Protocol interface manual

DeviceNet Master

Hilscher Gesellschaft für Systemautomation mbH
Rheinstraße 15
D-65795 Hattersheim
Germany

Tel. +49 (6190) 9907 - 0
Fax. +49 (6190) 9907 - 50

Sales: +49 (6190) 9907 - 0
Hotline and Support: +49 (6190) 9907 - 99

Sales email: sales@hilscher.com
Hotline and Support email: hotline@hilscher.com

Homepage: http://www.hilscher.com

Index	Date	Version	Chapter	Revision
1	01.03.98	1.000	all	created
2	30.07.98	1.010	all	-extention of device specific error codes in chapter 3.3 -changing the meaning of UCMM-Support bit in the device specific download parameters
3	28.09.98	1.020	all	- insertion of new command Reset-Service - correction of BUS-parameter download message structure
4	16.02.99	1.030	all	- new chapter 'General procedure how to get the DEVICE operative without SyCon' - new chapter 'Deleting existing data base in the DEVICE' - new chapter 'Starting and Stopping Communication during Runtime' - new parameter WatchDogTime in wramstart parameter structure - new chapter 'Example of Message Device Parameter Data Sets hexdecimal'
5	19.05.99	1.043		- extention of globalbits in Protocol States
6	30.06.99	1.051		Examples for Device data set in Hexadecimal format
7	29.03.00	1.058 1.060		- Explanation of the Automatic Network Scan message - new server function Explicit Message Channel access through Mailbox
8	12.04.01	1.072		- new function in DNM_Download to save data sets in FLASH memory. - new Command DNM_RestoreFromFlash. - new Command DNM_Upload. - extention of device parameter data set, electronic keying. - extention of master parameter data set, server parameter. - start/stop devices connection online. - extention of global state field, new bit array showing status of the explicit message connection of the devices. - extention of bErr_event definitions - status bytes for each slave device in the I/O area of DEVICE, configureable via warmstart init parameter -Access of SyCon Configuration DBM-file using dbm32.dll - new function DNM_Explicit_Message
9	16.10.01	1.081		- correction of dual-port memory address bExtSlaveStatus from 0x1EC4 to 0x1EC5 - new chapter UCMM services - extention, DNM_Bit_Strobe command can now handle bit strobe devices also via message services. - new parameters in Protocol Parameter structure, to configure Product Specific information Online
10	03.07.02	1.083		- enlarged the address range of Class ID in DNM_Get_Set_Attribute client command from 255 up to 65535
11	28.08.02	1.084		- new warmstart parameter Bus Off - error codes in DNMRestoreFlash- Answer-Message inserted
12	23.06.04	>=1.097		- new connections support : COS and CYCLIC with Acknowledge in slave parameter section - bus off stop behavior as parameter in Protcol Parameter section corrected

Although this software has been developed with great care and intensively tested, Hilscher Gesellschaft für Systemautomation mbH cannot guarantee the suitability of this software for any purpose not confirmed by us in writing.

Guarantee claims shall be limited to the right to require rectification. Liability for any damages which may have arisen from the use of this software or its documentation shall be limited to cases of intent.

We reserve the right to modify our products and their specifiactions at any time in as far as this contributes to technical progress. The version of the manual supplied with the software applies.

1 Indroduction

This manual describes the user interface of DeviceNet for the communication interfaces and the communication module. The aim of this manual is to support the integration of these devices into own applications based on device driver functions or direct access into the dual-port memory.

The general mechanism for the data transfer is protocol independent and described in the 'general definitions' of the toolkit manual.

All parameters and data have basically the description LSB/MSB. This corresponds to the convention of the Microsoft-C-compiler.

1.1 Protocol Signification

To manage the DeviceNet protocol 2 tasks are involved in the system. Therefore following entries for the protocol signification in the variables `TaskiName` of the dual-port memory are done:

```
Task2Name: 'PLC     '
Task3Name: 'DNM     '
```

1.2 The Process Data Interface

The DEVICE handles up to 3584 bytes send and 3584 bytes receive process data in the lower 7 kbyte of the dual-port memory. To exchange the data between the DEVICE and the HOST use the device driver function `DevExchangeIO()`or read and write directly into these locations.
After calling the function `DevExchangeIO()`the function decides on its own which handhake mechanism has to be used to read and write the process data in the rigth manner from and to the DEVICE. If these locations are accessed directly without using the device driver functionality, than you have to use the right handshake mechanism to ensure that the data is handed over safety and valid. See chapter 'IO Communication with a Process Image' in the 'toolkit general definitions' manual.

Parameter	Address	Description
abSndPd	000H-DFFH	Send Process Data (HOST ? DEVICE ? Network)
abRecvPd	E00H-1BFFH	Receive Process Data (Network ? DEVICE ? HOST)

2 Protocol Parameter

Some important parameters can be handed over to the DEVICE online. They have a higher priority than the static parameter in the internal FLASH memory, coming from SyCon-configuration tool data base for example.

2.1 Using Device Driver Function to Write

To hand over these parameters use the device driver function `DevPutTaskParameter()`. For parameter `usNumber` use value 2, because the parameters must handed over to the task 2. For parameter `usSize` use value 49, to fix the length of the structure. Point the parameter `pvData` to the following structure below.

```
typedef struct DNM_INIT_PARAMETERtag {
  unsigned char   bMode;
  unsigned char   bReserved[2];
  unsigned short  usWatchDogTime;
  unsigned char   bExtSlaveStatus;
  unsigned char   bEnableFlags;
  unsigned short  usVendorId;
  unsigned short  usProductType;
  unsigned short  usProductCode;
  unsigned char   bMajorRev;
  unsigned char   bMinorRev;
  struct
  {
    unsigned char bLength;
    unsigned char bName[32];
  } tProduct;
  unsigned char bBusOffStop;
} DNM_INIT_PARAMETER;

/* values for bMode */
#define DNM_SET_MODE_BUFFERED_DEVICE_CONTROLLED 1
#define DNM_SET_MODE_UNCONTROLLED               2
#define DNM_SET_MODE_BUFFERED_HOST_CONTROLLED   3

/* values for bEnableExSlaveStatus */
#define DNM_EXTENDED_STATUS_OFF 0
#define DNM_EXTENDED_STATUS_ON  1

/* values for bEnableBits */
#define DNM_ENA_VID_TYPE_MASK  0x01
#define DNM_ENA_PROD_TYPE_MASK 0x02
#define DNM_ENA_PROD_CODE_MASK 0x04
#define DNM_ENA_PROD_NAME_MASK 0x08
#define DNM_ENA_MINOR_REV_MASK 0x10
#define DNM_ENA_MAJOR_REV_MASK 0x20
```

After setting up the structure and fixing it with `DevPutTaskParameter()`, the warmstart command must be performed with the `DevReset()` function. The most important parameter in this function `usMode` must be set up to 3 = WARM-START. After the warmstart is finished without error the new parameters are active.

2.2 Direct Write Access into the Dual-Port Memory

First the parameters must be written down into the corresponding area of the dual-port memory. Then a warmstart command must be activated by setting the `Init` bit in the variable `DevFlags`. Then the DEVICE will set them valid. (See the chapter 'initialization of the DEVICE' in the toolkit manual 'general definitions' for handling of the init procedure).

structure element	type	address	parameter
bMode	byte	1EC0H	process data delivery (1,2,3)
usWatchDogTime	word	1EC3H	HOST-supervision time in multiples of a msec.
bExtSlaveStatus	byte	1EC5H	enables/disables extended slave status information in IO area of DEVICE
bEnableFlags	byte	1EC6H	enable Register for identity variables
usVendorId	word	1EC7H	Device Vendor ID Code
usProductType	word	1EC9H	Device Product Type
usProductCode	word	1ECBH	Device Product Code
bMinorRev	byte	1ECDH	Device Minor Revision
bMajorRev	byte	1ECEH	Device Major Revision
tProduct	byte	1ECFH	Device Name
bBusOffStop	byte	1EF0H	Bus Off Behaviour

Protocol parameter in area `Task2Parameter`

2.3 Explanation of theProtocol Parameters

The first parameter `bMode` fixes the handshake mode of the process data.. Valid entries are 1,2,3. See the chapter 'Handshake mode of process data delivery' in the toolkit manual 'general definitions', for the explanation of the different modes.

The parameter `usWatchDogTime` fixes the time in multiples of 1msec. the DEVICE has to supervise the HOST program if it has started the HOST-watchdog functionality once. Read in manual 'Toolkit General definitions' how to activate and deactivate the DEVICE and HOST supervision.

The parameter `bExtSlaveStatus` enables or disables an array of 64 bytes in the IO area of the DEVICE, one byte for each slave device in ascending order, to hand over all their current online error conditions to the HOST interface in a summarized format. The structure is updated every device handler cycle which is between 400µsec and 1msec. While the structure `DNM_DIAGNOSTICS`, which is explained in the chapter 'Protocol States', indicates an online error of one device only, this contigous array has the advantage to have all slave online conditions available in the dual-port. Because the array structure is placed in the IO area of the DEVICE the receive process data area is now 64 less big. That means only 3520 can be used as inputs only, if the structure is enabled.

Here the layout of the DEVICEs IO area if the structure is enabled:

Parameter	Address	Description
abSndPd	000H - DFFH	Send Process Data (HOST ? DEVICE ? Network)
abRecvPd	E00H - 1BBFH	Receive Process Data (Network ? DEVICE ? HOST)
abErrEvent	1BC0H - 1BFFH	current online error for each slave device.

Use the following definitions to enable or disable the structure:
`#define DNM_EXTENDED_STATUS_OFF 0`: disable structure (default)
`#define DNM_EXTENDED_STATUS_ON 1`: enable strcuture

The values for each `abErrEvent` byte are the same like they are used for the global `bErrEvent` byte in the `DNM_DIAGNOSTICS` structure, which is defined in the following chapter 'Protocol States'.

The variable `bEnableFlags` decides if the following identity variables `usVendorId`, `usProductType`, `usProductCode`, `bMinorRev`, `bMajorRev` and the `tProductName` are filled up with valid data.

D7	D6	D5	D4	D3	D2	D1	D0
reserved		MAJREV	MINREV	NAME	CODE	TYPE	VID
		Product Major Revision variable enabled	Product Minor Revision variable enabled	Product name variable enabled	Product code variable enabled	Product type variable enabled	Product Vendor ID

If the corresponding bit is logical 1, the variable contains valid data and is enabled. During the startup phase of the DEVICE, it will copy the internal configured values of `usVendorId, usProductType, usProductCode, bMinorRev, bMajorRev` and the `tProductName` into the corresponding dual-port memory locations. The HOST application can change these entries afterwards and perform a WARMSTART. During this WARMSTART the DEVICE will take over the new variables (only if the assigned enable bit is set) and makes it present within the socalled Identity Object Handler which is adressable via the DeviceNet network.

The parameter `usVendorId` is a DeviceNet specific unique number which is fixed by the ODVA control center for each DeviceNet manufacturer. The DEVICE itself uses this digit to report it 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. We as office Hilscher for example got the number 283dec as Vendor-ID.

The variable `usProductType` is the indication of the general type of product. The Hilscher standard value for this is 12 which is a Communications Adpater.

The variable `usProductCode` is the identification of a particular product within a device type.

The variable `bMinorRev` is one part of the revision which identifies the revision of the DEVICE. The revision attribute consists of Major and Minor Revisions and they are typically displayed as major.minor.

The variable `bMajorRev` 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.

The variable `tProductName` 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 the variable `bLength` which is the first byte in the structure `tProduct`.

The configuration byte `bBusOffStop` defines how the DEVICE process the bus-off report of the CAN chip. In case of value 1 the DEVICE will automatically reset the CAN chip and continue communicating. In case of value 0 the DEVICE holds the CAN chip in its bus-off state and does not communicate any further. Only a warmstart or coldstart command can restart the master system in this case.

3 Controlled Start and Stop of I/O Connection

The slave connection status can be changed by the HOST online if the DEVICE is in state OPERATE.
The table that can be written to consists of an array of 64 bit, each for one slave, to control the current I/O communication of the slave. The bit field is evaluated by the DEVICE at least every millisecond. If the corresponding bit is cleared = logial '0', the DEVICE will stop the communication to the device. A logical '1' will start the communication and the DEVICE will instantaneously tries to reestablish the communication.

3.1 Using Device Driver Function to Write

To hand over these status information use the device driver function `DevReadWriteDPMRaw()`. For parameter `usMode` use value 2 = `PARAMETER_WRITE` to select the write option. For parameter `usOffset` use value `2F8H`, since this parameter selects the offset within the last kilobyte of the dual-port memory. The length indicator `usSize` which represents the size of the structure must be set to value 8. Point the parameter `pvData` to the following structure below.

```
typedef struct DNM_SLAVE_CONN_STATUStag {
  unsigned char   abConnStatus[8];
} DNM_SLAVE_CONN_STATUS;
```

3.2 Direct Write Access into the Dual-Port Memory

Write the parameter to the offset below.

structure element	type	address	parameter
abConnStatus	bit array	1EF8H-1EFFH	connection status of the devices

Online Status parameter in area `Task2Parameter`

3.3 Explanation of the Status parameter

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
1EF8H	7	6	5	4	3	2	1	0
1EF9H	15	14	13	12	11	10	9	8
1EFAH	23	22	21	20	19	18	17	16
...								
1EFFH	63	62	61	60	59	58	57	56

Relation table between device MAC address and the Connection Status bit

4 Protocol States

The protocol states built by the DEVICE form the diagnostic interface between the HOST and the DeviceNet.
The established structure informs about global busstates as well as individual states of the managed devices. To hold the information preferably compact, the device specific informations are stated in bitfields.
The first 4 variables in the structure informs about global bus state informations. They are followed by a bus error event and and a bus off counter. After this field an unused reserved area of 8 byte is following. The next 16 bytes characterize each device as configured and the next 16 bytes each device as active or inactive, followed by 16 bytes which serve to refer the diagnostic bit of every device station.

