

Direct OSEK Network Management

Technical Reference

Nm_DirOsek

Version 1.28.00

Authors:	M. Schwarz, B. Jesse, .A.Gutlederer, S. Hoffmann, V. Ebner, B. Baudermann
Version:	1.28.00
Status:	released (in preparation/completed/inspected/released)

1 Document Information

1.1 History

Author	Date	Version	Remarks
V. Ebner	1998-08-15	1.00	first version
S. Hoffmann	1998-11-30		Minor changes
G. Meiss	1999-06-04		Layout / minor changes
B. Baudermann	2000-02-29	1.02	Complete revision of the document Automatic configuration by the generation tool added
A. Gutleiderer	2000-04-10	1.03	Minor changes
A. Gutleiderer	2000-10-17	1.04	Chapter 3.3.1.3 and Picture 2: corrected ApplNmBusSleep to ApplNmCanBusSleep Chapter 4.2.4 and 4.2.5: Changed text for TargetConfigTable and ConfigMaskTable. Same in API Description Chapter 4.4.5 and 4.4.6.
A. Gutleiderer	2000-11-21	1.05	Chapter 4.1: Added Source-Code bug fix Information
A. Gutleiderer	2001-09-04	1.06	Extended callback functions added Added Multiple ECU Example added
A. Gutleiderer	2001-07-25	1.07	Minor changes Revision of API-description
A. Gutleiderer	2001-07-30	1.08	Added CANdb database attributes description
A. Gutleiderer	2001-06-11	1.09	Revision of overview in Picture 3 Callback ApplNmBusOff in example added New screenshot in chapter 6.2 added Picture in chapter 3.4 corrected
A. Gutleiderer	2001-12-14	1.10	Callback function in chapter 4.7.3 ApplNmWaitBusCancel corrected to ApplNmWaitBusSleepCancel

A. Gutleiderer	2002-10-01	1.11	<p>Abbreviation "BCD" corrected in chapter 4.1.</p> <p>Replaced T_{Overrun} with $T_{\text{waitBusSleep}}$ in chapter 3.3.</p> <p>Added picture which describes the overview of the Network Management state machine compared with the OSEK Network Management Specification in chapter 9.</p>
A. Gutleiderer	2002-05-29	1.12	Complete Revision
B. Jesse	2003-01-21	1.20	kNmChannel removed, Revision
K. Emmert	2003-08-21	1.21	CanSleep has a return parameter, how to use it in the example code.
B. Jesse	2003-09-05	1.22	<p>Reference to current OSEK specification updated [refer chapter 2.1]</p> <p>Revision of [chapter 3.3.1.8]</p> <p>StopNM(), StartNM, ReadRingData() TransmitRingData() added [refer chapter 4.4]</p> <p>ApplNmIndRingData() added [refer chapter 4.6]</p> <p>Example Ring Data access added [refer chapter 8]</p>
B. Jesse	2004-09-07	1.23	Reference to current OSEK specification updated [refer chapter 2.1]
M. Schwarz	2005-07-21	1.24	<p>Complete review & rework.</p> <p>Changed document structure.</p> <p>Adaptation to new template</p>
M. Schwarz	2006-03-09	1.25	Adapted chapter 6.4
M. Schwarz	2006-11-10	1.26	Adapted chapter 4.2 for new GENy version
M. Schwarz	2007-03-21	1.27	Adapted description of callback ApplNmBusOff()
M. Schwarz	2008-09-15	1.27.01	ESCAN00029846: Updated OEM-specific documentation
K. Emmert, M. Schwarz	2009-05-05	1.28.00	<p>Template modifications</p> <p>Added feature "LimpHome Callbacks"</p>

Table 1-1 History of the document

1.2 Reference Documents

Index	Document
[Manual_CANdriver]	Vector CAN Driver User Manual, Version 1.6
[Manual_GenTool]	If you use CANgen: [OnlineHelp_CANgen] If you use GENy: [OnlineHelp_GENy]
[Manual_OsekNm]	OSEK Network Management User Manual, Version 1.02
[OnlineHelp_CANdb]	The online help of the database editor CANDb++
[OnlineHelp_CANgen]	The online help of the configuration tool CANgen
[OnlineHelp_GENy]	The online help of the configuration tool GENy
[Spec_OsekNm]	OSEK/VDX Network Management Concept and Application Programming Interface Version 2.5.3 of 2004-07-26 The specification can be downloaded from http://www.osek-vdx.org/
[TechRef_CANdriver]	Vector CAN Driver Technical Reference, Version 2.22
[TechRef_OsekNm]	Direct OSEK Network Management Technical Reference, Version 1.27.01 (this document)
[TechRef_OsekNm_OEM]	OEM-specific addition to this document

Table 1-2 Reference documents

1.3 Abbreviations & Acronyms

Abbreviation	Complete expression
CAN	Controller Area Network
CCL	Communication Control Layer
DLL	Dynamic Link Library
ECU	Electronic Control Unit
IL	Interaction Layer
NM	Network Management Note: Within this document, NM refers to OSEK Direct NM.
OEM	Original Equipment Manufacturer
OSEK/VDX	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (Open systems and the corresponding interfaces for automotive electronics) / Vehicle Distributed eXecutive

Table 1-3 Abbreviations & acronyms

1.4 Naming Convention

Naming	Description
Nm_DirOsek	Refers to the software component that handles the OSEK Direct NM.

Table 1-4 Naming convention



Please note

We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector's release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

1	Document Information	2
1.1	History	2
1.2	Reference Documents	4
1.3	Abbreviations & Acronyms	4
1.4	Naming Convention	5
2	Overview	11
2.1	Overview	11
2.2	Delivery Package	11
2.3	Concept	12
3	Integration	13
3.1	Involved Files	13
3.2	Include Structure	13
3.3	Necessary Steps to Integrate the NM in Your Project	14
3.4	Necessary Steps to Run the NM	14
4	Configuration	16
4.1	CANgen	16
4.2	GENy	19
4.2.1	Main Settings	20
4.2.2	Channel-specific Settings	23
5	Background Information	27
5.1	Aspects of ECU Applications	27
5.2	States and State Transitions of the NM	27
5.2.1	Overview	27
5.2.2	NmOn / NmOff	28
5.2.3	NmAwake / NmWaitBusSleep / NmBusSleep	29
5.2.4	NmNormal / NmLimpHome	29
5.2.5	NmRun / NmPrepSleep	29
5.2.6	NmActive / NmPassive	29
5.3	Standard Use Cases	31
5.4	Interaction with CAN driver	32
5.4.1	Selection of the Initialization Parameters of the CAN Driver	32
5.4.2	NM Messages	32
6	Basic Functionalities	33

6.1	Initialization	33
6.1.1	Order of Initialization	33
6.1.2	Options for Initialization	33
6.2	NmTask	34
6.3	Control the Transition to the BusSleep Mode	35
6.4	Determine and Monitor the Network Configuration	35
6.4.1	Retrieve Configuration	35
6.4.2	Compare Configuration	36
6.5	Provide Status Information	37
6.6	Bus Off Handling and Recovery	39
6.6.1	Default Recovery	39
6.6.2	Fast Bus Off Recovery	40
6.7	Data Exchange with NM User Data	40
6.8	Additional Features	41
6.8.1	Immediate Alive	41
6.8.2	Delay Reset Alive	41
6.8.3	Prepare Sleep Counter	41
6.8.4	LimpHome Notification	41
7	Special Use-cases	43
7.1	Multiple ECUs	43
8	Integration Hints	45
8.1	Communication Control Layer	45
9	Related Files	46
9.1	Static Files	46
9.2	Dynamic Files	46
10	API Description	47
10.1	Overview	47
10.2	Interface for the Application	48
10.3	Interface for the CAN Driver	60
10.4	OEM-specific API	62
10.5	Callbacks	63
10.5.1	Overview	63
10.5.2	Standard Callbacks	63
10.5.3	Extended Callbacks	68
10.5.4	Special-feature callbacks	73
10.5.5	OEM-specific Callbacks	74
10.6	Other Interfaces	74

10.6.1	Version of the Source Code	74
10.6.2	Parameters and Configuration	75
11	Working with the Code	76
11.1	Version Changes	76
11.2	Application Interface	76
12	CANdb attributes	77
13	FAQ.....	79
14	Example	80
15	Contact.....	82
15.1	Related Products	82
15.2	Additional Information	82

Illustrations

Figure 2-1	Concept of NM within the CANbedded stack.....	12
Figure 3-1	Include structure of Nm_DirOsek for tool GENy	13
Figure 3-2	Include structure of Nm_DirOsek for tool CAnGen	14
Figure 4-1	Configuration of NM options in CAnGen	16
Figure 4-2	Activation of component Nm_DirOsek in GENy	19
Figure 4-3	Configuration of general NM options in GENy	20
Figure 4-4	Configuration of channel-specific NM options in GENy	23
Figure 5-1	States of NM in hierarchical context	28
Figure 5-2	Internal state-machine of Nm_DirOsek	30
Figure 6-1	Handling of user ring data in NM messages	40
Figure 7-1	Selection of multiple ECU's with the configuration tool CAnGen	43
Figure 10-1	Overview of API and interface towards CAN driver	47
Figure 10-2	Overview of callback of the NM	63

Tables

Table 1-1	History of the document.....	3
Table 1-2	Reference documents.....	4
Table 1-3	Abbreviations & acronyms	4
Table 1-4	Naming convention	5
Table 4-1	Configuration options in CAnGen	18
Table 4-2	General configuration options in GENy	23
Table 4-3	Channel-specific configuration options in GENy.....	26
Table 5-1	States of NM	28
Table 5-2	Standard use cases	32
Table 6-1	Options for initialization.....	34
Table 6-2	Assignment between node address and bit/byte position for n stations	36
Table 6-3	Access on NM status information	38
Table 10-1	Parameter interface	75
Table 12-1	NM attributes in CANdb	78

Introduction

This document describes the handling of the software component “OSEK Direct Network Management (Nm_DirOsek)”.

That means:

- How to integrate and initialize
- How to configure
- How to operate
- Basic understanding of the NM component

Additionally, this document describes the API functions of the NM.

For details on the NM algorithms, please refer to the specification of the OSEK NM [Spec_OsekNm].

2 Overview

2.1 Overview

The NM provides network services for an application operating on a CAN bus. These services are mainly:

- control the transition to the BusSleep state (see chapter 6.3)
- determine and monitor the network configuration (see chapter 6.4)
- provide status information (see chapter 6.5)
- handle and recover Bus Off (see chapter 6.6)
- exchange user data in the NM messages (see chapter 6.7)

The most important task of the NM is the safe transition into a power saving BusSleep state. Therefore the NM is mainly used in communication systems which cannot derive the need for communication from a hardware setting like e.g. ignition on/off.

2.2 Delivery Package

The delivery package includes:

- source code and header files (see chapter "9 Related Files")
- configuration tool CANgen or GENy generator DLL
- user manual [Manual_OsekNm]
- technical reference (this document)
- optional: OEM specific technical reference [TechRef_OsekNm_OEM]

2.3 Concept

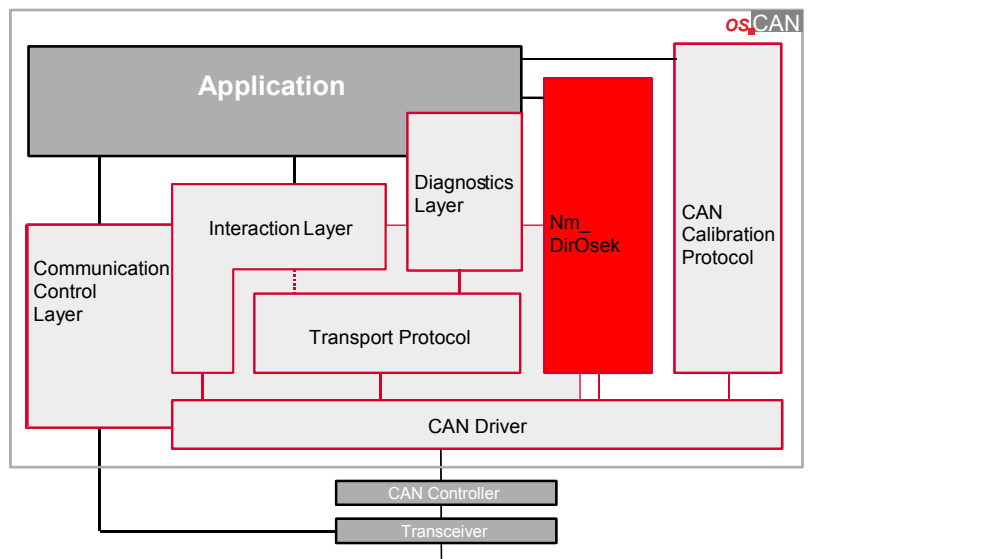


Figure 2-1 Concept of NM within the CANbedded stack

The NM needs to send and receive NM messages via the CAN bus. Therefore an underlying CAN driver is required to run the NM. The NM is initialized and controlled by the application. Status changes are signaled by the means of callback functions that have to be provided by the application (see chapter “10 API Description”).

3 Integration

3.1 Involved Files

To integrate the NM in your project, you need the files that are listed in chapter “8 Related Files”.

3.2 Include Structure

The include structure of the involved files is shown in Figure 3-1 (for GENy) and Figure 3-2 (for CANGen). Please note that the names of the files depend on the OEM.

