rexroth
A Bosch Company

NYCe 4000

Linear motion system
LMS User Manual

Reference
R911425761

Edition 02

Title

NYCe 4000
Linear motion system
LMS User Manual

Type of Documentation

Reference

Document Typecode

DOK-NY4000-LMS*USR*V54-RE02-EN-E

Internal File Reference

RS-3d0cbbb67fcab9bc0a347e866c1e72c7-2-en-US-10

Record of Revision

Edition	Release Date	Notes
DOK-NY4000-LMS*USR*V54-RE01-EN-E	08/2024	First edition 54V02
DOK-NY4000-LMS*USR*V54-RE02-EN-E	04/2025	2nd edition 54V04

List of trademarks

Products in this publication are referred to by their general trade names. In most, if not all cases, these designations are claimed as trademarks or registered trademarks of their respective companies.

Liability

The specified data is intended for product description purposes only and shall not be deemed to be a guaranteed characteristic unless expressly stipulated in the contract. All rights are reserved with respect to the content of this documentation and the availability of the product.

Published by

Bosch Rexroth AG

P.O. Box 7170 ■ 5605 JD Eindhoven ■ The Netherlands

http://www.boschrexroth.com

Table of Contents

Page

1 NYCe 4000 Linear Motion System ... 1
1.1 Introduction ... 1
1.2 Used abbreviations in this manual ... 1

2 System description ... 3
2.1 Introduction ... 3
2.2 LMS machine hardware ... 3
2.2.1 Introduction ... 3
2.2.2 Tracks ... 3
2.2.3 Carriers ... 4
2.2.4 Coils ... 4
2.2.5 LMS sensors ... 4
2.2.6 Configurations ... 5
2.2.7 Movement decomposition ... 6
2.3 LMS control hardware ... 6
2.3.1 Introduction ... 6
2.3.2 NY4074-based system ... 7
2.3.3 NY4079-based system ... 8
2.3.4 NY4089-based system ... 10
2.3.5 Node – coil connection ... 11
2.3.6 Node – LMS sensor connection ... 11
2.4 LMS software ... 16
2.4.1 Introduction ... 16
2.4.2 Application on a PC ... 16
2.4.3 Application for a PLC ... 17
2.4.4 CM service ... 18
2.4.5 Data retrieval guidelines for applications ... 19
2.4.6 LMS tools ... 19
2.5 Design guidelines for the NYCe 4000 LMS ... 19

3 Configuration ... 25
3.1 Introduction ... 25
3.2 Parameters in the CM configuration file ... 25
3.2.1 Introduction ... 25
3.2.2 System-wide parameters ... 25
3.2.3 Track parameters ... 25
3.2.4 Carrier parameters ... 29
3.2.5 Physical coil parameters ... 32
3.2.6 Sensor parameters ... 45

4 CM - Carrier Management ... 49
4.1 Introduction ... 49
4.2 System state ... 49

Page

4.3 Client state ... 50
4.3.1 Introduction ... 50
4.3.2 Application start ... 51
4.3.3 Application stop ... 51
4.4 Track states ... 52
4.5 Carrier states ... 54
4.6 Carrier controller states ... 57
4.7 Carrier Management functions ... 58
4.7.1 Introduction ... 58
4.7.2 Connect and disconnect an application ... 61
4.7.3 Initializing Carrier Management ... 62
4.7.4 Activating a track ... 62
4.7.5 Deactivating a track ... 63
4.7.6 Set track to idle state ... 63
4.7.7 Runtime Track Layout Changes ... 63
4.7.8 Homing a track ... 67
4.7.9 Get number of carriers and IDs ... 80
4.7.10 Convert carrier ID / carrier index ... 80
4.7.11 Get number of tracks ... 81
4.7.12 Get axis identifier of a coil ... 81
4.7.13 Detect carrier presence in an area, create an area without carriers ... 82
4.7.14 Carrier Position Adjustment ... 85
4.7.15 Assigning a carrier ID ... 87
4.7.16 Change the ID of a carrier ... 88
4.7.17 Read and write carrier parameters ... 88
4.7.18 Moving carriers against an endstop ... 89
4.7.19 Moving a carrier to a position ... 89
4.7.20 Moving a carrier adjacent to its predecessor ... 92
4.7.21 Moving a carrier along a profile ... 93
4.7.22 Moving a carrier at constant velocity ... 94
4.7.23 Stopping a carrier ... 94
4.7.24 Synchronized start of multiple carriers ... 95
4.7.25 Using feed override ... 96
4.7.26 Removing / Adding a carrier ... 97
4.7.27 Move a carrier from track to track ... 99
4.7.28 Removing / Adding / Cancelling a moving carrier from / to an LMS ... 105
4.7.29 Moving a carrier across tracks ... 108
4.7.30 Moving a carrier with collision avoidance ... 112
4.7.31 Camming a carrier on a track ... 126
4.7.32 Markers ... 131
4.7.33 Reading carrier data ... 133
4.7.34 Resetting errors ... 134
4.7.35 Clearing warnings ... 135
4.7.36 Handling special controller parameters for an area ... 135
4.7.37 Controlling carriers in velocity mode ... 136
4.7.38 Enabling / Disabling sensor detection ... 138

Page

4.7.39 Enabling / Disabling start triggers ... 139
4.7.40 Synchronize on an asynchonous operation ... 139
4.7.41 Event handling ... 142
4.7.42 Reading a variable ... 146
4.7.43 Reading configuration parameters ... 148
4.7.44 Conversion functions ... 150
4.7.45 Logging ... 151
4.7.46 Shutting down the LMS ... 152
4.8 Tuning of a coil axis ... 152

5 Error handling and recovery ... 155
5.1 Introduction ... 155
5.2 Synchronous errors and warnings ... 155
5.3 Asynchronous errors ... 167
5.4 Error handling at coil level ... 170
5.5 Error handling at CM level ... 172
5.6 Error recovery ... 174
5.6.1 Default recovery ... 174
5.6.2 Subsystem error recovery ... 174
5.6.3 Specific recovery ... 174

1 NYCe 4000 Linear Motion System

1.1 Introduction

This manual describes the installation, configuration and operation of the Rexroth NYCe 4000 Linear Motion System (LMS). This motion system is an extension to the standard NYCe 4000 system. It provides functions for linear motion purposes.

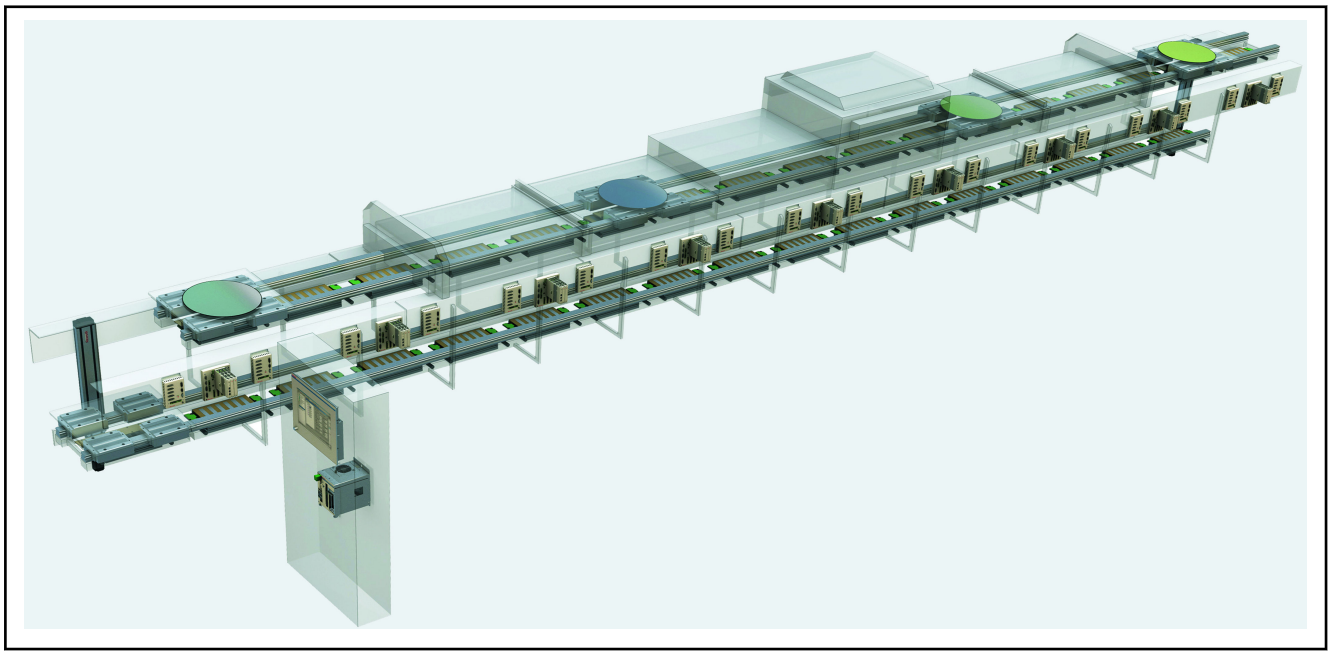


Fig. 1-1: Schematic presentation of a Linear Motion System (LMS)

An LMS consists of one or more tracks. Each track consists of coils and position sensors. Carriers are placed on the track and each carrier has a magnet plate. Under control of NYCe 4000 LMS hardware, coils generate magnetic fields to move every carrier to the commanded position.

LMS sensor

This manual uses the term "LMS sensor" for the sensor that is used in an LMS to detect the position of a carrier. The LMS sensor can be based on two different technologies. One technology uses two Hall sensor elements, the other technology uses two magneto-resistive sensor elements. The term "LMS sensor" is used when a general reference can be made, "Hall sensor" and "MR sensor" is used when a specific reference must be used.

1.2 Used abbreviations in this manual

The following table gives a summary of abbreviations used in this manual.

Abbreviation	Description
API	Application Programming Interface
CM	Carrier Management
DriveLink	SERCOS III connection based drives (IndraDrive and ctrlX DRIVE)
Hall sensor	A sensor based on two Hall elements

Abbreviation	Description
IEEE	Institute of Electrical and Electronics Engineers
LMS	Linear Motion System
MCU	Motion Control Unit
MR sensor	A sensor based on two magneto-resistive (MR) elements
MUX	Multiplexer
NY4114	MCU with 3 IEEE802.3 (ethernet) connections
NY4120/10	PWM Drive Module for LMS
NY4150/10	SERCOS III Master Module
PLC	Programmable Logic Controller
PTC	Positive Temperature Coefficient (sensor)
RTLC	Runtime Track Layout Changes
SAC	Single Axis Control (subsystem)
SERCOS III	Third generation SERCOS (Serial Real time Communication System) drive communication protocol based on the Ethernet physical hardware architecture
XML	Extensible Markup Language

Tab. 1-1: Summary of used abbreviations

2 System description

2.1 Introduction

This chapter gives an overview of the Linear Motion System (LMS). The LMS basically consists of three parts.

- machine hardware - the parts that physically perform the movements, such as tracks with carriers, coils and LMS sensors (chapter 2.2 "LMS machine hardware" on page 3)
- control hardware - the NYCe 4000 components that control the movements (chapter 2.3 "LMS control hardware" on page 6)
- software - the software interface for tools and applications that command the LMS (chapter 2.4 "LMS software" on page 16)

See the NYCe 4000 LMS Design Guide for detailed information of the machine hardware and control hardware.

Drawings in this manual are for explanation purposes only, and are not drawn to an accurate scale.

2.2 LMS machine hardware

2.2.1 Introduction

The machine hardware is based on an inverse linear motor principle, and basically consists of the following parts (see fig. 2-1 "Coil, carrier, magnets and sensors" on page 3 for a typical layout).

- One or more tracks
- One or more carriers with a magnet plate that move along the track
- Coils, which drive the carriers
- LMS sensors, which produce carrier position information

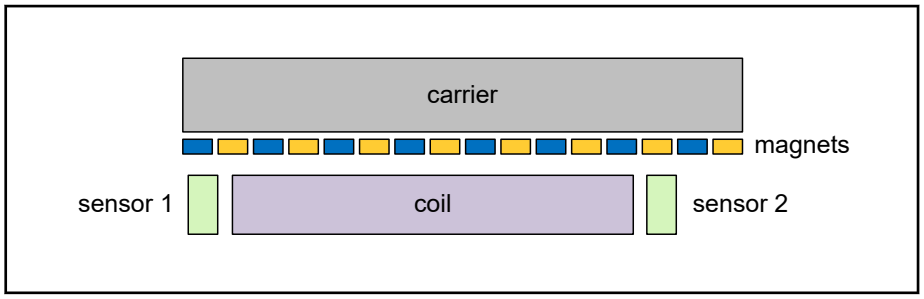


Fig. 2-1: Coil, carrier, magnets and sensors

2.2.2 Tracks

The type of a track can be a linear track or an oval track.

The carrier (or more carriers) can move in both directions on a linear track till the end stop position, see fig. 2-2 "Example of a linear track with end stops" on page 4.

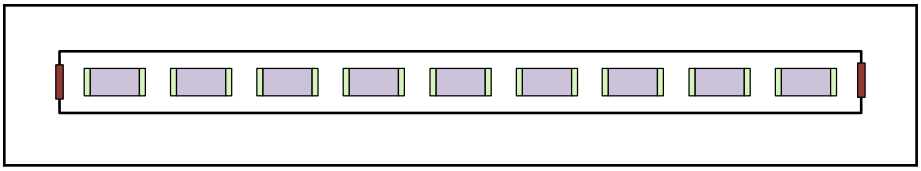


Fig. 2-2: Example of a linear track with end stops

The oval track layout enables a cyclic movement of a carrier (or more carriers), see fig. 2-3 "Example of an oval track" on page 4. As an oval track layout has no start and end position, the track is called a modulo track.

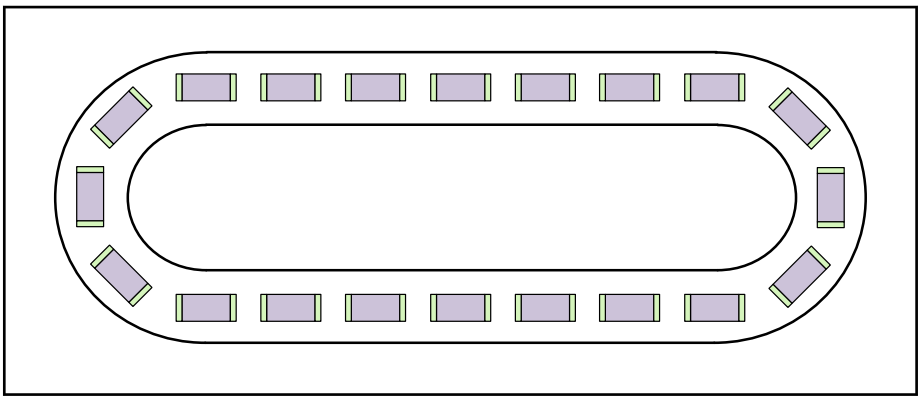


Fig. 2-3: Example of an oval track

2.2.3 Carriers

The general function of the LMS is to move product carriers from one position to another position on a track or to another track. The carriers have magnets, for example, at the bottom or on a side (alternating north and south poles). The movement of a carrier is established by controlling the coils, which are positioned in proximity of the magnets. As mentioned, this is an inverse linear motor principle.

If the LMS uses MR sensors, the carrier also has a magnetic scale.

2.2.4 Coils

The coils can be controlled by CM independently of each other. This implies that multiple carriers can be moved along the same track, as long as the required coil currents for movements of different carriers are not conflicting. In practice, this means that either each coil only moves one carrier, or it moves two carriers at the same velocity with a mutual distance which equals the distance between two consecutive pairs of magnets. The control signal is based on the position of one of the two carriers only, which implies that a second carrier above the same coil moves along with the controlled carrier.

Coils incorporate a PTC sensor to detect a too high temperature of the coil.

2.2.5 LMS sensors

The position of a carrier is sensed locally by sensors. Each sensor consists of a pair of Hall sensor elements or a pair of MR elements. The pair of sensors gives (by approximation) a sine and cosine signal. Using interpolation and correction a sufficiently accurate position is obtained. The Hall sensors use the magnetic field of the magnet plates of the carrier. The MR sensors use

the magnetic field of a magnetic scale mounted on the carrier. The LMS sensor signals are also used for commutation.

2.2.6 Configurations

Two configurations are supported concerning the physical location of the pair of LMS sensors and the coils.

- Configuration 1. LMS sensor pair - coil - LMS sensor pair

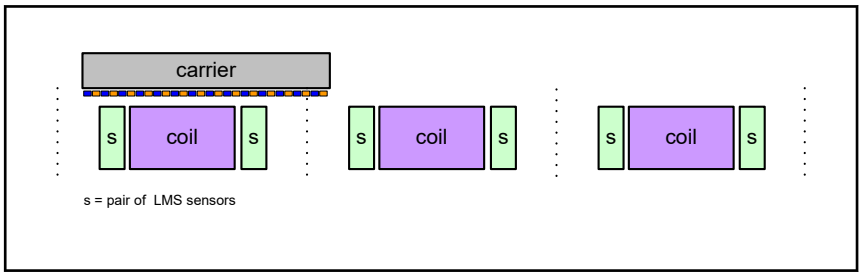


Fig. 2-4: LMS sensor configuration 1

A pair of LMS sensors is attached at both sides of the coil.