4.1 Using Device Driver Functions

Use the device driver function `DevGetTaskState()` to read the states. For parameter `usNumber` use value 2, because the parameter must be read from the task state area of the task 2. For parameter `usSize` use value 64, which is the length of the structure below. Point `pvData` to the following structure.

```
typedef struct DNM_DIAGNOSTICtag {
  struct
  {
    unsigned char bCtrl     : 1;
    unsigned char bAClr     : 1;
    unsigned char bNonExch  : 1;
    unsigned char bFatal    : 1;
    unsigned char bEvent    : 1;
    unsigned char bNRdy     : 1;
    unsigned char bDupMAC   : 1;
    unsigned char bPerDup   : 1;
  } bGlobalBits;

  unsigned char   bDNM_State;

  struct
  {
    unsigned char bErr_Dev_Adr;
    unsigned char bErr_Event;
  } tError;

  unsigned short  usBus_Error_Cnt;
  unsigned short  usBus_Off_Cnt;
  struct
  {
    unsigned char bSrvExpl   : 1;
    unsigned char bSrvIO     : 1;
    unsigned char bReserved  : 6;
  } bSrvStatus;
  unsigned char   abReserved[7];

  unsigned char   abDv_cfg  [16];
  unsigned char   abDv_state[16];
  unsigned char   abDv_diag [16];
} DNM_DIAGNOSTICS;
```

4.2 Direct Read Access in Dual-Port Memory

Read the bus status structure directly from the following dual-port memory location:

variable	type	address	short signification
bGlobalBits	1 byte	1F40H	global error bits
bDNM_State	1 byte	1F41H	main state of the DEVICE system
bErrDevAdr	1 byte	1F42H	faulty device address
bErrEvent	1 byte	1F43H	corresponding error specification
usBus_Error_Cnt	word	1F44H	number of detected low transmission qualities
usBus_Off_Cnt	word	1F46H	number of canceled transmissions and CAN-chip reinitializations
bSrvStatus	byte	1F48H	current status of the server connection
reserved	7 bytes	1F49H-1F5FH	reserved area
abDv_Cfg	16 bytes	1F50H-1F5FH	see the table below
abDv_State_Expl	8 bytes	1F60H-1F67H	see the table below
abDV_State_IO	8 bytes	1F68H-1F6FH	see the table below
abDv_Diag	16 Bytes	1F70H-1F7FH	see the table below

Global bus state field in area `Task2State`

4.3 Explanation of the Protocol States

- bGlobalBits

D7	D6	D5	D4	D3	D2	D1	D0
PDUP	DMAC	NRDY	EVE	FAT	NEXC	ACLR	CTRL

CONTROL-ERROR:
parameterization error

AUTO-CLEAR-ERROR:
DEVICE stopped the communication to all devices and reached the auto-clear end state

NON-EXCHANGE-ERROR
At least one device has not reached the data exchange state and no process data are exchange with it.

FATAL-ERROR:
Because of heavy bus error, no further bus communication is possible

EVENT-ERROR:
The DEVICE has detected bus short circuits. The number of detected events are fixed in the bus_error_cnt variable. The bit will be set when the first event was detected and will not be deleted any more.

HOST-NOT-READY-NOTIFICATION:
Indicates if the HOST program has set its state to operative or not. If the bit is set the HOST program ist not ready to communicate

DUPLICATE-MAC-ID detected
Indicates that the DEVICE has found another device in the network which has the same MAC ID

DUPLICATE-MAC-ID check is performed
As long this bit is set the DEVICE is involved in handling the duplicate MAC-ID check. A duplicate MAC-ID check will be ready, if the DEVICE finds other DeviceNet devices in the network against the check can be performed.

The bit field serves as collective display of global event indications. Notified errors can either occur at the DEVICE itself or at the handled devices. To distinguish the different errors the variable `Err_dev_adr` contains the error location (address=MAC-ID), while the variable `Err_event` is fixing the corresponding error number. If more than one error is determined, the error location will contain always the device with lowest MAC-ID.

- Variable `bDNM_State`

This variable represents the main state of the DEVICE system. Following values are possible:

00H: state OFFLINE
40H: state STOP
80H: state CLEAR
C0H: state OPERATE

- Variable `bErrDevAdr`

If either the bits `CTRL`, `ACLR` or `NDATA` are set, this variable fixes the closer location of the error. If the source of the error is determined inside the DEVICE itself, the value 255 is written in. Else the faulty device address = MAC-ID is written in directly.

- Variable `bErrEvent`

This variable is a closer specification of the error of the faulty participant reported in `Err_dev_adr` . See the table below for further explanations of the error codes.

- Variable `bBus_Error_Cnt`

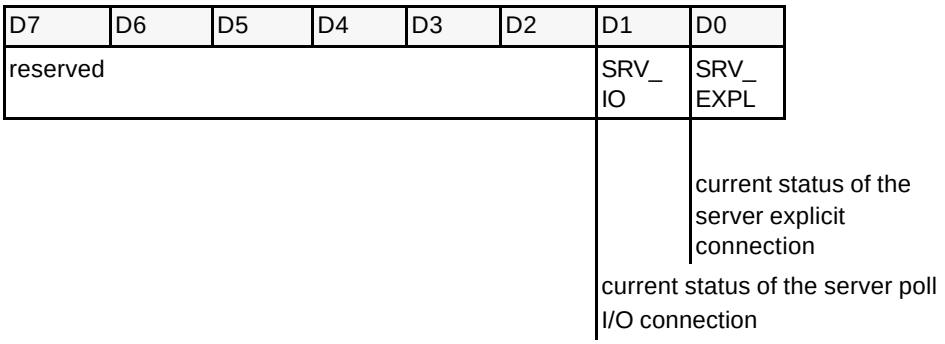
This variable will be incremented whenever the error frame counter of the used Philips CAN chip has reached warning limit because of a disturbed bus communication.

- Variable `bBus_off_cnt`

This variable will be incremented when the CAN chip reports that it is not longer involved in bus activities because of its overstepped bus error frame counter. The chip must be reinitialised then which is done automatically by the DEVICE afterwards.

- Variable `bSrvStatus`

The DEVICE is able to be a IO server at the same time beeing a master. The statu of the server poll IO connection and the explicit connection can be read from this variable.



- Variable reserved

This data block is reserved.

- Variable `abDv_Cfg`

This variable is a field of 16 bytes and contains the paremeterization state of each device station. The following table shows, which bit is related to which device station address:

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
1F50H	7	6	5	4	3	2	1	0
1F51H	15	14	13	12	11	10	9	8
1F52H	23	22	21	20	19	18	17	16
...								
1F57H	63	62	61	60	59	58	57	56

Table of the relation between device address and the Dv_Cfg bit

If the `abDv_cfg` bit of the corresponding device is logical

'1', the device is configured in the DEVICE, and serviced in its states.

'0', the device is not configured in the DEVICE

- Variable `abDv_State_Expl`

This variable is a field of 8 bytes and contains the explicit connection status of each device station. The following table shows, which bit is related to which device address:

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
1F60H	7	6	5	4	3	2	1	0
1F61H	15	14	13	12	11	10	9	8
1F62H	23	22	21	20	19	18	17	16
...								
1F67H	63	62	61	60	59	58	57	56

Table of the relation between device address and the Dv_State_Expl bit

If the `abDv_State_Expl` bit of the corresponding device is logical

'1', the devices explicit connection is in established state

'0', the devices explicit connection is not ins established state

The values in variable `abDv_State_Expl` are only valid, if the DEVICE runs the main state OPERATE.

- Variable `abDv_State_IO`

This variable is a field of 8 bytes and contains the I/O connection status of each device station. The following table shows, which bit is related to which device address:

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
1F68H	7	6	5	4	3	2	1	0
1F69H	15	14	13	12	11	10	9	8
1F6AH	23	22	21	20	19	18	17	16
...								
1F6FH	63	62	61	60	59	58	57	56

Relation table between device address and the Dv_State_IO bit

If the `abDv_State_IO` bit of the corresponding device is logical
'1', the devices I/O connection is in established state
'0', the devices I/O connection is not ins established state

- Variable `abDv_Diag`

This variable is a field of 16 bytes containing the diagnostic bit of each device. The following table shows the relationship between the device address and the corresponding bit in the variable `abDv_diag`.

Bit / Offset	D7	D6	D5	D4	D3	D2	D1	D0
1F70H	7	6	5	4	3	2	1	0
1F71H	15	14	13	12	11	10	9	8
1F72H	23	22	21	20	19	18	17	16
...								
1F77H	63	62	61	60	59	58	57	56

Table of the relationship between device address and the Dv_diag bit

If the `abDv_diag` bit of the corresponding device is logical
'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 were change in the internal diagnostic buffer.

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

	abDv_State_IO = 0	abDv_state_IO = 1
abDv_Diag = 0	- device not operative, no process data exchange between DEVICE and device - device is not configured	- device is present on the network, device guarding active - process data exchange between DEVICE and device happens as configured
abDv_Diag = 1	- device is not operating, device guarding failed or configuration fault detected. - DEVICE provides new diagnostic data in the internal diagnostic buffer to be read out by HOST	- device is present on the bus, device guarging is active, process data exchange - DEVICE provides new diagnostic data in the internal diagnostic buffer to be read out by HOST

Relationchip between abDv_State_IO bit and abDv_Diag bit

The following error numbers are valid for `bErrevent`, if `bErrRemAdr` is 255:

bErrEvent	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 detetcted in the network	configuration or network
58	no device entry found in the actual configuration database	download error in the current data base,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	master has stopped bus communication because of CAN based bus off error. Parameter bBusOffStop in paramter area has been configured to ON.	network error
226	master firmware downloaded to slave EC1 device	User Error

The following error numbers are valid for `bErrEvent`, if `bErrRemAdr` is unequal 255:

bErrEvent	signification	error source	remarks
0	no error		
1	device guarding failed, after device was operational	device	check if device is still running
30	device access timeout	device	device do not respond, check the baudrate and its MAC-ID
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 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 DEVICE) 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 / DEVICE	use single device diagnostic to get real consumed connection size
39	connection already in request	device	connection will be automatically released
40	number of CAN-message data bytes in read produced or consumed connection size response unequal 4	device	device don't have operability with DEVICE and norm description
41	predefined master slave connection already exits	device / DEVICE	connection will be automatically released
42	length in polling device response unequal produced connection size	device	
43	sequence error in device polling response	device	two first 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	sequence error in device COS or cyclic response	device	two first segments in multiplexed transfer were received

bErr_event	signification	error source	remarks
48	fragment error in device COS or cyclic response	device	fragmentation counter while multiplexed transfer differs from the awaited one
49	sequence error in device COS or cyclic response	device	middle or last segment was received before the first segment
50	length in COS or cyclic device response unequal produced connection size	device	
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/Oconfiguration 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,WORD,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 possible address range of 3584 bytes	configuration	the process data image is limited to 3584 bytes
70	configured input address of one module oversteps the possible address range of 3584 bytes	configuration	the process data image is limited to 3584 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

5 The Message Interface

The following send and receive messages are exchanged with the DEVICE via its mailboxes in the structure like it is described in the chapter 'definition of the message interface' in the toolkit manual.

To put and get messages to respectively from the DEVICE through its mailboxes use the device driver functions `DevPutMessage()` or `DevGetMessage()`. In direct access into the dual-port memory you have to write the message in the `DevMailbox` or read the message out from the `HostMailbox` by using the handshake mechanism described in the toolkit manual.

The structures of this messages and its values are described in the sections below.

5.1 The PLC-Task

The PLC-Task manages the process input and output data and handle the steering of the DNM cycles corresponding the parameterization. Therefore the task communicates with the DNM-Task and starts the device handling according to the parametrized handshake operation mode. The task has implemented the following functions:

- activates DNM cycles.
- do handshake sychronization with the user application

The task manages following message command:

DNM_Shared_memory	write consistent data block into the send process data `SndPd` or read consistent data block from `RecvPd`.