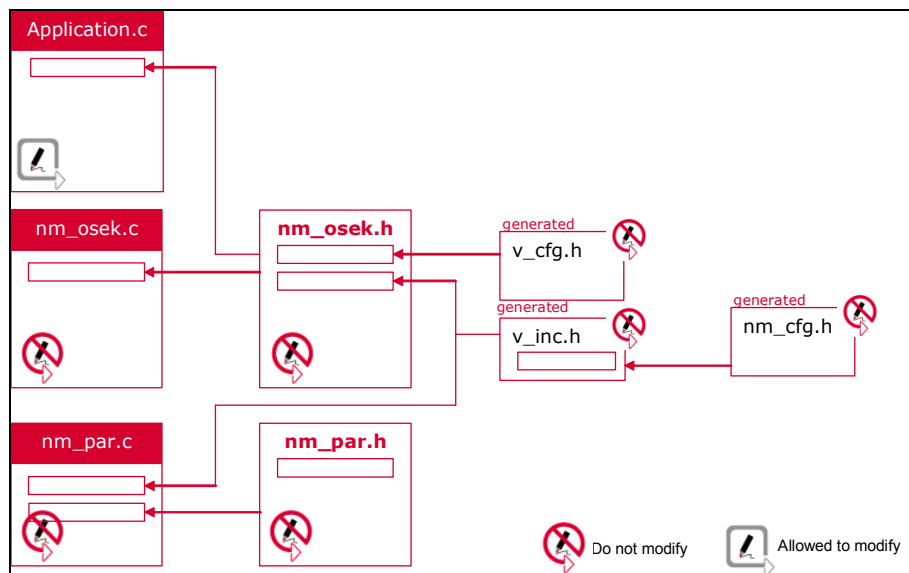


Figure 3-1 Include structure of Nm_DirOsek for tool GENy

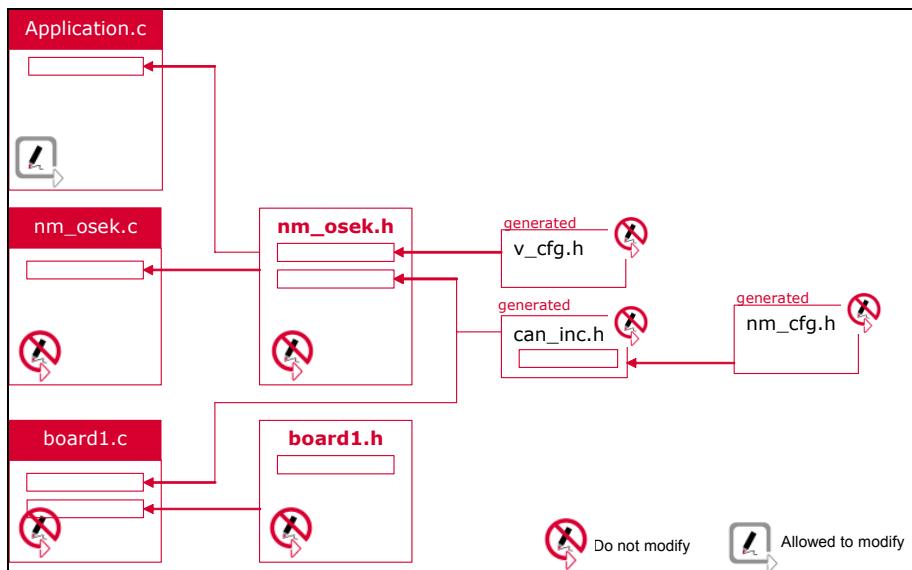


Figure 3-2 Include structure of Nm_DirOsek for tool CANgen

3.3 Necessary Steps to Integrate the NM in Your Project

Following steps may be necessary to integrate the NM in your project:

- Copy the NM related files into your project tree.
- Make these files available in your project settings, e.g. set the correct paths in your makefile.
- In order to make the NM available to your application, include the header file nm_osek.h into all files that make use of NM services and functions.
- Start the configuration tool and configure the NM according to your needs (see chapter “4 Configuration”).
- Generate the configuration files (see chapter “4 Configuration”).
- Implement all necessary callbacks in your application (see chapter “10.5 Callbacks”).
- Build your project (compile & link).

3.4 Necessary Steps to Run the NM

The NM should already have been integrated in your project and the building process should complete without any errors.

There are two main steps that have to be performed:

Initialization

- Initialize the CAN Driver by calling `CanInitPowerOn(...)` after each reset during start-up and before initializing the NM. Interrupts have to be disabled until the complete initialization procedure is done.
- Initialize the NM by calling `NmOsekInit(...)` for the desired operation mode during start-up (see chapter “6.1 Initialization”).

Cyclic call of task functions

- Add a cyclic function call of `NmTask()` to your runtime environment. Ensure that the call cycle matches the value $T_{NmCycle}$, which is configured in the configuration tool (typically 20ms). (see chapter “6.2 NmTask” and chapter “4 Configuration”)

Add the required NM services to your application, e.g. call API `GotoMode(BusSleep)` as soon as your application does not require CAN communication anymore.



Info

If you are using a CCL within the CANbedded stack, the initialization and the cyclic call of the task functions can be handled by the CCL. Please refer to the documentation of the CCL

4 Configuration

The configuration options of the NM are stored in the file nm_cfg.h. This file is created with the help of a PC-based configuration tool. Settings for the NM can be selected in the GUI. These settings are used to generate the configuration file, which is needed to compile the component.

Some options are fixed, based on information from the CAN database (DBC-file). Some other options depend on the OEM and cannot be changed. For details, please refer to the additional, customer-specific documentation [TechRef_OsekNm_OEM].

The NM is supported by two configuration tools: CANGen and GENy. Please refer to the sub-chapter that is applicable for you.

4.1 CANGen

Figure 4-1 gives an example on how the configuration GUI of tool CANGen looks like.

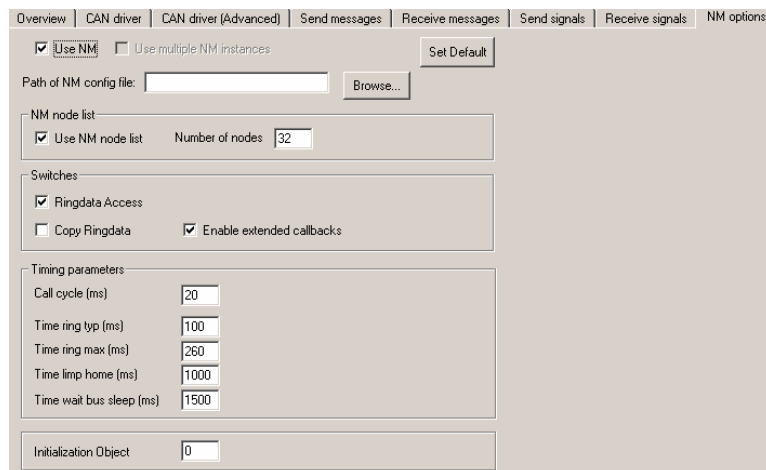


Figure 4-1 Configuration of NM options in CANGen

Attribute	Description
Use NM	Activates the usage of the NM. NM could be de-activated to overwrite the settings in the DBC-file. If this box is checked, the configuration tool will generate code that is necessary for the NM. If this box is not checked, no NM specific code will be generated.
Use multiple NM instances	Has to be activated if the NM (code-replicated) is used on several CAN channels.
Path of NM configuration file	A configuration file is generated by CANGen. If the user wants to extend or overwrite settings in the generated configuration file, he can specify a path to an user-defined configuration file. The user-defined configuration file will be included at the end of the generated file. Thus definitions in the user-defined configuration file can overwrite and extend definitions in the generated configuration file.

Use NM node list	<p>This attribute enables/disables a feature to monitor and store the currently available nodes of the logical ring in a bit field.</p> <p>If this attribute is enabled, API functions GetConfig() and CmpConfig() are provided by the NM.</p> <p>Hints on usage:</p> <p>Enable this attribute if you have to monitor and/or to check the present nodes. Otherwise you can disable this attribute in order to save memory and runtime resources.</p>
Number of nodes	<p>This attribute defines the number of nodes that have to be covered by the node list.</p> <p>This value is used to calculate the size of the bit field.</p> <p>Value limitations:</p> <ul style="list-style-type: none"> Value must be an integer multiple of '8' and must not exceed the value of CANdb attribute 'NmMessageCount'. <p>Attribute limitation:</p> <p>This attribute is only necessary if attribute "Use NM node list" is activated.</p>
Ring Data Access	<p>This attribute enables/disables the services to read and write the Ring Data in the NM ring messages.</p> <p>If this attribute is enabled, the API functions 'ReadRingData()' and 'TransmitRingData()' are provided by the NM.</p>
Copy Ring Data	<p>This attribute enables/disables an automatic copy mechanism:</p> <p>Ring data of received NM ring messages is automatically copied into the transmit buffer of the own NM message, if the own node is addressed.</p> <p>The transmit buffer can be modified by API 'TransmitRingData ()'.</p> <p>The content of the transmit buffer will be used when the own NM message is sent.</p> <p>If the application does not modify the buffer, the received Ring Data will be sent.</p> <p>Enable this attribute if the received Ring Data should be automatically passed on to the next NM node, e.g. if your NM node does not have to change the ring data.</p>
Enable extended callbacks	<p>This attribute enables/disables the extended callback functions of the NM.</p> <p>The NM provides general callbacks that indicate the major transitions of the NM state machine. Extended callbacks indicate additional state transitions that are not covered by the general callbacks.</p> <p>These indications can be used to inform higher layers about the state changes and to trigger certain events (e.g.. start/stop IL message handling).</p> <p>The extended callbacks are:</p> <ul style="list-style-type: none"> AppINmBusOffEnd() AppINmBusStart() AppINmWaitBusSleep() AppINmWaitBusSleepCancel() <p>If the extended callbacks are enabled, the application has to provide them.</p>

Call cycle [ms]	<p>This attribute defines the cycle time of NmTask().</p> <p>Value limitations:</p> <ul style="list-style-type: none"> Value range: 1.255 [ms] Typical value: 10ms or 20ms (depends on OEM) <p>Please make sure that the value of this attribute matches the actual call cycle of NmTask() in your system.</p>
Time ring typ [ms]	<p>This attribute defined the typical time interval between two ring messages (T_{Typ}).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> Value range: 1..n [ms] Value may not exceed 255*‘Cycle Time’ <p>Value must be an integer multiple of attribute ‘Cycle Time’.</p>
Time ring max [ms]	<p>This attribute defines the maximum time interval between two ring messages (T_{Max}).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> Value range: 1..n [ms] Value may not exceed 255*‘Cycle Time’ Value must be an integer multiple of ‘Cycle Time’. <p>Value must be greater than the value of attribute ‘TTyp’.</p>
Time limp home [ms]	<p>This attribute defines the time interval between two limp home messages (T_{Error}).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> Value range: 1..n [ms] Value may not exceed 255*‘Cycle Time’ Value must be an integer multiple of ‘Cycle Time’. <p>Value is typically: ‘T_{Error}’=10*‘TTyp’</p>
Time wait bus sleep [ms]	<p>This attribute defines the time between the detection of a SleepAcknowledge and the transition to BusSleep mode ($T_{WaitBusSleep}$).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> Value range: 1..n [ms] Value may not exceed 255*‘Cycle Time’ Value must be an integer multiple of ‘Cycle Time’. Value must be greater than ‘TErrror’. Value must be greater than ‘TMax’. <p>Value is typically: ‘$T_{WaitBusSleep}$’ = 1.5 * ‘TErrror’</p>
Initialization Object	<p>This attribute defines the handle of the initialization object that will be used to re-initialize the CAN driver (see [TechRef_CANdriver]).</p> <p>The initialization object covers the contents of the hardware registers of the CAN controller, e.g. the Bit Timing registers.</p>

Table 4-1 Configuration options in CANgen

4.2 GENy

A general description of GENy can be found in [OnlineHelp_GENy].

The Nm_DirOsek is listed in the dialogue “System Configuration” if

- the used DBC-file contains the necessary attributes (see chapter “12 CANdb attributes”)
- the license covers the component.

The Nm_DirOsek can be assigned to the existing channels by checking the corresponding box (Figure 4-2).

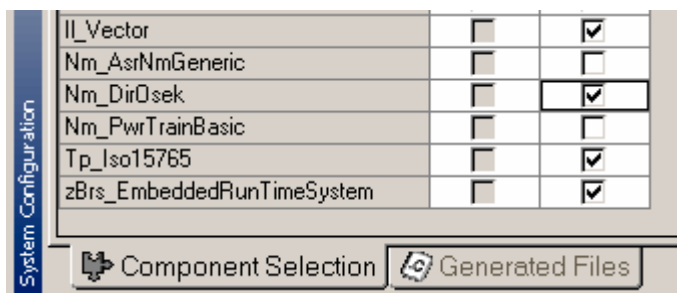


Figure 4-2 Activation of component Nm_DirOsek in GENy

The configuration is split into a general and a channel-specific part. Some configuration options depend on the OEM and won't be available in your tool.

Please also refer to the online help system of tool GENy to get more information on the configuration options for the NM.

4.2.1 Main Settings

Figure 4-3 illustrates which general configuration options of the NM can be set for Nm_DirOsek.

Configurable Options	Nm_DirOsek
General Settings	
Indexed Component	<input type="checkbox"/>
User Config File	* <input type="text" value="..."/>
Extended Callback	<input type="checkbox"/>
Node Monitoring	<input type="checkbox"/>
Number of Nodes	32*
Bus Diagnostics	
Bus Diagnostics	<input type="checkbox"/> *
Notification of Error Bit Changes	<input type="checkbox"/> *
Transceiver's Error Pin Verification at Transmission	<input type="checkbox"/> *
Extended Specification	
Immediate Alive	<input type="checkbox"/>
Fast Bus Off Recovery	<input type="checkbox"/>
NM Extensions	
OEM specific Adaption Layer	<input type="checkbox"/>
Ring Data	
Access	<input type="checkbox"/>
Copy to Transmit Buffer	<input type="checkbox"/>
Copy to Local Buffer	<input type="checkbox"/>
Signal API	<input type="checkbox"/>
User Data Protocol	
User Data Access	<input type="checkbox"/> *
Broadcast Data	
Sending of Alive Message	<input type="checkbox"/> *
Reception of Alive Message	<input type="checkbox"/> *
Sending of Ring Message	<input type="checkbox"/> *
Reception of Ring Message	<input type="checkbox"/> *
Ring Data	
Sending of Ring Message	<input type="checkbox"/> *
Reception of Ring Message	<input type="checkbox"/> *

Figure 4-3 Configuration of general NM options in GENy

Attribute	Description
Indexed Component	Selects if the indexed or non-indexed version of the NM component is used. Must be set if the NM is used on more than one CAN channel.