The LMS sensor only gives a valid signal when the magnetic field of a carrier is sufficiently close to the sensor. As the carrier length exceeds the distance between the outer sensors, at least one of the sensor pairs always gives valid position/commutation data for the relevant coil. The area in which the first or second sensor is valid (non-exclusive "or") is referred to as the sensor range, see fig. 2-6 "Sensor and control range" on page 6.

- Configuration 2. LMS sensor pair – coil with LMS sensor pair – LMS sensor pair

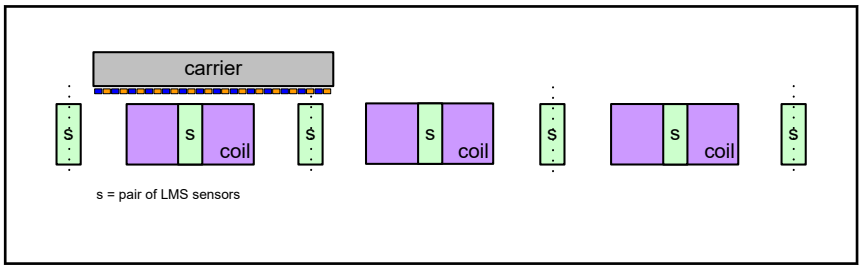


Fig. 2-5: LMS sensor configuration 2

In this configuration, a coil consists of

– Shared LMS sensor pair (left)
– Coil with LMS sensor pair
– Shared LMS sensor pair (right)

The pairs of LMS sensors at both sides of the coil are shared with the neighboring coil. As the carrier is always detected by at least one sensor, at least one of the sensors gives valid position/commutation data for the relevant coil. The area in which the first, second or third sensor is valid (non-exclusive "or") is referred to as the sensor range, see fig. 2-6 "Sensor and control range" on page 6.

2.2.7 Movement decomposition

The coils are installed on the track at a mutual distance of the length of the magnet plate, by approximation. This makes it possible to realize carrier movements along the track while maintaining a constant force. To this end, the movement of a carrier is decomposed in a number of movements "above" each of the coils.

The setpoint movement at coil level therefore may have an initial and final velocity not equal to 0. The carrier is controlled as soon as the magnet plate enters the coil control range (this is the position range in which the currents in the coils affect the behavior of the carrier). This is when the front of the magnet plate is just above the coil until the end of the magnet plate is just about to leave the coil. See fig. 2-6 "Sensor and control range" on page 6 for a visualization of sensor range (in blue) and coil control range (in red). The position of a carrier is defined as the position of the center of the magnet plate. Note that the coil control range is entirely within the sensor range.

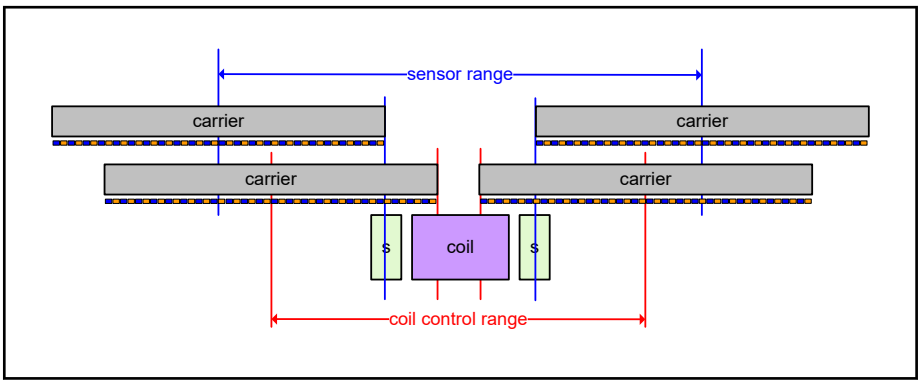


Fig. 2-6: Sensor and control range

2.3 LMS control hardware

2.3.1 Introduction

The LMS machine hardware is controlled by NYCe 4000 hardware. The NYCe 4000 (electrical) hardware consists of the following components.

- One or more of the following NYCe 4000 nodes (any mix is possible)
  - NY4074 - LMS 2 Drive Slot System Housing

    The system housing has one motion control unit (MCU) and one or two NY4120/10 drive modules.
  - NY4079 - LMS System Housing for SERCOS III

    The system has one motion control unit (MCU) and one NY4150/10 SERCOS III Master module.
  - NY4089 - LMS System Housing for 8 SERCOS axes

    The system has one motion control unit (MCU) and one NY4150/10 SERCOS III Master module. Sensors are connected to the NY4961 Sensor box.
  - NY4013, NY4023, NY4033, NY4043, and NY4053 system housings for STD and DriveLink drive axes. Every system housing has one motion control unit (MCU).
- NY4981/00 - MR sensor modules
- NY4980/00 and NY4980/10 - Hall sensor modules

The sensors can be connected to any sensor input on the standard housings (NY4074, NY4079), or the NY4961 Sensor box. However, for reasons of clarity and easy service, it is advised to follow a modular concept where possible.

- Coils
- Magnetic scale (for MR sensors only)
- Power supplies

☞ NYCe 4000 LMS uses a specific drive module, NY4120/10. Make sure you use the correct drive module type.

☞
- NYCe 4000 STD supports the IndraDrive Cs, IndraDrive C, and ctrlX DRIVE for NYCe 4000 axes.
- NYCe 4000 LMS only supports the IndraDrive Cs and ctrlX DRIVE for LMS axes.

See the NYCe 4000 Release Bulletin for the supported versions of the DriveLink drive types.

☞ You can have a mix of LMS drives (NY4120/10) and DriveLink drives (controlled via NY4150/10) in one system, however, a mix of LMS drives and DriveLink drives in a single track or single node is not supported.

Note that carriers may not be synchronized in a system with a mix of drive types, or different node frequencies in a multi-node network.

See NYCe 4000 Hardware System Manual for a detailed description of the MCU and drive modules.

See NYCe 4000 Standard Housings & Accessories manual for a detailed description of the system housings, Hall sensors modules, MR sensor modules, magnetic scales, and coils.

2.3.2 NY4074-based system

The LMS control based on the NY4074 system housing consists of one MCU and one or two NY4120/10 drive modules. The MCU is the NY4114. The NY4120/10 drive module can be used with maximum 72V DC nominal servo drive voltage and a continuous current of $7A_{rms}$ per coil, and maximum peak current of $14A_{rms}$ per drive module. These restrictions limit the maximum obtainable acceleration and velocity of the carrier(s). If the application requires higher accelerations and velocities, you must choose the DriveLink drive solution supported with the NY4079 system housing.

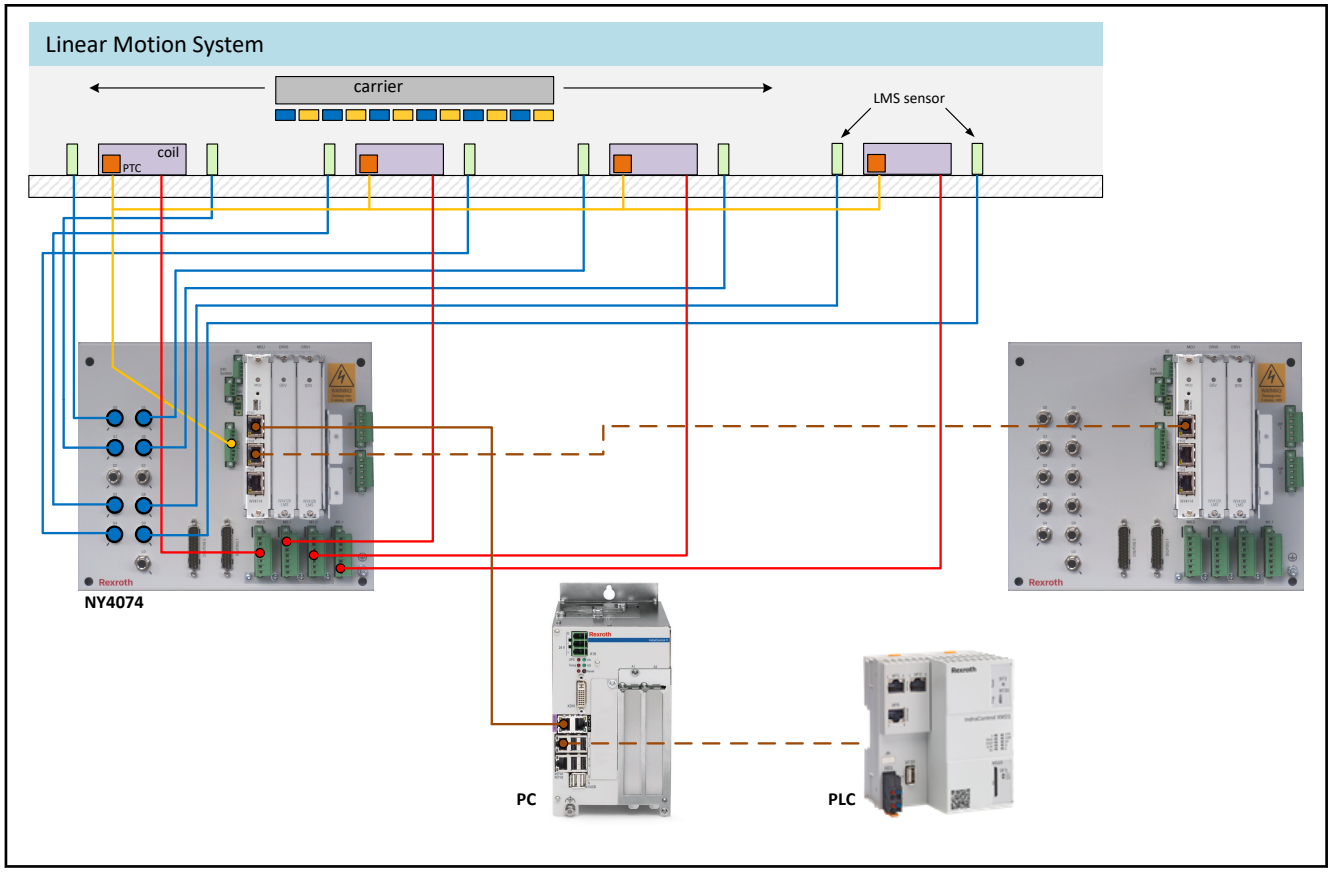


Fig. 2-7: NY4074 system overview with integrated drive modules

fig. 2-7 "NY4074 system overview with integrated drive modules" on page 8 shows the electric wiring of 4 coils with their PTC and the associated sensors to the NY4074 system housing. Further, the connection with a PC and optional a PLC is shown. The NY4074 contains one MCU (NY4114) and one or two NY4120/10 LMS drive modules. The system housing with an MCU and drive modules is called a (NYCe 4000) node. With this configuration you can connect up to 4 coils. The NY4074 has 10 analog inputs to connect LMS sensors of the track. The node is connected to a PC with a standard ethernet cable to one of the ports on the MCU. The PC can be a standard industrial PC or an INDRAMOTION MLC VP SET. The INDRAMOTION is a PC with an MLC incorporated.

2.3.3 NY4079-based system

The LMS control based on the NY4079 system housing consists of one MCU and one NY4150/10 SERCOS Master module. The MCU is the NY4114. The NY4150/10 connects the DriveLink drives via the SERCOS network. The DriveLink drives are available for higher voltage and higher current ranges. The coil and PTC are connected to the DriveLink drive.

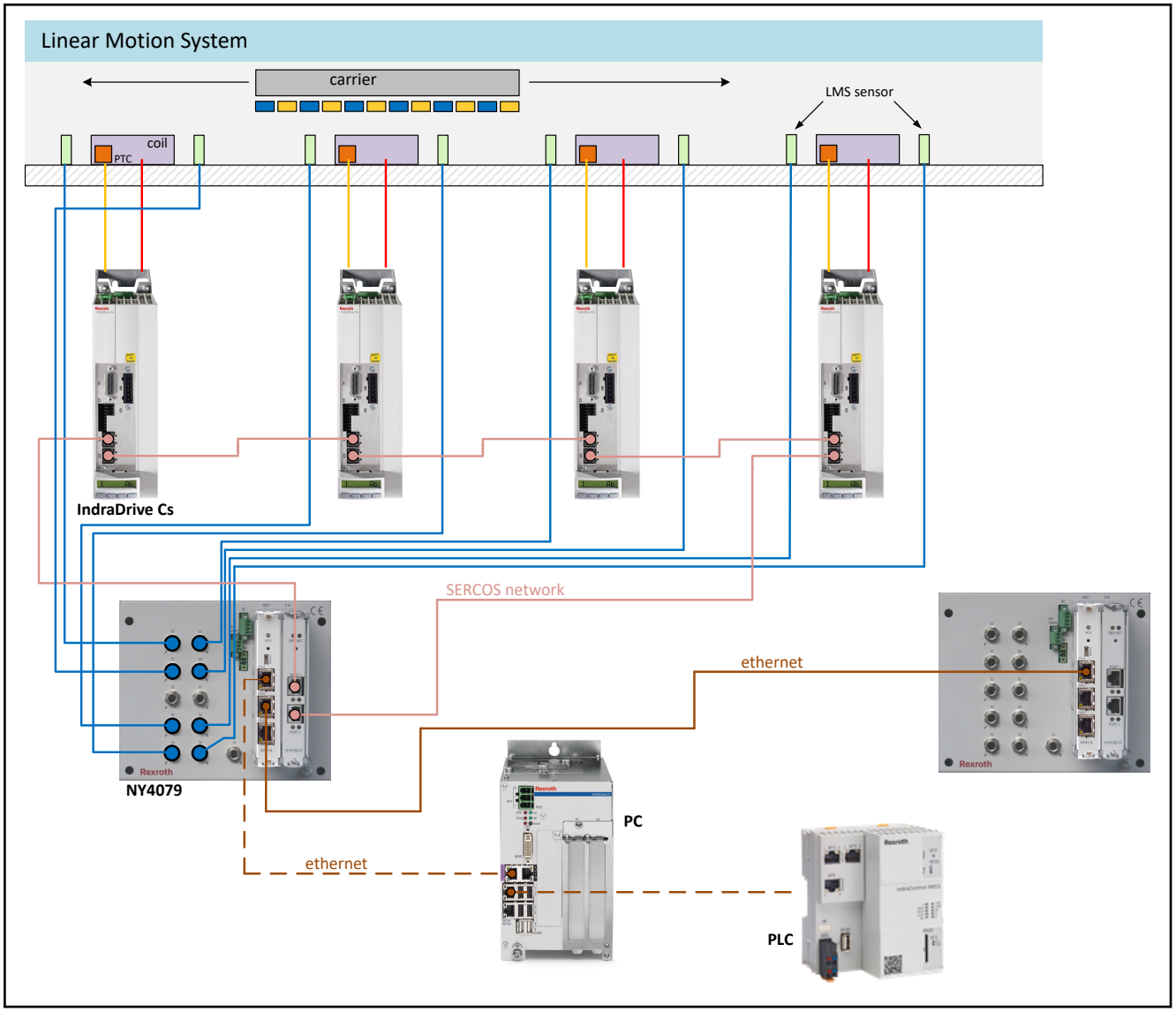


Fig. 2-8: NY4079 system overview with SERCOS network and DriveLink drives

fig. 2-8 "NY4079 system overview with SERCOS network and DriveLink drives" on page 9 shows the electric wiring of 4 coils with their PTC and the associated sensors to the DriveLink drives and the NY4079 system housing. Further, the connection with a PC and optional a PLC is shown. The NY4079 contains one MCU (NY4114) and one NY4150/10 SERCOS Master module. The system housing with an MCU and SERCOS Master module is called a (NYCe 4000) node. With this configuration you can connect up to 4 DriveLink drives. The NY4079 has 10 analog inputs to connect LMS sensors of the track. The node is connected to a PC with a standard ethernet cable to one of the ports on the MCU. The PC can be a standard industrial PC or an INDRAMOTION MLC VP SET. The INDRAMOTION is a PC with an MLC incorporated. The DriveLink drives are connected in a closed loop SERCOS network to the NY4150/10 SERCOS Master module.

The required software to commission the LMS and all (carrier management) functions for application programs run on the PC. Optionally, you can use a PC with a separate PLC to implement the application with a PLC. Bosch Rexroth has PLCs (named MLC) available, such as the MLC L45, MLC L65,

and XM hardware, etc. These PLCs also support carrier management functions.

2.3.4 NY4089-based system

The LMS control based on the NY4089 system housing consists of one MCU and one NY4150/10 SERCOS Master module. The MCU is the NY4114. The NY4150/10 connects the DriveLink drives via the SERCOS network. The DriveLink drives are available for higher voltage and higher current ranges. The coil and PTC are connected to the DriveLink drive.