5.1.1 DNM_Shared_memory

	command message			
Message header	variable	type	value	signification
	msg.rx	byte	2	receiver = PLC-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	 8 8+m	length of the message read access write access
	msg.nr	byte	j	number of message (optional)
	msg.a	byte	0	no answer
	msg.f	byte	0	no error
	msg.b	byte	17	command = DNM_Shared_Memory
	msg.e	byte	0	unused
Extended message header	msg.device_adr	byte	0	device address unused
	msg.data_area	byte	 0 1 2	data area: msg.Function decides the data area receive process data area send process data area
	msg.data_adr	word	 0-1791 0-3583 0-1791	address offset refer to the data type word-offset address Byte-offset address word-offset address if bit access
	msg.data_idx	byte	0-15	bitposition within the word offset address if bit access
	msg.data_cnt	byte	m	count of read or write data referring to the data type
	msg.data_type	byte	 6 5,10 14	data type: TASK_TDT_UINT16: word TASK_TDT_UINT8: octet-string TASK_TDT_BIT: bit
	msg.function	byte	 1 2	function: TASK_TFC_READ = read access TASK_TFC_WRITE = write access
Data	msg.d[0]	byte	x	write access: first data to be written read access: unused
	...	...	...	...
	msg.d[m-1]	byte	z	write access: last data to be written read access: unused

The command serves the user program to write data of a definite length into the send process data buffer or to read from the receive process data. The command can be used in all process data handshake modes.
With the command the data types word, bytes or bits can be selected. The firmware of the DEVICE guarantees that the read or write access of the data will be done safely during two active process data transfers.
This command is an other possibility to get access to the process data in the dual-port memory. Its disadvantage is the slower access then the direct reading or writing the dual-port memory. But the main advantage is that you have the access to

the process data also via a message, when an old driver for example has already message functionality in it. We use this message in all diagnostic tools too.

The data type is fixed in the byte `msg.DataType`. Only the values decimal 6 for words, 5 or 10 for byte strings and 14 for bits are allowed.

The read access is distinguished from the write access in the byte `msg.Function`. A 1 is valid for read access and 2 for write access.

The data area to be read from, is fixed in `msg.DataArea`. 1 is valid for the receive process data buffer and 2 for the send process data buffer. In case of the value 0, the value placed in `msg.Function` decides the data area automatically.

The count of the data to be read or to be written is fixed by the value of `msg.DataCnt`. The count refers to the chosen data type. Maximum permitted values are 119 for words, 239 for byte and 255 for bits.

The offset address is fixed in the word `msg.DataAdr`. The specified address must be refered on the chosen data type and is interpreted from the DEVICE as the relative address to the start address in the send process data or the receive process data. The maximum values are decimal 1791 for word, 3583 for byte and 1791 for bit access. In case of bit access the value in `msg.DataIdx` additionally fixes the relative offset in the word to be read or to be written. For the other accesses the value doesn't have any meaning.

The data at `msg.d[]` are unused if read access is chosen, while in case of a write access the send data must be written into it. Words must be written in Intel format - LSB before MSB - and bits must be put in there in packed form. For example to write 5 bits, the first databyte `msg.d[0]` must be placed in the bits 0-4 to be valid.

answer message to the user			
variable	type	value	signification
msg.rx	byte	16	receiver = user at HOST
msg.tx	byte	2	transmitter = PLC-Task
msg.ln	byte	 8+m 8 0	length of the message read access write access error
msg.nr	byte	j	number of message
msg.a	byte	17	answer = DNM_Shared_Memory
msg.f	byte	0 f	no error error nummber see following table
msg.b	byte	0	no command
msg.e	byte	0	unused
msg.device_adr	byte	0	device address unused
msg.DataArea	byte	 0 1 2	data area: datafnc decides the data area receive process data area send process data area
msg.DataAdr	word	 0-1791 0-3583 0-1791	address offset refer to the data type word-offset address Byte-offset address word-offset address if bit access
msg.DataIdx	byte	0-15	bit position within the word offset address if bit access
msg.DataCnt	byte	m	count of read or write data referring to the data type
msg.DataType	byte	 6 5,10 14	data type: TASK_TDT_UINT16: word TASK_TDT_UINT8: octet-string TASK_TDT_BIT: bit
msg.Function	byte	 1 2	function: TASK_TFC_READ = read access TASK_TFC_WRITE = write access
msg.d[0]	byte	x	read access: first data be read write access: unused
...	...	...	...
msg.d[m-1]		z	read access: last data be read write access: unused

In the answer message the `msg.f` byte reports back the information, if the command could be executed. If the error byte is 0, then the service could be finished without any error.

The extended telegram header of the answer message is not be changed if the command could be executed correctly, so that the user can assign the answer message unequivocally to the sent command message.

The convention for the data `msg.d[]` coming back in the answer message is the same one as the convention for the command messages. That means that words will be placed in Intel format and bits are placed in packed format.

The byte `msg.f` in the answer message can have the following values:

error number msg.f	signification
0	no error, command executed
162	illegal address area
163	data address together with the data count results an overflow in the buffer length
165	illegal data count
166	unknown data type
167	unknown function

5.2 The DNM-Task

The DNM-Task includes the three functional blocks:

- device handler
- DNM message queue management
- mapping of the physical addresses of the data to the logical addresses of the data in the dual-port memory.

For every device station one state machine is implemented. These indiviual state machines named device handler are managing the several states of each device like getting diagnostic, doing parameterization, doing configuration and doing the process data exchange.

The DNM-Task has implemented the following messages for the HOST:

`DNM_Start_Seq`	start download of parameters
`DNM_End_Seq`	end of download
`DNM_Download`	download of parameters
`DNM_Device_Diag`	read internal saved diagnostic data of one device
`DNM_Get_Set_Attriute`	read or write a definite attribute at the device

The DNM-Task needs the configuration data for the DeviceNet.

Normally the configuration will be downloaded by the SyCon-DNM configuration tool statically into the FLASH memory. The task reads out this data block when the DEVICE starts up. If all parameters are valid the task starts its device handlers and goes into the mode OPERATE.

If no static download of the configuration data is wished, all these data can be handed over online to the DEVICE by a message download from the HOST program. But before doing this, you have to prevent the DEVICE to start up with possible downloaded static parameters. This can be done by deleting the data base 'DNM' with the ComPro tool before. Then the DEVICE starts up without finding any configuration data file and the online message download can be done by the HOST program like it is described in the following chapters.
NOTE! If no data base exists on the DEVICE, the DEVICE must be initialized with protocol parameters first (see chapter: protocol parameters) before the message download can be done, to fix the process data handshake mode.

ANNEX: We recommend to order the DeviceNet specification, when the online download of the parameters is used. The most configuration data containments of the messages have exactly the same meaning and functionality like it is described in the norm specification.

5.2.1 Starting and Stopping of whole Communication during Runtime

5.2.1.1 Using Device Driver Function to Write

Use the function `DevSetHostState()` together with the parameter `HOST_NOT_READY` to stop the network communication. Use the parameter `HOST_READY` to start or restart the communication.

5.2.1.2 Direct Write Access in Dual-Port Memory

To start and stop the communication of the DEVICE you have to clear and set the bit `NotRdy` in the cell `bDevFlags`. Clearing the bit will start the network communication while setting the bit will stop it.

ATTENTION: Stopping the communication will always cause a reset of the network modules output data.

5.2.2 Deleting Existing Data Base in the DEVICE

Normally the configuration will be downloaded by the SyCon configuration tool statically into the FLASH memory. The DEVICE reads out this data block during its startup. If all parameters are valid the DEVICE starts its slave handlers and goes into the mode OPERATE. Then the message download procedure like it is described in the chapters below can not be used any more.

If no static download of the configuration data is wished, all these data can be handed over online to the DEVICE by a message download from the HOST program using the functions Start,End and Download. But before doing this, you have to prevent the DEVICE to start up with possible downloaded static parameters. This can be done by deleting the data base by message service before. Then the DEVICE starts up without finding any configuration data base and then the online message download can be proceeded by the HOST program like it is described in the following chapters.
IMPORTANT NOTE! If no data base exists within the DEVICE, the DEVICE must be initialized with protocol parameters (see chapter: protocol parameters) before the message download is done, to fix the process data handshake mode etc.

	command message			
	variable	type	value	signification
Message Header	msg.rx	byte	0	receiver = RCS-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	2	length of the message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	no error
	msg.b	byte	6	command = data base access
	msg.e	byte	0	extention, not used
Service header	msg.d[0]	Byte	4	mode = delete data base
	msg.d[1]	Byte	8	startsegment of the data base

answer message			
variable	type	value	signification
msg.rx	byte	16	receiver = user at HOST
msg.tx	byte	0	transmitter = RCS-Task
msg.ln	byte	1	length of message
msg.nr	byte	j	number of the message
msg.a	byte	6	answer = data base access
msg.f	byte	f	error, state
msg.b	byte	0	no command
msg.e	byte	0	extension

The time for deleting the data base depents on the used FLASH memory, so sending back the answer message can take up to 3 seconds

5.2.3 DNM_Start_Seq

	command message			
	variable	type	value	signification
Message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	4	length of the message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	no error
	msg.b	byte	67	command = DNM_Start_Seq
	msg.e	byte	k	extension
DNM_START_SEQ_REQUEST	msg.d[0]	Byte	0	Req_Adr, unused
	msg.d[1]	Byte	1-126	Area_Code, device address
	msg.d[2]	Word	0-65535	Timeout, mutiple of 1 ms

The command starts a blocked download in the stated `Area_Code`, if the device parameter file to download is larger than one download message can contain. To complete the download the command `DNM_End_Seq` must be called after finishing the download sequence.

answermessage			
variable	type	value	signification
msg.rx	byte	16	receiver = user at HOST
msg.tx	byte	3	transmitter = DNM-Task
msg.ln	byte	1	length of message
msg.nr	byte	j	number of the message
msg.a	byte	67	answer = DNM_Start_Seq
msg.f	byte	f	error, state
msg.b	byte	0	no command
msg.e	byte	k	extension
msg.d[0]	byte	240	Max_Len_Data_Unit

The value `Max_Len_Data_Unit` fixes the maximum length of the parameter `Data` for each following `DNM_Download` message.

Possible values for `msg.f` are the following:

error number msg.f	signification
0	no error
52	Area_Code unknown

See below the corresponding structures in the header file:

```
DNM_START_SEQ_REQUEST
DNM_START_SEQ_CONFIRM
```

5.2.4 DNM_End_Seq

	command message			
	variable	type	value	signification
Message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	1	length of the message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	no error
	msg.b	byte	69	command = DNM_End_Seq
	msg.e	byte	k	extension
DNM_END_SEQ_REQUEST	msg.d[0]	byte	0	Req_Adr, unused

The command finalize the blocked download to the internal buffer and activates the previously sequenced downloaded parameter data and its consistency check.

answer message			
variable	type	value	signification
msg.rx	byte	16	receiver = user at HOST
msg.tx	byte	3	transmitter = DNM-Task
msg.ln	byte	0	length of message
msg.nr	byte	j	number of the message
msg.a	byte	69	answer = DNM_End_Seq
msg.f	byte	f	error, state
msg.b	byte	0	no command
msg.e	byte	k	extension

The values for `msg.f` are equal to the value returned in the `DNM_Download` answer message:

See below the corresponding structure in the header file:

`DNM_END_SEQ_REQUEST`

5.2.5 DNM_Download

	command message			
	variable	type	value	signification
Message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	m + 4	length of the message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	no error
	msg.b	byte	68	command = DNM_Download
	msg.e	byte	k	extension
DNM_DOWNLOAD_REQUEST	msg.d[0]	byte	0	Req_Adr, unused
	msg.d[1]	byte	0-63 127 255	Area_Code, device address DEVICE bus parameters DEVICE bus parameters and store all data in FLASH memory afterwards
	msg.d[2]	word	0-760	Add_Offset, if sequenced device parameter download is used
	msg.d[4-240]	m bytes	0-255	Data[240] per message

This command allows to hand over the DEVICE bus parameters or the device parameter data files. This is commendable, if no static data base exists on the DEVICE and the parameterization should happen from the HOST program without using SyCon-DNM tool.
Two ways to download data files have been implemented. A data file can be downloaded either in one call (single download) or if it is to large in block calls (sequenced download) into an internal download area (length 1000 bytes). After the download cycle is finished completly the specified data is checked and copied afterwards into the task access area. Then the next download can be started into the freed download area.
The parameter `Area_Code` fixes the destination area (DEVICE bus parameter or slave parameter file). A special value is 255. It loads the busparameter like value 127 but stores all data including bus parameter and device parameter data set in the FLASH memory. The data can be restored by using the command `DNM_RestoreFromFlash`.
The offset in the download area where the data will be copied from the message is fixed in the variable `Add_Offset`.
If a device data file shall be transferred sequenced, the command `DNM_Start_Seq` must be activated before to initialize the download sequence. The sequence will be finished after the command `DNM_End_Seq` is called. Even then the parameters will be checked and be set valid if no error is recognized. The download of the bus parameters needs no sequenced download.
After the bus parameter are loaded the DEVICE starts immediatly with the network communication.