User Config File	<p>A configuration file is generated by GENy.</p> <p>If you want to extend or overwrite settings in the generated configuration file, you can specify a path to an user-defined configuration file.</p> <p>The user-defined configuration file will be included at the end of the generated file. Thus definitions in the user-defined configuration file can overwrite and extend definitions in the generated configuration file.</p>
Extended Callback	<p>This attribute enables/disables the extended callback functions of the NM.</p> <p>The NM provides general callbacks that indicate the major transitions of the NM state machine. Extended callbacks indicate additional state transitions that are not covered by the general callbacks.</p> <p>These indications can be used to inform higher layers about the state changes and to trigger certain events (e.g.. start/stop IL message handling).</p> <p>The extended callbacks are:</p> <ul style="list-style-type: none"> ■ ApplNmBusOffEnd() ■ ApplNmBusStart() ■ ApplNmWaitBusSleep() ■ ApplNmWaitBusSleepCancel() <p>If the extended callbacks are enabled, the application has to provide them.</p>
Node Monitoring	<p>This attribute enables/disables a feature to monitor and store the currently available nodes of the logical ring in a bit field.</p> <p>If this attribute is enabled, API functions GetConfig() and CmpConfig() are provided by the NM.</p> <p>Hints on usage:</p> <p>Enable this attribute if you have to monitor and/or to check the present nodes. Otherwise you can disable this attribute in order to save memory and runtime resources.</p>
Number of Nodes	<p>This attribute defines the number of nodes that have to be covered by the node list.</p> <p>This value is used to calculate the size of the bit field.</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value must be an integer multiple of '8' and must not exceed the value of CANdb attribute 'NmMessageCount'. <p>Attribute limitation:</p> <p>This attribute is only necessary if attribute 'Node Monitoring' is activated.</p>
Bus Diagnostics	<p>This attribute enables/disables the verification of logical and physical bus status at message transmission and reception.</p> <p>The logical status represents the return value of CAN driver API 'CanTransmit()'.</p> <p>The physical status represents the bus transceiver's error pin during message reception. The NM executes callback 'ApplNmCanErrorPinReceive()' at each reception of a valid NM message. This callback has to be provided by higher layers and has to return the status of the transceiver's error pin.</p> <p>If this attribute is enabled, the logical and physical status are collected during reception or transmission of a NM message.</p>

Notification of Error Bit Changes	<p>This attribute enables/disables the verification of the transceiver's error pin during the reception of any NM message.</p> <p>If this attribute is enabled, the NM executes the callback 'ApplNmCanErrorPinReceive()' at each reception of a valid NM message. This callback has to be provided by higher layers and has to return the status of the transceiver's error pin.</p> <p>Disable this attribute if your transceiver does not have an error pin.</p> <p>If this attribute is disabled, memory resources and runtime can be saved.</p>
Tranceiver's Error Pin Verification at Transmission	<p>This attribute enables/disables the verification of the transceiver's error pin during the transmission of the NM message.</p> <p>If this attribute is enabled, the NM executes the callback 'ApplNmCanErrorPinSend()' at each transmission of the node's NM message. This callback has to be provided by higher layers and has to return the status of the transceiver's error pin.</p> <p>Disable this attribute if your transceiver does not have an error pin.</p> <p>If this attribute is disabled, memory resources and runtime can be saved.</p>
Immediate Alive	<p>This attribute enables/disables the immediate transmission of an ALIVE message</p> <p>If an ECU receives a SleepAcknowledge, but is not ready for sleep any more (e.g. due to an internal wake event) the node may not enter BusSleep mode. There are two possibilities on how the NM proceeds:</p> <ul style="list-style-type: none"> ■ If this attribute is disabled, the NM node will stay in its previous mode. When time 'TMax' has elapsed, the node will detect a missing NM message and changes to its startup phase. ■ If this attribute is enabled, the NM immediately changes to its startup phase and an ALIVE message is transmitted.
Fast Bus Off Recovery	<p>This attribute activates the usage of the 'Fast Bus Off Recovery' algorithm (see chapter "6.6.2 Fast Bus Off Recovery").</p> <p>This algorithm uses two timings for Bus Off recovery: Timing parameter 'TErrorFast' is used 'Number of retries after Bus Off' times, before switching the timeout to 'TError'.</p> <p>Related attributes:</p> <p>If activated, please check and set the timing values 'TErrorFast' and 'Number of retries after Bus Off' on the channel-specific configuration page of the component Nm_DirOsek.</p>
LimpHome Notification	<p>This attribute enables/disables notification callbacks for entry/exit of NM LimpHome state. (see chapter "6.8.4 LimpHome Notification")</p>
Access	<p>This attribute enables/disables the services to read and write the Ring Data in the NM ring messages.</p> <p>If this attribute is enabled, the API functions 'ReadRingData()' and 'TransmitRingData()' are provided by the NM.</p>
Copy to Transmit Buffer	<p>This attribute enables/disables an automatic copy mechanism:</p> <p>Ring data of received NM ring messages is automatically copied into the transmit buffer of the own NM message, if the own node is addressed.</p> <p>The transmit buffer can be modified by API 'TransmitRingData ()'.</p> <p>The content of the transmit buffer will be used when the own NM message is sent.</p> <p>If the application does not modify the buffer, the received Ring Data will be sent.</p> <p>Enable this attribute if the received Ring Data should be automatically passed on to the next NM node, e.g. if your NM node does not have to change the ring data.</p>

Copy to Local Buffer	<p>This attribute enables/disables an automatic copy mechanism:</p> <p>Ring data of received NM ring messages is automatically copied into a local buffer. The contents of this buffer can be read by the application.</p> <p>However, the data will not be copied automatically to the transmit buffer. Every byte of the transmit buffer is initialized with a default pattern. The transmit buffer can be modified by API TransmitRingData().</p> <p>Enable this attribute if you have to process the received Ring Data and if there is no need to automatically pass on this data to the next NM node.</p> <p>This attribute can only be enabled if attribute 'Copy to Transmit Buffer' is not enabled.</p> <p>If this attribute is enabled, an additional buffer is provided by the NM. This results in higher memory consumption.</p>
----------------------	--

Table 4-2 General configuration options in GENy

4.2.2 Channel-specific Settings

The channel-specific configuration options are shown in Figure 4-4.

Configurable Options	Channel 0
Type of bussystem	CAN
Manufacturer	BMW
<input type="checkbox"/> NmOsekDirect	
<input type="checkbox"/> Information	
NmMessageCount	128
NmBaseAddress	0x480
<input type="checkbox"/> Standard Settings	
Initialization object	0*
<input type="checkbox"/> Timing Parameters [ms]	
NM call cycle [ms]	20*
Time Ring Typ [ms]	100*
Time Ring Max [ms]	260*
Time Limp Home [ms]	1000*
Time Wait Bus Sleep [ms]	1500*
<input type="checkbox"/> Network Access	
Delay Reset Alive [ms]	0*
Prepare Sleep Counter	1*
<input type="checkbox"/> Misc	
Fast Retry Time [ms]	40*
Number of Retries after BusOff	10*

Figure 4-4 Configuration of channel-specific NM options in GENy

Attribute	Description
Type of bussystem	Fixed to 'CAN'
Manufacturer	This setting can't be modified in GENy. It is an OEM-specific value that is stored in the DBC-file in attribute 'Manufacturer'.

Nm Message Count	This setting can't be modified in GENy. It is an OEM-specific value that is stored in the DBC-file in attribute 'NmMessageCount'.
Nm Base Address	This setting can't be modified in GENy. It is an OEM-specific value that is stored in the DBC-file in attribute 'NmBaseAddress'.
Cycle Time [ms]	<p>This attribute defines the cycle time of NmTask().</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 1.255 [ms] ■ Typical value: 10ms or 20ms (depends on OEM) <p>Please make sure that the value of this attribute matches the actual call cycle of NmTask() in your system.</p>
T _{Typ} [ms]	<p>This attribute defined the typical time interval between two ring messages (T_{Typ}).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 1..n [ms] ■ Value may not exceed 255* 'Cycle Time' ■ Value must be an integer multiple of attribute 'Cycle Time'.
T _{Max} [ms]	<p>This attribute defines the maximum time interval between two ring messages (T_{Max}).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 1..n [ms] ■ Value may not exceed 255* 'Cycle Time' ■ Value must be an integer multiple of 'Cycle Time'. ■ Value must be greater than the value of attribute 'T_{Typ}'.
T _{Error} [ms]	<p>This attribute defines the time interval between two limp home messages (T_{Error}).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 1..n [ms] ■ Value may not exceed 255* 'Cycle Time' ■ Value must be an integer multiple of 'Cycle Time'. ■ Value is typically: 'T_{Error}'=10*T_{Typ}
T _{WaitBusSleep} [ms]	<p>This attribute defines the time between the detection of a SleepAcknowledge and the transition to BusSleep mode (T_{WaitBusSleep}).</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 1..n [ms] ■ Value may not exceed 255* 'Cycle Time' ■ Value must be an integer multiple of 'Cycle Time'. ■ Value must be greater than 'T_{Error}'. ■ Value must be greater than 'T_{Max}'. ■ Value is typically: 'T_{WaitBusSleep}' = 1.5 * 'T_{Error}'

T _{ErrorFast} [ms]	<p>This attribute defines the time interval between two limp home messages if fast Bus Off recovery is used.</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 1..n [ms] ■ Value may not exceed 255* 'Cycle Time' ■ Value must be an integer multiple of 'Cycle Time'. <p>Access limitations:</p> <ul style="list-style-type: none"> ■ This option is only available if attribute "Fast Bus Off Recovery" is activated on the Nm_DirOsek general page.
Number of retries after Bus Off	<p>This attribute defines how often time 'TErrorFast' has to be used before switching to the OSEK-conform timing 'TError'.</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 0..255 <p>Attribute limitations:</p> <ul style="list-style-type: none"> ■ Attribute is only available if "Fast Bus Off Recovery" is activated on the Nm_DirOsek general page.
T _{StartDelay} [ms]	<p>This attribute defines a time delay between the initial ALIVE and the consecutive RING messages within the startup of the node.</p> <p>A NM node starts its network communication with a start up sequence: It sends an ALIVE message, waits for time 'TTyp' and then sends its first RING message.</p> <p>The delay between the ALIVE and RING message time 'TTyp' can be lengthened in steps of the NM task cycle ('Cycle Time').</p> <p>This asynchronous network startup avoids the creation of a partial ring and improves the assignment of the initial token ownership.</p> <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 0..n [ms] ■ Value may not exceed 255* 'Cycle Time' ■ Value must be an integer multiple of 'Cycle Time'. ■ Value must be less than 'TMax'-'TTyp' <p>Please also see chapter "6.8.2 Delay Reset Alive".</p>

Prepare Sleep Counter	<p>This attribute defines how many complete ring cycles with a permanent set SleepIndication flag have to occur before the SleepAcknowledge flag is sent.</p> <p>The SleepAcknowledge flag is transmitted if a NM node that is ready for sleep receives the token and no NM message with cleared SleepIndication flag has occurred since this node sent out its last NM ring message.</p> <p>Hints on usage:</p> <ul style="list-style-type: none"> ■ The value of this attribute should be set to '1' according to the specification of the OSEK Direct NM. ■ However, especially for the sleep synchronization through gateways, a counter can be used to get an additional delay. This counter prevents the NM node from any transmission of a SleepAcknowledge flag until the counter reaches zero. The counter is decremented each time a SleepAcknowledge flag should be sent by the own node. <p>Value limitations:</p> <ul style="list-style-type: none"> ■ Value range: 1..255 <p>Please also see chapter "6.8.3 Prepare Sleep Counter".</p>
Initialization object	<p>This attribute defines the handle of the initialization object that will be used to re-initialize the CAN driver (see [TechRef_CANdriver]).</p> <p>The initialization object covers the contents of the hardware registers of the CAN controller, e.g. the Bit Timing registers.</p>

Table 4-3 Channel-specific configuration options in GENy

5 Background Information

This chapter provides background information that help to understand the NM internal mechanisms and the interaction with other components.

5.1 Aspects of ECU Applications

Normally an ECU can take part in network communication. An application can either provide or use the information contained in the bus traffic.

If a NM node does not require communication any more, it sets the flag `SleepIndication` within its NM message (call of `GotoMode(BusSleep)`). This state of the node is also called "Local Application".

If all NM nodes have signaled that they do not require the communication any more, the bus can enter its `BusSleep` state.

5.2 States and State Transitions of the NM

There are several states and sub-states within the NM:

- `NmOn/NmOff`
- `NmAwake/NmWaitSleep/NmBusSleep`
- `NmNormal/NmLimpHome`
- `NmRun/NmPrepSleep`
- `NmActive/NmPassive`

5.2.1 Overview

The mentioned states have an hierarchical linkage (see Figure 5-1).

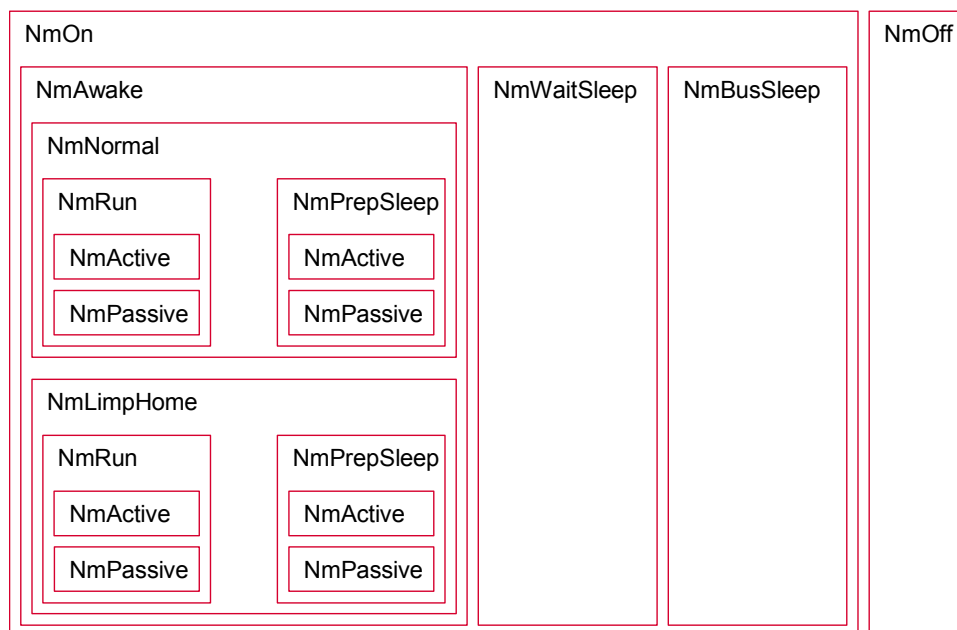


Figure 5-1 States of NM in hierarchical context

Reference name	Application status	NM states	Bus	CAN
Normal	The application requires communication.	NmOn NmAwake NmRun	AWAKE	Online
SleepInd	The application does not require communication anymore.	NmOn NmAwake NmPrepSleep	AWAKE	Online
WaitBusSleep	The application does not require communication.	NmOn NmWaitSleep	AWAKE	Online
Sleep	The application does not require communication.	NmOn NmBusSleep	ASLEEP	Offline
PowerOff	N/A	N/A	ASLEEP	N/A

Table 5-1 States of NM

5.2.2 NmOn / NmOff

The state NmOn is the normal state of the NM: The NM is ready to take part in the logical ring. It sends its NM messages and monitors the network. Internal state transitions can occur. NmOn is entered after initialization or if StartNM() is called.

The NM can be set to state NmOff by calling StopNM(). Now the NM is disabled. A disabled NM does not take part in the logical ring. There are no NM messages from this

node and no internal state transitions of the NM. Especially, there will be no transition to BusSleep!

5.2.3 NmAwake / NmWaitBusSleep / NmBusSleep

State NmWaitBusSleep is entered if all participating NM nodes have signaled their readiness for BusSleep, i.e. a NM message with a set SleepAcknowledge flag was received or transmitted.

This state is kept for some time before the transition to NmBusSleep occurs automatically.

State NmAwake is entered if there is any need for communication.

The state changes between NmAwake and NmBusSleep are performed internally, depending on the status of the NM.

5.2.4 NmNormal / NmLimpHome

NmNormal is the default state if the NM is awake. In this state, the NM runs the algorithm to establish and run the logical ring. The NM can also determine the current network configuration.