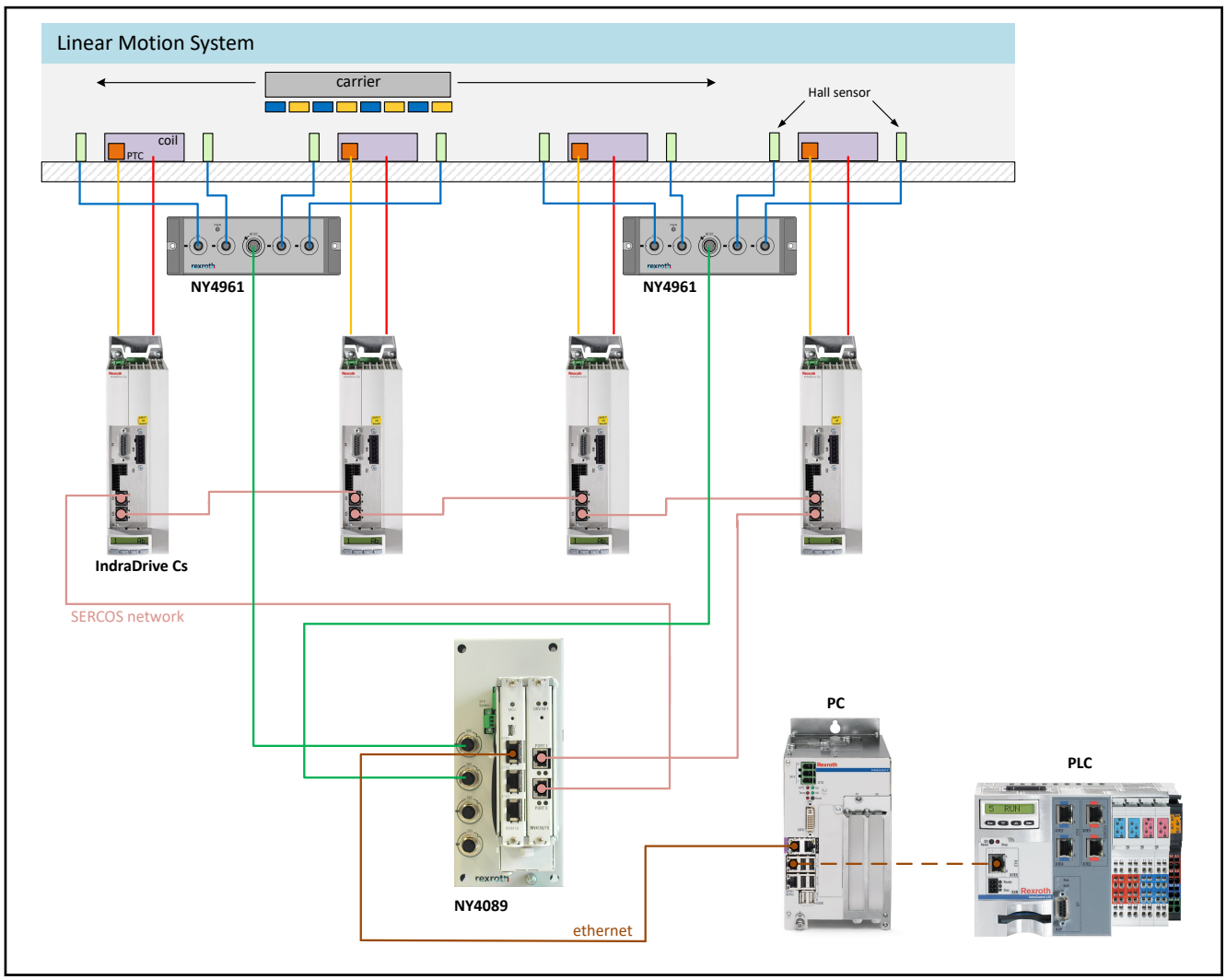


Fig. 2-9: NY4089 system overview with SERCOS network, DriveLink drives and NY4961 Sensor box

fig. 2-9 "NY4089 system overview with SERCOS network, DriveLink drives and NY4961 Sensor box" on page 10 shows the electric wiring of 4 coils with their PTC and the associated sensors to the DriveLink drives and the NY4089 system housing. Further, the connection with a PC and optional a PLC is shown. The NY4089 contains one MCU (NY4114) and one NY4150/10 SERCOS Master module. The system housing with an MCU and SERCOS Master module is called a (NYCe 4000) node. With this configuration you can connect up to 8 DriveLink drives when the node frequency is 1 kHz, and up to 6 axes when the node frequency is 2 kHz. The

NY4089 has 4 connectors to connect up to 4 NY4961 Sensor boxes. Each NY4961 Sensor box can connect up to 4 LMS sensors. The node is connected to a PC with a standard ethernet cable to one of the ports on the MCU. The PC can be a standard industrial PC or an INDRAMOTION MLC VP SET. The INDRAMOTION is a PC with an MLC incorporated. The DriveLink drives are connected in a closed loop SERCOS network to the NY4150/10 SERCOS Master module.

The required software to commission the LMS and all (carrier management) functions for application programs run on the PC. Optionally, you can use a PC with a separate PLC to implement the application with a PLC. Bosch Rexroth has PLCs (named MLC) available, such as the MLC L45, MLC L65, and XM hardware, etc. These PLCs also support carrier management functions.

2.3.5 Node – coil connection

The motor output of an NY4120/10 drive module is connected to the coil in a straight-forward manner. The three phases of the output must be connected to the three motor phases. The NY4120/10 drive module can drive two coils. The motor headers on the NY4074 system housing are called M0.0 and M0.1 for the first drive slot, and M1.0 and M1.1 for the second drive slot, see tab. 2-1 "NY4074 node – coil connections" on page 11.

Always use shielded cables to connect the drive module to the coils.

NY4074 header	Connection
M0.0	1st coil connection from NY4120/10 in drive slot 0 (DRV0)
M0.1	2nd coil connection from NY4120/10 in drive slot 0 (DRV0)
M1.0	1st coil connection from NY4120/10 in drive slot 1 (DRV1)
M1.1	2nd coil connection from NY4120/10 in drive slot 1 (DRV1)

Tab. 2-1: NY4074 node – coil connections

The motor coil temperature sensor(s) must be connected to the PTC header on the NY4074.

If an NY4079 or NY4089 system housing is used, the coils are connected to the motor connections of the DriveLink drive, and the motor coil temperature sensor(s) are connected to the DriveLink drive.

2.3.6 Node – LMS sensor connection

The Hall sensor module is available with a straight attached cable, or with a cable attached at 90 degrees angle to the sensor housing. The MR sensor module is available with a straight attached cable. The LMS sensors are connected to the NY4074 and NY4079 system housing with a standard M8 connector. The LMS sensors are connected to the NY4089 system housing via the NY4961 LMS Sensor box. Up to 4 LMS sensors can be connected to a NY4961 LMS Sensor box with the standard M8 connector, and up to 4 NY4961 LMS Sensor boxes can be connected to the NY4089 system housing with the standard M12 connector.

Order number	Component name	Description
R911173569	NYA04.1-LMS-HALLSENSOR-24-180-NY4980/00	NY4980/00 LMS Hall sensor module, straight
R911173570	NYA04.1-LMS-HALLSENSOR-24-90-NY4980/10	NY4980/10 LMS Hall sensor module, angled
R911174592	NYA04.1-LMS-MRSENSOR-5-180-NY4981/00	NY4981/00 LMS MR sensor module, straight

Tab. 2-2: LMS sensor modules

The connections on the base plate of the NY4074 and NY4079 system housing are identified by the text "S0", "S1", ..., "S9". One connection, named "L0" is available on the base plate for a shared sensor connection.

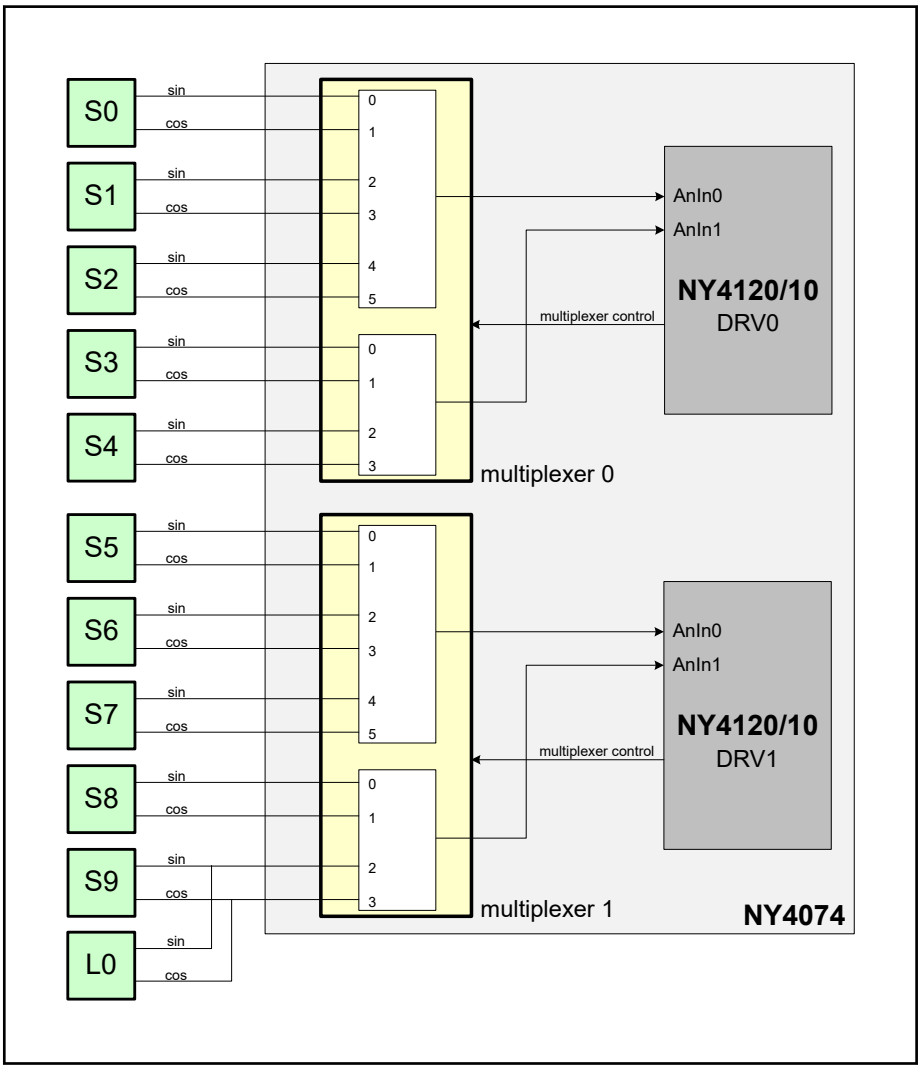


Fig. 2-10: Sensor connection on NY4074

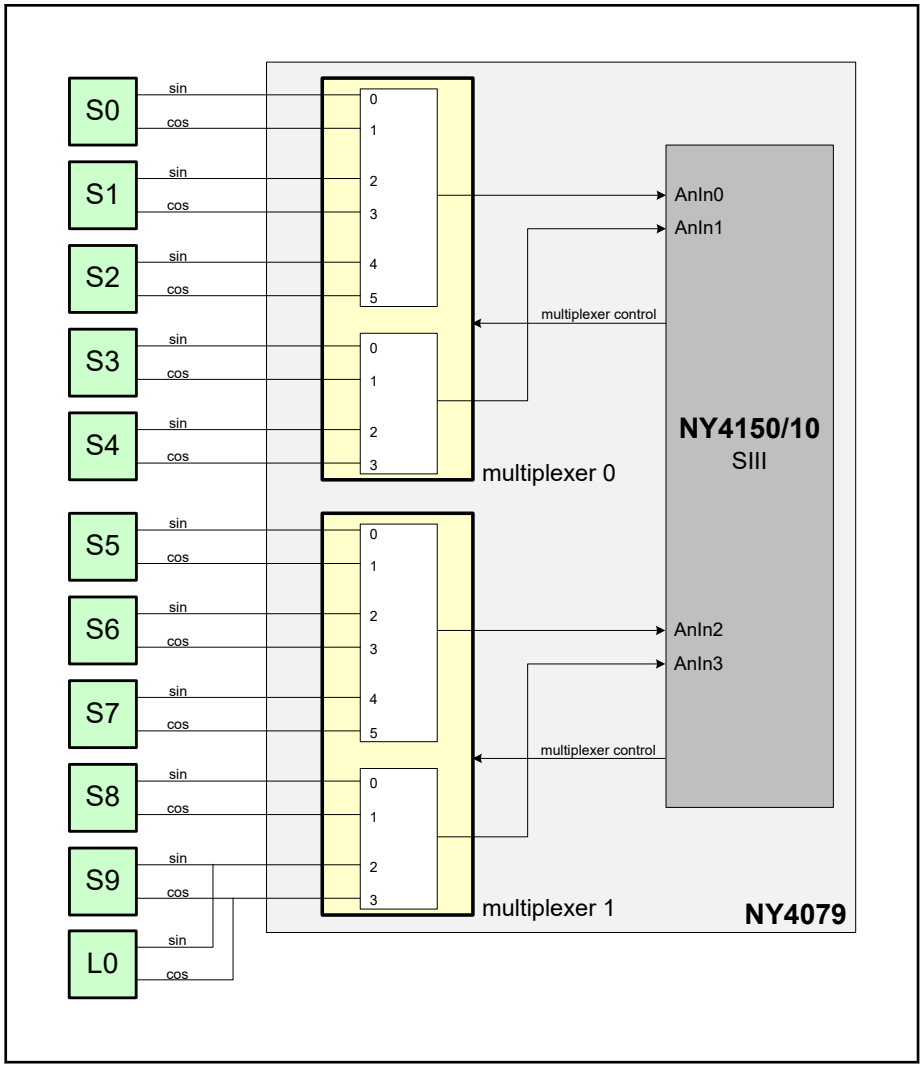


Fig. 2-11: Sensor connection on NY4079

The 10 sensor connections (S0 .. S9) are connected to 2 multiplexers. The sin and cos signal of 5 sensors are connected to multiplexer 0 and the sin and cos signal of the other 5 sensors are connected to multiplexer 1. The outputs of the multiplexers are connected to the analog inputs of the NY4120/10 drive module. If the NY4079 system housing is used, the outputs of the multiplexers are connected to the NY4150/10 SERCOS III Master module. A sensor signal is identified by means of a multiplexer channel number, slotID, and an analog input ID. This is stored in the configuration file.

Sensor	Multiplexer	Slot ID		Analog input ID	
connection	ID	NY4074	NY4079	NY4074	NY4079
S0	0, 1	0	0	0	0
S1	2, 3	0	0	0	0
S2	4, 5	0	0	0	0
S3	0, 1	0	0	1	1
S4	2, 3	0	0	1	1

Sensor	Multiplexer	Slot ID		Analog input ID	
connection	ID	NY4074	NY4079	NY4074	NY4079
S5	0, 1	1	0	0	2
S6	2, 3	1	0	0	2
S7	4, 5	1	0	0	2
S8	0, 1	1	0	1	3
S9	2, 3	1	0	1	3

Tab. 2-3: Sensor, multiplexer, slot, and analog input relationship

In case of an NY4089 system housing with NY4961 LMS Sensor boxes, the first four LMS sensors are connected to the first LMS Sensor box sensor connectors E0, E1, E2, and E3. The first LMS Sensor box is connected with the NODE connector to the SB0 connector on the NY4089 system housing. The next 4 LMS sensors are connected to the second LMS Sensor box sensor connectors E0, E1, E2, and E3. The second LMS Sensor box is connected with the NODE connector to the SB1 connector on the NY4089 system housing. Likewise, the third and fourth LMS Sensor box is connected to, respectively, the SB2 and SB3 connector on the NY4089 system housing.

The cable attached to the sensor has a length of approximately 20 cm. Extension cables to connect the LMS sensor module to the NY4074 or NY4079 system housing are available in several lengths, see following table.

Component name	Order number	Cable length	Short name
NYA04.1-SENSORCABLE-M-F-00.6M-NY4951/00	R911174517	0.6 m	NY4951/00
NYA04.1-SENSORCABLE-M-F-01.0M-NY4951/50	R911421065	1.0 m	NY4951/50
NYA04.1-SENSORCABLE-M-F-01.5M-NY4951/10	R911174518	1.5 m	NY4951/10
NYA04.1-SENSORCABLE-M-F-02.0M-NY4951/60	R911421066	2.0 m	NY4951/60
NYA04.1-SENSORCABLE-M-F-03.0M-NY4951/20	R911174519	3.0 m	NY4951/20
NYA04.1-SENSORCABLE-M-F-04.0M-NY4951/70	R911421067	4.0 m	NY4951/70
NYA04.1-SENSORCABLE-M-F-05.0M-NY4951/30	R911174520	5.0 m	NY4951/30
NYA04.1-SENSORCABLE-M-F-09.0M-NY4951/40	R911174521	9.0 m	NY4951/40

Tab. 2-4: Sensor extention cables male-female

A shared sensor connection cable is needed if a sensor is shared between two nodes.

Component name	Order number	Cable length	Short name
NYA04.1-SHARED-CABLE-M-M-01.0M-NY4951/70	R911174516	1.0 m	NY4951/70

Tab. 2-5: Shared sensor cable male-male

If the distance between NY4074 or NY4079 system housings is too large for the NY4951/70 shared sensor cable, you can extend the length with a suitable LMS sensor extension cable.

Standard two sensor configuration

In a standard two sensor configuration, with a sensor on both sides of one coil, any sensor can be connected to any sensor connection. fig. 2-12 "Standard 2 sensor configuration" on page 15 shows a logical connection approach.

