in the device.
17	Service fragmentation sequence not in progress	Fragmentation sequence for this service is not currently active for this data.
18	No stored attribute data	The attribute data of this object has not been saved prior to the requested service.
19	Store operation failure	The attribute data of this object could not be saved due to a failure during the storage attempt.
1A	Routing failure, request packet too large	The service request packet was too large for transmission on a network in the path to the destination. The routing device was forced to abort the service.

General Status Code (specified hexadecimally)	Status Name	Description
1B	Routing failure, response packet too large	The service response packet was too large for transmission on a network in the path from the destination. The routing device was forced to abort the service.
1C	Missing attribute list entry data	The service did not supply an attribute in a list of attributes that was needed by the service to perform the requested behavior.
1D	Invalid attribute value list	The service returns the list of attributes containing status information for invalid attributes.
1E	Embedded service error	An embedded service caused an error.
1F	Vendor specific error	A vendor specific error has occurred. This error should only occur when none of the other general error codes can correctly be applied.
20	Invalid parameter	A parameter which was associated with the request was invalid. The parameter does not meet the requirements of the CIP specification and/or the requirements defined in the specification of an application object.
21	Write-once value or medium already written	An attempt was made to write to a write-once medium for the second time, or to modify a value that cannot be changed after being established once.
22	Invalid reply received	An invalid reply is received. Possible causes can for instance be among others a reply service code not matching the request service code or a reply message shorter than the expectable minimum size.
23-24	Reserved	Reserved for future extension of CIP standard
25	Key failure in path	The key segment (i.e. the first segment in the path) does not match the destination module. More information about which part of the key check failed can be derived from the object specific status.
26	Path size Invalid	Path cannot be routed to an object due to lacking information or too much routing data have been included.
27	Unexpected attribute in list	It has been attempted to set an attribute which may not be set in the current situation.
28	Invalid member ID	The Member ID specified in the request is not available within the specified class/ instance or attribute
29	Member cannot be set	A request to modify a member which cannot be modified has occurred
2A	Group 2 only server general failure	This DeviceNet-specific error cannot occur in EtherNet/IP
2B-CF	Reserved	Reserved for future extension of CIP standard
D0-FF	Reserved for object class and service errors	An object class specific error has occurred.

Table 130: General Error Codes according to CIP Standard

7 Quick Connect

The “Quick Connect” if supported on a device allows the device go online immediately after sending of the first duplicate MAC ID check CAN frame. The shortest time to establish IO exchange will be possible if both master and slave support “quick connect”. In case of only one of the master or slave support “quick connect”, the effect of reduced establishment-time will be minimum. Further information about “quick connect” will be provided in Volume 3 of the DeviceNet specification.

The “quick connect” on the master is enabled if one of the slave is configured with `DN_FAL_MSK_DEV_PRM_ENABLE_QUICK_CONNECT` (see Device Parameter EnableFlags). This “quick connect” could also be configured using Sycon.NET (see DeviceNet Master DTM OI xx EN.pdf for further details).

8 Appendix

8.1 List of Tables

8.2 List of Figures

8.3 Contacts

Headquarters

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France
Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support
Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India
Hilscher India Pvt. Ltd.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

Italy
Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support
Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan
Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support
Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea
Hilscher Korea Inc.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
E-Mail: info@hilscher.kr

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA
Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support
Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com