State NmLimpHome is entered if a fatal bus error occurs or if the NM can't send or receive NM messages. In this state, the NM cyclically tries to send so-called LimpHome messages. If message transmission and reception works again (e.g. bus error has been removed), this state is left again.

The state changes between NmNormal and NmLimpHome are performed internally, depending on the status of the NM.

5.2.5 NmRun / NmPrepSleep

If the NM is in state NmNormalActive, the node sends its NM message according to the specification.

NmPrepSleep is entered when GotoMode(Sleep) is called. This API indicates that this node does not need communication any more and the NM node waits for the network to enter BusSleep mode.

NmRun is entered when GotoMode(Awake) is called.

5.2.6 NmActive / NmPassive

The state NmActive is the normal state of the NM: The NM actively takes part in the logical ring. NmActive can be activated by calling API TalkNM().

NmPassive is only meant for testing purpose, as it prevents the NM from active participation in the ring. NmPassive can be entered by calling API SilentNM(). If an ECU is in NmPassive state, it remains in its previous main state. It listens to the bus without sending any own NM message. However, the ECU can enter BusSleep on demand.

There is no token inspection and there are no state transitions. However, application messages can still be sent.

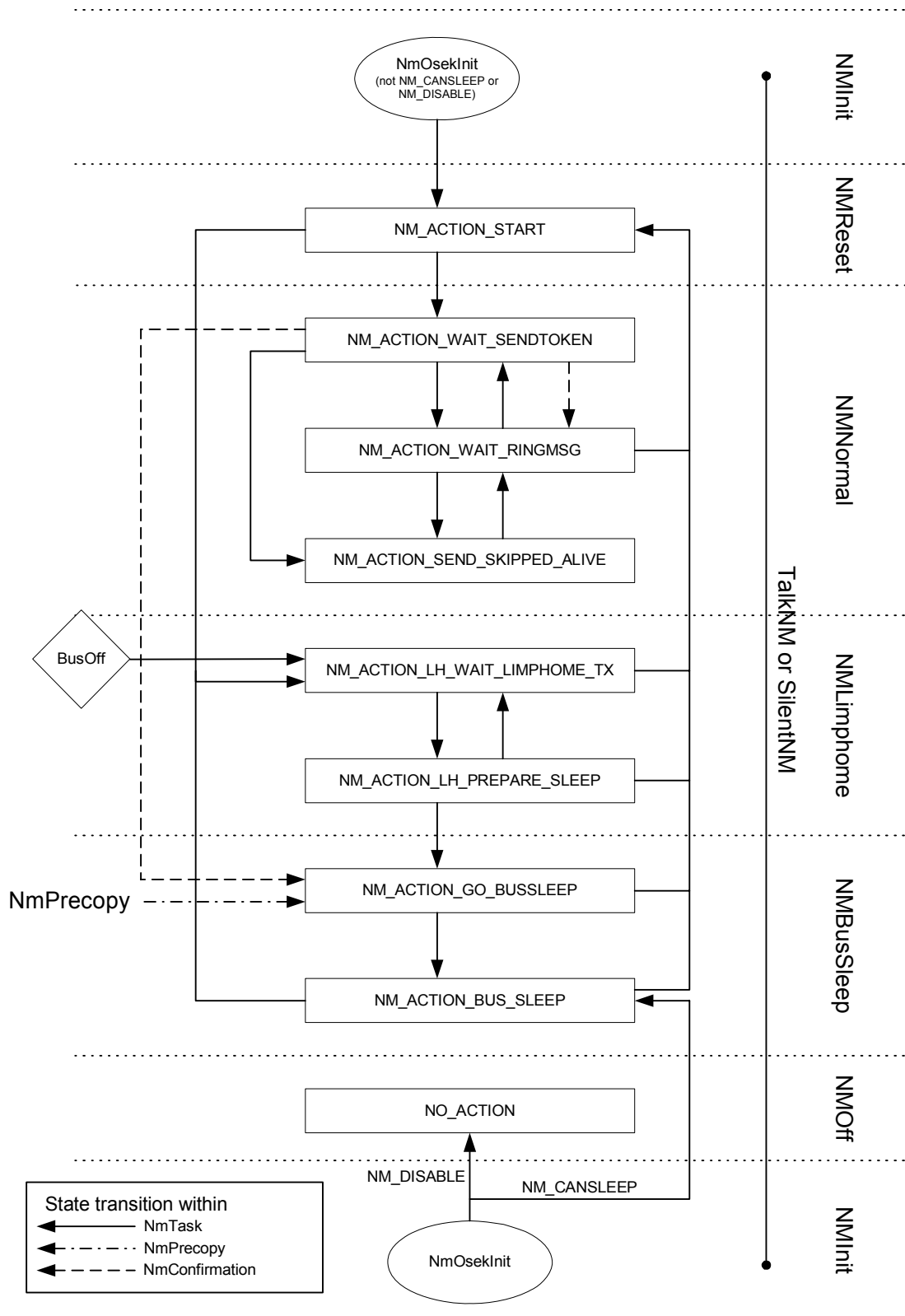


Figure 5-2 Internal state-machine of Nm_DirOsek

5.3 Standard Use Cases

Reference	Description	associated NM services
start (active) network operation	<p>The application requires CAN communication and starts the NM.</p> <p>The application is not ready to sleep.</p> <p>It informs the NM, that it requires the bus.</p> <p>The NM activates the transceiver and CAN controller with the help of callback <code>AppINmCanNormal()</code> which has to be provided by the application.</p>	NmOsekInit (NM_NORMAL)
start (passive) network operation	<p>The application detects, that it is required to provide CAN communication and starts the NM.</p> <p>The application is ready to sleep. It informs the NM, that it does not require the bus actively.</p> <p>The NM activates the transceiver and CAN controller with the help of callback <code>AppINmCanNormal()</code> which has to be provided by the application.</p>	NmOsekInit (NM_SLEEPIND)
start local operation	<p>The application does not require CAN communication. The NM is started in SLEEP mode.</p> <p>The communication path is switched off.</p>	NmOsekInit (NM_CANSLEEP)
set sleep indication	The application is ready to sleep and does not need the bus anymore.	GotoMode(BusSleep)
cancel sleep indication	The application is no more ready to sleep and needs the bus again.	GotoMode(Awake)
stop the network operation	<p>All ECUs in the network are ready to sleep.</p> <p>The NM has set all ECUs into the BusSleep mode.</p> <p>The application has to provide the service <code>AppINmCanSleep()</code>. This callback sets the communication hardware (CAN controller, bus transceiver) to a state from which they can be woken up by the CAN bus.</p> <p>The NM announces the successful transition to BusSleep mode with the help of <code>AppINmCanBusSleep()</code>.</p>	—
external wake-up caused by starting network operation	<p>The application operates in local mode.</p> <p>An external wake-up event occurs.</p> <p>The application detects that communication is required.</p> <p>As the application requires the bus itself, the NM is started without a set SleepIndication flag.</p>	NmOsekInit (NM_NORMAL) or GotoMode(Awake)

demand network operation by the CAN bus	<p>The application operates in local mode. The bus is asleep.</p> <p>The CAN controller detects a dominant level on the CAN bus. The CAN driver signals the wake-up event by calling <code>ApplCanWakeUp()</code>.</p> <p>The NM node has to participate in the bus traffic because it was woken up by the bus.</p> <p>As the application does not require communication itself, the NM is started with a set <code>SleepIndication</code> flag.</p>	NmOsekInit (NM_SLEEPIND)
local wake-up caused by an event without communication demand	<p>The application operates in local mode.</p> <p>An external wake-up event occurs. This event can be processed locally. There is no need to activate the communication. Therefore the NM is started in <code>BusSleep</code> mode.</p>	NmOsekInit (NM_CANSLEEP)
switch off the CPU (Power-Off)	The application switches off the CPU with the help of a „power on logic“.	—

Table 5-2 Standard use cases

5.4 Interaction with CAN driver

5.4.1 Selection of the Initialization Parameters of the CAN Driver

The CAN driver and the CAN controller have to be initialized by the application (`CanInitPowerOn(...)`) before the NM is initialized. Several sets of initialization parameters can be defined in the configuration tool (see [Manual_GenTool]).

The CAN controller will be re-initialized by the NM (`CanInit(InitObject)`) after a fatal bus error (Bus Off) and before the `BusSleep` mode is entered. The NM determines the current set of initialization parameters from the setting in parameter `kNmCanPara`. This parameter is configured automatically by the configuration tool (see chapter “4 Configuration”).

5.4.2 NM Messages

The NM needs an underlying CAN driver to send and receive its NM messages. The content of the NM message depends on the OEM (see [TechRef_OsekNm_OEM]). The message is assembled within the NM and the request for transmission is triggered according to the NM timing specification.

The parameter `kNmTxObj` contains the handle of a transmit object which is used by the NM to send NM messages:

```
CanTransmit(kNmTxObj)
```

The NM station address of the ECU is stored in parameter `kNmEcuNr`. This parameter is used internally to implement the NM algorithms, e.g. to initialize the logical successor or to detect the condition “skipped over”.

Please also see chapter “7.1 Multiple ECUs”

6 Basic Functionalities

6.1 Initialization

6.1.1 Order of Initialization

The bus transceiver, the CAN controller and the CAN driver have to be initialized once after a power-on reset and before starting the NM.

The application has to activate the transceiver component (switching into „normal mode“) and the CAN driver (CanInitPowerOn()) before calling NmOsekInit().



Example

```
main()
{
    DI(); /* lock interrupts */
    ...
    TrcvInitPowerOn(); /* initialize bus transceiver */
    CanInitPowerOn(); /* initialize CAN controller & driver */
    NmOsekInit( NM_NORMAL ); /* initialize NM */
    ...
    EI() /* restore interrupts */
    ...
    for(;;)
}
```



Info

Interrupts have to remain disabled until the complete initialization procedure is finished!

6.1.2 Options for Initialization

The NM is initialized by calling NmOsekInit(<parameter>). The parameters are used to switch directly to certain NM states during initialization (see Figure 6-1).

Parameter	State of the NM	Notes
NM_DISABLE	NmOff	The NM is offline: There is no NM functionality. It is not possible to access the CAN. It is possible to re-initialize the NM later on.

NM_NORMAL	NmOn, NmAwake, NmNormal, NmActive	The NM node takes part in network operation: It actively sends its own NM messages. The NM is not ready for BusSleep.
NM_NMPASSIVE	NmOn, NmAwake, NmNormal , NmPassive	The NM node does not take part in network operation: No NM messages are sent by this node. The NM is not ready for BusSleep. Note: This option is only recommended for test purpose!
NM_SLEEPIND	NmOn, NmAwake, NmNormalPrepSleep, NmActive	The NM node takes part in network operation: It actively sends its own NM messages. The NM is prepared for BusSleep.
NM_SLEEPIND_ NMPASSIVE	NmOn, NmAwake, NmNormalPrepSleep, NmPassive	The NM node does not take part in network operation: No NM messages are send. The NM is prepared for BusSleep. Note: This option is only recommended for test purpose!
NM_CANSLEEP	NmOn, NmSleep	There is no network operation: The bus is asleep. The CAN driver and bus transceiver are offline or in standby mode. Note: This option has to be selected after a hardware reset from BusSleep mode, if the CAN components should directly be initialized with the readiness for waking up.

Table 6-1 Options for initialization

6.2 NmTask

NmTask() has to be called cyclically by the application with a constant time period. The cycle time can be configured by the application (typically: $T_{NmCycle} = 20ms$, OEM dependent). The settings of the NM specific timers $T_{Limphone}$, $T_{RingTyp}$, $T_{RingMax}$, $T_{WaitBusSleep}$ und $T_{NmCycle}$ are configured in the configuration tool (see chapter “4 Configuration”). These parameters are normally defined by the OEM. For details on this timing, please refer to the specification [Spec_OsekNm] and the requirements of the OEM [TechRef_OsekNm_OEM].



Info

The function NmTask() may not be called in interrupt context.

6.3 Control the Transition to the BusSleep Mode

The NM ensures that all active ECUs within a network will enter the BusSleep mode simultaneously.

This is required because an ECU will be woken up by any message on the CAN bus. That is the reason why a transition of a single ECU into the BusSleep mode only makes sense, if all the other ECUs are also ready for BusSleep mode and will not send further messages.

The NM algorithm (see [Spec_OsekNm]) detects the need to enter BusSleep mode or to wake up the bus again. That means that the NM decides when the CAN channel must be activated or deactivated.

As the NM has no information on the connected hardware, the application must handle the underlying components. Therefore the application has to provide the callback functions ApplNmCanNormal() and ApplNmCanSleep(). They have to handle the activation and deactivation of the bus transceiver and CAN controller.

The NM is ready for BusSleep, if the application decides that no more CAN communication is needed and therefore calls GotoMode(BusSleep), or if the NM has been initialized with NmOsekInit(NM_SLEEPIND). The readiness for BusSleep remains until NM service GotoMode(Awake) is called or the NM is (re)-initialized (e.g. due to hardware reset).

The NM calls the function ApplNmCanBusSleep() after the transition into the BusSleep mode. The call of this application function is important for the application to show that the BusSleep mode has been achieved. The successful transition to BusSleep mode can also be checked by the status information of the NM (e.g. NmStateBusSleep(NmGetStatus())).

6.4 Determine and Monitor the Network Configuration

The network configuration is determined in the so-called “Start Phase”. Each NM node regards itself as ready for operation.

After the “Start Phase”, the NM monitors the network. This “Monitoring Phase” is used to detect new nodes as well as the loss of existing nodes.

This feature can be enabled/disabled in the configuration tool. In order to save runtime and memory resources, do not enable this feature unless you need it.

6.4.1 Retrieve Configuration

The NM collects and provides information about the network nodes that are currently active (see API GetConfig()).

The network configuration of the NM is updated each time a NM message is received, i.e. in ISR-context. Please be aware of this behavior if multiple decisions in the application are necessary to gain a result (e.g. multiple nested “if”). Maybe it is necessary to copy the current network configuration to a temporary variable.

The configuration is discarded when the ring is not stable anymore, e.g. if a node gets lost or a bus error occurs.

6.4.2 Compare Configuration

The NM provides the possibility to compare the current network configuration with a reference configuration (see API CmpConfig()).

This reference configuration is stored in two arrays that have to be provided by the application:

- **TargetConfigTable** Indicates the required status of the NM nodes. If the corresponding bit is set ('1'), the associated NM node has to be present. If the bit is not set ('0'), the node may not be present.
- **ConfigMaskTable** Indicated which NM nodes have to be checked. If the corresponding bit is set ('1'), the presence of the associated NM node has to be evaluated. If the bit is not set ('0'), the node is not of interest for the comparison.

The assignment between the node address and the corresponding bit/byte position is depicted in Table 6-2.

Byte	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
...
n/8	n-1	n-2	n-3	n-4	n-5	n-6	n-7	n-8

Table 6-2 Assignment between node address and bit/byte position for n stations

Bit 0 of byte 0 represents the NM node with station address 0. Bit 7 of the most significant byte represents the NM node with the highest station address.