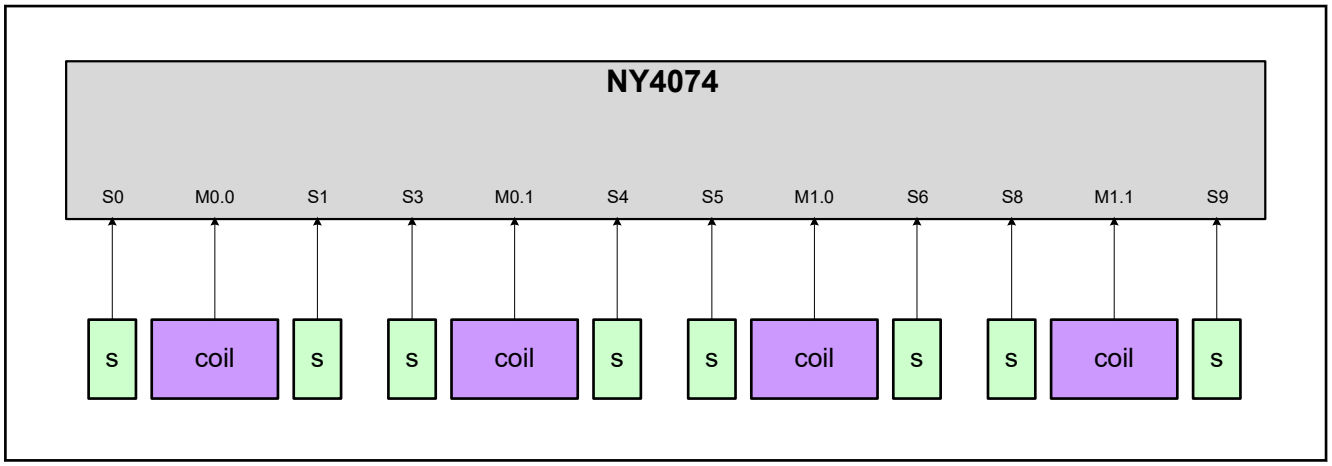


Fig. 2-12: Standard 2 sensor configuration

If an NY4079 system housing is used, the coils are connected to the motor connections of the DriveLink drive.

Standard three sensor configuration

In a standard three sensor configuration, a coil has two sensors at both sides of the coil and one sensor in the center of the coil. The sensors at the ends are shared with the neighboring coils. For coil 2, the sensor will be configured as the right sensor and for coil 3 the sensor will be configured as the left sensor or vice versa, depending on track direction. fig. 2-13 "Standard 3 sensor configuration" on page 15 shows a logcal connection approach.

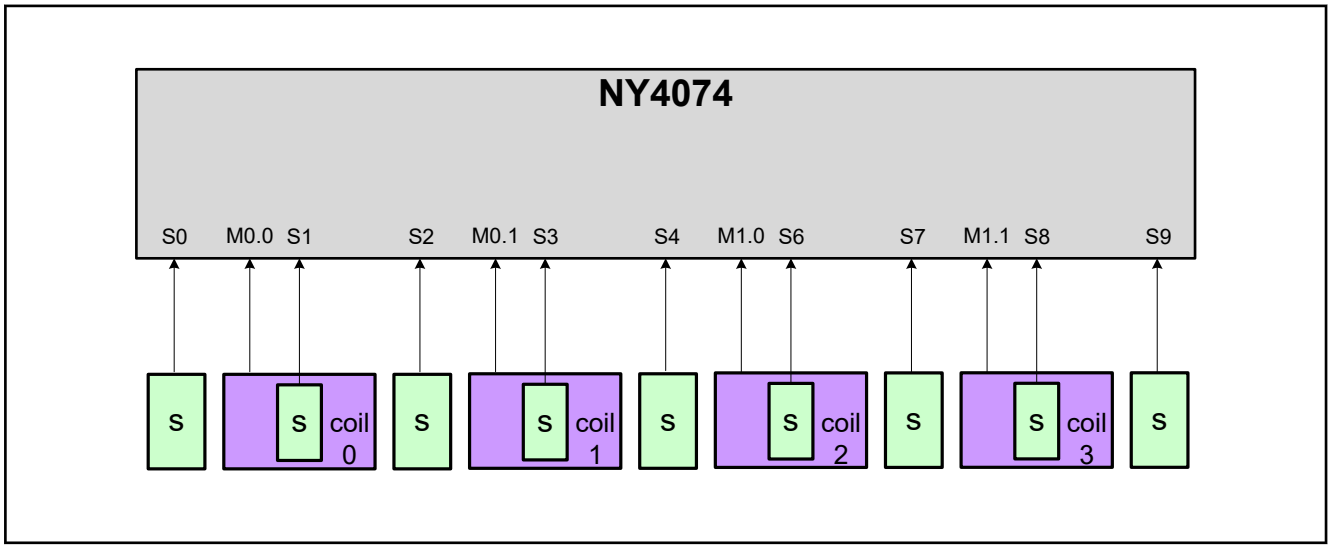


Fig. 2-13: Standard 3 sensor configuration

If an NY4079 system housing is used, the coils are connected to the motor connections of the DriveLink drive.

Looping shared sensors

When a sensor is shared between two coil units that are controlled by drive modules in different nodes, the shared sensor must be connected to both drive modules. For this purpose header "L0" is available on the base plate of the NY4074 and NY4079 system housing. fig. 2-14 "Connecting a shared sensor to two system housings" on page 16 shows a logical connection approach. Only the use of connector S9 and L0 is mandatory.

☞ The connector "L0" is internally connected to connector "S9". For that reason you must connect the shared sensor to S9.

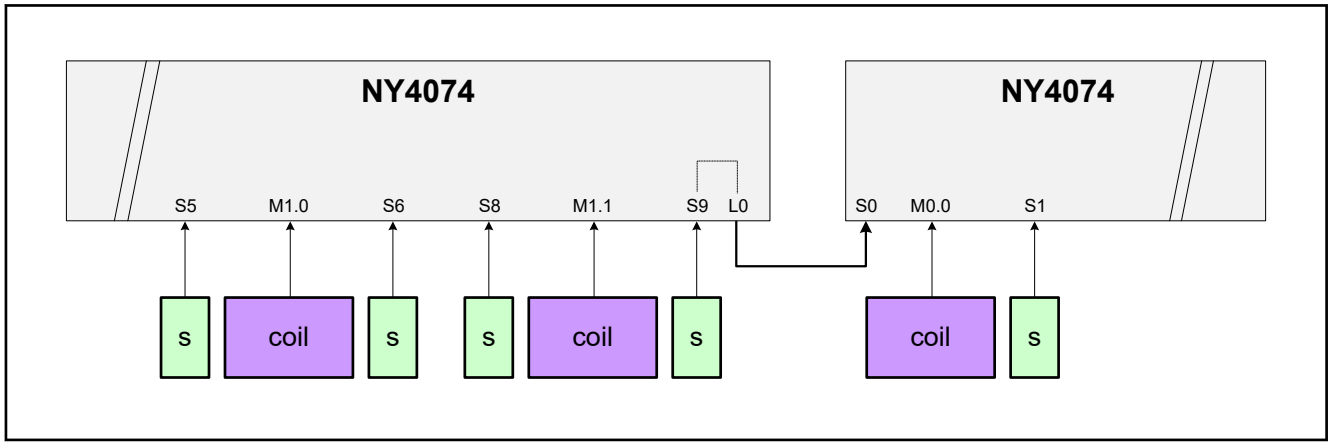


Fig. 2-14: Connecting a shared sensor to two system housings

If an NY4079 system housing is used, the coils are connected to the motor connections of the DriveLink drive.

2.4 LMS software

2.4.1 Introduction

An LMS application can execute on a PC or on a PLC. The LMS application that executes on a PC is written in the C/C++, .NET, or Python programming language. The LMS application that executes on a PLC is written in the PLC programming language.

The NYCe 4000 Software LMS Release also includes tools to configure and commission LMS tracks. These tools can be used when the LMS application runs on the PC.

☞ The installation procedure of the NYCe 4000 Software LMS software on the PC is described in the NYCe 4000 Software LMS Release Bulletin, chapter 3, "NYCe 4000 Software installation".

The installation procedure of the NYCe 4000 Software LMS node package in the node is described in the NYCe 4000 Tools Manual, chapter 2.2 "First time start-up of the NYCe 4000 System".

2.4.2 Application on a PC

An LMS application calls Carrier Management (CM) functions defined in the n4kcm.dll. The CM functionality is available to the C application via the

header files cmapi.h, cmenums.h, cmevents.h, cmtypes.h, cmvariables.h, cm_event_conversion.h, cm_var_conversion.h, cmparameters.h and cm_par_conversion.h. These header files are located in the folder "dev\inc" in the folder that you specified during the installation of the NYCe 4000 software. The default path used during the NYCe 4000 software installation is C:\Program Files\Rexroth\NYCe4000.

The n4kcm.dll contains all CM functionality, which is defined by function prototypes and data types. Every client loads its own instance of the n4kcm.dll. This makes it possible to have several programs running concurrently. The maximum number of clients supported concurrently is defined by `CM_MAX_NR_OF_CLIENTS`. See fig. 2-15 "CM architecture for PC applications" on page 17 for an architecture overview. See chapter 4 "CM - Carrier Management" on page 49 for more information on the Carrier Management software.

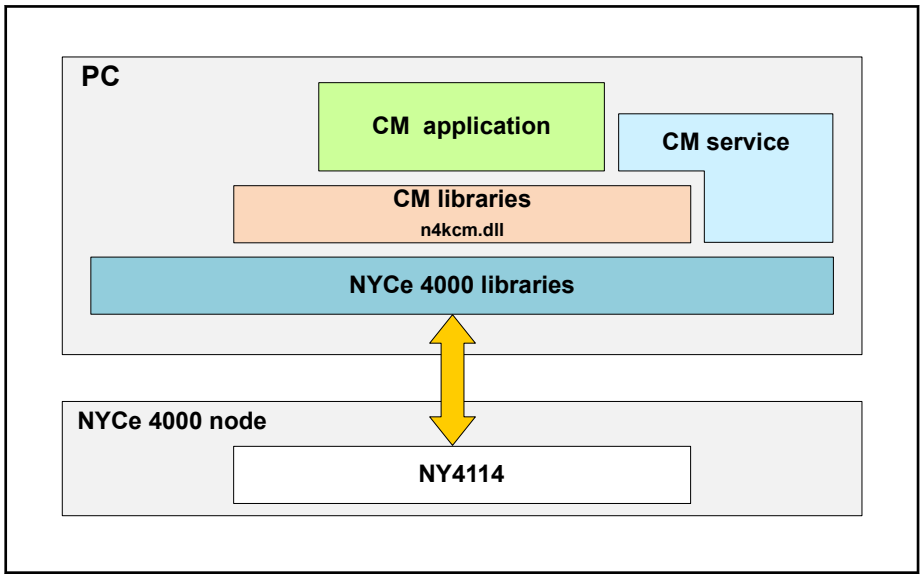


Fig. 2-15: CM architecture for PC applications

2.4.3 Application for a PLC

An LMS application can also be developed for a PLC. A PLC application program communicates with CM via a TCP/IP socket. This mechanism separates the PLC application from the execution of the requested CM commands to ensure that the cyclic PLC program can continue. The n4kcmsocketclient.library provides the communication functionality to communicate with the socket server. This library must be linked with the PLC application. The NYCeCmSocketServer executable takes care of the communication with the PLC via the TCP/IP socket. See fig. 2-16 "CM architecture for PLC applications" on page 18 for an architecture overview.

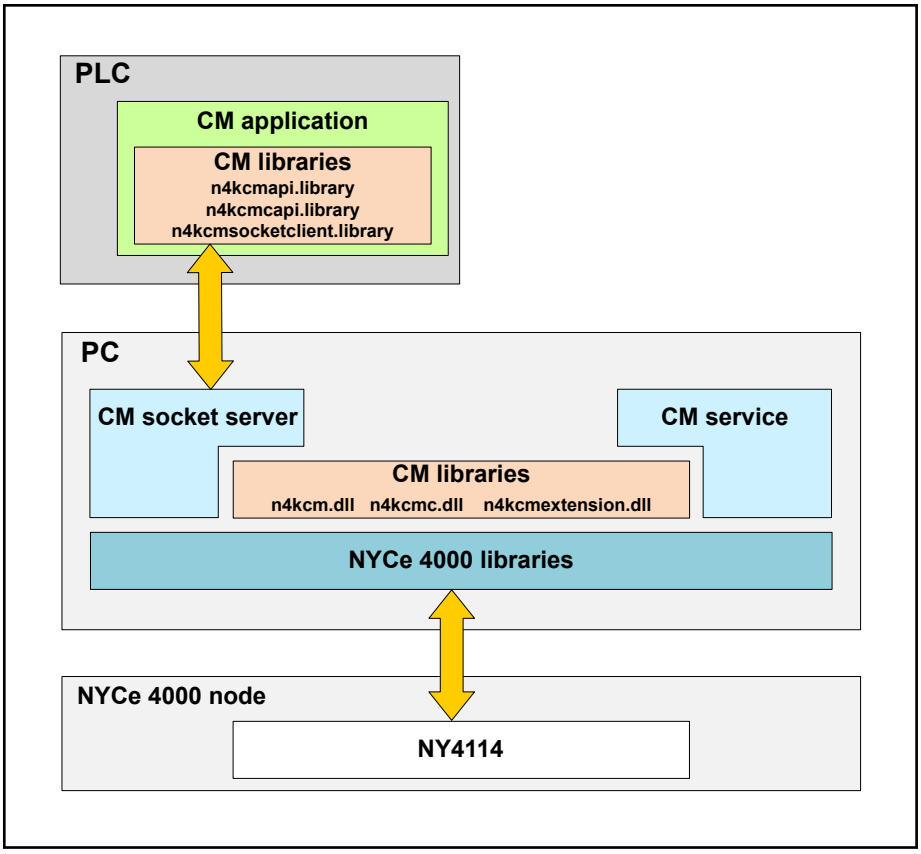


Fig. 2-16: CM architecture for PLC applications

The n4kcm.dll implements the uniform interface as required by the PLC, and provides functionality to call CM API functions directly from the PLC. The n4kcmextension.dll implements the uniform interface as required by the PLC, and provides functionality to call NYCe 4000 API functions directly from the PLC. For more details about the integration of PLCs and an LMS see the NYCe 4000 LMS IndraLogic Interface Manual.

2.4.4 CM service

CM service is implemented as a service and runs in the background. CM service starts automatically when the PC is started. The CM service keeps track of the data for each carrier in the LMS. The CM service processes internal events from the underlying NYCe 4000 subsystems and maintains the shared memory as result of the internal events and the state changes caused by commands that are executed by the LMS client application(s). Further, the CM service generates the CM events which LMS client applications can use for synchronization. The application can use the events to execute a defined response.

CM service uses TCP port number 5678 for commands, the port number range 5860 .. 5876 for events, and 5960 .. 5976 for logging. You can change these default port numbers in the registry if these port numbers are already in use. Note that administrator rights are required to change these entries in the registry. The port number registry keys are of type `DWORD` and are named

- `HKEY_LOCAL_MACHINE\SOFTWARE\Rexroth\NYCe4000\Cm \N4kCmService\CommandPort`

- `HKEY_LOCAL_MACHINE\SOFTWARE\Rexroth\NYCe4000\Cm\N4kCmService\EventPortBase`
- `HKEY_LOCAL_MACHINE\SOFTWARE\Rexroth\NYCe4000\Cm\N4kCmService\LogPortBase`

The default server connection points are thus, respectively,

- "TCP:127.0.0.1:5678"
- "TCP:127.0.0.1:5860" .. "TCP:127.0.0.1:5876"
- "TCP:127.0.0.1:5960" .. "TCP:127.0.0.1:5976"

The shared memory contains the system status and offers a current state "view" which can be retrieved via the n4kcm library. As each program links with n4kcm library, each program shares the same "view" on the LMS.

2.4.5 Data retrieval guidelines for applications

Data retrieval is often needed in an LMS application on the PC or on the PLC. Some examples are reading carrier data or polling the position of a carrier for a HMI (Human Machine Interface). If data is read in an infinite while loop, the CPU load can increase which in turn will affect n4kcmservice. The increase in communication can also affect the node communication. For these reasons, the following recommendations apply for data retrieval in LMS applications.

- HMI and other not time-critical data: not faster than every 100 ms.
- Polling of data used in the application: not faster than every 20 ms.

2.4.6 LMS tools

The NYCe 4000 Software Release LMS also includes tools to configure and commission LMS tracks. Besides the tools to configure and tune standard NYCe 4000 functionality (see NYCe 4000 Tools Manual), the Software Release LMS also includes tools for configuration and tuning of LMS tracks. See NYCe 4000 LMS Tools Manual for detailed information.

2.5 Design guidelines for the NYCe 4000 LMS

This chapter gives brief information about an LMS design from a hardware-technical point of view. However, there are also physical and software limitations imposed on the mechanical design of an LMS. The following table lists components with general guidelines for the design of an LMS. The hardware, physical, and software capabilities and limitations need to be considered for the mechanical design of an LMS with the desired performance.

See the NYCe 4000 LMS Design Guide for detailed information of the machine hardware and control hardware.

Please consult Bosch Rexroth before you start the realization of the mechanical design of an LMS to prevent costly redesigns in the mechanical design. Also, consult Bosch Rexroth if you want to modify a designed existing LMS track.