See below the corresponding structure in the header file:

```
DNM_DOWNLOAD_REQUEST
```

answer message			
variable	type	value	signification
msg.rx	byte	16	receiver = user at HOST
msg.tx	byte	3	transmitter =DNM-Task
msg.ln	byte	0	length of message
msg.nr	byte	j	number of the message
msg.a	byte	68	answer = DNM_Download
msg.f	byte	f	error, state
msg.b	byte	0	no command
msg.e	byte	k	extension

Possible values for `msg.f` are the following:

error number msg.f	signification
0	no error
57	duplicate MAC-ID detected in network
59	device address already configured in actual configuration
60	whole size indicator of one device data set is corrupt
61	size of the additional table for predefined master slave connections is corrupt
62	size of predefined master slave I/Oconfiguration table is corrupt
63	predefined master slave I/O configuration does not correspond to the additional table
64	size indicator of parameter data table corrupt
65	number of inputs declared in the additional table does not correspond to the number in the I/O configuration table
66	number of outputs declared in the additional table does not correspond to the number in the I/O configuration table
67	unknown data type in I/O configuration detected
68	data type of a defined I/O module in a connection does not correspond with the defined data size
69	configured output address of one module oversteps the possible address range of 3584 bytes
70	configured input address of one module oversteps the possible address range of 3584 bytes
71	one predefined connection type is unknown
72	multiple connections defined in parallel
73	the configured Exp_Packet_Rate value of one connection is less than the Prod_Inhibit_Time value
74	parameter field DNM_PRED_MSTSL_CFG_DATA inconsistent
76	value for usRecFragTimer in data set out of range

5.2.6 DNM_Upload

	command message			
	variable	type	value	signification
Message header	msg.rx	byte	3	receiver = USR_INTF-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	5	length of the message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	no error
	msg.b	byte	80	command = DDLM_Upload
	msg.e	byte	0	extention, not used
DNM_UPLOAD_REQUEST	msg.d[0]	byte	0	Req_Adr, unused
	msg.d[1]	byte	0-63 127	Area_Code, device address DEVICE bus parameters
	msg.d[2..3]	word	0-760	Add_Offset
	msg.d[4]	byte	1-240	Data_Len

This command allows to read back the master bus parameters or the device parameter data files.

The difference to the `DNM_Download` command is, that the functions `DNM_Start_Seq` and `DNM_End_Seq` can not be used, but the data consistency is guaranteed anyway, but not supervised via the programmable timer algorithm.

The parameter `Area_Code` fixes the source area and can be either the master bus parameter file = 127 or a device parameter file = 0-63. The start offset within the corresponding upload area where the data will be read from must be fixed in the variable `Add_Offset`. The number of data bytes that shall be read in maximum with the service have to be inserted in the `Data_Len` variable.
If the variable `Add_Offset` plus `Data_Len` overstep the maximum actual loaded file length then the DEVICE will return no error, but return all the rest data beginning at `Add_Offset` position.

See below the corresponding structure in the header file:

```
DNM_UPLOAD_REQUEST
```

answer message			
variable	type	value	signification
msg.rx	byte	16	receiver = user at HOST
msg.tx	byte	3	transmitter = DNM-Task
msg.ln	byte	x	length of message
msg.nr	byte	j	number of the message
msg.a	byte	80	answer = DNM_Upload
msg.f	byte	f	error, state
msg.b	byte	0	no command
msg.e	byte	0	extention, not used
msg[0...x-1]	byte string		data of whished data set file

The `msg.d[x]` area will contain the wished data field that was addressed within the command message. I

Possible values for `msg.f` are the following:

error number msg.f	signification
0	no error
152	unknow command, DEVICE needs newer firmware
20	CON_LR, local resource not available, requested bus parameter are not present and available because card is not configured
21	CON_IV, parameter fault in request
52	CON_NI, area_code unknown
53	CON_EA, overstep of the buffer length
55	CON_IP, faulty parameter detected
57	CON_SI, sequence error
59	CON_DI, data incomplete or faulty

5.2.7 DNM_RestoreFromFlash

	command message			
	variable	type	value	signification
Message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	1	length of the message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	no error
	msg.b	byte	82	command = DNM_RestoreFromFlash
	msg.e	byte	0	extention, not used
DNM_RESTORE_FROM_FLASH	msg.d[0]	byte		RestoreMethod
			0	reload FLASH contents to RAM
			1	reload FLASH contents to RAM and start network with found parameters

This command serves to read back all saved download parameter data sets from FLASH back to RAM at once, which where previously stored during execution of `DNM_Download` command.
There are 2 methods which can be selected in the `RestoreMethod` byte of the message. The first method is just reload the contents of the FLASH back to RAM of the DEVICE. In this case no network communication is started, but all data files can be uploaded for example with the `DNM_Upload` command.
The second method is to restore the data sets like in method 1, but with the network start afterwards. The DEVICE will behave in this case like a HOST programm would have loaded all data sets and the bus parameter via the `DNM_Download` message commands.

The answer message informs the HOST program that the command was executed

answer message			
variable	type	value	signification
msg.rx	byte	16	receiver = user at HOST
msg.tx	byte	3	transmitter = DNM-Task
msg.ln	byte	0	length of message
msg.nr	byte	j	number of the message
msg.a	byte	82	answer = DNM_RestoreFromFlash
msg.f	byte	0	No error
		1	No Flash found at all
		2	No Configuration found in FLASH
		3	Bus Parameter Data inconsistent
		4	One of the Slave Parameter Dataset is inconsistent
msg.b	byte	0	no command
msg.e	byte	0	extention, not used

5.2.7.1 The Download of the DEVICE Parameter

5.2.7.1.1 Coding of the DEVICE Parameter Data Set

Header Structure	variable name	type	explanation
BUS_DNM	bOwnMacId	byte	MAC-ID of the DEVICE, range 0-63
	usVendorId	word	Hilscher Device Vendor ID = 283
	bBaudrate	byte	fixes the baud rate
	bReserved	byte	reserved value, don't care
	usReserved	word	reserved value, don't care
	bAutoClear	byte	auto clear mode on / off
	bSrvConsConnSize	byte	I/O consume connection size for server function
	usConsOffset	word	Offset in input process data area for consume I/O data
	bSrvProdConnSize	byte	I/O produce connection size for server function
	usProdOffset	word	Offset in output process data area for produce I/O data

The `bOwnMacId` fixes the DeviceNet specific MAC address under which the DEVICE will present itself later in the network after its initialization.

The `usVendorId` is a 'DeviceNet Vendor Association' specific unique number for each manufacturer of DeviceNet devices. The number will be used by the DEVICE later to report it on each incoming and outgoing 'duplicate MAC-ID check' message. We have got our own Vendor_ID Hilscher = 283dec. from the Vendor Association, but you are not forced to use this ID for your applications too.

The parameter `bBaudrate` is a value between 1 and 3. See the following table for the variety of baudrate definitions:

```
#define DNM_BAUD_125   3    : 125kBaud
#define DNM_BAUD_250   2    : 250kBaud
#define DNM_BAUD_500   1    : 500kBaud
```

The parameter `bAutoClear` defines the system behavior if one as active classified device has been disconnected. If `bAutoClear` mode is active then the DEVICE stops the communication to all other devices too, else it tries to restart the missed device and keeps running.

```
#define DNM_ACLR_INACTIVE 0
#define DNM_ACLR_ACTIVE   1
```

The DEVICE is able to be a server at the same time beeing a master. This allows other masters in the network to exchange I/O data with the DEVICE too. To enable the server functionality the parameter `bSrvConsConnSize` or `bSrvProdConnSize` must be set unequal zero. Both values represent the size of I and O data that gets exchanges through this connection. For each direction a corresponding input `(usConsOffset)` and output `(usProdOffset)` offset

must be configured where the DEVICE reads the produce data from and where it writes the consumed data into.

See the below the corresponding structure in the header file: `BUS_DNM`

5.2.7.1.2 Download Message of the DEVICE Bus Parameters

	command message			
Message header	variable	type	value	signification
	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	18	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	68	command = DNM_Download
	msg.e	byte	k	extension
Service header	msg.d[0]	byte	0	Req_Adr, not used
	msg.d[1]	byte	127	Area_Code = DEVICE bus parameter
	msg.d[2-3]	word	0	Add_Offset
BUS_DNM	msg.d[4]	byte	0-63	Own_MAC_ID
	msg.d[5-6]	word	0 - 65535	Vendor_ID Hilscher Device = 283dec.
	msg.d[7]	byte	0-3	Baud_rate 125k= 3, 250k=2, 500k=1, 1M = 0
	msg.d[8]	byte	0	reserved
	msg.d[9-10]	word	0	reserved
	msg.d[11]	byte	0, 1	Auto_Clear off = 0, on =1
	msg.d[12]	byte	0-255	consumed size for server I/O connection
	msg.d[13-14]	word	0-3583	byte offset in input process data area for server I/O connection
	msg.d[15]	byte	0-255	produced size for server I/O connection
	msg.d[16-17]	word	0-3583	byte offset in output process data area for server I/O connection

5.2.7.2 Download of the Device Parameter Data Sets

5.2.7.2.1 Coding of the Device Parameter Data Set

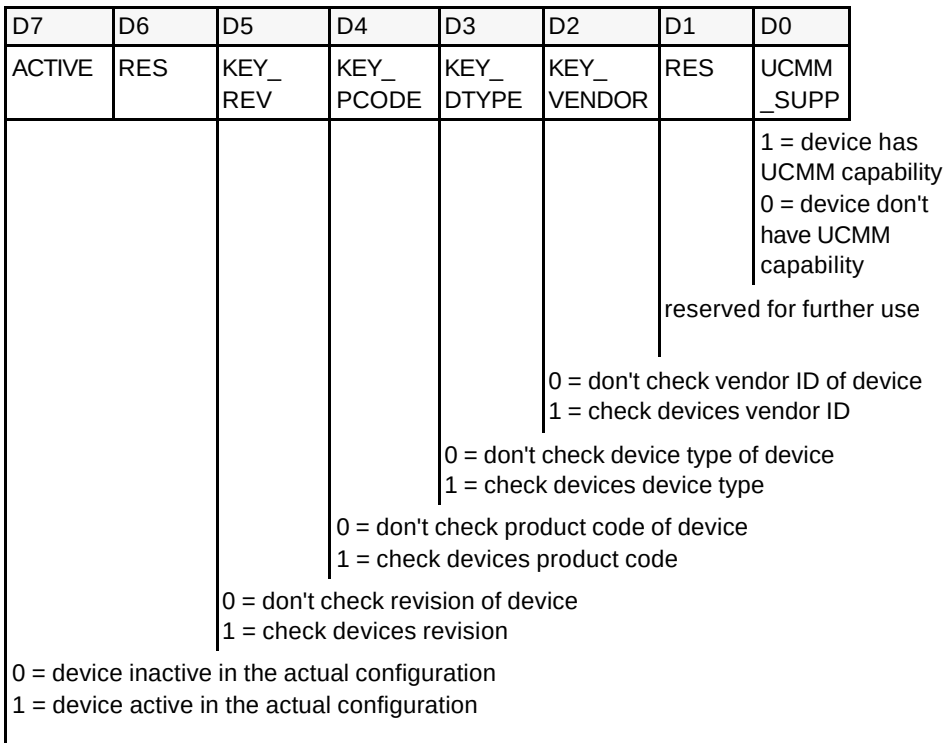
Header Structure	variable name	type	explanation
DNM_DEV_PRM_HEADER	usDevParaLen	word	length of whole data set inclusive the length (2 bytes) of the size indicator
	bDvFlag	byte	bit field, to activate the parameter data set and to indicate which kind of connections are supported, enable disable device keying
	bUcmmGroup	byte	if device supported Dynamic Connections(UCMM) the connection group must specified here
	usRecFragTimer	word	timer value for in multiples of msecond: - slave device times out and DEVICE 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
DNM_PRED_MSTSL_CFG_DATA	usPredMstSlCfg DataLen	word	Length of the following predefined master slave connection field inclusive the length (2 bytes) of the size indicator
	DNM_PRED_MSTSL_ CONNOBJ	structure	predefined connections which shall be established to this device see corresponding HEADER file for structure
DNM_PRED_MSTSL_ADD_TAB	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
DNM_PRED_MSTSL_CFG_DATA	usAttrDataLen	word	Length of the following set attribute data field inclusive the length (2 bytes) of the size indicator
	DNM_SET_ATTR_ DATA	structure	Attributes which shall be changed via explicit messaging before doing I/O messaging see corresponding HEADER file for structure

DNM_UCMM_CONN_OBJ_CFG_DAT	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.
DNM_UCMM_CONN_OBJ_ADD_TA	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
	ausIOOffsets	word array	offset address table for I/O data in the dual-port memory I/O area. Not supported

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

```
usDevParaLen =    2( size indicator itself )
                  + 14 ( header bytes ) +
                  usPredMstSlCfgDataLen +
                  usAddTabLen +
                  usAttDataLen +
                  2 ( usCfgDataLen fixed )+
                  4 ( usAddTabLen fixed )
```

This variable is followed by a special bit field called `bDvFlag`, declaring the parameter data set as active or inactive. Only if the active bit is set the DEVICE will activate the network access for this device.



The bit `UCMM_SUPP` indicates if the device supports the dynamic establishment of explicit messaging connections. Depending on this bit value the DEVICE 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 value `usVendorID`, `usDeviceType`, `usProductCode`, `bMajorRevision` and `bMinorRevision` from the device data set with the physical present values in the devices Identity Object during its initalization. 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 DEVICE is in mode OPERATE and the bit is cleared then the device is not handled and remains in unconnected state for the specific device.