An ECU which should be present in the reference configuration is marked with '1' at the corresponding position in both tables.

An ECU which should not be present in the reference configuration is marked with '1' at the corresponding position in ConfigMaskTable and '0' in the TargetMaskTable.

An ECU which should not be part of the comparison is marked with '0' at the corresponding position in ConfigMaskTable.

CmpConfig() compares the current configuration with the reference only for those nodes that are marked with '1' in ConfigMaskTable.



Example

```
#include "nm_osek.h"
```

```
/* ROM tables provided by the application */
```

```
NmConfigType TargetConfigTable[2] =
```

```
{
```

```
    { 0x69, /* NM node 6,5,3,0 */ /* Configuration: Net1 */
```

```

    0x00,
    0x80, /* NM node 23 */
    0x80 /* NM node 31 */
},
{ 0x02, /* NM node 1 */ /* Configuration: LimpHome */
  0x12, /* NM node 12,9 */
  0x11, /* NM node 20,16 */
  0x40 /* NM node 30 */
},
}
NmConfigType ConfigMaskTable[2] =
{
  { 0xF9, /* NM node 7..3,0 */ /* Configuration: Net1 */
    0x00,
    0x80, /* NM node 23 */
    0x80 /* NM node 31 */
  },
  { 0x0F, /* NM node 3..0 */ /* Configuration: LimpHome */
    0x12, /* NM node 12,9 */
    0x11, /* NM node 20,16 */
    0xC1 /* NM node 31,30,24 */
  },
}
/* application */
...
#define LIMPHOME_DESIRED_CONFIG 1
...
if( CmpConfig( LIMPHOME_DESIRED_CONFIG )==1 )
{...;}

```

6.5 Provide Status Information

The NM provides information on the status of the NM node and the network itself.

This status information can be accessed with the help of APIs `GetStatus()` or `NmGetStatus()`. These two services differ in the way the result is passed back. The result itself is of type `NmStatusType` (see Table 6-3).

Bit	Name	Interpretation	Access macro(s)
0	Ring Stable	Indicates if the current network is stable. Stable means, that the configuration has not changed during the last ring cycle. 0 ring is not stable 1 ring is stable	NmStateRingStable (NmStatus)
1	Bus Off Error	Indicates if a Bus Off has occurred 0 no Bus Off error 1 Bus Off error occurred. CAN is locked. CAN driver is offline.	NmStateBusError (NmStatus)
2	Active	Indicates the operation mode of the NM 0 NMPassive (see chapter 5.2.6) 1 NMActive	NmStateActive (NmStatus) NmStatePassive (NmStatus)
3	LimpHome	Indicates if the NM is in the mode LimpHome. LimpHome is entered, if the NM can't send or receive NM messages. 0 not in state LimpHome 1 in state LimpHome	NmStateLimpHome (NmStatus)
4	BusSleep	Indicates if the bus is asleep or not 0 not in state BusSleep 1 in state BusSleep	NmStateBusSleep (NmStatus)
5	WaitBusSleep	Indicates if the NM currently waits for BusSleep 0 NM is not in state NmWaitBusSleep 1 NM is in state NmWaitBusSleep	NmStateWaitBusSleep (NmStatus)
6	TxRingData Allowed	Indicates if a write access on the user ring data is allowed or not. 0 no write access on ring data 1 write access on ring allowed Note: This flag is not supported by all OEMs	NmStateTxRingDataAllowed (NmStatus)
7	SleepIndication	Indicates if the application has signaled readiness for Sleep or not. 0 GotoMode(Awake) was called 1 GotoMode(SleepInd) was called	NmStateBusSleepInd (NmStatus) NmStateBusWakeUp (NmStatus)

Table 6-3 Access on NM status information

The OSEK specification mentions a status flag that indicates if the NM is started or not (NMOn, NMOff) [Spec_OsekNm]. This status flag is not supported by the NM component.

The status information should be evaluated only with the help of the macros given in the table (also see header file of Nm_DirOsek).

These macros return TRUE ('1') if the condition is met and FALSE ('0') if not. The meaning of these symbolic constants is described in [Spec_OsekNm].



Example

...	...
NmStatusType netState;	NmStatusType netState;
 netState = NmGetStatus();	 GetStatus(&netState);
 if(NmStateActive(netState) == 1)	 if(NmStateActive(netState) == 1)
{...;}	{...;}
else	else
{...;}	{...;}
...	...



Info

The state of the NM is updated each time a NM message is received or transmitted, i.e. in ISR-context. Please be aware of this behavior if the application uses multiple decisions to get a result (e.g. multiple nested "if"). Maybe it is necessary to copy the current status to a temporary variable.

6.6 Bus Off Handling and Recovery

The Bus Off recovery aims to re-initialize the CAN communication after a bus error.

A Bus Off condition is detected by the CAN driver. The NM is notified by the CAN driver's callback function ApplCanBusOff().

The recovery starts with setting the CAN driver to offline mode (CanOffline()). Afterwards the CAN controller is initialized (CanResetBusOffStart()). If the NM is prepared for BusSleep and the bus is already asleep (or about to enter the BusSleep state), the Bus Off recovery is ended (CanResetBusOffEnd()) and the node enters NmBusSleep.

Otherwise the timer for Bus Off recovery is started and the application is informed that a Bus Off has occurred (ApplNmBusOff()).

When the Bus Off recovery timer has elapsed, the CAN driver is set into online mode again (CanResetBusOffEnd(), CanOnline()) and the application can be optionally notified about the event with the help of callback ApplNmBusOffEnd(). If the NM is still active, it tries to send a LimpHome message.

There are two algorithms for Bus Off recovery. They can be selected in the configuration tool (see chapter "4 Configuration"):

6.6.1 Default Recovery

The default algorithm waits for the configured recovery time $T_{LimpHome}$ (e.g. 1000ms). If this time has elapsed, the NM tries to send a LimpHome message.

6.6.2 Fast Bus Off Recovery

This algorithm uses a smaller value of the recovery timer (e.g. 40ms) for a number of times (e.g. 10 times). Afterwards the normal recovery time (e.g. 1000ms) is used. The timing parameters can be configured in the configuration tool.

This algorithm is not conform with the OSEK specification [Spec_OsekNm].

This algorithm is only available for some OEMs. Please refer to [TechRef_OsekNm_OEM].

6.7 Data Exchange with NM User Data

The OSEK specification [Spec_OsekNm] defines a mechanism to exchange user data within NM messages. The user data can be used to realize functions which go beyond the OSEK specification.

However, the OSEK specification does not define any procedure how to process this data in the individual ECU. Therefore it is left up to the application or respectively to the person who is responsible for a bus system.

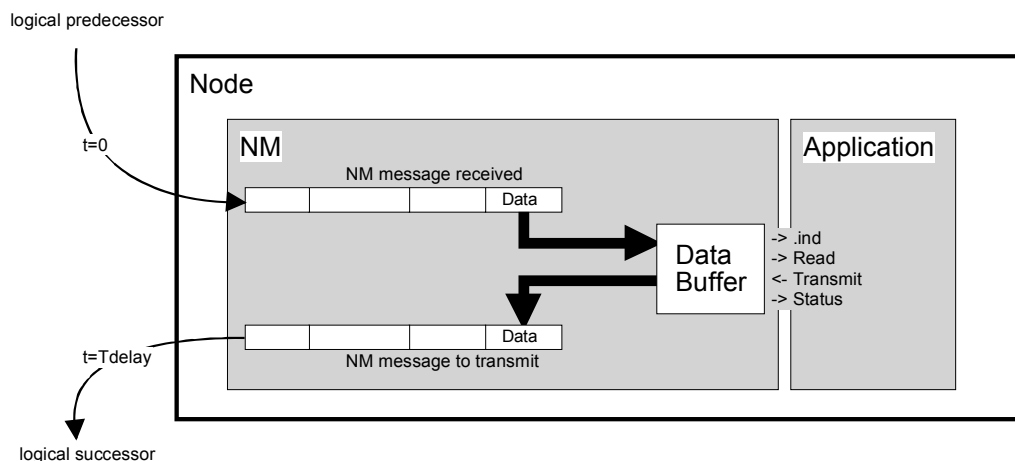


Figure 6-1 Handling of user ring data in NM messages

Figure 6-1 shows the OSEK mechanism for the transmission of user data in the logical ring. The application can read this data on a stable ring at any time. The application is able to change the data after the reception of the NM token, that means if the own station is addressed.

The application has to provide the indication function `ApplNmIndRingData()`. This function is called by the NM if a `RingMessage` was received, the own station got the token and the ring is stable. The indication will not occur, if a `SleepAcknowledge` flag was received or if the own token was already passed on.

The user data can be modified by the function `TransmitRingData()` as long as the NM status bit `TxRingDataAllowed` is set. The function returns `E_OK` if the write access is permitted, otherwise it returns `E_NotOK`.

The user ring data is initialized during `NmOsekInit()`.

Please note that the mechanism for exchanging user data strongly depends on the OEM. Please refer to the OEM-specific NM manual [TechRef_OsekNm_OEM].

This feature can be enabled/disabled in the configuration tool. In order to save runtime and memory resources, do not enable this feature unless you need it.

6.8 Additional Features

The Nm_DirOsek provides additional, OEM-specific features. Please check with your OEM, if these features are applicable for you.

6.8.1 Immediate Alive

If an ECU transmits or receives a SleepAcknowledge, it begins with the transition into BusSleep. If an internal resource requests the bus within this time ($T_{\text{WaitBusSleep}}$), the ECU waits T_{RingMax} to send its Alive message with a cleared SleepIndication flag (according to strict OSEK interpretation).

If the Immediate Alive feature is enabled, the Alive message will be sent immediately within the next NM task cycle.

6.8.2 Delay Reset Alive

A NM node starts its network communication with a start up sequence: It sends an Alive message, waits for T_{Typ} and then sends its first Ring message.

The delay between the Alive and Ring message (T_{Typ}) can be lengthened in steps of the NM task cycle. This asynchronous network startup avoids the creation of a partial ring and improves the assignment of the initial token ownership.

Please make sure, that the resulting $T_{\text{TypInitial}}$ is not larger then T_{RingMax} .

6.8.3 Prepare Sleep Counter

According to the OSEK specification, a transition to BusSleep is started if the SleepAcknowledge flag is detected. The SleepAcknowledge flag is transmitted if a NM node that is ready for sleep receives the token and no NM message with cleared SleepIndication flag has occurred since this node sent out its last NM ring message.

Especially for the sleep synchronization through gateways a counter can be used.

This counter prevents the NM node from any transmission of a SleepAcknowledge flag until the counter reaches zero.

The counter is decremented each time a SleepAcknowledge flag should be send.

6.8.4 LimpHome Notification

This is an optional feature. It is not available in all setups.

If available, it can be configured within GENy (see chapter 4.2.1)

If enabled, the NM informs the application about any state transition into or from state LimpHome with the callback functions ApplNmLimpHomeEnter() and ApplNmLimpHomeExit().

These callbacks can be used to track the time the NM stays in state LimpHome.

**Info**

ApplNmLimpHomeExit() is also called when the NM changes to state WaitBusSleep.

The NM internally remembers the LimpHome state and if WaitBusSleep is aborted, LimpHome is entered again (together with an occurrence of ApplNmLimpHomeEnter())

This is a different behavior compared to the LimpHome flag that can be accessed with NmGetStatus(): This flag is also set in state WaitBusSleep.

7 Special Use-cases

7.1 Multiple ECUs

Multiple ECUs are control units which are assembled several times within the CAN network with the same software. Example: Seat in the front on the left hand side and on the right hand side.

The application has to decide at start-up which ECU is currently installed and has to set-up these parameters dynamically (e.g. if it is a left door or right door).

Configuration

Multiple ECUs are supported in the configuration tool. The tool allows to select more than one ECU (see Figure 7-1).

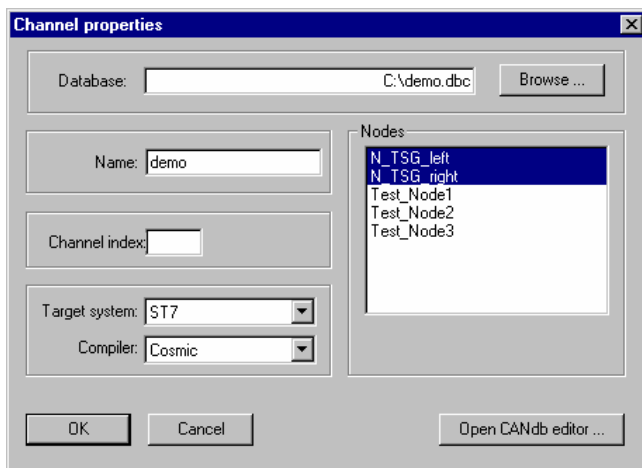


Figure 7-1 Selection of multiple ECU's with the configuration tool CANgen

The configuration tool creates two arrays that contain the ECU number and the Tx-handle of the NM message

board1.c:

```
...
V_MEMROM0 V_MEMROM1 canuint8 V_MEMROM2 NmEcuNr_Field[kComNumberOfNodes] = {
    kNmEcuNrN_TSG_right, kNmEcuNrN_TSG_left };
V_MEMROM0 V_MEMROM1 canuint8 V_MEMROM2 NmTxObj_Field[kComNumberOfNodes] = {
    kNmTxObjN_TSG_right, kNmTxObjN_TSG_left };
...
```

nm_cfg.h:

```
...
#define kNmEcuNrN_TSG_right    2
#define kNmEcuNrN_TSG_left    1
#define kNmTxObjN_TSG_right    19
#define kNmTxObjN_TSG_left    20
...
```

Furthermore, the generated node-specific files (e.g. board1.c/.h) contain following definitions and declarations:

```
#define kComNumberOfNodes      2

typedef enum {
    COM_N_TSG_right
, COM_N_TSG_left
} tComMultipleECUNames;

canuint8 comMultipleECUCurrent;

#define ComSetCurrentECU(ecu) comMultipleECUCurrent=(ecu)
```

It is up to the application to set the parameters of the currently used ECU before calling NmOsekInit().



Example

```
#include "can_inc.h"
#include "node.h"
#include "nm_osek.h"

main()
{
    ActivateTransceiver();
    CanInitPowerOn(0);

    /* Select node N_TSG_left */
    ComSetCurrentECU(COM_N_TSG_left);

    NmOsekInit(NM_NORMAL);

    EnableInterrupts();

    for (;;)          /* Main loop */
    {
        ...
    }
}
```

8 Integration Hints

8.1 Communication Control Layer

The Communication Control Layer (CCL) handles the communication within the CANbedded stack. This includes especially the NM.

The CCL handles following functionalities of the NM:

- Initialization (NmOsekInit())
- Start/Stop of communication (GotoMode())
- Optional: Call of cyclic task function (NmTask())

Please refer to the technical reference of the CCL for more information.