Component	Design information
System	The supported maximum node frequency depends on the number of coils controlled by a node and the type of the coils. These choices and the type of sensor also limit the maximum attainable velocity. node frequency / maximum speed Hall sensors / maximum speed MR sensors 2 kHz: 5 m/s / 2.5 m/s 1 kHz: 2.5 m/s / 1.25 m/s The maximum supported speed mentioned in the table applies to the NYCe 4000 control. Commutation needs a minimum number of points for noiseless current control. The hardware may impose additional limitations (motor specification, air gap, applied voltage, etc.).
Tracks	A track has a straight shape. Please consult Bosch Rexroth if the design has an oval shape. The NYCe 4000 LMS software supports up to 64 tracks per system (CM_MAX_NR_OF_TRACKS). Tracks can be combined in a system. Overlap between tracks is allowed as part of the RTLC functionality. The maximum number of coils in a system is defined with CM_MAX_NR_OF_TOTAL_COILS. The maximum number of coils in a track is defined with CM_MAX_NR_OF_COILS_PER_TRACK. Both definitions are set at 164. Coils may be different per track, but all coils within a track must have the same coil triplet pitch, see chapter 3.2.5 "Physical coil parameters" on page 32. For the track to track carrier movement functionality additional requirements apply, see chapter 4.7.27 "Move a carrier from track to track" on page 99.

Component	Design information
Motors	Motors consist of a magnet plate and corresponding coils. • Magnet plate Magnet plates with 24 mm pole pitch are supported. The maximum length of a magnet plate is 1024 * pole pitch = 24576 mm. The length of the magnet plate for one carrier is limited to be controlled by at maximum 3 coils. The maximum temperature for standard magnet plates is 70 °C. • Coil Coils with 2 sensors (see Fig. A and B), and coils with 3 sensors are supported. The distance between the centers of coils is equal to the length of the magnet plate (d in Fig. A) or less (M * pole pitch, where M = 1, 2, … (d' in Fig. B). A distance between the centers of the coils equal to the length of the magnet plate is preferred (see Fig. A), because distances smaller than the length of the magnet plate cause gain variations (see Fig. B). Contact Bosch Rexroth if the track design has distances smaller than the length of the magnet plate. The center of a coil does not need to be the mechanical center with respect to the outer sensor construction (distance a ≠ distance b). The deviation in placement of coils should preferably be less than ±1 mm. Deviation causes force disturbances and has a negative effect on the performance in the area between the coils.

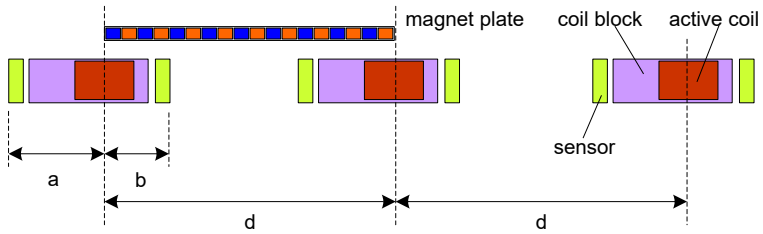


Fig. A distance d between centers of coils = length of magnet plate

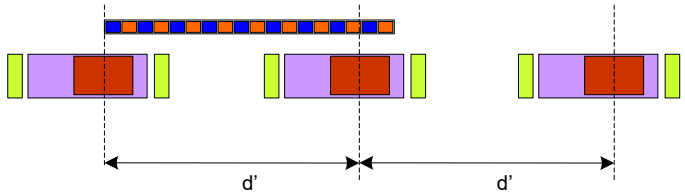


Fig. B distance d' between centers of coils < length of magnet plate

Component	Design information
Air gap (general)	Some application may require material between the carriers and the coils. Materials with a low conductivity are preferred between the Hall sensors / coils and the magnet plates in the track, for example stainless steel, AISI 316L (1.4404). Materials with a high conductivity (for example aluminum) have a negative influence on the transfer of magnetic flux between coils and magnets as well as from magnets to Hall sensors. This affects control performance and measurement accuracy, particularly at high speed. A constant air gap between each Hall sensor / each coil and the magnet plate over the track is preferred.
Air gap (for sensors)	The distance between the Hall sensor and the magnet plate has effect on the signal quality. The allowed air gap for 24 mm pole pitch magnet plates is 6.5 ± 0.5 mm. This implies that the 24 mm pole pitch motor with integrated Hall sensors can only operate with an air gap of 6.0 … 7.0 mm. The distance between the MR sensor and the magnetic scale must be between 0.5 mm and 1.4 mm.
Air gap (for coils)	For 24 mm pole pitch motors, the allowed air gap between the coil and the magnet plate is 0.5 … 5.0 mm. Force decreases with air gap distance, see the table. A larger air gap will cause less cogging. In the following table the nominal force exerted by a coil is set to 100%. The continuous force depends non-linearly on the air gap between the coil and the magnet plate for the 24 mm pole pitch motors. For 24 mm pole pitch motors, the following data applies. Air gap (mm) Continuous force --- 0.5 100% 1.0 85% 2.0 63% 3.0 47% 4.0 35% 5.0 27%
Sensors	Each carrier in a track is at all times detected by at least one sensor. This means that • the distance between two Hall sensors must be less than the length of the magnet plate minus 1 pole pitch (of the magnet plate). • the distance between two MR sensors must be less than the length of the magnetic scale minus 6 pole pitch (of the magnetic scale).

Component	Design information
Carriers	The maximum number of carriers supported in a system and in a single track is `CM_MAX_NR_OF_CARRIERS`. The length of a carrier must be equal to the length of the magnet plate plus M * pole pitch (where M = 0, 1, 2 …). Thus, the length of a carrier is always equal to or larger than the length of the magnet plate. The maximum length of a carrier is 25 m. The length of all carriers must be identical in a track, and the length of all magnet plates must be identical in a track. The following rules apply for adding a carrier to a track. • The position deviation between the specified position in software and the actual position where the carrier is placed on the track must be smaller than ½ pitch of the magnet. • None of the sensors will be underneath or close to the edge of the magnet plate.
Mechanical stops	**Begin stop and end stop are advised for safety reasons.** Begin stop and end stop are required for commissioning a linear track. The stops are used as reference to determine the position of the sensors and coils. The sensor near a mechanical stop should be covered by a pole pitch or magnet pitch if the carrier hits this stop. The maximum of additional mechanical stops (besides the begin stop and end stop) supported in a track is 10. These mechanical stops may be fixed or removable.
Homing	Positioning of the sensors should be done in such a way that after homing none of the sensors will be underneath or close to the edge of the magnet plate. Specifically, after homing to a mechanical stop the following applies. • The sensors are not able to detect two magnet plates at the same time. • The status of the sensor must always be the same.
Motion profiles	The motion profiles are 2nd order profiles or 3rd order profiles. Feed override cannot be changed during movements.
Carriers moving close to each other	The distance between magnet plates that are above one and the same coil is preferably always N * pole pitch (where N = 0, 1, 2 …). This rule implies that carriers move at the same speed above one coil, and stop at the same time. Thus, the distance between carriers above a coil is not allowed to decrease or increase.
NYCe 4000 configuration	Carrier Management (CM) software is executed on a PC, a stand-alone operation (node without PC) is not possible. In a multi-node configuration all nodes must operate at the same node frequency.

Tab. 2-6: NYCe 4000 LMS components – guidelines

3 Configuration

3.1 Introduction

This chapter describes the parameters that are defined in the configuration file of the Carrier Management for the LMS. You can determine or specify the values of these parameters using the NYCeLmsCommissioning tool (see the LMS Commissioning Tool User Manual). With tools supplied on the DVD and installed on your host, for example NYCeConfigurator and NYCeTuner, you can change all parameters and I/O (hardware). Make sure that you read the relevant chapters in the Tools Manual and the Hardware System Manual to understand the effects if you change a certain setting.

☞ The 5 Volt digital inputs and the analog inputs of the NY4120/10 (Dig5Vin0a, Dig5Vin0b, Dig5Vin0c, Dig5Vin1a, Dig5Vin1b, Dig5Vin1c, AnIn0, AnIn1) are reserved for LMS and are not allowed to be used in the application.

3.2 Parameters in the CM configuration file

3.2.1 Introduction

The configuration file has a tree structure. First, system-wide parameters are defined for the system such as the number of tracks. Next, a structure with all coils and sensors of the LMS are defined. Finally, a structure of all tracks is defined. Each track definition consists of parameters for that track, and one or more references to the structure of coils and sensors within the LMS for all coils in the track.

3.2.2 System-wide parameters

System-wide parameters are mentioned for all tracks, all coils and all sensors of the system, where applicable.

Parameter	Description
syntaxVersion	A version number that defines a specific structure
nrOfTracks	Number of tracks defined in the system
machineName	Name of the machine

Tab. 3-1: System-wide parameters

3.2.3 Track parameters

Track parameters are specified for each track and for all coils and sensors within that track, where applicable.

Parameter	Description
nrOfCoils	Number of coils on the track
nrOfSharedCoilsBegin	The total number of shared coils at the begin of the track
nrOfSharedCoilsEnd	The total number of shared coils at the end of the track
nrOfBumpers	Number of bumpers on the track
interpFactor	Interpolation factor of the measurement scale.

Parameter	Description
sensorType	Type of LMS sensor
motorMagnetPolePitch	The pitch of the magnets of the carrier from north to north magnet (see fig. 3-1 "Carrier and magnet plate related parameters" on page 28)
motorMagnetNrOfPolePairs	Number of pole pairs of the carrier (see fig. 3-1 "Carrier and magnet plate related parameters" on page 28)
motorMagnetPoleConfiguration	Polarity of the leading magnet at the positive side of the carrier (0=north and 1=south, fig. 3-1 "Carrier and magnet plate related parameters" on page 28). The positive side is the side in the positive direction of the track. The polarity can be determined with the sensor output signals.

motorMagnetEccentricity	The offset of the carrier with respect to the magnet plate. If the center of the carrier is at the lower side in relation to the center of the magnet plate, the parameter has a negative value, if the center of the carrier is at the upper side in relation to the center of the magnet plate, the value is a positive number, and if the center of the carrier equals the center of the magnet, the value is zero (see fig. 3-1 "Carrier and magnet plate related parameters" on page 28). In the figure the carrier is at the upper side and the parameter is a positive number.
	The following restriction applies:
	`abs(motorMagnetEccentricity) ≤`
	`(carrierLength - magnetPlateLength)/2`
	where `magnetPlateLength` equals `polePitch * nrOfPolePairs`

measurementScaleSensorPeriod	The period of the sensor signal of the measurement scale in [pu]
measurementScaleNrOfPeriods	The number of periods of the measurement scale
measurementScaleEccentricity	The off-center mounting of the measurement scale with respect to the carrier in [pu]. If the center of the carrier is at the lower side in relation to the center of the measurement scale, the parameter has a negative value, if the center of the carrier is at the upper side in relation to the center of the measurement scale, the value is a positive number, and if the center of the carrier equals the center of the measurement scale, the value is zero
carrierLength	The length of the carrier (see fig. 3-1 "Carrier and magnet plate related parameters" on page 28)
coilTripletPitch	Distance between two successive triplets of coil windings (see fig. 3-9 "CoilTripletPitch" on page 35)
trackType	Type of track (linear = 0, modulo = 1)
trackBeginPos	• Linear track: position of the carrier (calculated at the center of the magnet plate of the carrier), when the carrier is placed against the negative mechanical end stop (see fig. 3-11 "Position parameters of the track" on page 36). This begin position is always the track begin position without any shared coils • Modulo track: the minimum position for modulo calculation.
trackEndPos	• Linear track: position of the carrier (calculated at the center of the magnet plate of the carrier), when the carrier is placed against the positive mechanical end stop (see fig. 3-11 "Position parameters of the track" on page 36). This end position is always the track end position without any shared coils. • Modulo track: the maximum position for modulo calculation.
homeMode	Homing mode (see chapter 4.7.8 "Homing a track" on page 67)

Parameter	Description
homeVelocity	Velocity of the carriers during homing of the carriers
homeTimeout	Time-out value in which a home procedure should be finished
homeMoveCheckDelay	The time delay used between the start of the home procedure and the check that carriers are actually moving
maxVelocity	Maximum velocity of a carrier
maxAcceleration	Maximum acceleration of a carrier
maxJerk	Maximum jerk of a carrier
logical coil folder(s)	Each coil folder can hold multiple alternative coil definitions. These parameters must be defined for each alternative coil. • name : reference to the physical coil name • coilCoggingParFile : file with the cogging compensation table for that coil • coilPosition : position of the coil on the track (see fig. 3-8 "CoilPosition" on page 34) • trackBeginPos : the track begin position including the shared coil(s) at the begin of the track. Must only be specified when a shared coil is used at the begin of a track. • trackEndPos : the track end position including the shared coil(s) at the end of the track. Must only be specified when a shared coil is used at the end of a track.
bumper folder(s)	• lowerSidePos : position of the first carrier against the lower side of a bumper when homed in positive direction. • upperSidePos : position of the first carrier against the upper side of a bumper when homed in negative direction.
safeDistance	The minimum distance between two carriers for collision avoidance functionality.
process area folder(s)	• id : identifier of the defined process area. • minPosition : the minimum position of a defined process area. • maxPosition : the maximum position of a defined process area.

Tab. 3-2: Track parameters

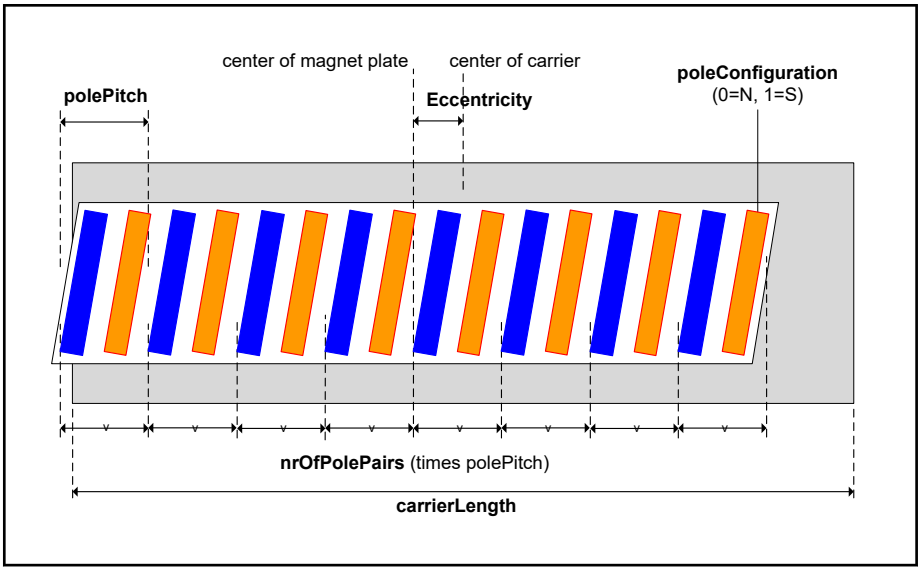


Fig. 3-1: Carrier and magnet plate related parameters

The length of the carrier can be equal to the length of the magnet plate or larger than the length of the magnet plate, as is shown in fig. 3-1 "Carrier and magnet plate related parameters" on page 28. The length of the magnet plate is the product of `polePitch` and `nrOfPolePairs`.

For a magnet plate, the pole pitch equals the distance between two identical poles (North → North or South → South). See fig. 3-2 "pole pitch definition of a magnet plate" on page 28 for the definition of pole pitch of a magnet plate and the signal output from the Hall sensor.

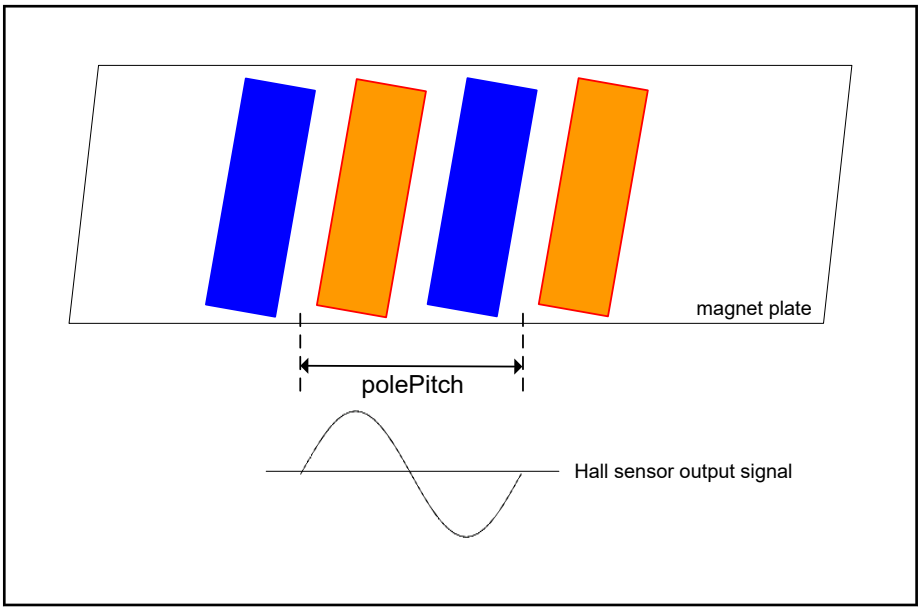


Fig. 3-2: pole pitch definition of a magnet plate

For a magnetic measurement scale, the pole pitch equals the distance between two poles (North → South or South → North). See fig. 3-3 "pole pitch definition of a magnetic measurement scale" on page 29 for the definition of pole pitch of a magnetic measurement scale and the signal output from the MR sensor.