The 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 DNM_GROUP_1 0
#define DNM_GROUP_2 1
#define DNM_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 wished 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 transfered 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 DEVICE to retry the establishment of the devices connection after the connection was lost. The value range goes from 1 to 65535.

The DNM_PRED_MSTSL_CFG_DATA structure informs the DEVICE 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 transfered. The explicit message channel is established to each device anyway and must not be configured. So if it is wished the only the explicit message channel shall be established by the DEVICE, then the DNM_PRED_MSTSL_CFG_DATA structure must be left empty. That means the parameter usPredMstSlCfgDataLen must be set to the value 2.

Header Structure	variable name	type	explanation
DNM_PRED_MSTSL_CONNOBJ	bConnectionType	byte	type of connection that shall be established
	bWatchdogTimeoutAction	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
	DNM_IO_MODULE	structure	process data modul definition
	... for multiple connections	...	...
DNM_PRED_MSTSL_CONNOBJ	bConnectionType	byte	type of connection that shall be established
	bWatchdogTimeoutAction	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
	DNM_IO_MODULE	structure	process data modul definition

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. So one connection to a device is usual.

see structures
DNM_PRED_MSTSL_CONNOBJ ,
DNM_PRED_MSTSL_IO_OBJ_HEADER,
DNM_I_O_MODULE

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 DNM_TYPE_CYCLIC                0x08
#define DNM_TYPE_CHG_OF_STATE          0x04
#define DNM_TYPE_BIT_STROBED           0x02
#define DNM_TYPE_POLLED                0x01
#define DNM_TYPE_CHG_OF_STATE_ACK      0x10
#define DNM_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 DNM_TRANSMISSION_TO_TIMEOUT 0x00
#define DNM_AUTO_DELETE             0x01
#define DNM_AUTO_RESET              0x02
```

The 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 DEVICE suppresses new data production until the timer has expired. This method prevents that the device is overloaded with to fast incoming requests.
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 1000dec 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 and doing 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 occured 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 socalled Input / Output modules are defined in the directly following structure DNM_IO_MODULE.

The table DNM_IO_MODULE defines which type of process data and in sum how many data shall be transferred through this connection. The structure DNM_IO_MODULE 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 TASK_TDT_BOOLEAN  1 /* bit module        */
#define TASK_TDT_UINT8    5 /* byte module       */
#define TASK_TDT_UINT16   6 /* word module       */
#define TASK_TDT_UINT32   7 /* long module       */
#define TASK_TDT_STRING  10 /* byte array module */
```

To distinguish if 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 DEVICE, 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 DNM_OUTPUT 0x80
```

The bDataType is followed by the bDataSize indicator of the module. Except for the bDataType TASK_TDT_STRING all other bDataTypes have fixed bDataSize values. The DEVICE checks if the type and its size are corresponding.

```
TASK_TDT_BOOLEAN -> bDataSize = 1
TASK_TDT_UINT8   -> bDataSize = 1
TASK_TDT_UINT16  -> bDataSize = 2
TASK_TDT_UINT32  -> bDataSize = 4
```

In case of TASK_TDT_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 DEVICE don't care the order of the modules definition especially not if input and output modules are mixed defined. One entry in the DNM_IO_MODULE table must result a corresponding entry in the DNM_PRED_MSTSL_ADD_TAB structure which contains the dual-port memory offset address where to place 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 separate for input and output and in the sum it results 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 got real present value, than the access to this device is denied wholly.

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,

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

if the device data shall be handled transparent as byte strings in the process data image areas of the DEVICE in both directions. The table DNM_PRED_MSTSL_ADD_TAB 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 a plugable I/O rack for example, which can contain modular I/O submodules like analog or digital ones. in mixed constellations. So in this case the analog values can be placed to a different location in the process image then 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 = TASK_TDT_UINT16, input
bDataSize	02 hex = 2 byte
bDataType	05 hex = TASK_TDT_UINT8, input
bDataSize	01 hex = 1 byte
bDataType	86 hex = TASK_TDT_UINT16, output
bDataSize	02 hex = 2 byte
bDataType	05 hex = TASK_TDT_UINT8, input
bDataSize	01 hex = 1 byte
bDataType	85 hex = TASK_TDT_UINT8, output
bDataSize	01 hex = 1 byte
bDataType	07 hex = TASK_TDT_UINT32, input
bDataSize	04 hex = 4 byte

In this example the table `DNM_PRED_MSTSL_ADD_TAB` must contain 6 process data offset addresses one for each module.

see structure `DNM_IO_MODULE` in the header file.

One entry in the `DNM_IO_MODULE` table must result a corresponding entry in the `DNM_PRED_MSTSL_ADD_TAB`. In this table the offset address in the dual-port memory of each process data is held down, where the DEVICE has to start later 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 was received. See the following structure:

variable name	type	explanation
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

see structure `DNM_PRE_MSTSL_ADD_TAB` in the header file.

The `ausIOOffsets` have to be placed in order to each configured I/O module in the table `DNM_IO_MODULE` so that the DEVICE has a relationship between both tables and can associate them together later when doing the I/O exchange. For an output process data module it results a corresponding output offset, for an input process data module it results a corresponding input offset.
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 `TASK_TDT_BOOLEAN` process

data. There an offset must be set up like the following figure illustates:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Bit offset			Byte offset												
			byte offset in the dualport memory 0 - 3583 dec												
corresponding bit offset in the byte 0 -7dec															

To complete the last example above, here is a related DNM_PRE_MSTSL_ADD _TAB 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

The DNM_PRED_MSTSL_CFG_DATA structure can be understood as the parameter data containing structure of each device. The DEVICE takes all configured attributes with their configured values from this table and compares the contents while the device's startup with the physically present attribute values of the device. If the compared pair of values are different the DEVICE sends the configured new value to the device with the DeviceNet 'Set_Attribute' write command. DNM devices store such parameter data normally into an electronic eraseable 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 DEVICE does not stop the parameterization and finalizes it by taking the next configured entry. A denied write request will only be remarked in the internal device specific diagnostic structure, which can be read out with the DNM_Device_Diag message.

	variable name	type	explanation
DNM_SET_ATTR_DATA	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[...]	octect string	data field
	... if more attributes shall be changed	...	...
DNM_SET_ATTR_DATA	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[...]	octect string	data field

see structure `DNM_SET_ATTR_DATA` 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 uneuqal is fixed in the variable `bDataCnt`. If no attributes shall be changed while the device's network startup procedure, 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 `DNM_EXPL_SET_ATTR_DATA` entry should have the following data entries:

variable name	contents
usAttrDataLen	0009 hex
usClassID	0003 hex
usInstanceID	0001 hex
bAttributeID	01 hex
bData Cnt	01 hex
abData[0]	10 hex

The next table `DNM_UCMM_CONN_OBJ_CFG_DATA` is used for further extention of the firmware of the DEVICE. 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 `DNM_UCMM_CONN_OBJ_ADD_TAB` and its length must be reserved also in the device data set. `usAddTabLen` must be set up to a length of 4 fixed.

5.2.8 Example of Message Device Parameter Data Sets Hexdecimal

No Input and output process data, explicit only connection:

03 10 22 08 00 00 44 00, message header
00 0a 00 00, Device Adr = 10 for example
1E 00 80 00 e8 03 00 00 00 00 00 00 00 00 00 00, DNM_DEV_PRM_HEADER
02 00 , DNM_PRED_MSTSL_CFG_DATA = empty, only length itself = 2
04 00 00 00 , no input and output offset in DNM_PRED_MSTSL_ADD_TAB
02 00 , empty DNM_SET_ATTIBUTE_DATA table, only length itself = 2
02 00 , empty DNM_UCMM_CONN_OBJ_CFG_DATA table, only length itself
04 00 00 00, no offsets in DNM_UCMM_CONN_OBJ_ADD_TAB

Polling connection, 7 byte input , 8 bytes output:

03 10 31 08 00 00 44 00, message header
00 02 00 00, Device Adr = 2 for example
2D 00 80 00 40 06 00 00 00 00 00 00 00 00 00 00, DNM_DEV_PRM_HEADER
0D 00 01 00 0A 00 00 00 02 0A 07 8A 08, DNM_PRED_MSTSL_CFG_DATA
08 00 01 01 00 00 00 00, offset = 0 in DNM_PRED_MSTSL_ADD_TAB
02 00 , empty DNM_SET_ATTIBUTE_DATA table, only length itself = 2
02 00 , empty DNM_UCMM_CONN_OBJ_CFG_DATA table, only length itself
04 00 00 00, no offsets in DNM_UCMM_CONN_OBJ_ADD_TAB

Polling connection UCMM group 3 capable device, 7 byte input 8 bytes output

03 10 31 08 00 00 44 00, message header
00 02 00 00, Device Adr = 2 for example
2D 00 81 03 40 06 00 00 00 00 00 00 00 00 00 00, DNM_DEV_PRM_HEADER
0D 00 01 00 14 00 1E 00 02 0A 07 8A 08, DNM_PRED_MSTSL_CFG_DATA
08 00 01 01 14 00 19 00, I-offset=20, O-offset=25 in DNM_PRED_MSTSL_AD-
D_TAB
02 00 , empty DNM_SET_ATTIBUTE_DATA table, only length itself = 2
02 00 , empty DNM_UCMM_CONN_OBJ_CFG_DATA table, only length itself
04 00 00 00, no offsets in DNM_UCMM_CONN_OBJ_ADD_TAB

Bit-Strobe connection, 8 bytes input

03 10 2D 08 00 00 44 00, message header
00 02 00 00, Device Adr = 2 for example
29 00 80 00 40 06 00 00 00 00 00 00 00 00 00 00, DNM_DEV_PRM_HEADER
0B 00 02 00 14 00 1E 00 01 0A 08, DNM_PRED_MSTSL_CFG_DATA
06 00 01 00 00, input offset = 0 in DNM_PRED_MSTSL_ADD_TAB
02 00 , empty DNM_SET_ATTIBUTE_DATA table, only length itself = 2
02 00 , empty DNM_UCMM_CONN_OBJ_CFG_DATA table, only length itself
04 00 00 00, no offsets in DNM_UCMM_CONN_OBJ_ADD_TAB

5.2.9 DNM_Device_Diag

	command message			
message header	variable	type	value	signification
	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	66	command = DNM_Device_Diag
	msg.e	byte	0	extension
extended message header	msg.device_adr	byte	k	Device_Adr
	msg.DataArea	byte	0	data area, unused
	msg.DataAdr	word	0	data address, unused
	msg.DataIdx	byte	0	data index, unused
	msg.DataCnt	byte	0	data count, unused
	msg.DataType	byte	0	data type, unused
	msg.Function	byte	0	function, unused

The command serves to read the diagnostic structure of a device. The wished address of the device must be fixed in `msg.device_adr` corresponding to the real address of the device in the network. The execution of the command results no network access, because the DEVICE saved the last diagnostic of any device in an internal buffer. The status structure can be ordered anytime from the DEVICE.

The corresponding device diagnostic bit in the 'global bus status field' of the dual-port memory indicates, if a newly changed diagnostic information within the internal status structure is available for the specific device and have to be read out. So if the bit is set, the HOST program have to request this diagnostic information via `DNM_Device_Diag` command and wait for the response message. Before sending back the response diagnostic message the bit in the dual-port memory will be released automatically by the DEVICE. If the bit is still present after receiving the response message, the diagnostic buffer contents of the specific device has changed again during the access.

	answer message			
message header	variable	type	value	signification
	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	15	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	66	answer = DNM_Device_Diag
	msg.f	byte	0	error, state
	msg.b	byte	0	no command
	msg.e	byte	0	extension
extended message header	msg.device_adr	byte	k	Rem_Adr
	msg.DataArea	byte	0	data area, unused
	msg.DataAdr	word	0	data address unused
	msg.DataIdx	byte	0	data index unused
	msg.DataCnt	byte	7	data count = length of diagnosis structure
	msg.DataType	byte	0	data type, unused
	msg.Function	byte	1	function read TASK_TFC_READ
diagnostic data	msg.d[0]	byte		bDevStatus1
	msg.d[1]	byte		bDevMainState
	msg.d[2]	byte		bOnlineError
	msg.d[3]	byte		bGeneralErrorCode
	msg.d[4]	byte		bAdditionalCode
	msg.d[5]	word		usTimeout

The reading of the diagnostic information of a device causes the reset of the corresponding diagnostic bit in the 'global bus status field' of the dual-port memory. Should the device address in `msg.device_adr` of the requested message be out of range, the answer message delivers the error code 161. Otherwise no error is reported and the message contains valid diagnostic data.

See below the corresponding structure in the header file:

```
DNM_SINGLE_DEVICE_DIAGNOSTIC
```

Devicestatus_1:

D7	D6	D5	D4	D3	D2	D1	D0
Deact	Res	Res	UCMM_ Support	Cfg_ Fault	Prm_ Fault	Res	No_Re sponse

D0: device not responding

D2: device had denied the access to at least one configured attribute to write in

D3: differences between device produced and consumed connection size to the resulting configured ones

D4: reserved

D5: reserved

D6: reserved

D7: device is deactivated in actual configuration and not handled

`bDevMainState:`

For each device the DEVICE has a socalled state machine handler active, which handles device specific its startup procedure, the process data transmission and the error handling depending on its behavior and its configuration. Each state a device can have active in its handler is represented by a number in `bDevMainState` with the following meaning:

Value	Definition	Explanation
0	DNM_DV_ENTER	State machine enter
1	DNM_DV_INACTIVE	Device inactive, not handled
2	DNM_MANAGE_DUP_MAC	Own MAC ID, state waiting for all incoming duplicate MAC-ID requests
3	DNM_DV_INIT_PRED_MSTS	initialize internal predefined master slaves strucutres
4	DNM_DV_ALLOC_PRED_MSTSL_CONN	allocated predefined master slave connection set request
5	DNM_DV_WAIT_FOR_ALLOC	wait for predefined master slave allocation connection response
6	DNM_DV_RELEASE_MSTSL_CONN	release predefined master slave connection set request
7	DNM_DV_WAIT_FOR_RELEASE	wait for predefined master slave release connection response
8	DNM_DV_INIT_IO_CONF	initialize internal I/O configurated structures
9	DNM_DV_ALLOC_IO_CONN	allocate configured I/O connection request
10	DNM_DV_WAIT_FOR_IO_ALLOC	wait for I/O allocation response
11	DNM_DV_RELEASE_IO_CONN	release I/O connection request
12	DNM_DV_WAIT_FOR_IO_RELEASE	wait for I/O connection release response
13	DNM_DV_READ_CONSUMED_SIZE	read consumed connection size
14	DNM_DV_WAIT_CONS_SIZE	wait for read consumed connection size response
15	DNM_DV_CHECK_CONS_SIZE	compare cosumed connection size with internal configured one
16	DNM_DV_READ_PRODUCED_SIZE	read produced connection size
17	DNM_DV_WAIT_READ_PROD	wait for read produced connection size response
18	DNM_DV_CHECK_PROD_SIZE	compare produced connection size with internal configured one
19	DNM_DV_CONFIGURE_CONNECTION	configure the I/O connection structures and register it
20	DNM_DV_SET_EXPCT_RATE	set expected packet rate
21	DNM_DV_WAIT_EXPCT_RATE	wait for set expected packet rate response
22	DNM_DV_END_IO_POLL	I/O poll request 1'st time
23	DNM_DV_WAIT_IO_POLL	wait for I/O poll response
24	DNM_DV_END_IO_POLL_2	I/O poll request 2'nd time
25	DNM_DV_WAIT_IO_POLL_2	wait for I/O poll response

26	DNM_DV_END_IO_POLL_3	I/O poll request 3'rd time
27	DNM_DV_WAIT_IO_POLL_3	wait for I/O poll response
28	DNM_DV_HRTB_TIME_OUT	heart beat timeout to the device
30	DNM_DV_OPEN_EXPLICIT_CONN	open unconnected explicit connection request 1'st time
31	DNM_DV_OPEN_EXPLICIT_RES	wait for unconnected explicit connection response
32	DNM_DV_OPEN_EXPLICIT_CONN_2	open unconnected explicit connection request 2'nd time
33	DNM_DV_OPEN_EXPLICIT_RES_2	wait for unconnected explicit connection response
34	DNM_DV_CHECK_CLOSE_CON	close unconnected connection request
35	DNM_DV_WAIT_CLOSE_RES	wait for close unconnected connection response
36	DNM_DV_RELEASE_CONNECTION	release all established connections request
37	DNM_DV_WAIT_RELEASE_ALL	wait for connection relase response
38	DNM_DV_USR_EXPL_CONN	open user unconnected explicit connection request
39	DNM_DV_USR_EXPLICIT_RES	wait for user explicit connection response
40	DNM_DV_USR_PRED_MSL_CONN	user predefined master slave allocate connection request
41	DNM_DV_WAIT_FOR_USER_ALLOC	wait for user allocation response
42	DNM_DV_CHECK_USR_CLOSE_CON	user close unconnected connection request
43	DNM_DV_WAIT_USR_CLOSE_RES	wit for user close unconnected response
44	DNM_DV_GET_SET_ATTRIBUTE	get or set user defined attribute request
45	DNM_DV_WAIT_GET_SET_ATTR	wait for user defined get or set attribute response
46	DNM_DV_CHECK_GET_SET_RESPONSE	send or wait fragmented get or set attribute

`bOnlineError:`

In this byte the actual online error of the device station is held down. See the table `Err_Event` of the global bus status field for possible entries.

`bGeneralErrorCode`:

The `bGeneralErrorCode` will only be handled in one `bOnlineError` case to give detailed information about the error source and location. See the following table showing the relation between the `bOnlineError` and the `bGeneralErrorCode`. The values are defined in the DeviceNet specification and inserted in the `bGeneralErrorCode` variable transparent on each incoming error response message

Online_Error	Explanation	General_Error_Codes
35	device rejects requested command with an error response	error code containment of the response 2 = resources unavailable 8 = service not supported 9 = invalid attribute value 11 = already in request mode 12 = object state conflict 14 = attribute not settable 15 = privileg violation 16 = device state conflict 17 = reply data to large 19 = not enough data 20 = atttibute not supported 21 = too much data 22 = object does not exist

`bAdditionalCode`:

This additional error information is only valid, if the `bGeneralErrorCode` contains a value unequal 0. The value `bAdditionalCode` is filled transparent like the `bGeneralErrorCode` above, with the additional error code of each incoming error response message of the device.

`usTimeout:`

If a device is supervised by the expected packet rate of a connection and times out then, the timer `usTimeout` 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 DEVICE trys always to reestablish the connection immediatly.

5.2.10 DNM_Get_Set_Attribute as Client

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8 8+240max	length of message read access write access
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	17	command = DNM_Get_Set_Attribute
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addresses
	msg.data_area	byte	0-255	data area, Class ID, Low byte
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_idx	byte	0-255	data index, Attribute ID
	msg.data_cnt	byte	0 1-240	data count, unused in read access length of attribute data in write access
	msg.data_type	byte	0-255	data type, Class ID, High byte
	msg.function	byte	1 2	function code TASK_TFC_READ TASK_TFC_WRITE
data to write	msg.d[0-239max]	byte array	xxx	attribute data in write access

This message serves to read out or to write in HOST actively a definite attribute of a connected device as client. The address of the device must be fixed in `msg.device_adr`. Its value ranges from 0 to 63. Which of the possible 65535 Class IDs of that device shall be addressed must be fixed in `msg.data_area` and `msg.data_type`. The Low byte must be written into `msg.data_area` and the High byte into `msg.data_type`. The Instance ID must be inserted in `msg.data_adr`. The Attribute ID which shall be accessed must be defined in `msg.data_idx`. To distinguish the access type the `msg.Function` have to be written with the `TASK_TFC_READ` value for read access or with `TASK_TFC_WRITE` for write access. In case of the write command the data have to be inserted in `msg.d[0]` area. Don't forget to assimilate the length of the message in `msg.Ln` and `msg.data_cnt` then.

If the wished device is handled internally, its actual state is left for that time the command takes to handle in the network. During this time other incoming messages, like for example 'Change of state' message from the device, are thrown away and be lost. Afterwards the internal device handler retakes the old state of the device before the command was executed.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	8 8+240max	length of message write access read access
	msg.nr	byte	j	number of the message
	msg.a	byte	17	answer = DNM_Get_Set_Attribute
	msg.f	byte	see table	error, state
	msg.b	byte	0	no command
	msg.e	byte	0	extension
extended message header	msg.device_adr	byte	0-63	device address, MAC ID
	msg.data_area	byte	0-255	data area, Class ID, Low byte
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_Idx	byte	0-255	data index, Attribute ID
	msg.data_cnt	byte	 1-240 0-240	data count number of bytes be written length of attribute data in read access
	msg.data_type	byte	0-255	data type, Class ID, High byte
	msg.function	byte	 1 2	function code TASK_TFC_READ TASK_TFC_WRITE
data read out	msg.d[0-239max]	byte array	xxx	attribute data in read access

The answer message serves the HOST program to get information if the wished command has been be executed without an error. See the table below for the backcoming possible error values. In case of read access the message includes the read data in the area `msg.d[xxx]`.

error number msg.f	signification
0	no error
2	device: resources unavailable
8	device: service not available
9	device: invalid attribute value
11	device: already in request mode
12	device: object state conflict
14	device: attribute not settable
15	device: a permission check failed
16	device: state conflict, device state prohibits the command execution
17	no response from the network device, timeout
19	device: not enough data received
20	device: attribute not supported
21	device: to much date received
22	device: object does not exists
23	device: reply data too large, internal buffer to small
48	DEVICE: device not configured, Explicit message channel not established
50	DEVICE: format error in response telegram
54	DEVICE: another get or set command still active
55	DEVICE: MAC-ID in msg.device_adr out of range
57	DEVICE: sequence error in fragmented response sequence
200	DEVICE: not configured, no data base found

5.2.11 DNM_Get_Set_Attribute as Server

	command message indication			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	8 8+240max	length of message read access write access
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	17	command = DNS_Get_Set_Attribute
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0	device address, unused
	msg.data_area	byte	100-199	data area, Class ID
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_idx	byte	0-255	data index, Attribute ID
	msg.data_Cnt	byte	0 1-240	data count, unused in read access length of attribute data in write access
	msg.data_type	byte	0	data type, unused
	msg.function	byte	1 2	function code TASK_TFC_READ TASK_TFC_WRITE
data to write to HOST	msg.d[0-239max]	byte array	xxx	attribute data in case of write access

If the Explicit Message channel between master DEVICE and any other client capable device is in established state, a defined range of accessible Class-IDs are routed by the DEVICE trough the mailboxes, if the access to them happens. The DEVICE can be an Explicit Message Server in this case, that means requested Get- and Set Attribute commands of the other client device can be answered by the HOST application. This means for the HOST program to look cyclically for incoming `DNS_Get_Set_Attribute` Indication messages like described above and answer them by a corresponding answer message.
Which of the possible 100 Class IDs (Device Net Specification area for User Defined Class ID = 100 up to 199) is requested by the client is indicated in `msg.data_area`, while the Instance ID is inserted in `msg.data_adr`. The Attribute ID which is accessed is inserted in `msg.data_idx`. To distinguish the access type the `msg.function` is indicating `TASK_TFC_READ` value for read = Get Attribute access or `TASK_TFC_WRITE` for write access = Set Attribute command. In case of a Set command the data the master wants to write is inserted in `msg.d[0]` area. The number of bytes that shall be written are fixed in `msg.data_cnt`. Per message in maximum 240 bytes can be transferred.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8 8+240max 9	length of message write access read access in case of msg.f = 31
	msg.nr	byte	j	number of the message
	msg.a	byte	17	answer = DNS_Get_Set_Attribute
	msg.f	byte	see table	error, state
	msg.b	byte	0	no command
	msg.e	byte	0	extension
extended message header	msg.device_adr	byte	0	device address, unused
	msg.data_area	byte	100-199	data area, Class ID
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_idx	byte	0-255	data index, Attribute ID
	msg.data_Cnt	byte	1-240 0-240	data count number of bytes be written length of attribute data in read access
	msg.data_type	byte	0	data type, unused
	msg.function	byte	1 2	function code TASK_TFC_READ TASK_TFC_WRITE
data read out from HOST	msg.d[0-239max]	byte array	xxx	attribute data in read access
alternative addtitional error code, if msg.f = 31	msg.d[0]	byte		additional error code

An answer message must be generated on each received indication message. This message serves the DEVICE to send back the corresponding and outstanding response message to the client. Normally such Get- and Set Attribute commands are supervised by the client via a timer. That means a timeout error will be generated on the client side if a defined time has expired and no answer was received during this time. Because it's now on the HOST application to send back the corresponding answer message so that the DEVICE can route it back to the client, the HOST application should not waste time and send back the message immediatly after having finished up the analysis of the indication message.

In case of read access the message must include the read data in the area `msg.d[...]`. Furthermore the HOST application has the possibility to send back access errors back to master, for example if a requested attribute does not exist. This must be done in `msg.f` then. See the following table for possible error numbers. DeviceNet allows to send back a general error code and an additional error code in the error response message. All errors in the next table will result an additional error code of 255 except the error value 31. If this error is returned, you can define an additional error code at `msg.d[0]` position, which is sent back to the client together with the general error code 31.

error number msg.f	signification
0	no error
2	resources unavailable, Class ID invalid
8	service not available, Read or Write not supported to Class
9	invalid attribute value, attribute not supported
14	attribute not settable, attribute has no write permission
15	access denied, general error to deny access
16	state conflict, HOST state prohibits the command execution
19	not enough data received, master has send to less data
20	attribute not supported,
21	to much data received, master has send to much data
31	vendor specific error code. An addition error code can be placed in msg.d[0]

5.2.12 DNM_Reset

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	77	command = DNM_Reset
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addressed
	msg.data_area	byte	0-255	data area, Class ID
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	0	data count, unused
	msg.data_type	byte	0	data type, unused
	msg.function	byte	0	function code, unused

This command invokes the DeviceNet Reset Service of the specific Class and Instance.
The address of the device must be fixed in `msg.device_adr`. Its value ranges from 0 to 63. Which of the possible 255 Class IDs of that device shall be addressed must be fixed in `msg.data_area`, while the Instance ID must be inserted in `msg.data_adr`.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	8	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	77	answer = DNM_Reset
	msg.f	byte	see table	error, state
	msg.b	byte	0	no command
	msg.e	byte	0	extension
extended message header	msg. device_adr	byte	0-63	device address, MAC ID
	msg.data_area	byte	0-255	data area, Class ID
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	0	data count, unused
	msg.data_type	byte	0	data type, unused
	msg.function	byte		function code, unused

The answer message informs the HOST program if the Reset Service could be executed succesfully. See the table below for the possible error codes.

error number msg.f	signification
0	no error
2	device: resources unavailable
8	device: service not available
11	device: already in request mode
12	device: object state conflict
15	device: a permission check failed
16	device: state conflict, device state prohibits the command execution
17	no response from the device, timeout
50	DEVICE: format error in response telegram
54	DEVICE: another reset command still active
55	DEVICE: MAC-ID in msg.device_adr out of range
64	DEVICE:the explicit channel is currently in use and occupied by the master firmware itself
200	DEVICE: not configured, no data base found

5.2.13 DNM_Explicit_Message

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8+x	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	79	command = DNM_Explicit_Message
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addressed
	msg.data_area	byte	0-255	data area, Class ID
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	x	data count, length of service data field
	msg.data_type	byte	0	data type, unused
	msg.function	byte	0-31	Function, service code, see DeviceNet Spec Vol I, Appendix G
service data field	msg.d[0...x-1]	byte array		optional request service data field

With this service it is possible to send any explicit message to a specific node. The services that can be used are described in the DeviceNet Specification Volume I in the Appendix G.
The request parameter of the explicit message are at first the slave that shall be addressed. This value must be specified in the `msg.device_adr` variable. This is followed by the class ID which must be set in `msg.data_area` and the instance ID in `msg.data_adr`. The DEVICE itself handles the service code in the `msg.function` variable transparent and does not check for any valid codes. The service data field which can be set in the message too, is sent to the specified slave fully transparent. The meaning of this field is always explained and defined together with the service in the DeviceNet Spec. If the service data field is used, the length of the message must be adjusted in `msg.ln` and `msg.data_cnt`.
d

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	8+x	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	79	answer = DNM_Explicit_Message
	msg.f	byte	see table	error, state
	msg.b	byte	0	no command
	msg.e	byte	0	extension
extended message header	msg.device_adr	byte	0-63	device address, MAC ID
	msg.DataArea	byte	0-255	data area, Class ID
	msg.DataAdr	word	0-255	data address, Instance ID
	msg.DataIdx	byte	0	data index, unused
	msg.DataCnt	byte	x	data count, length of the response service data field
	msg.DataType	byte	0	data type, unused
	msg.Function	byte	0-31	Function, service code
service data field	msg.d[0...x-1]	byte array		optional response service data field

The backcoming response message informs the HOST if the request was successfully serviced. If optional response service data was received, the DEVICE inserts this data transparent in the `msg.d[...]` area of the message and reflects the length in `msg.data_cnt`.

error number msg.f	signification
0	no error
2	device: resources unavailable
8	device: service not available
11	device: already in request mode
12	device: object state conflict
15	device: a permission check failed
16	device: state conflict, device state prohibits the command execution
17	no response from the device, timeout
50	DEVICE: format error in response telegram
54	DEVICE: another reset command still active
55	DEVICE: MAC-ID in msg.device_adr out of range
200	DEVICE: not configured, no data base found

5.2.14 Messaging to UCMM capable devices

5.2.14.1 DNM_Open_UCMM

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	84	command = DNM_Open_UCMM
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addressed
	msg.data_area	byte	0	data area, unused
	msg.data_adr	word	0	data address, unused
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	0	data count, unused
	msg.data_type	byte	0	data type, unused
	msg.function	byte	0	function, unsused

This service opens an UCMM connection via the unconnected explicit message port of the specified device. The device addess (MAC-ID) the service shall be addressed to must be set in `msg.device_adr`.

For opening the explicit channel the DEVICE as client uses the predefined GROUP3 unconnected port message identifiers on the network. The request itself automatically requests for a GROUP3 allocation. The server then allocates the connected explicit channel from its GROUP3 message ID pool that will be used in conjunction with its own MAC-ID to generate the Connection ID that is specified when it transmits a message across this connection. (server that needs GROUP1 or GROUP2 message connections can not be handled).

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	 10 8 9	length of message, positive response, negative response with no additional error negative response from server with additional error information
	msg.nr	byte	j	number of the message
	msg.a	byte	84	command = DNM_Open_UCMM
	msg.f	byte	see error table	error, status
	msg.b	byte	0	no command
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addressed
	msg.data_area	byte	0	data area, unused
	msg.data_adr	word	0	data address, unused
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	 2 1	data count, no error additional error available
	msg.data_type	byte	0	data type, unused
	msg.function	byte	0	function, unsused
Connection Instance ID received in UCMM connection response	msg.d[0-1]	word		Connection Instance ID
optional additional error code	msg.d[0]	byte		additional error code, coming from the returned error response message

The response message informs the HOST if the request was successfully serviced. In case of an error the value `msg.f` contains the error information. If the request was rejected by the server device, further additional error information is available in `msg.d[0]`. This value is directly taken from the error response message of the server device. Together with the error code in `msg.f` which reflects the general error code, both values together identifies the encountered error.

5.2.14.2 DNM_UCMM_Explicit_Message

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8+x	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	85	command = DNM_UCMM_Explicit_Message
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addressed
	msg.data_area	byte	0-255	data area, Class ID
	msg.data_adr	word	0-255	data address, Instance ID
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	x	data count, length of service data field
	msg.data_type	byte	0	data type, unused
	msg.function	byte	0-255	Function, service code, see DeviceNet Spec Vol I, Appendix G
service data field	msg.d[0...x-1]	byte array		optional request service data field

With this service it is possible to send any explicit message to a specific device via a previous installed explicit channel with the command `DNM_Open_UCMM`. The DEVICE itself reminds the ID and Group parameter which were exchanged before during the establishment of the UCMM connection. The services that can be used are described in the DeviceNet Specification Volume I in the Appendix G.
The request parameter of the explicit message are at first the slave that shall be addressed. This value must be specified in the `msg.device_adr` variable. This is followed by the class ID which must be set in `msg.data_area` and the instance ID in `msg.data_adr`. The DEVICE itself handles the service code in the `msg.function` variable transparent and does not check for any valid codes. The service data field which can be set in the message too, is sent to the specified slave fully transparent. The meaning of this field is always explained and defined together with the service in the DeviceNet Spec. If the service data field is used, the length of the message must be adjusted in `msg.ln` and `msg.data_cnt`.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	8+x 8 9	length of message, positive response with optional data field, negative response, negative response with additional error
	msg.nr	byte	j	number of the message
	msg.a	byte	79	answer = DNM_Explicit_Message
	msg.f	byte	see error table	error, state
	msg.b	byte	0	no command
	msg.e	byte	0	extension
extended message header	msg.device_adr	byte	0-63	device address, MAC ID
	msg.DataArea	byte	0-255	data area, Class ID
	msg.DataAdr	word	0-255	data address, Instance ID
	msg.DataIdx	byte	0	data index, unused
	msg.DataCnt	byte	x 1	data count, length of the response service data field negative response with additonal error
	msg.DataType	byte	0	data type, unused
	msg.Function	byte	0-255	Function, service code
service data field or	msg.d[0...x-1]	byte array		optional response service data field

additional error code in case of an error response	msg.d[0]	byte		additional error

The response message informs the HOST if the request was successfully serviced. If optional response service data was received, the DEVICE inserts this data transparent in the `msg.d[...]` area of the message and reflects the length in `msg.data_cnt`.

5.2.14.3 DNM_Close_UCMM

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	8	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	86	command = DNM_Close_UCMM
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addressed
	msg.data_area	byte	0	data area, unused
	msg.data_adr	word	0	data address, unused
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	0	data count, unused
	msg.data_type	byte	0	data type, unused
	msg.function	byte	0	function, unsused

This service closes a previous installed UCMM connection channel which was installed with the `DNM_Open_UCMM` command. For this the DEVICE uses the unconnected explicit message port of the specified device. The device addess (MAC-ID) the service shall be addressed to must be set in `msg.device_adr`.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	 8 8 9	length of message, positive response, negative response with no additional error negative response from server with additional error information
	msg.nr	byte	j	number of the message
	msg.a	byte	86	command = DNM_Close_UCMM
	msg.f	byte	see table	error, status
	msg.b	byte	0	no command
	msg.e	byte	0	extension, unused
extended message header	msg.device_adr	byte	0-63	device address, MAC-ID of device to be addressed
	msg.data_area	byte	0	data area, unused
	msg.data_adr	word	0	data address, unused
	msg.data_idx	byte	0	data index, unused
	msg.data_Cnt	byte	 0 1	data count, no error additional error available
	msg.data_type	byte	0	data type, unused
	msg.function	byte	0	function, unsused
optional additional error code	msg.d[0]	byte		additional error code, coming from the returned error response message

The response message informs the HOST if the request was successfully serviced. In case of an error the value `msg.f` contains the error information. If the request was rejected by the server device, further additional error information is available in `msg.d[0]`. This value is directly taken from the error response message of the server device. Together with the error code in `msg.f` which reflects the general error code, both values together identifies the encountered error.

5.2.14.4 Error response definitons of UCMM response messsages

error number msg.f	signification
0	no error
2	device: resources unavailable
8	device: service not available
11	device: already in request mode
12	device: object state conflict
15	device: a permission check failed
16	device: state conflict, device state prohibits the command execution
17	no response from the device, timeout
50	DEVICE: format error in response telegram
54	DEVICE: another reset command still active
55	DEVICE: MAC-ID in msg.device_adr out of range
200	DEVICE: not configured, no data base found

5.2.15 DNM_Auto_Config

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	2	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	78	command = DNM_Auto_Config
	msg.e	byte	0	extension, unused
DNM_AUTO_CONFIG_REQ	msg.d[0]	byte	1,2,3	baudrate
	msg.d[1]	byte	0-63	master MAC-ID

This command message serves to force an automatic network scan of the connected devicenet network. The DEVICE will analyse the network in all connected slave devices and establish to each found one the explicit message channel automatically and if these devices also supporting I/O messages it will establishes these connections too. After the network scan is finished all establishes connections remainin the established state. Because of this it is possible afterwards to use the `DNM_Get_Set_Attribute` message to access via the established explicit message channel to all objects within the devices.

The parameter `msg.d[0]` fixes the baudrate the DEVICE shall do the automatic network scan and is a value between 1 and 3. See the following table for the variety of baudrate definitions:

```
#define DNM_BAUD_125   3    : 125kBaud
#define DNM_BAUD_250   2    : 250kBaud
#define DNM_BAUD_500   1    : 500kBaud
```

The second parameter of the message must be fixed in `msg.d[1]` and sets the MAC-ID of the DEVICE. It has a range of 0 up to 63. The DEVICE will afterwards perfom a duplicate MAC-ID check and will return an error message if a duplicate slave MAC-ID with the same MAC-ID was detected.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	0 128	length of message, if msg.f <> 0 if msg.f = 0
	msg.nr	byte	j	number of the message
	msg.a	byte	78	answer = DNM_Auto_Config
	msg.f	byte	see table	error, state
	msg.b	byte	0	no command
	msg.e	byte	0	extension
DNM_AUTO_CONFIG_CONFIRM	msg.d[0]	byte	0-255	supported functions of slave 0
	...	...	...	...
	msg.d[63]	byte	0-255	supported functions of slave 63
	msg.d[64]	byte	1,2,3	UCMM_Group number of slave 0
	...	...	...	...
	msg.d[127]	byte	1,2,3	UCMM_Group number of slave 63

If no error was detected the return message includes now 128 bytes of information collected during the automatic network scan.
Each slave is represented in one byte supported functions from `msg.d[0]` up to `msg.d[63]` and one byte group number from `msg.d[64]` up to `msg.d[127]`, if it supports the DeviceNet UCMM capability.

The `msg.d[0]` to `msg.d[63]` will contain per slave following bit structure:

D7	D6	D5	D4	D3	D2	D1	D0
res	UCMM	Cyclic	Change of state	Res	BitStrobed	Polled	Explicit

- D0: Device supports explicit channel
- D1: the slave supports the Polled I/O connection
- D2: the slave supports the Bitstrobed I/O connection
- D3: reserved
- D4: the slave supports the Change of state I/O connection
- D5: the slave supports the Cyclic I/O connection
- D6: the slave device has UCMM capability
- D7: reserved bit

If the supported function byte of a slave indicates that the slave supports UCMM, then the correspondig `msg.d[SlaveMACID+64]` = `UCMM_Group` number must be examined too to get the information through which DeviceNet message pool the connection was established.

The `UCMM_Group` defines the message group which the slave is assigned to. The following values are defined:

```
#define DNM_GROUP_1 0
#define DNM_GROUP_2 1
#define DNM_GROUP_3 3
```

error number msg.f	signification
57 = TASK_F_DUPLICATE_MAC_ID	the DEVICE has detected another device in the network which has the same configured MAC-ID. Please change either the MAC-ID parameter in the request message or change the address of the other slave station in the network
75 = TASK_F_NO_CAN	the DEVICE was not capable to perfom the Duplicate_MAC-ID check. This can either result from a wrong configured baudrate or a physical connection problem to the network.

5.2.16 Sending and receiving Bit-Strobe via Messages

The standard bit strobe communication of the DEVICE is configured via SyCon configuration software normally or via the `DNM_Download` message procedure online. If the bit strobe connection is configured in this way, then the DEVICE handles on one hand the sending of the bit strobe output command message automatically depending on the configured expected packet and collects on the other hand all incoming bit strobe response messages and places their contents into the process data input image area. But there is never a sychronization between HOST application and DEVICE handler itself and both processes are running fully independant.
For that reason the HOST application can be sychronized now to the DEVICE by a special message command called `DNM_Bit_Strobe`. Enabling this special bit strobe command message follows two steps. In the first configuration message called `DNM_Cfg_Bit_Strobe` that must be sent once by the HOST, the DEVICE gets configured which of the bit strobe slave devices shall be routed and tunneled to the `DNM_Bit_Strobe` response message. Instead of placing the backcoming bit strobe responses into the input process area, the DEVICE collects the bit strobe response data and builds one `DNM_Bit_Strobe` response message with them. Up to 31 slave devices can be handled in this way. Whenever the HOST application sends now the `DNM_Bit_Strobe` command message with the bit strobe output data as data, the DEVICE returns after a configureable time all collected bit strobe input data in one message.

5.2.16.1 Confguring the Bit-Strobe DNM_Cfg_Bit_Strobe

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	X+1	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	87	command = DNM_Cfg_Bit_Strobe
	msg.e	byte	0	extension, unused
configuration set up of bit strobe response message	msg.d[0]	byte	0-63	MAC-ID of first bit strobe device
	...			
	msg.d[X]	byte	0-63	MAC-ID of last bit strobe device

This message configures the bit strobe response message. It must be sent to the DEVICE at least once, to enable the execution of the `DNM_Bit_Strobe` command.

For each slave device that data shall appear later in the DNM_Bit_Strobe response message, one MAC-ID entry must be set in the msg.d[...] area of the configuration message.
The data position within the DNM_Bit_Strobe response message begins at relative position 0 and for each further configured MAC-ID the offset position is incremented by the value 8, which is the maximum possible value for a bit strobe response. This means if for example a MAC-ID has the position 3 within the message, then the data will be placed later at byte postion 3*8 = 24. It is not necessary to keep any order in the MAC-IDs that are configured. The order can be mixed. The value msg.ln must be set to the number of configured bit strobe devices.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	0	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	87	answer = DNM_Cfg_Bit_Strobe
	msg.f	byte	0	no error
	msg.b	byte	0	no command
	msg.e	byte	0	extension

The answer message inform about the success of the set up process. After receiving this answer message the DNM_Bit_Strobe command message can be used now.

5.2.16.2 The DNM_Bit_Strobe command

The DNM_Bit_Strobe command message is used by the HOST application to send the DeviceNet bit strobe output message. This message is 8 byte in length and is sent by the DEVICE completely transparent to the DeviceNet network. Next to this output message a timout value defines how long the DEVICE has to wait for all incoming bit strobe response message until it sends back the DNM_Bit_Strobe response message back to the HOST. Every incoming bit strobe slave response will be placed within the response message at exactly that position where it was previously configured via the DNM_Cfg_Bit_Strobe message.

	command message			
	variable	type	value	signification
message header	msg.rx	byte	3	receiver = DNM-Task
	msg.tx	byte	16	transmitter = user at HOST
	msg.ln	byte	10	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	0	no answer
	msg.f	byte	0	error, status
	msg.b	byte	88	command = DNM_Bit_Strobe
	msg.e	byte	0	extension, unused
Timeout value	msg.d[0-1]	word	0-65535	timeout in multiples of 1msec
DEVICE bit strobe output message	msg.d[2]	byte	0-255	bit strobe output for MAC-ID 0 to 7
	msg.d[3]	byte	0-255	bit strobe output for MAC-ID 8 to 15
	msg.d[4]	byte	0-255	bit strobe output for MAC-ID 16 to 23
	msg.d[5]	byte	0-255	bit strobe output for MAC-ID 24 to 31
	msg.d[6]	byte	0-255	bit strobe output for MAC-ID 32 to 39
	msg.d[7]	byte	0-255	bit strobe output for MAC-ID 40 to 47
	msg.d[8]	byte	0-255	bit strobe output for MAC-ID 48 to 55
	msg.d[9]	byte	0-255	bit strobe output for MAC-ID 56 to 63

The response message is sent back to the HOST application when the configured timeout value timed out. All collected bit strobe response messages will be placed at their configured position in the response message. The message itself contains also a 4 byte bit indication list, which of the bit strobe responses within the message contains valid and received data and which not. If the corresponding bit is 0 no bit strobe response was received and if it is logical 1 the response was received.

	answer message			
	variable	type	value	signification
message header	msg.rx	byte	16	receiver = user at HOST
	msg.tx	byte	3	transmitter = DNM-Task
	msg.ln	byte	4+(X-1)	length of message
	msg.nr	byte	j	number of the message
	msg.a	byte	88	answer = DNM_Bit_Strobe
	msg.f	byte	0	no error
	msg.b	byte	0	no command
	msg.e	byte	0	extension
response status as bit array	msg.d[0]	byte	0-255	response status bits for response 0 to 7
	msg.d[1]	byte	0-255	response status bits for response 8 to15
	msg.d[2]	byte	0-255	response status bits for response 16 to 23
	msg.d[3]	byte	0-255	response status bits for response 24 to 31
bit strobe response of first configured MAC-ID	msg.d[4]	byte	0-255	byte 0 of bit strobe response
	msg.d[5]	byte	0-255	byte 1of bit strobe response
	msg.d[6]	byte	0-255	byte 2of bit strobe response
	msg.d[7]	byte	0-255	byte 3of bit strobe response
	msg.d[8]	byte	0-255	byte 4of bit strobe response
	msg.d[9]	byte	0-255	byte 5of bit strobe response
	msg.d[10]	byte	0-255	byte 6of bit strobe response
	msg.d[11]	byte	0-255	byte 7of bit strobe response
...	...	...	...	...
bit strobe response of last configured MAC-ID	msg.d[4+(X-7)]	byte	0-255	byte 0 of bit strobe response
	msg.d[4+(X-6)]	byte	0-255	byte 1of bit strobe response
	msg.d[4+(X-5)]	byte	0-255	byte 2of bit strobe response
	msg.d[4+(X-4)]	byte	0-255	byte 3of bit strobe response
	msg.d[4+(X-3)]	byte	0-255	byte 4of bit strobe response
	msg.d[4+(X-2)]	byte	0-255	byte 5of bit strobe response
	msg.d[4+(X-1)]	byte	0-255	byte 6of bit strobe response
	msg.d[4+X]	byte	0-255	byte 7of bit strobe response

6 Access of SyCon Configuration DBM-file using dbm32.dll

A DBM file, which can be created with SyCon option Export or with dbm32.dll function DbmExportCfgToDbm(), is a Hilscher defined data base format. Within such a data base different data base tables can exist. They are all distinguished by an ASCII character string of 32 bytes in maximal length. Each table can have at least one data set. If for example the bus parameter shall be accessed to, this table will consist of only one data set. The table where the configuration of the different bus devices is held will consist of one or more records.
The DBM file can be created based on a Windows ACCESS data base only, which is usually the configuration file itself. Of course the coded tables within the resulting DBM file will not contain every information that is held within the mother ACCESS data base.
Each DBM file has a unique 8 byte ASCII character string to distinguish each DBM file for each fieldbus which is held in the table GLOBAL. This character string is also stored in the DEVICE itself and is nameing the FLASH segment and is compared during the download procedure with the name of the DBM file. If both strings are matching, the download can be performed.

6.1 Getting the FLASH Segment, the Internal Name of the DBM file

Use the function DbmGetFieldbusName() to get the name of the fieldbus.
For the DeviceNet DEVICES the following string will be returned:

HEX	44	4E	4D	20	20	20	20	20
ASCII	D	N	M					

6.2 Getting and Accessing the Table Names

Use the function `DbmGetTableDesc()` to get an overview of all tables which are included in the file and the corresponding number of data sets per table.

The function will find the following relevant table names:

HEX	47	4C	4F	42	41	4c	0
ASCII	G	L	O	B	A	L	

HEX	50	52	4F	4A	45	4B	54	0
ASCII	P	R	O	J	E	K	T	

HEX	42	55	53	5F	44	50	4D	0
ASCII	B	U	S	_	D	N	M	

HEX	50	4C	43	5F	50	41	52	41	4D	0
ASCII	P	L	C	_	P	A	R	A	M	

HEX	53	4C	5F	44	50	45	53	0
ASCII	D	E	V	I	C	E	S	

HEX	44	45	53	43	52	49	50	54	0
ASCII	D	E	S	C	R	I	P	T	

Each name is terminated with a 0 in the returned structure. For each table also a handle is returned which be must used for the later access to each table.

6.2.1 The relevant Tables and their Structure

Use the function `DbmGetRecords()` now to get the containments of each relevant table. If a table consist of more records, this function will return all records of this tables in one call.
After accessing the recordfile, be sure to free the allocated buffer memory with the function `DbmFreeBuffer()`.

6.2.1.1 The relevant Table BUS_DP, getting Bus Parameter Record

The table `BUS_DNM` will consist of one record.
The pointer `pbRecordData` in the structure `REC` of the table `BUS_DNM` points to a structure which is already defined in the chapter 'Coding of the DEVICE Parameter Data Set' .

6.2.1.2 The relevant Table DEVICES, getting Device Parameter Records

The table `DEVICES` will consist of at least one record. Use the returned value `lRecordStructCount` to get the whole number of record entries of the tables `DEVICES`.
Each pointer `pbRecordData` in the structures `REC` points to a structure which is almost defined in the chapter 'Coding of the Device Parameter Data Set'. But the structure described in this chapter is extended by one leading byte which is the configured `MAC_ID`.

Structure each pointer `pbRecordData` will be assigned to:

variable name	type	explanation
MAC_ID	**byte**	**Address of Device 0-63**
Device_Data_Set	structure	Structure already defined in the chapter 'Coding of the Device Parameter Data Set'.

7 General Procedure how to get the DEVICE operative without SyCon

Like in the chapters above described, the DEVICE supports the online configuration without using the SyCon configuration tool. That means the DEVICE must be initalized in its protocol parameters first (see. chapter protocol parameters), then a warmstart must be proceeded. After that the network specific parameter must be download via message functionality `Start_Seq, Download, End_Seq`. By using these functions, the network device specific parameters must be downloaded first and then the bus parameter must follow. The download of the bus parameter is the trigger point for the DEVICE to start its network activity the first time. The download of the data is initiated in RAM memory first step. When downloading the bus parameter file at last, a parameter decides if all parameter shall be saved in FLASH memory. This data can be reread with the `Restore-FromFlash` function and uploaded with the `Upload` function.
The just described procedure does only work, if the DEVICE isn't configured by SyCon configuration tool, else the found FLASH configuration will always have higher priority against the HOST defined configuration download parameter. To ensure that the DEVICE itself is not preconfigured by SyCon with a static FLASH configuration, for example if you have receive a new delivered one, you have to proceed the following initial sequence to get every DEVICE working:

7.1 Using Device Driver Functions

1. `DevOpenDriver()`: Enable the link of the application to the device driver
2. `DevInitBoard()`: Link application to the specific DEVICE
3. `DevPutTaskParameter()`: Set up the protocol parameter
4. `DevReset(WARMSTART)`: Execute a warm start command to DEVICE
5. `DevGetBoardInfo(GET_DRIVER_INFO)`: Read driver state
6. Examine the variable `bHostFlags` in the backcoming driver state structure
7. If `bHostFlags` indicates the bits `RDY` and `RUN` then the DEVICE is configured by SyCon. Then execute Delete database message to DEVICE by using `DevPutMessage()` and `DevGetMessage()` procedure. Goto step 3 again
8. Use now `DevPutMessage()` and `DevGetMessage()` procedure to download the network specific configuration. After the download of the bus parameter data set the DEVICE automatically starts up the network.

7.2 Using direct Access to the Dual-Port Memory

1. Examine the cell `bHostFlags` directly. If cell indicates the bits `RDY` and `RUN` then the DEVICE is configured by SyCon. Then execute `delete database` message (see chapter in this manual) to DEVICE by using corresponding message algorithm described in the `toolkit general definitions` manual. Goto step 1. If cell indicates `RDY` only then goto step 2.
2. Write protocol specific parameter into corresponding `Task2Parameter` area.
3. Write WARMSTART = 0x40 into cell `bDevFlags` to execute the DEVICE's warmstart.
4. Now download the network specific configuration via message procedure. After the download of the bus parameter data set the DEVICE automatically starts up the network.