Info

Please be aware that the functionalities handled by the CCL may not be handled any more by the application.

9 Related Files

9.1 Static Files

nm_osek.c	Source file of the NM. Contains all API and algorithms.
nm_osek.h	Header file of the NM. Contains all static prototypes for the API and definitions, such as symbolic constants.

**Info**

The names of the static files depend on the OEM and may vary from the default names stated above. Please refer to [TechRef_OsekNm_OEM].

**Info**

It is not allowed to change the NM source code during the integration into the application. Please contact the Vector hotline if any problems arise.

9.2 Dynamic Files

The dynamic files are generated by the configuration tool.

CANgen

nm_cfg.h Configuration file

Further data is generated in the ECU-specific source and header files (e.g. board1.c/.h)

GENy

nm_cfg.h Configuration file

nm_par.c Source file of the dynamic part of the NM

nm_par.h Header file of the dynamic part of the NM

**Info**

Do not change anything within the dynamic files, as the changes will be overwritten during the next generation process.

10 API Description

The API of the NM consists of services which are realized by function calls. These services can be called wherever they are required. They transfer information to the NM or take over information from the NM.

10.1 Overview

Figure 10-1 gives an overview on the standard API of the NM and its connection to the underlying CAN driver and the application. OEM-specific extensions are described in [TechRef_OsekNm_OEM].

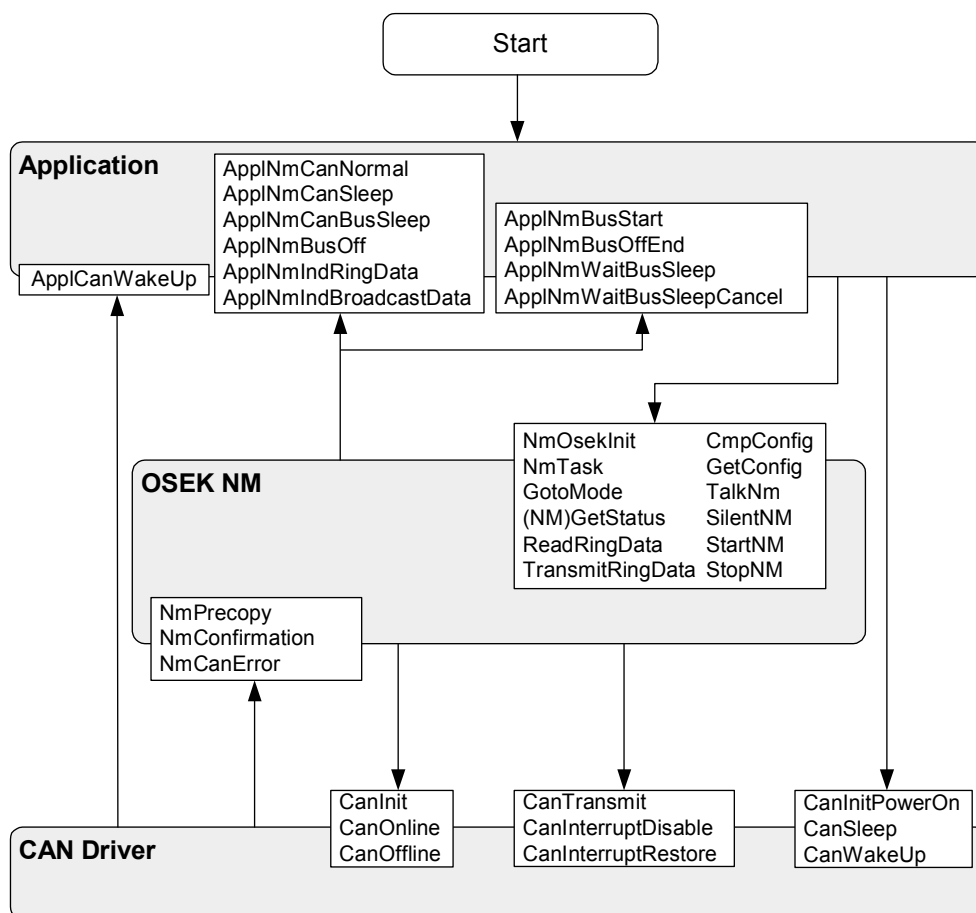


Figure 10-1 Overview of API and interface towards CAN driver

Remarks:

- The function ApplCanWakeup() is directly called during the wake up by the CAN bus from the CAN driver. This function is not supported by all CAN drivers.
- The Function ApplNmIndRingData() is not supported by all OEMs.
- The API of the CAN driver is described in [Manual_CANdriver].

All service functions of the indexed NM must be called with the channel parameter. The channel parameter represents the physical CAN channel on which the NM is used. Possible values are: 0...max(CAN channel).

Functions are not re-entrant unless state otherwise.

10.2 Interface for the Application

NmOsekInit

Prototype	
Standard	void NmOsekInit (NmInitType initMode)
Multiplied	void NmOsekInit_X (NmInitType initMode) with X=0...number of CAN channels
Indexed	void NmOsekInit (CanChannelHandle channel, NmInitType initMode)
Parameter	
channel initMode	Represents the handle of the assigned CAN channel (see chapter 10.1). Selects the initial operation mode <ul style="list-style-type: none"> ■ NM_DISABLE ■ NM_NORMAL ■ NM_NMPASSIVE ■ NM_SLEEPIND ■ NM_SLEEPIND_NMPASSIVE ■ NM_CANSLEEP Please also refer chapter "initialization".
Return code	
-	-
Functional Description	
<p>This function initializes the NM. The initial operation mode can be selected with the help of parameter initMode.</p> <p>This function is called during the initialization of the system.</p> <p>Afterwards the NM is initialized and runs in the selected operation mode.</p>	

Particularities and Limitations

- NmOsekInit() must be called before any other NM service is called.
- The CAN driver and the bus transceiver have to be initialized in advance (see chapter „6.1 Initialization“)
- During the initialization, global interrupts have to be disabled.
- NmOsekInit(NM_NORMAL) can be replaced by StartNM().
- The parameters kNmTxObj, kNmEcuNr and kNmCanPara have to be defined in advance.
- Call context: task level or interrupt level. (Example for interrupt level: NM is initialized with NmOsekInit(NM_SLEEPIND) inside ApplCanWakeUp(). NOTE: It is not possible/allowed to call NmOsekInit(NM_CANSLEEP) inside ApplCanWakeUp(), as the wakeup request is still pending!)
- The application has to ensure that NmTask() is not interrupted by the function NmOsekInit()! Otherwise all internal flags are reset and further processing of NmTask() will lead to an inconsistent system.
- Please also see: NmTask()
- It is not allowed to call NmOsekInit(NM_CANSLEEP) within ApplCanWakeUp().

NmTask

Prototype

Standard	void NmTask (void)
Multiplied	void NmTask_X (void) with X=0...number of CAN channels
Indexed	void NmTask (CanChannelHandle channel)

Parameter

channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
---------	---

Return code

-	-
---	---

Functional Description

This function handles the main tasks of the NM, like internal state handling.

NmTask() has to be called cyclically with a constant call period. The call period must match the configured setting the configuration tool. Please refer to chapter “4 Configuration”.

Particularities and Limitations

- The NM has to be initialized correctly.
- Call context: task level.
- Important note: This function must not be called in interrupt context!
- Please also see NmOsekInit()

GotoMode(BusSleep)

Prototype	
Standard	<code>void GotoMode (NMModeName mode)</code>
Multiplied	<code>void GotoMode_X (NMModeName mode)</code>
Indexed	<code>void GotoMode (CanChannelHandle channel, NMModeName mode)</code>
Parameter	
channel mode="BusSleep"	Represents the handle of the assigned CAN channel (see chapter 10.1). Indicates that the application is ready to sleep.
Return code	
-	-
Functional Description	
<p>This function triggers the transition to BusSleep mode: The NM is informed that the application is ready to enter BusSleep.</p> <p>The transition to BusSleep is synchronized with all other ECUs. If all ECUs are ready for BusSleep, the NM performs the transition of all ECUs to local mode.</p> <p>If the NM node is in LimpHome mode, no further NM messages are sent.</p> <p>See chapter "6.3 Control the Transition to the BusSleep Mode".</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ GotoMode(BusSleep) may be called immediately after StartNM() or NmOsekInit(). The transition to the state BusSleep by GotoMode(BusSleep) is also possible in NMPassive mode. ■ Call context: task level. ■ Please also see: GotoMode(Awake), TalkNM(), SilentNM() 	

GotoMode(Awake)

Prototype	
Standard	<code>void GotoMode (Awake)</code>
Multiplied	<code>void GotoMode_X (Awake)</code>
Indexed	<code>void GotoMode (CanChannelHandle channel, Awake)</code>
Parameter	
channel mode="Awake"	Represents the handle of the assigned CAN channel (see chapter 10.1). Indicates that the application is not ready to sleep.
Return code	
-	-
Functional Description	
<p>This function cancels the transition to BusSleep mode: The NM is informed that the application requires bus communication.</p> <p>The NM enters NORMAL mode.</p> <p>If the CAN bus is asleep, it is woken. If the NM is about to prepare BusSleep, this preparation is aborted.</p> <p>If the NM node is in LimpHome mode, no LimpHome messages are sent.</p> <p>See chapter "6.3 Control the Transition to the BusSleep Mode".</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: task level. ■ Please also see: GotoMode(BusSleep), TalkNM(), SilentNM() 	

GetConfig

Prototype	
Standard	void GetConfig (NmConfigType *dataPtr)
Multiplied	void GetConfig_X (NmConfigType *dataPtr)
Indexed	void GetConfig (CanChannelHandle channel, NmConfigType *dataPtr)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
*dataPtr	Gives an address in memory where the current configuration shall be written to. The size of the reserved memory space can be configured in the configuration tool.
Return code	
-	-
Functional Description	
<p>This function retrieves the current network configuration and returns it to the application. The information is written to the memory location that is given by *dataPtr.</p> <p>The calling station is always set active within the configuration (see chapter “6.4 Determine and Monitor the Network Configuration”).</p> <p>The state of the NM is not changed.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ This function depends on the OEM. Please refer to the additional OEM-specific documentation. ■ This function is only available if feature “NmNodeList” is activated in the configuration tool. ■ Call context: task level. ■ Please also see chapter “6.4 Determine and Monitor the Network Configuration”. ■ Please also see: CmpConfig() 	

CmpConfig

Prototype	
Standard	NmReturnType CmpConfig (NmConfigType vuint8 idx)
Multiplied	NmReturnType CmpConfig_X (NmConfigType vuint8 idx)
Indexed	NmReturnType CmpConfig (CanChannelHandle channel, vuint8 idx)
Parameter	
channel idx	<p>Represents the handle of the assigned CAN channel (see chapter 10.1).</p> <p>Index to an element of type NmConfigType in the arrays for configuration and mask.</p> <p>The index selects an element from TargetConfigTable and ConfigMaskTable for comparison with the current configuration.</p> <p>See chapter “6.4.2 Compare Configuration”.</p>
Return code	
NmReturnType	<p>TRUE (1) : The current configuration matches the desired configuration.</p> <p>FALSE (0) : The current configuration differs from the desired configuration.</p>
Functional Description	
<p>This function compares the current configuration of the NM with a given configuration.</p> <p>The calling station is always set active within the configuration (see “6.4 Determine and Monitor the Network Configuration”).</p> <p>The return code identifies whether the current configuration matches the reference or not .</p> <p>The state of the NM is not changed.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ This function depends on the OEM. Please refer to the additional OEM-specific documentation. ■ This function is only available if feature “NmNodeList” is activated in the configuration tool. ■ Call context: task level. ■ Please also see chapter “6.4 Determine and Monitor the Network Configuration”. ■ Please also see: GetConfig() 	

Example

```
#include "nm_osek.h"

in ROM
desired configuration of control units
NmConfigType TargetConfigTable[2] =
{
    { 0x69,      /* ECU 6, 5, 3, 0 */
      0x00,
      0x80,      /* ECU 23 */
      0x80      /* ECU 31 */
    },
    { 0x02,      /* ECU 2 */
      0x12,      /* ECU 13, 10 */
      0x11,      /* ECU 20, 16 */
      0x40      /* ECU 30 */
    }
};

in ROM
masks for configuration
NmConfigType ConfigMaskTable[2] =
{
    { 0xF9,      /* ECU 31 to 27, 24 */
      0x00,
      0x80,      /* ECU 15 */
      0x80      /* ECU 7 */
    },
    { 0x0F,      /* ECU 0 to 3 */
      0x12,      /* ECU 12, 9 */
      0x11,      /* ECU 20, 16 */
      0xC1      /* ECU 31, 30, 24 */
    }
};

in Program
...
#define LIMPHOME_DESIRED_CONFIG 1
...
if( CmpConfig (LIMPHOME_DESIRED_CONFIG) )
then ...
```

NmGetStatus

Prototype	
Standard	NmStatusType NmGetStatus (void)
Multiplied	NmStatusType NmGetStatus_X (void)
Indexed	NmStatusType NmGetStatus (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
NmStatusType	Holds the current status of the NM.
Functional Description	
<p>This function returns the status of the NM.</p> <p>The NM status is not changed by calling NmGetStatus().</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: task level. ■ Please refer to chapter “6.5 Provide Status Information”. ■ Please also see: GetStatus() ■ If the NM node is in LimpHome, NmGetStatus() also indicates that the ring is not stable. ■ If the network is in bus sleep state, the stability of the ring can't be evaluated. Therefore the corresponding information may not be evaluated in that case. ■ This function is not part of the OSEK NM API. It has been added by Vector to simplify status queries. ■ The state of the NM is updated each time a NM message is received or transmitted, i.e. in ISR-context. Please be aware of this behavior if the application uses multiple decisions (e.g. multiple nested “if”) to get a result. (Hint: copy current state in a temporary variable) 	
Example	
<pre>#include "nm_osek.h" /* name of header depends on OEM */ static NmStatusType NetState; NetState = NmGetStatus(); /* get status */ if (NmStateActive(NetState)) /* evaluate status */ { ...; } else { ...; } ... /* or */ if (NmStatePassive(NmGetStatus())) /* get and evaluate status */ { ...; } else { ...; }</pre>	

GetStatus

Prototype	
Standard	void GetStatus (NmStatusType *dest)
Multiplied	void GetStatus_X (NmStatusType *dest)
Indexed	void GetStatus (CanChannelHandle channel, NmStatusType *dest)
Parameter	
channel *dest	Represents the handle of the assigned CAN channel (see chapter 10.1). Pointer to a variable where the NM status shall be written to.
Return code	
-	-
Functional Description	
Please refer to the description of NmGetStatus(). The current status of the NM is written to the memory address that is given by *dest.	
Particularities and Limitations	
<ul style="list-style-type: none">■ The NM has to be initialized correctly.■ Call context: task level.■ Please also see: NmGetStatus ()■ The state of the NM is updated each time a NM message is received or transmitted, i.e. in ISR-context. Please be aware of this behavior if the application uses multiple decisions (e.g. multiple nested “if”) to get a result. (Hint: copy current state in a temporary variable)	
Example	
<pre>#include "nm_osek.h" /* name of header depends on OEM */ static NmStatusType NetState; GetStatus(&NetState); /* save status */ if (NmStateBusSleep(NetState)) { ...;} else { ...;}</pre>	