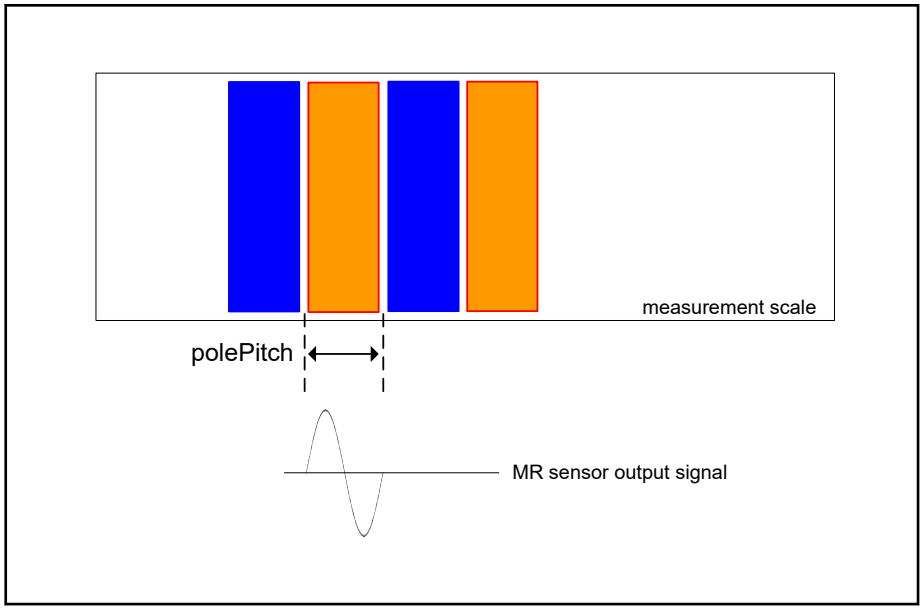


Fig. 3-3: pole pitch definition of a magnetic measurement scale

All parameters of the velocity control loop (see fig. 3-4 "Velocity control loop" on page 29) are defined in the LMS XML configuration file. There is one exception: `CONTROLLER_OUTPUT_INVERT`. This parameter is not in the LMS configuration file, it is a SAC parameter. You can set the parameter `CONTROLLER_OUTPUT_INVERT` with the NYCeTuner tool, see the NYCe 4000 Tools Manual.

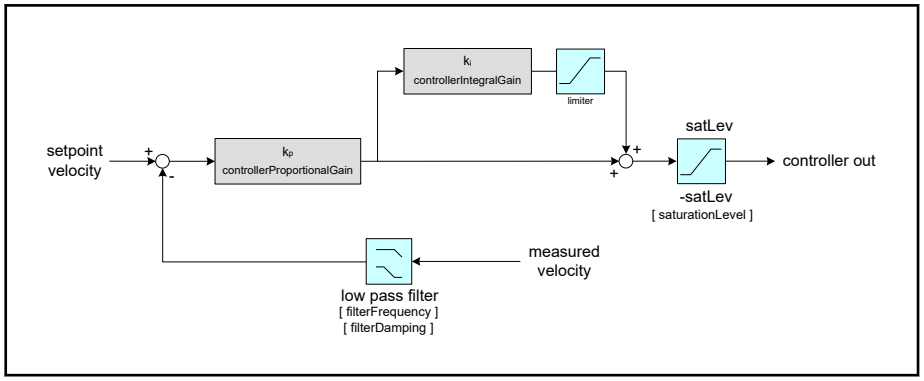


Fig. 3-4: Velocity control loop

3.2.4 Carrier parameters

Carrier parameters are specified for each carrier within a track.

Parameter	Description
id	Identifier of the carrier.
measurementScaleOffset	This parameter is only used in an MR sensor based LMS. The value must be within +/- `motorMagnetPolePitch` / 12.
productOffset	This parameter defines the exact placement position of a product on the carrier.

Tab. 3-3: Carrier parameters

How to determine measurementScaleOffset

An LMS carrier on an MR sensor based track has a motor magnet plate for control and a measurement magnetic scale for sensing the position. For different carriers, the measurement scale may have small variations in accurate mounting position, relative to the (middle of) the carrier. In fig. 3-5 "Magnetic scale positioning accuracy over carriers" on page 30, two carriers are placed above the same coil and sensor. Carrier 0 is considered to be the "reference carrier", with which commissioning is performed. The following offset must be taken into account for carrier 1 to fulfill accuracy requirements.

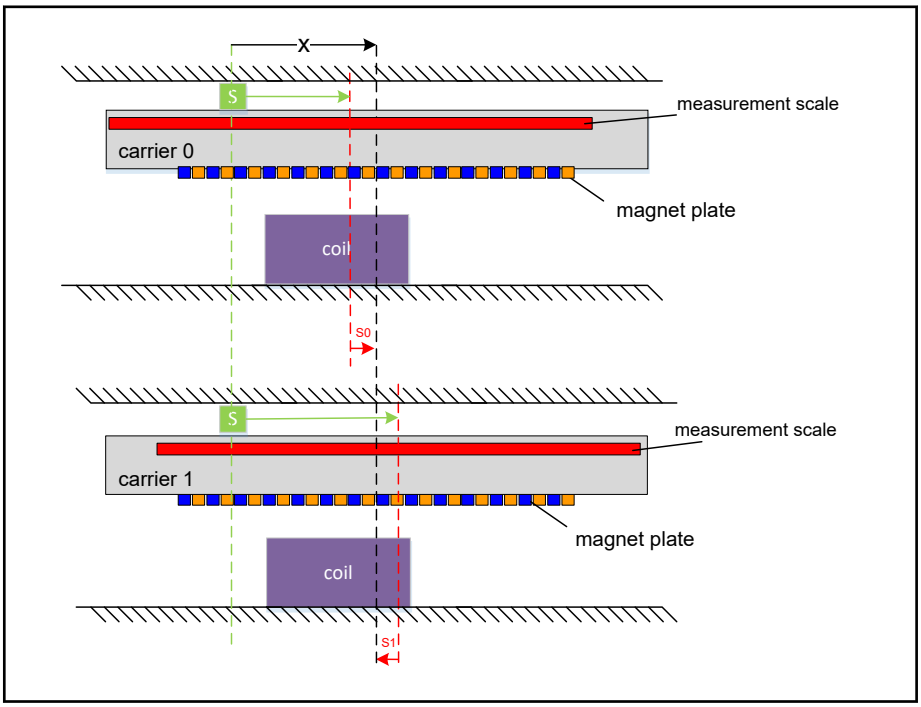


Fig. 3-5: Magnetic scale positioning accuracy over carriers

The distance from the carrier center to the center of the measurement scale is indicated with a red arrow, and is S0 for carrier 0 and S1 for carrier 1. The position indicated by the sensor is determined by the position of the carrier, when the middle of the sensor is above the middle of the measurement scale. As carrier 0 is the "reference carrier", an offset of S1 – S0 must be subtracted from the measured position of carrier 1 to have the same position value at physically the same position.

Commissioning can be performed with one (reference) carrier. Subsequently, for all other carriers in the LMS an offset must be determined. In general, applications have their own solution to determine the offset, usually with an external high-accuracy measurement device. If this is not available, the following method can be used for a rough determination of an offset per carrier. The described method is not supported by any tool, as it is not sufficiently accurate in general. Set the offset of all carriers to 0. Measure the position of the reference carrier at the track begin position (or at track end position, a bumper, or a manually marked position). Measure the position value for an other carrier at physically the same position. The difference of the measured position values determines `measurementScaleOffset`.

Positioning of the magnetic scale

The magnetic scale of **all** carriers in the LMS must be positioned such that all are within a span of 2 mm. The positioning of the magnetic scale of carrier D can never fit within a 2 mm span, compared to carriers A, B, and C.

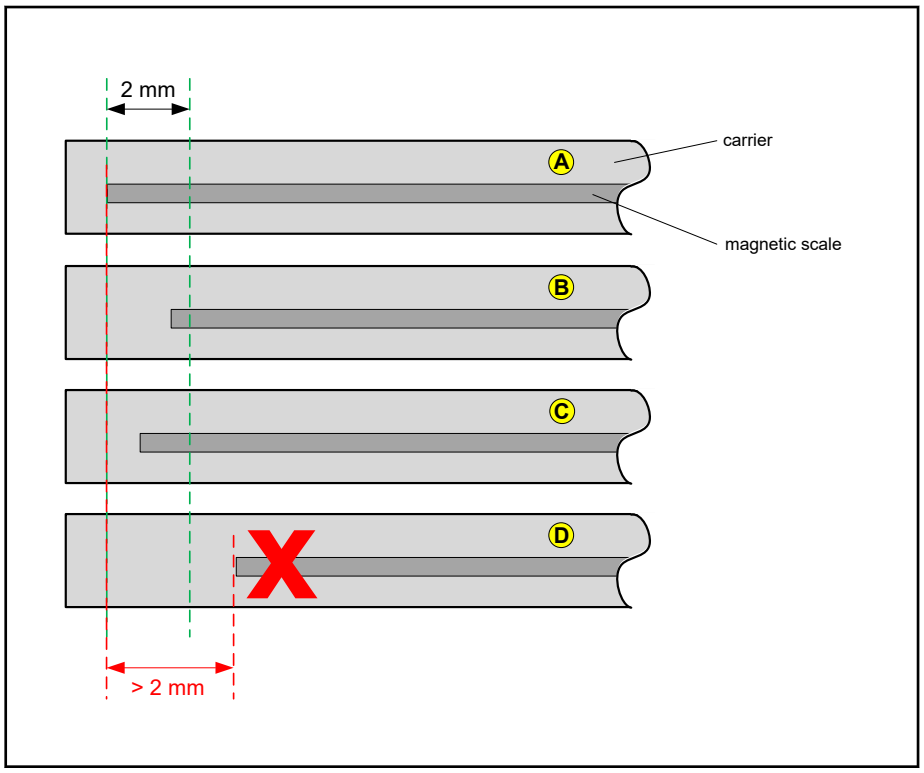


Fig. 3-6: Positioning of the magnetic scale on carriers in one LMS

How to determine productOffset

The position of a product on a carrier is usually accurately defined by some means of an alignment provision on the carrier. However, small differences of the placement of the alignment provision on different carriers are possible. These small differences between carriers must be corrected for each carrier. One carrier in the LMS may be the "reference carrier" and this carrier may have a `productOffset` 0. For all other carriers in the LMS the `productOffset` (of the alignment provision) is determined relative to the "reference carrier".

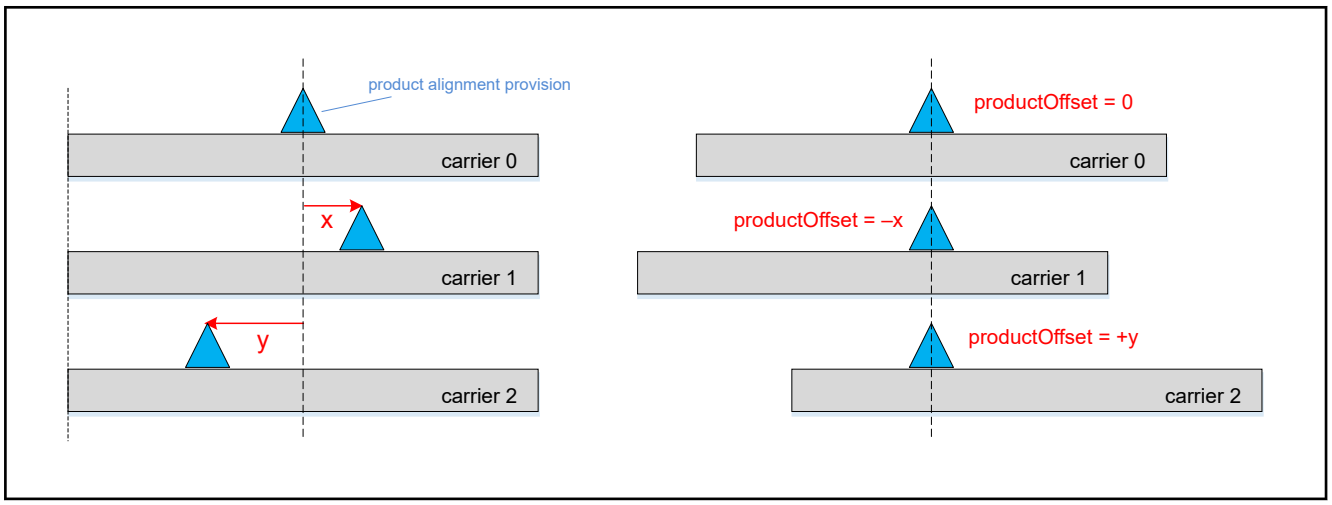


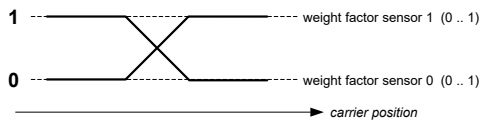
Fig. 3-7: Product positioning differences between different carriers

In fig. 3-7 "Product positioning differences between different carriers" on page 31, 3 carriers are positioned at exactly the same position. Carrier 0 is considered to be the "reference carrier". Due to a small position difference of the alignment provision on carrier 1 and carrier 2, the position of the product on carrier 1 and carrier 2 has that same position difference. A product offset can be defined for up to 128 carriers. With the parameter `productOffset` this difference can be corrected. To align the product on carrier 1 with the product on the reference carrier (carrier 0), the carrier would have to move in the negative direction. Therefore, the `productOffset` for carrier 1 is a negative number. Likewise, carrier 2 would have to move in the positive direction, therefore, the `productOffset` for carrier 2 is a positive number. The position difference must be determined with an external high-accuracy measurement device.

3.2.5 Physical coil parameters

Coil parameters are specified for each coil and for all sensors used by that coil, where applicable.

Parameter	Description
name	Coil name, physical coil name, or axis name of the coil configured by NYCeConfigurator
coilParFile	Parameter file (tuning parameters) belonging to that coil
defaultTrack	The track to which this coil is assigned
nrOfCoilTriplets	Number of coil triplets in the coil-housing (see fig. 3-9 "CoilTripletPitch" on page 35)
coilTripletCoverageFraction	Fraction of the outer triplet that must be covered by a carrier before the coil starts controlling the carrier (see fig. 3-10 "CoilTripletCoverageFraction and controlHysteresis" on page 35). This parameter defines the possible control range of the coil: min: coilPosition – (½ * nrOfPolePairs * polePitch) – (½ * coilTripletPitch * nrOfCoilTriplets) + (coilTripletCoverageFraction * coilTripletPitch) max: coilPosition + (½ * nrOfPolePairs * polePitch + (½ * coilTripletPitch * nrOfCoilTriplets) – (coilTripletCoverageFraction * coilTripletPitch) Value range is between 0 and 1. Typical value is 0.5. The value has an effect on cogging.
controlHysteresis	Hysteresis for leaving the control range (see fig. 3-10 "CoilTripletCoverageFraction and controlHysteresis" on page 35).

Parameter	Description
positionOffset	The distance between the magnetic center of the coil and the midway distance between the sensor elements of the sensor (d1 = d2). See fig. 3-12 "positionOffset" on page 36. Note that the magnetic center (line ②) and mechanical center (line ①) of the coils are not identical. Also note that the sensor element is not in the middle of the sensor housing. Therefore, the orientation of the sensor has an effect on the `positionOffset` value as can be seen in figure B. A positive value means an offset in positive direction, a negative value means an offset in negative direction, 0 means no offset.
sensorPosOffset	Defines two areas, lower than a minimum position and higher than a maximum position, where the sensor cannot see the carrier (see fig. 3-13 "Sensor ranges" on page 37, and the description "Configuration of sensorPosOffset").
sensorSwitchMargin	Defines an area above a sensor where the carrier may or may not be detected by the sensors. The transition occurs in this area, see fig. 3-13 "Sensor ranges" on page 37, and the description "Configuration of sensorSwitchMargin".
sensorTransitionZone	Size of the zone where the carrier position is determined as a weighed average of the positions of the two sensor pairs. min: (sensorPos0 + sensorPos1)/2 – ½ * sensorTransitionZone max: (sensorPos0 + sensorPos1)/2 + ½ * sensorTransitionZone This results in a smooth take-over between sensors.  Note: The take-over position is not necessarily the same as the coil position
controllerIntegralGain ①	Controller integrator
controllerProportionalGain ①	Controller gain
controllerProportionalGainInError ①	Controller gain used when the open loop error handler is active
filterFrequency ①	Filter frequency of the low pass filter (LPF)
filterDamping ①	Filter damping of the low pass filter (LPF)
saturationLevel ①	Maximum controller output (abs)

Parameter	Description
saturationLevelInError ①	Maximum controller output (abs) when the open loop error handler is active
nrOfSensors	Number of sensors
sensor folder(s)	See chapter 3.2.6 "Sensor parameters" on page 45

① These parameters are used in the velocity control loop (see fig. 3-4 "Velocity control loop" on page 29).

Tab. 3-4: Coil parameters

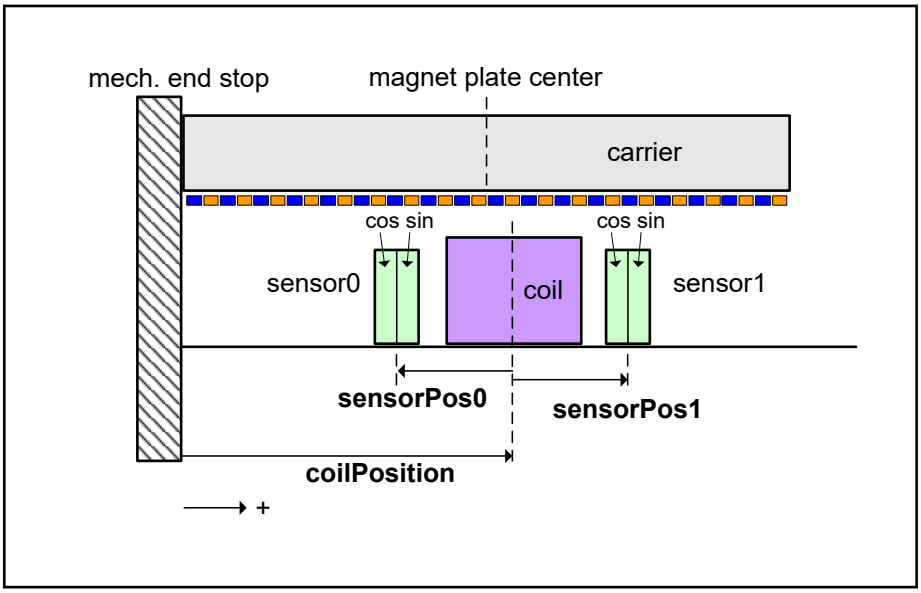


Fig. 3-8: CoilPosition

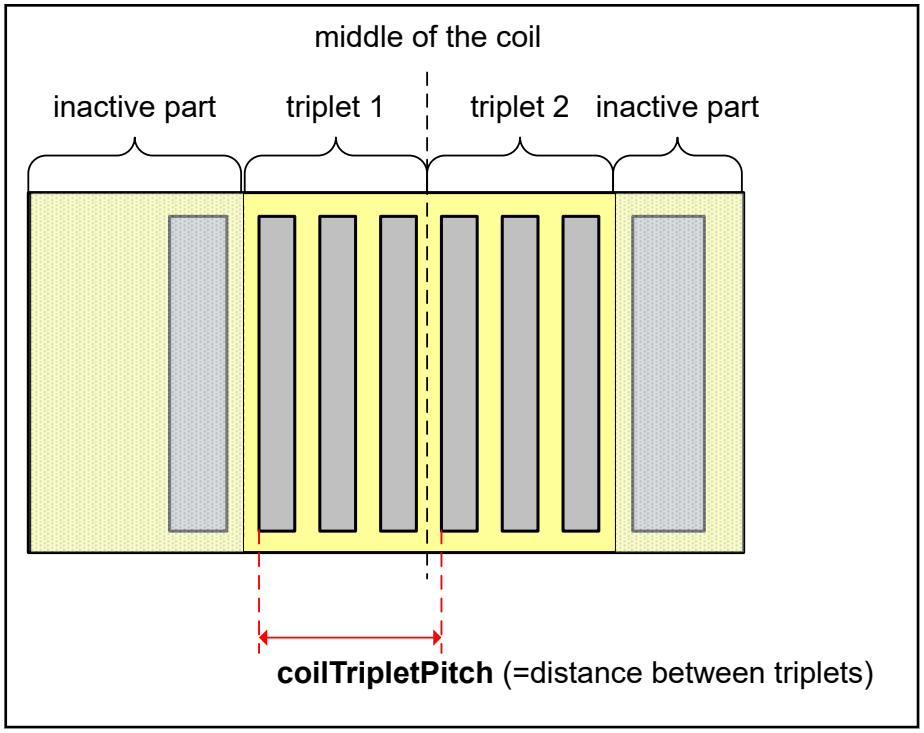


Fig. 3-9: CoilTripletPitch

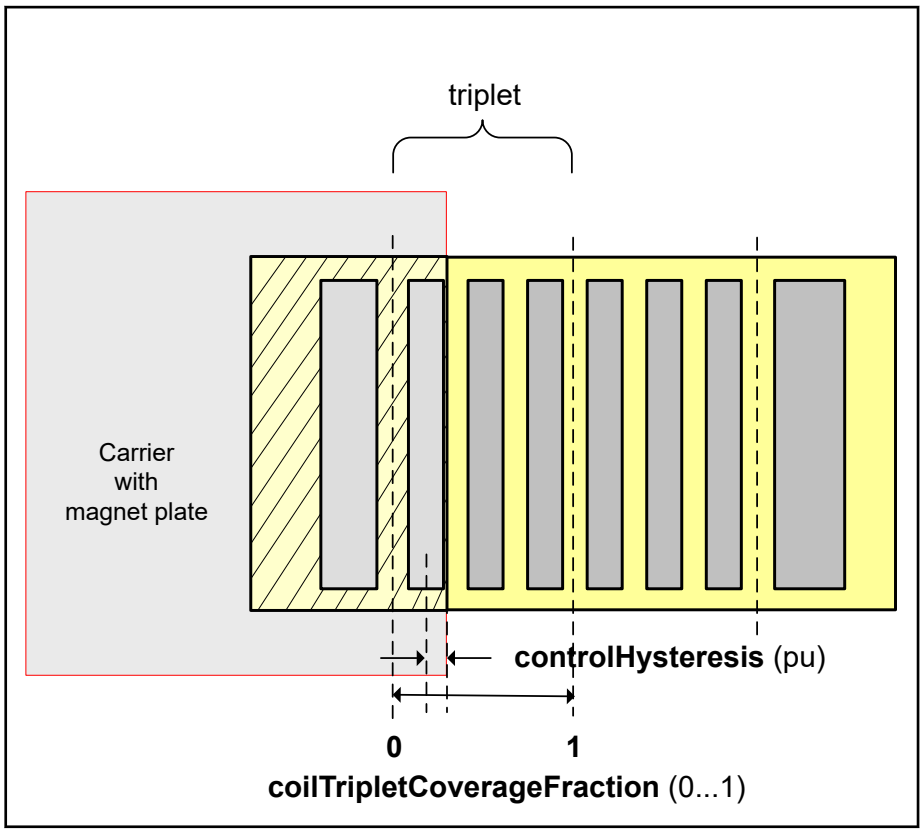


Fig. 3-10: CoilTripletCoverageFraction and controlHysteresis

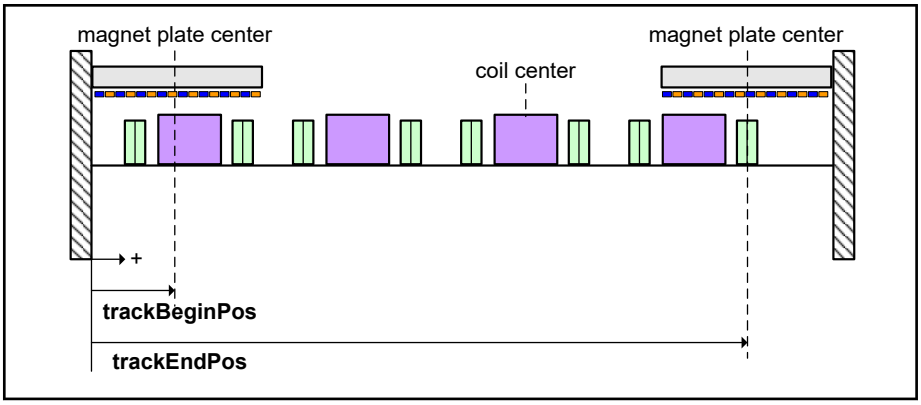


Fig. 3-11: Position parameters of the track

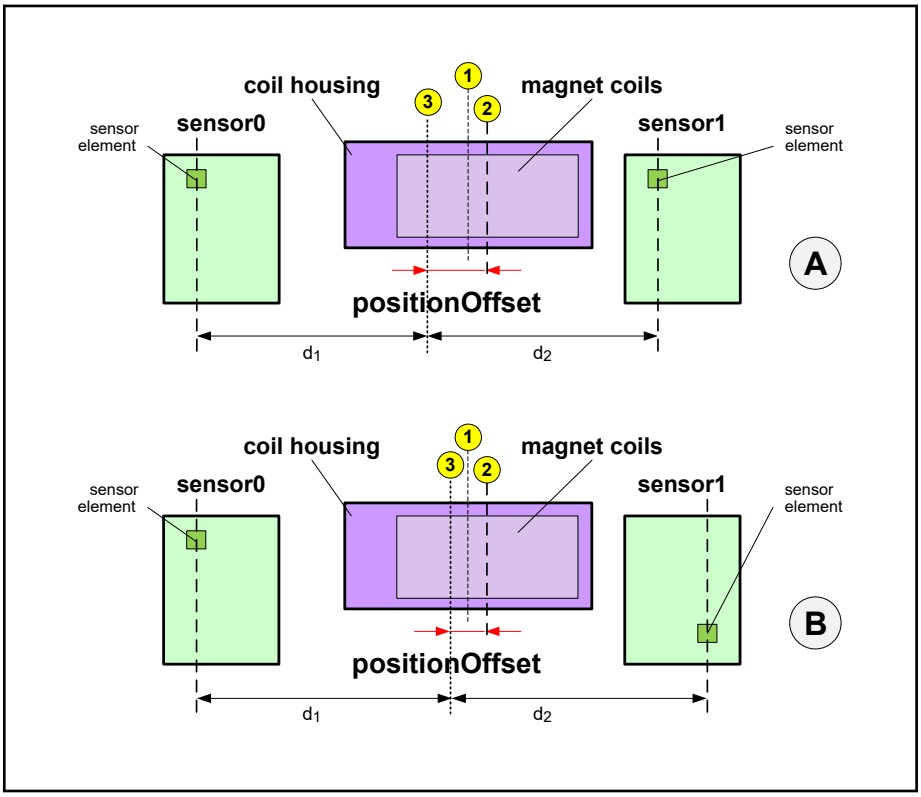


Fig. 3-12: positionOffset

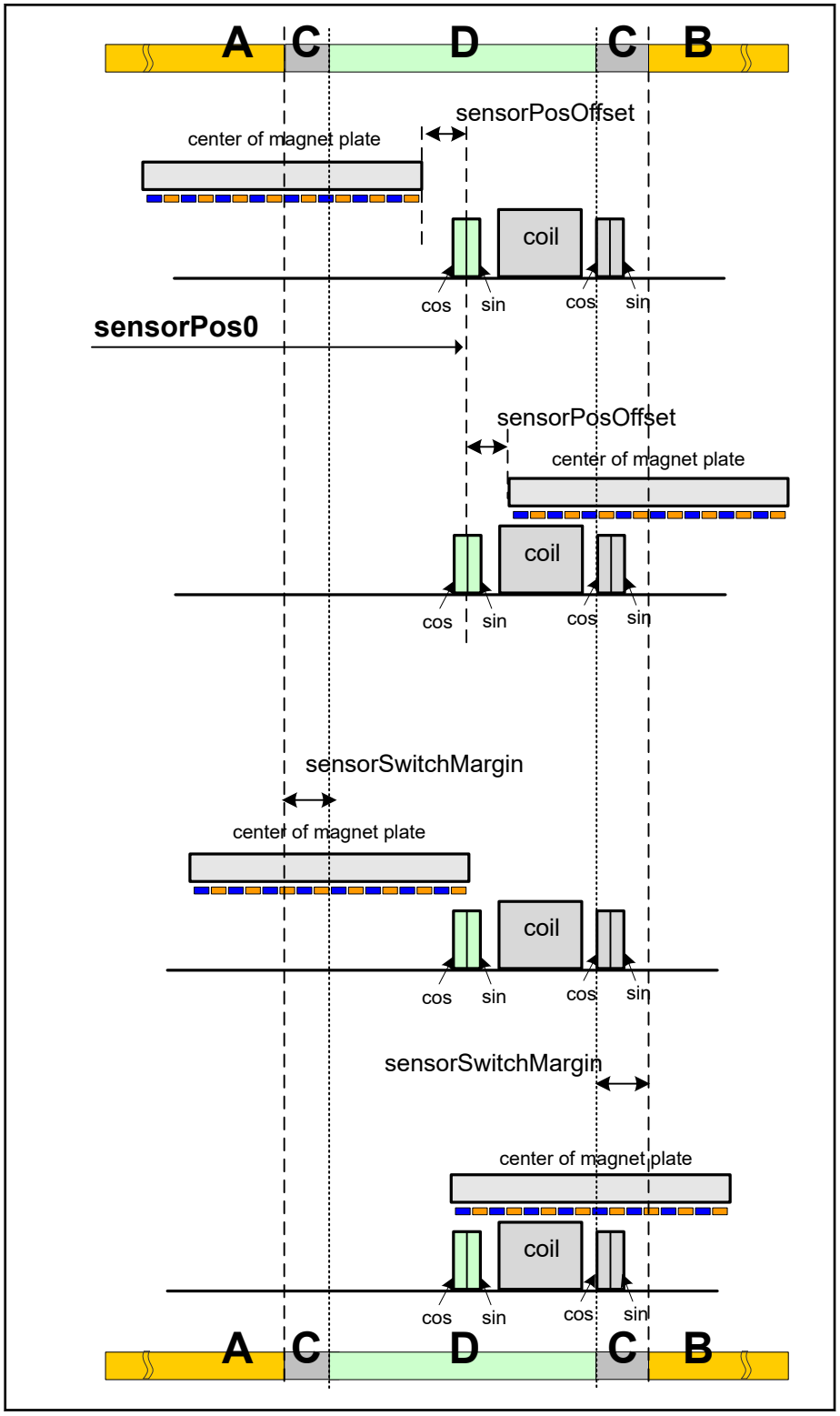


Fig. 3-13: Sensor ranges

In the ranges "A" and "B", the detection by the sensor 0 is guaranteed `FALSE`.

In the range "D", the detection by the sensor 0 is guaranteed `TRUE`.

In the ranges "C", the detection by the sensor 0 is `TRUE` or `FALSE`, the transition occurs in this area.

Configuration of sensorPosOffset

The `sensorPosOffset` is a parameter in the LMS configuration XML file that the user can set. The parameter `sensorPosOffset` is needed to control adjacent moving carriers. Adjacent moving carriers are carriers that move concatenated with no gap or a very small gap between the carriers. A sensor will continuously detect a carrier when the first of two adjacent moving carriers has left the upper or lower sensor range. At this moment it is no longer correct to continue to respectively increase or decrease the sensor position. The sensor already detects the second carrier, which is moving adjacent to the first carrier. If a sensor reads a position outside the sensor range, it is assumed that an adjacent carrier is already detected by this sensor. Depending on which sensor range is reached, the sensor index and position are updated as follow.

- sensor position > upper sensor range
  - index is changed to the index of the lower side neighbor sensor
  - sensor position is decreased with one carrier length.
- sensor position < lower sensor range
  - index is changed to the index of the upper side neighbor sensor
  - sensor position is increased with one carrier length.

`sensorPosOffset` depends on the detected and undetected sensor positions when a carrier moves above a sensor. The precondition that a sensor must detect approximately (n - ¼)*`polePitch` of a carrier, with n = number of pole pairs of one carrier, must be met. The parameters that affect this precondition are `sensorDetectCrit` and `sensorDetectHist`, see tab. 3-5 "Sensor parameters" on page 45.

The lower sensor range (`minSensorPos`) and upper sensor range (`maxSensorPos`) are defined by `sensorPosOffset`:

```
minSensorPos = parSensorPos - (½ magnetPlateLength) - sensorPosOffset
```

```
maxSensorPos = parSensorPos + (½ magnetPlateLength) + sensorPosOffset
```

To avoid sensor errors, you must initialize `sensorPosOffset` and `sensorSwitchMargin` with the values for ideal hardware:

```
sensorPosOffset = ¼ polePitch
```

```
sensorSwitchMargin = ½ polePitch
```

After you have set these values you can determine the best `sensorPosOffset` value. The `sensorPosOffset` has two criteria. These criteria apply for adjacent moving carriers in positive direction, respectively negative direction. The `sensorPosOffset` must meet both criteria.

1. `sensorPosOffset` criterion for adjacent moving carriers in positive direction.

   When carrier A is at sensor position > `maxSensorPos`, it is assumed that there is a carrier B moving adjacent with carrier A, see fig. 3-14 "Sensor position offset, movement in positive direction" on page 39. The sensor index and position are updated as follow.

   - index is changed to the index of the lower side neighbor sensor
   - sensor position is decreased with one carrier length.

The criterion that must be met after update:

new sensor position > $sensorDetectedPos_{when\ moving\ in\ positive\ direction}$.

This means that when the sensor is updated, the position of carrier B must be at least greater than $sensorDetectedPos_{when\ moving\ in\ positive\ direction}$. The $sensorDetectedPos_{when\ moving\ in\ positive\ direction}$ is the position read by the sensor to change from undetected to detected, when a carrier is moving in positive direction onto the sensor. The general criterion is:

$maxSensorPos - magnetPlateLength > sensorDetectedPos_{when\ moving\ in\ positive\ direction}$.