SilentNM

Prototype	
Standard	void SilentNM (void)
Multiplied	void SilentNM_X (void)
Indexed	void SilentNM (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
This function disables the active participation of the NM node: No more NM messages are transmitted by the station. However, the NM listens to the NM messages on the bus.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ This function should only be used for test purposes! ■ Call context: task level ■ Please also see: TalkNM() 	

TalkNM

Prototype	
Standard	void TalkNM (void)
Multiplied	void TalkNM_X (void)
Indexed	void TalkNM (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
This function enables the active participation of the NM node: NM messages are transmitted. The NM is active.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ This function should only be used for test purposes! ■ Call context: task level ■ Please also see: SilentNM() 	

StopNM

Prototype	
Standard	<code>void StopNM (void)</code>
Multiplied	<code>void StopNM_X (void)</code>
Indexed	<code>void StopNM (CanChannelHandle channel)</code>
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>This function will disable the NM: No more state transitions will be done. The CAN Driver is online. Any re-initialization must be done by calling StartNM().</p> <p>Important note:</p> <p>The current implementation of the NM shows a special behavior after <i>StopNM()</i> and Bus Off. The state Off will be left and the module runs into Passive Mode. The Bus Off notification could not be switched off, because it is essentially required by the ECU. If the application wants to turn back to the state Off it should call <i>StopNM()</i> again at the end of <i>AppNmBusOff()</i> or <i>AppNmBusOffEnd()</i>, if Extended Callbacks are used. Do not use StopNM() in both functions !</p> <p>Otherwise the Application could not work properly and the Interaction Layer is furthermore stopped.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: task level or interrupt level ■ The application has to ensure that NmTask() is not interrupted by the function StopNM()! Otherwise internal flags are reset and further processing of NmTask() will lead to an inconsistent system. ■ Please also see: StartNM(), NmTask() 	

StartNM

Prototype	
Standard	void StartNM (void)
Multiplied	void StartNM_X (void)
Indexed	void StartNM (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
This function will enable the NM: The NM enters NORMAL mode and participates in the ring.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: task level or interrupt level ■ The application has to ensure that NmTask() is not interrupted by the function StartNM()! Otherwise internal flags are reset and further processing of NmTask() will lead to an inconsistent system. ■ Please also see: StopNM(), NmTask() 	

ReadRingData

Prototype	
Standard	StatusType ReadRingData (vuint8 *ringData)
Multiplied	StatusType ReadRingData_X (vuint8 *ringData)
Indexed	StatusType ReadRingData (CanChannelHandle channel, vuint8 *ringData)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
*ringdata	Pointer to the memory space where the ring data shall be copied to.
Return code	
StatusType	E_NotOK : received data is not copied E_OK : received data is copied
Functional Description	
The RingData or UserData will be copied from the local NM buffer to the memory space of the application. This memory space is given by *ringData. The copy procedure fails, if the ring is not stable.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: task level or within ApplNmIndRingData() ■ Please also see: ApplNmIndRingData(), TransmitRingData() 	

TransmitRingData

Prototype	
Standard	StatusType TransmitRingData (vuint8 *ringData)
Multiplied	StatusType TransmitRingData X (vuint8 *ringData)
Indexed	StatusType TransmitRingData (CanChannelHandle channel, vuint8 *ringData)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
*ringdata	Pointer to the memory space where the ring data shall be read from.
Return code	
StatusType	E_NotOK : transmission not successful E_OK : transmission successful
Functional Description	
<p>The RingData or UserData will be copied from a memory space of the application to the local transmission buffer. The application memory is given by *ringData.</p> <p>The copy procedure fails, if the token was lost.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: task level or within ApplNmIndRingData() ■ Please also see: ApplNmIndRingData(), ReadRingData() 	

10.3 Interface for the CAN Driver

The interface for the CAN driver has some limitations, especially when the NM is used together with an operating system like OSEK OS:

- ISR level for CAN TX and CAN RX must be equal
- Interrupts must be mutually
- Nested calls of this API are not allowed

NmConfirmation

Prototype	
Standard	void NmConfirmation (CanTransmitHandle tmtObject)
Multiplied	void NmConfirmation_X (CanTransmitHandle tmtObject)
Indexed	void NmConfirmation (CanTransmitHandle tmtObject)
Parameter	
tmtObject	
Return code	
-	-
Functional Description	
This is the post processing function for the NM. It is automatically called by the CAN driver after a NM-message has been transmitted.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ Call context: interrupt level (CAN Tx ISR) 	

NmPrecopy

Prototype	
Standard	vuint8 NmPrecopy (CanTransmitHandle tmtObject)
Multiplied	vuint8 NmPrecopy_X (CanTransmitHandle tmtObject)
Indexed	vuint8 NmPrecopy (CanTransmitHandle tmtObject)
Parameter	
tmtObject	
Return code	
vuint8	CopyEnable: Indicates whether the received data shall be copied by the CAN driver. Always returns FALSE, i.e. the data is not copied by the CAN driver.
Functional Description	
This function is automatically called by the CAN driver to pre-process received NM messages.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ Call context: interrupt level (CAN Rx ISR) 	

NmCanError

Prototype	
Standard	<code>void NmCanError (void)</code>
Multiplied	<code>void NmCanError_X (void)</code>
Indexed	<code>void NmCanError (CanChannelHandle channel)</code>
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>This function notifies the NM about the occurrence of a Bus Off in the CAN driver.</p> <p>The NM re-initializes the CAN controller. Pending transmit messages are removed. The CAN driver is CanOffline(). The NM calls ApplNmBusOff to notify the application about this event. The NM enters the Limphome state.</p>	
Particularities and Limitations	
<ul style="list-style-type: none">■ The NM has to be initialized correctly.■ Required functions: ApplNmBusOff■ Call context: interrupt level (CAN error ISR)	

10.4 OEM-specific API

Please refer to [TechRef_OsekNm_OEM].

10.5 Callbacks

10.5.1 Overview

Figure 10-2 gives an overview of the callbacks that occur at certain state transitions. It also indicates the order of their occurrences.

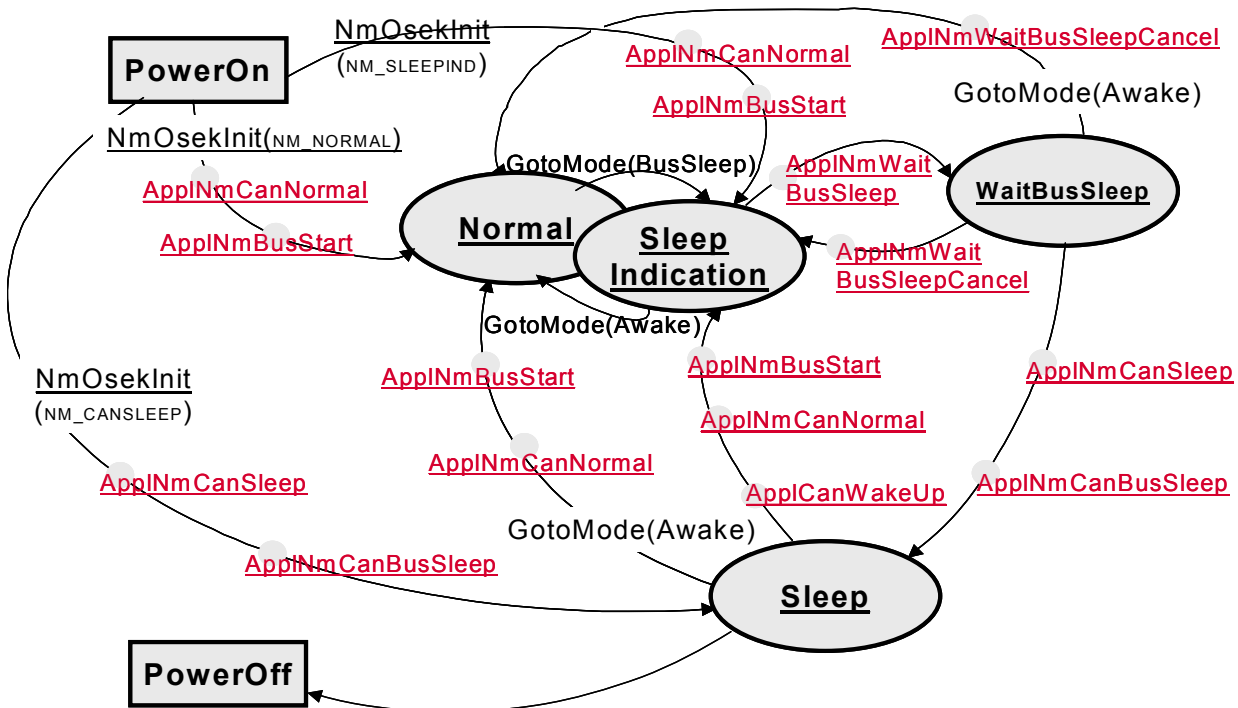


Figure 10-2 Overview of callback of the NM

10.5.2 Standard Callbacks

The NM uses a set of callback functions:

- `ApplNmBusOff()`
- `ApplNmCanBusSleep()`
- `ApplNmCanNormal()`
- `ApplNmCanSleep()`

These callback functions have to be provided by the application. They must match the required interfaces described below. This can be ensured by including the header file in the modules that provide the required callback functions. If the interfaces do not match, unexpected run-time behavior may occur.

It is not allowed to change the interrupt status within the callbacks.

ApplNmCanBusSleep

Prototype	
Standard	void ApplNmCanBusSleep (void)
Multiplied	void ApplNmCanBusSleep_X (void)
Indexed	void ApplNmCanBusSleep (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>The NM calls this functions to notify the application that the CAN bus has gone to BusSleep. After this function has been called, no cyclic NM messages should be sent.</p> <p>The application runs in local mode. As there is no need for communication at that moment, the application might disable the bus transceiver within this callback.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within the function NmTask() and NmOsekInit() while interrupts are blocked. ■ Please also see: GotoMode(BusSleep), GotoMode(Awake) 	
Example	
<pre>void ApplNmCanBusSleep() { /* switch the transceiver to SLEEP mode */ SetTrcvToSleep(); }</pre>	

ApplNmBusOff

Prototype	
Standard	void ApplNmBusOff (void)
Multiplied	void ApplNmBusOff_X (void)
Indexed	void ApplNmBusOff (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>The NM calls this function to allow the application to have a OEM-specific Bus Off handling. A Bus Off has occurred and the TX path of the CAN driver has been disabled by CanOffline(). The CAN controller has been re-initialized.</p> <p>The NM runs in LimpHome mode.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within the function NmCanError() in CAN error ISR context ■ ApplNmBusOff() and ApplNmBusOffEnd() are not symmetric. It can happen, that ApplNmBusOff() is called but ApplNmBusOffEnd() isn't, e.g. when a BusOff occurs while the NM is in state WaitBusSleep or Sleep. ■ Please also see: NmCanError() ■ ApplNmBusOff() and ApplNmBusOffEnd() are not symmetric. This is because of a special BusOff handling when the BusOff occurs while the NM is in sleep mode or about to enter sleep mode. 	
Example	
<pre>void ApplNmBusOff() { NmStatusType status; status = NmGetStatus(); if((NmStateWaitBusSleep(status) == 0x00) && (NmStateBusSleep (status) == 0x00)) { /* handle IL */ ILTxWait(); } else { /* the IL has already been stopped */ } }</pre>	

ApplNmCanNormal

Prototype	
Standard	<code>void ApplNmCanNormal (void)</code>
Multiplied	<code>void ApplNmCanNormal_X (void)</code>
Indexed	<code>void ApplNmCanNormal (CanChannelHandle channel)</code>
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>The NM calls this functions to inform the application that the NM is ready to be used. It is up to the application to</p> <ul style="list-style-type: none"> ■ switch the CAN transceiver to NORMAL mode ■ switch the CAN protocol chip in the ready state <p>The function ApplNmCanNormal() has to be provided by the application due to the fact that the hardware is not known by the NM.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within function NmTask() and NmOsekInit() while interrupts are blocked ■ Please also see: ApplNmCanSleep() 	
Example	
<pre>void ApplNmCanNormal() { /* switch the transceiver to NORMAL mode */ SetTrcvToNormal(); /* switch the CAN driver to NORMAL mode */ CanWakeUp(); /* switch IL to active mode */ IlRxStart(); IlTxStart(); }</pre>	

AppINmCanSleep

Prototype	
Standard	void AppINmCanSleep (void)
Multiplied	void AppINmCanSleep_X (void)
Indexed	void AppINmCanSleep (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>The NM calls this functions to inform the application that the CAN bus has entered its SLEEP mode.</p> <p>It is up to the application to</p> <ul style="list-style-type: none"> ■ switch the CAN protocol chip from NORMAL mode to SLEEP mode (if supported by the hardware) ■ switch the CAN-Transceiver from NORMAL Mode to STANDBY mode <p>The function AppINmCanSleep() has to be provided by the application due to the fact that the hardware not known by the NM.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within function NmTask() and NmOsekInit() while interrupts are blocked ■ Please also see: AppINmCanBusSleep() 	
Example	
please refer to the example in chapter "14 Example"	

10.5.3 Extended Callbacks

The NM supports extended callback functions in order to inform higher modules about the current state of the NM:

- ApplNmBusOffEnd()
- ApplNmBusStart()
- ApplNmWaitBusSleep ()
- ApplNmWaitBusSleepCancel()

These callbacks can e.g. inform the Vector Interaction Layer about the transition to the BusSleep state. This information can be used to start, stop or re-start the timeout monitoring for messages or signals.

The additional callbacks can be activated in the configuration tool (see chapter “4 Configuration”).

If the extended callbacks are enabled, the application has to provide them.

AppINmBusStart

Prototype	
Standard	void AppINmBusStart (void)
Multiplied	void AppINmBusStart_X (void)
Indexed	void AppINmBusStart (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>The NM calls this function to inform the application to switch into NORMAL mode and activate the communication channel.</p> <p>This function is called when switching from</p> <ul style="list-style-type: none"> ■ PowerOn to Normal ■ PowerOn to SleepIndication ■ Sleep to Normal ■ Sleep to SleepIndication <p>Note: This function is called together with function AppINmCanNormal(), i.e. both callbacks occur in the same context of the NM.</p> <p>The purpose of this second callback is, that the application can perform additional actions when bus communication is requested. However, transceiver handling has to be done inside AppINmCanNormal().</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within the function NmTask() and NmOsekInit() while interrupts are blocked ■ Have to be provided only if extended callbacks are enabled. ■ Please also see: AppINmCanSleep(), AppINmCanBusSleep() 	
Example	
<pre>#if defined(NM_ENABLE_EXTENDED_CALLBACK) void AppINmBusStart() { } #endif</pre>	

AppINmWaitBusSleep

Prototype	
Standard	void AppINmWaitBusSleep (void)
Multiplied	void AppINmWaitBusSleep_X (void)
Indexed	void AppINmWaitBusSleep (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
<p>The NM calls this function to inform the application about the internal switch to state WaitBusSleep.</p> <p>Mode Sleep will be entered as soon as the NM timer $T_{\text{WaitBusSleep}}$ has elapsed.</p>	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within the function NmTask() while interrupts are blocked ■ Has to be provided only if extended callbacks are enabled. ■ Please also see: AppINmWaitBusSleepCancel() 	
Example	
<pre>#if defined(NM_ENABLE_EXTENDED_CALLBACK) void AppINmWaitBusSleep() { /* handle the IL */ IlRxStop(); IlTxStop(); } #endif</pre>	

AppNmWaitBusSleepCancel

Prototype	
Standard	void AppNmWaitBusSleepCancel (void)
Multiplied	void AppNmWaitBusSleepCancel_X (void)
Indexed	void AppNmWaitBusSleepCancel (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
The NM calls this function to inform the application that the state NmWaitBusSleep has been left because communication has been started again..	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within the function NmTask() while interrupts are blocked ■ Have to be provided only if extended callbacks are enabled. ■ Please also see: AppNmWaitBusSleep() 	
Example	
<pre>#if defined(NM_ENABLE_EXTENDED_CALLBACK) void AppNmWaitBusSleepCancel () { /* handle the IL */ IlRxStart(); IlTxStart(); } #endif</pre>	

ApplNmBusOffEnd

Prototype	
Standard	void ApplNmBusOffEnd (void)
Multiplied	void ApplNmBusOffEnd_X (void)
Indexed	void ApplNmBusOffEnd (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
The NM calls this function to inform the application that state Bus Off is left.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within the function NmTask() while interrupts are blocked ■ Have to be provided only if extended callbacks are enabled. ■ ApplNmBusOff() and ApplNmBusOffEnd() are not symmetric. It can happen, that ApplNmBusOff() is called but ApplNmBusOffEnd() isn't, e.g. when a BusOff occurs while the NM is in state WaitBusSleep or Sleep. ■ Please also see: ApplNmBusOff() 	
Example	
<pre>#if defined(NM_ENABLE_EXTENDED_CALLBACK) void ApplNmBusOffEnd() { /* handle the IL */ IlTxRelease(); } #endif</pre>	

10.5.4 Special-feature callbacks

The special-feature callbacks can be activated in the configuration tool (see chapter “4 Configuration”). If the corresponding features are enabled, the application has to provide the assigned callbacks as well.

AppNmIndRingData

Prototype	
Standard	<code>void AppNmIndRingData (void)</code>
Multiplied	<code>void AppNmIndRingData_X (void)</code>
Indexed	<code>void AppNmIndRingData (CanChannelHandle channel)</code>
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	-
Functional Description	
The NM calls this function to inform the application that the RingData or UserData are valid. The application can then copy and transmit the RingData, until T_{typ} has expired.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. ■ Call context: Called within the function NmTask() while interrupts are blocked. ■ Has to be provided if the NM is configured to support access on ring data. ■ Please also see: ReadRingData() 	

AppNmSendSleepAck

Prototype	
Standard	<code>vuint8 AppNmSendSleepAck (void)</code>
Multiplied	<code>vuint8 AppNmSendSleepAck_X (void)</code>
Indexed	<code>vuint8 AppNmSendSleepAck (CanChannelHandle channel)</code>
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	Indicates if the SleepAcknowledge flag should be sent or not.
Functional Description	
This callback allows the application to decide if the SleepAcknowledge flag should be sent or not. Depending on the return value of the callback, the NM sets the SleepAcknowledge flag to '0' or '1'.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. 	

AppINmLimpHomeEnter

Prototype	
Standard	void AppINmLimpHomeEnter (void)
Multiplied	void AppINmLimpHomeEnter_X (void)
Indexed	void AppINmLimpHomeEnter (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	
Functional Description	
This callback informs the application that the NM has entered state LimpHome. Refer to chapter “6.8.4 LimpHome Notification”.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. 	

AppINmLimpHomeExit

Prototype	
Standard	void AppINmLimpHomeExit (void)
Multiplied	void AppINmLimpHomeExit_X (void)
Indexed	void AppINmLimpHomeExit (CanChannelHandle channel)
Parameter	
channel	Represents the handle of the assigned CAN channel (see chapter 10.1).
Return code	
-	
Functional Description	
This callback informs the application that the NM has left state LimpHome. Refer to chapter “6.8.4 LimpHome Notification”.	
Particularities and Limitations	
<ul style="list-style-type: none"> ■ The NM has to be initialized correctly. 	

10.5.5 OEM-specific Callbacks

Please refer to the OEM-specific documentation [TechRef_OsekNm_OEM].

10.6 Other Interfaces

10.6.1 Version of the Source Code

The version of the source code of the NM is stored in three BCD-coded constants within the NM source file:

```
V_MEMROM0 V_MEMROM1 vuint8 V_MEMROM2 kNmMainVersion    =
    (vuint8)(NM_DIROSEK_VERSION >> 8);
V_MEMROM0 V_MEMROM1 vuint8 V_MEMROM2 kNmSubVersion      =
    (vuint8)(NM_DIROSEK_VERSION & 0xFF);
V_MEMROM0 V_MEMROM1 vuint8 V_MEMROM2 kNmReleaseVersion =
    (vuint8)(NM_DIROSEK_RELEASE_VERSION);
```

Example - Version 3.31.00 is registered as:

```
kNmMainVersion    = 0x03;
kNmSubVersion     = 0x31;
kNmReleaseVersion = 0x00;
```

This information can be accessed by the application at any time.

10.6.2 Parameters and Configuration

Table 10-1 gives parameters that are needed for the NM to run properly.

Name	Description
kNmCanPara	This parameter identifies the handle of the parameter set that shall be used to initialize the CAN controller. Its value is set automatically by the configuration tool.
kNmEcuNr	This parameter contains the NM station number of the ECU. Its value is set automatically by the configuration tool, based on information from the network database. Please also see chapter “7.1 Multiple ECUs”.
kNmTxObj	This parameter identifies the handle for the transmit object that is used to send the NM messages. Its value is set automatically by the configuration tool based on information from the network database. Please also see chapter “7.1 Multiple ECUs”.
TargetConfigTable	This table stores target configuration of the NM. The table has to be provided by the application. It is used by the API CmpConfig(). The contents of this table is not modified by the NM. Please note that this table depends on the OEM. Please refer to the additional OEM-specific documentation [TechRef_OsekNm_OEM]. Please also see chapter “6.4”.
ConfigMaskTable	This table stores the masks for the target configuration of the NM. The table has to be provided by the application. It is used by the API CmpConfig(). The contents of this table is not modified by the NM. Please note that this table depends on the OEM. Please refer to the additional OEM-specific documentation [TechRef_OsekNm_OEM]. Please also see chapter “6.4”.

Table 10-1 Parameter interface

11 Working with the Code

11.1 Version Changes

Changes of the NM are listed in the history section at the beginning of the header and source code files.

11.2 Application Interface

The application uses the API of the NM. The necessary interfaces can be looked up in the NM header file. This file contains

- preprocessor definitions
- type definitions
- prototypes for API functions
- prototypes for necessary callback function

12 CANdb attributes

The configuration tool needs additional information to handle the configuration options for the NM. These information are stored in attributes within the CAN database (DBC-file). An attribute can belong to the database, to a node, to a message or to a signal.

Please refer to the online help system of the CANdb editor [[OnlineHelp_CANdb]] to learn how to define and set these attributes.

There are different types of attributes: String, Hex, Integer-Values or Enumeration. When using enumerations, please take care of the order ("No", "Yes").

These attributes are normally provided by the OEM. The user does not have to change them.

Table 12-1 gives an overview of the general attributes needed for the NM. There are also some OEM-specific attributes. Please refer to the additional OEM documentation for more information on typical values [TechRef_OsekNm_OEM].

Attribute Name	Valid for	Type	Value	Description	Used by
NmType	Database	String	depends on OEM	This attribute defines the OEM-specific type of the NM.	GenTool
NmBaseAddress (old: NWM-Basisadresse)	Database	Hex	Range: 0x000... 0x7FF Default: see OEM	<p>This attribute defines the base address of the NM messages (e.g. 0x400).</p> <p>Only a certain number of nodes can participate in the NM. The CAN identifiers of NM messages are kept in a certain range. This range starts with the ID given by attribute NmBaseAddress. The size of the range is given by attribute NmMessageCount.</p> <p>There is a general requirement that the value of attribute NmBaseAddress must be chosen in a way that masking is possible. This can be achieved if NmBaseAddress is an integer multiple of the number of nodes (see NmMessageCount).</p> <p>Example: The NM uses CAN message identifiers 0x400-0x43F. => NmBaseAddress=0x400 => NmMessageCount=64 (=0x3F) Note: 0x400 (=1024) is an integer multiple of 64.</p>	GenTool, CANoe


NmMessageCount (old: NWMMessageCount)	Database	Integer	range: see OEM (1..256) Default: see OEM	This attribute defines the maximum number of nodes within the NM. The value is required to define the range precopy-function of the CAN driver's precopy function that is used to receive the NM messages. There is the requirement that the value of attribute NmMessageCount is 2^n (where n is a natural number). Example: see NmBaseAddress	GenTool, CANoe
NmNode (old: NWM-Knoten)	Node	Enumeration	"No", "Yes"	This attribute defines if the corresponding node uses the NM ("Yes") or not ("No").	GenTool
NmStationAddress (old: NWM-Stationsadresse)	Node	Hex	range: 0..n where n= NmMessageCount -1	<p>This attribute defines the station address of the ECU. The value correlates with the identifier of the NM message of this node: NmStationAddress + NmBaseAddress = CANID(NmNode) Example: NmBaseAddress=0x400, NmStationAddress=0x12 => CANID(NmNode)=0x412</p> <div>  <p>Info The configuration tools do not refer to this value when accessing the node address. Instead they use the CAN identifier of the assigned NM message and calculate the node address with the help of the NM base address.</p> </div>	CANoe
NmMessage (old: NWM-Botschaft)	Message	Enumeration	"No", "Yes"	This attribute defines if the corresponding message is a NM message ("Yes") or not ("No").	GenTool

Table 12-1 NM attributes in CANdb

Column "Used by" indicates which attributes are evaluated by the GenTool (CANgen or GENy) or by the CANoe node layer DLL.

13 FAQ

Q: Which specifications are covered by Nm_DirOsek

A: V2.5.3

Q: Are there any deviations from the OSEK specification?

A: There are no deviations, only orthogonal extensions.

14 Example



Example

```
#include "can_inc.h"
#include "node.h"
#include "nm_osek.h"

/* Your generated header */
/* Name of header file depends on OEM */

void SetTrcvToNormal(void)
{
    /* Switch transceiver to NORMAL mode */
    NOT_STANDBY = 1;
    ENABLE = 1;
    /* Stand-By pin of transceiver */
    /* Enable pin of transceiver */
}

void SetTrcvToSleep(void)
{
    /* Switch transceiver to SLEEP mode */
    ENABLE = 0;
    NOT_STANDBY = 0;
}

void ApplCanWakeUp(void)
{
    /* called by CAN-Driver */
    NmOsekInit(NM_SLEEPIND);
}

void ApplNmBusOff(void)
{
    /* called by Network Management */
    /* NM informs Application about a Bus Off */
}

void ApplNmCanNormal(void)
{
    /* called by Network Management */
    SetTrcvToNormal();
    CanWakeUp();
}

void ApplNmCanSleep(void)
{
    /* called by Network Management */
    /* Switch CAN Controller to sleep mode */
    vuInt8 returnCode;
    returnCode = CanSleep();
    if( returnCode != kCanOk)
    {
        noSleep = 1; /*global variable*/
    }
    else
    {
        noSleep = 0; /*global variable*/
        /* Switch CAN Controller to standby mode */
        SetTrcvToStandby();
    }
    return;
}

void ApplNmCanBusSleep(void)
{
    /* called by Network Management */
    vuInt8 waitCount;
    #define MAX_WAIT_LOOP 40
    /* adapt this value to processor speed */

    if( noSleep == 1 )
    {
        noSleep = 0; /*global variable*/
        GotoMode( Awake );
    }
    else
    {
        /* Switch CAN Controller to sleep mode */
    }
}
```

```

    ENABLE = 1;
    for (waitCount = 0; waitCount < MAX_WAIT_LOOP; waitCount++)
    {
        ; /* make sure this loop is not removed by
compiler */
    }
    ENABLE = 0;
}

#if defined ( NM_RINGDATA_ACCESS_USED )
void ApplNmIndRingData(void)
{
    vuint8 RingData[6];

    #if defined ( NM_COPY_RINGDATA_USED )
    if(ReadRingData(RingData) == E_OK)
    {
        /* Ring Data test, clear received pattern */
        RingData[0] = 0xFF;
        RingData[1] = 0xFF;
        RingData[2] = 0xFF;
        RingData[3] = 0xFF;
        RingData[4] = 0xFF;
        RingData[5] = 0xFF;

        (void)TransmitRingData(RingData);
    }
    #else
    /* Only transmission possible, no interpretation of the Ring Data required */
    RingData[0] = 0xFF;
    RingData[1] = 0xFF;
    RingData[2] = 0xFF;
    RingData[3] = 0xFF;
    RingData[4] = 0xFF;
    RingData[5] = 0xFF;

    (void)TransmitRingData(RingData);
    #endif /* NM_COPY_RINGDATA_USED */
}
#endif /* NM_RINGDATA_ACCESS_USED */

main()
{
    DisableInterrupts();
    SetTrcvToNormal();
    CanInitPowerOn(0); /* Initialize CAN-Driver */
    NmOsekInit(NM_NORMAL); /* Initialize NM in normal mode */
    EnableInterrupts();

    for (;;) /* Main loop */
    {
        if (Flag10ms) /* depends on OEM */
        {
            NmTask();
            Flag10ms = 0;
        }
    }
}

.

```

15 Contact

15.1 Related Products

- Vector offers the CANbedded component “Communication Control Layer (CCL)”. This component handles the NM and provides a more generic interface to the application.
- Vector offers OEM-specific DLLs for CANoe. These DLLs can be used for simulated network nodes running a NM.

15.2 Additional Information

- Vector offers training classes for CANbedded components. These training classes include a presentation and introduction of the NM.

Visit our website for more information on

- > News
- > Products
- > Demo software
- > Support
- > Training data
- > Addresses

www.vector-informatik.com