With
$maxSensorPos = parSensorPos + (½\ magnetPlateLength) + sensorPosOffset$

substituted:

$parSensorPos + (½\ magnetPlateLength) + sensorPosOffset - magnetPlateLength > sensorDetectedPos_{when\ moving\ in\ positive\ direction}$

$sensorPosOffset > (½\ magnetPlateLength) + (sensorDetectedPos_{when\ moving\ in\ positive\ direction} - parSensorPos)$.

Adding a safety margin of $^1/_8$ $polePitch$, the formula becomes:

$sensorPosOffset = (½\ magnetPlateLength) + (sensorDetectedPos_{when\ moving\ in\ positive\ direction} - parSensorPos) + 1/8\ polePitch$.

Remember, if $sensorPosOffset < 0$ then $sensorPosOffset = 0$, since $sensorPosOffset \geq 0$.

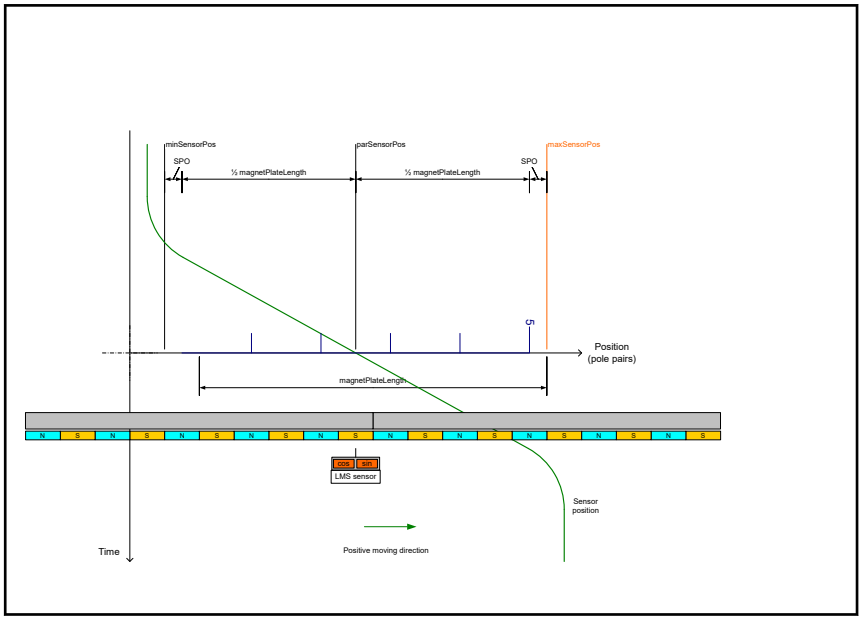


Fig. 3-14: Sensor position offset, movement in positive direction

2. sensorPosOffset criterion for adjacent moving carriers in negative direction

   When carrier B is at sensor position < $minSensorPos$, it is assumed that there is a carrier A moving adjacent with carrier B, see fig. 3-15

"Sensor position offset, movement in negative direction" on page 40.
The sensor index and position are updated as follow.

- index is changed to the index of the upper side neighbor sensor
- sensor position is increased with one carrier length.

The criterion that must be met after update:

$New\ sensor\ position < sensorDetectedPos_{when\ moving\ in\ negative\ direction}$.

This means that when the sensor is updated, the position of carrier A must be at least less than $sensorDetectedPos_{when\ moving\ in\ negative\ direction}$. The $sensorDetectedPos_{when\ moving\ in\ negative\ direction}$ is the position read by the sensor to change from undetected to detected when a carrier is moving in negative direction onto the sensor. The general criterion is:

$minSensorPos + magnetPlateLength < sensorDetectedPos_{when\ moving\ in\ negative\ direction}$.

With $minSensorPos = parSensorPos - (½\ magnetPlateLength) - sensorPosOffset$

substituted:

$parSensorPos - (½\ magnetPlateLength) - sensorPosOffset + magnetPlateLength < sensorDetectedPos_{when\ moving\ in\ negative\ direction}$

$sensorPosOffset > (½\ magnetPlateLength) + (parSensorPos - sensorDetectedPos_{when\ moving\ in\ positive\ direction})$

Adding a safety margin of $^1/_8$ $polePitch$, the formula becomes:

$sensorPosOffset = (1/2\ magnetPlateLength) + (parSensorPos - sensorDetectedPos_{when\ moving\ in\ positive\ direction}) + 1/8\ polePitch$

Remember, if $sensorPosOffset < 0$ then $sensorPosOffset = 0$, since $sensorPosOffset \geq 0$.

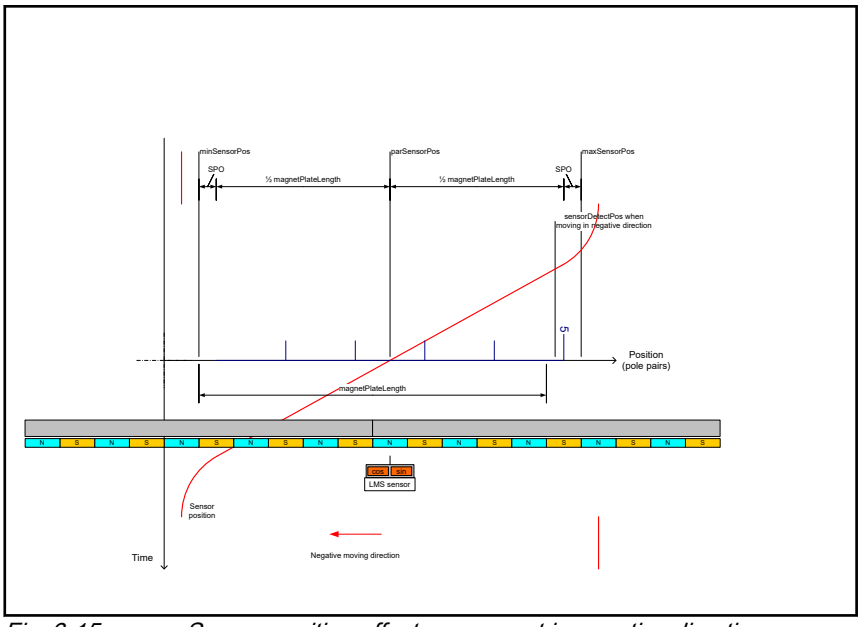


Fig. 3-15: Sensor position offset, movement in negative direction

You should measure

- $sensorDetectedPos_{when\ moving\ in\ positive\ direction}$

  The $sensorDetectedPos_{when\ moving\ in\ positive\ direction}$, is the position read by the sensor to change from undetected to detected when a carrier is moving in positive direction onto the sensor.

- $sensorDetectedPos_{when\ moving\ in\ negative\ direction}$

  The $sensorDetectedPos_{when\ moving\ in\ negative\ direction}$, is the position read by the sensor to change from undetected to detected when a carrier is moving in negative direction onto the sensor.

Calculate the sensorPosOffset for adjacent moving carriers in positive, respectively negative direction as follow:

$sensorPosOffset\_P = (½\ magnetPlateLength) + (parSensorPos - sensorDetectedPos_{when\ moving\ in\ positive\ direction}) + 1/8\ polePitch$

$sensorPosOffset\_N = (½\ magnetPlateLength) + (sensorDetectedPos_{when\ moving\ in\ positive\ direction} - parSensorPos) + 1/8\ polePitch$

$sensorPosOffset = maximum\ of\ sensorPosOffset\_P\ and\ sensorPosOffset\_N$

This explains the initial value used for the sensorPosOffset for ideal hardware. If the $sensorDetectedPos_{when\ moving\ in\ negative\ direction}$ and $sensorDetectedPos_{when\ moving\ in\ positive\ direction}$ are symmetrical in relation to parSensorPos, then

$sensorDetectedPos_{when\ moving\ in\ positive\ direction} - sensorDetectedPos_{when\ moving\ in\ negative\ direction} = magnetPlateLength - ¼\ polePitch$

$sensorPosOffset > 1/8\ polePitch$

Adding a safety margin of 1/8 polePitch, sensorPosOffset becomes

$sensorPosOffset = 1/8\ polePitch + 1/8\ polePitch = ¼\ polePitch$

which is the initially assumed value.

Configuration of sensorSwitchMargin

The sensorSwitchMargin is a parameter in the LMS configuration XML file that the user can set. The sensor switch margin is needed to check if the sensor position, when a sensor is switching from detected to undetected, is outside the detected range. The detected range is defined by a lower and upper sensor switch margin. Depending on the movement direction, the sensor undetected position is verified with a sensor switch limit.

- When moving in positive direction
  - sensor undetected position > upper sensor switch limit
- When moving in negative direction
  - sensor undetected position < lower sensor switch limit

sensorSwitchMargin depends on the detected and undetected sensor positions when a carrier moves above a sensor. The precondition that a sensor must detect approximately (n - ¼)*polePitch of a carrier, with n = number of pole pairs of one carrier, must be met. The parameters that affect this precondition are sensorDetectCrit and sensorDetectHist, see tab. 3-5 "Sensor parameters" on page 45.

The lower and upper sensor switch limits are defined by the `sensorSwitchMargin`, based on `minSensorPos` and `maxSensorPos`.

`Lower sensor switch limit = minSensorPos + sensorSwitchMargin`

`Upper sensor switch limit = maxSensorPos - sensorSwitchMargin`

You must initialize `minSensorPos` and `maxSensorPos`. To avoid sensor errors, initialize the `sensorPosOffset` and `sensorSwitchMargin` with values for ideal hardware.

`sensorPosOffset = ¼ polePitch`

`sensorSwitchMargin = ½ polePitch`

After you have set these values you can determine the best `sensorSwitchMargin` value. The `sensorSwitchMargin` has two criteria. These criteria apply for carriers moving in positive direction, respectively negative direction. The `sensorSwitchMargin` must meet both criteria.

1. `sensorSwitchMargin` criterion for carriers moving in positive direction

   The criterion to meet after the sensor becomes undetected is (see fig. 3-16 "Sensor switch margin, movement in positive direction" on page 43)

   $sensorUndetectedPos_{when\ moving\ in\ positive\ direction}$ > upper sensor switch limit

   The general criterion is:

   $sensorUndetectedPos_{when\ moving\ in\ positive\ direction} > maxSensorPos - sensorSwitchMargin$

   $sensorSwitchMargin > maxSensorPos - sensorUndetectedPos_{when\ moving\ in\ positive\ direction}$

   Adding a safety margin of ¼ `polePitch`

   $sensorSwitchMargin = maxSensorPos - sensorUndetectedPos_{when\ moving\ in\ positive\ direction} + ¼\ polePitch$

   Remember, if `sensorSwitchMargin` < 0 then `sensorSwitchMargin` = 0, since `sensorSwitchMargin` ≥ 0.

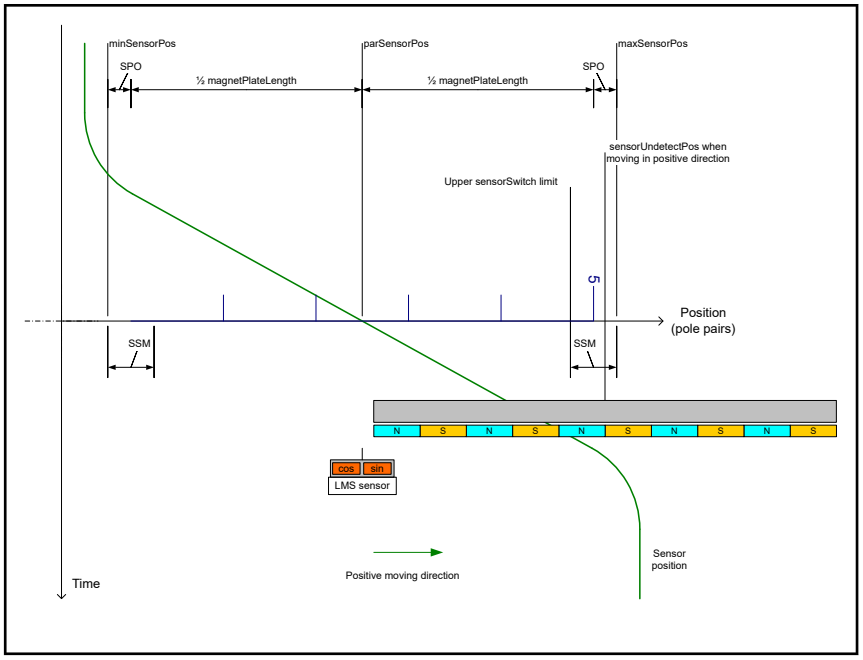


Fig. 3-16: Sensor switch margin, movement in positive direction

2. `sensorSwitchMargin` criterion for carriers moving in negative direction

   The criterion to meet after the sensor becomes undetected is (see fig. 3-17 "Sensor switch margin, movement in negative direction" on page 44)

   $sensorUndetectedPos_{\text{when moving in negative direction}}$ < Lower sensor switch limit

   The general criterion is:

   $sensorUndetectedPos_{\text{when moving in negative direction}} < minSensorPos + sensorSwitchMargin$

   $sensorSwitchMargin > sensorUndetectedPos_{\text{when moving in positive direction}} - minSensorPos$

   Adding a safety margin of ¼ `polePitch`

   $sensorSwitchMargin = sensorUndetectedPos_{\text{when moving in positive direction}} - minSensorPos + ¼\ polePitch$

   Remember, if `sensorSwitchMargin` < 0 then `sensorSwitchMargin` = 0, since `sensorSwitchMargin` ≥ 0.

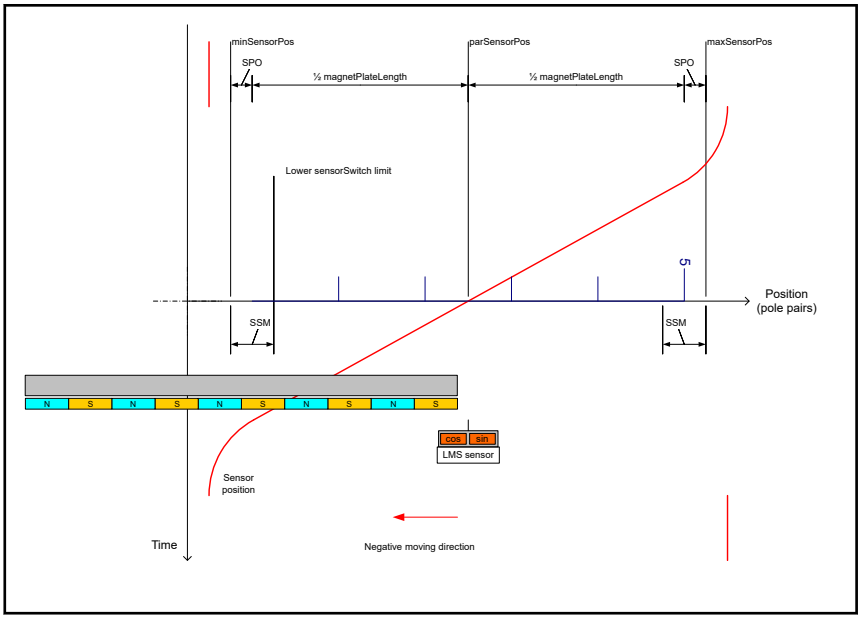


Fig. 3-17: Sensor switch margin, movement in negative direction

You should measure

- `sensorUndetectedPos`$_{\text{when moving in positive direction}}$

  The `sensorUndetectedPos`$_{\text{when moving in positive direction}}$, is the position read by the sensor to change from detected to undetected when a carrier is moving of the sensor in positive direction.

- `sensorUndetectedPos`$_{\text{when moving in negative direction}}$

  The `sensorUndetectedPos`$_{\text{when moving in negative direction}}$, is the position read by the sensor to change from detected to undetected when a carrier is moving of the sensor in negative direction.

Calculate the `sensorSwitchMargin` for carriers moving in positive, respectively negative direction as follow:

`sensorSensorSwitchMargin_P = maxSensorPos - sensorUndetectedPos`$_{\text{when moving in positive direction}}$ `+ ¼ polePitch`

`sensorSensorSwitchMargin_N = sensorUndetectedPos`$_{\text{when moving in positive direction}}$ `- minSensorPos + ¼ polePitch`

`sensorSwitchMargin = maximum of sensorSensorSwitchMargin_P and sensorSensorSwitchMargin_N`

This explains the initial value used for the `sensorSwitchMargin` for ideal hardware. If the `sensorUndetectedPos`$_{\text{when moving in positive direction}}$ and `sensorUndetectedPos`$_{\text{when moving in negative direction}}$ are symmetrical in relation to `parSensorPos`, then

`sensorUndetectedPos`$_{\text{when moving in positive direction}}$ `-` `sensorUndetectedPos`$_{\text{when moving in negative direction}}$ `= magnetPlateLength - ½ polePitch`

`sensorSensorSwitchMargin > ¼ polePitch`

Adding a safety margin of ¼ `polePitch`

`sensorSensorSwitchMargin = ¼ polePitch + ¼ polePitch = ½ polePitch`

Which is the initially assumed value.

3.2.6 Sensor parameters

Sensor parameters are specified for each sensor of the system.

Parameter	Description
sensorPos	Position of the sensor relative to the coil position (see fig. 3-8 "CoilPosition" on page 34)
cmCommutationAngle	The magnet field of the coil must be in proper phase with respect to the magnet field on the carrier to optimize external force. This condition is met by measuring the phase of the magnet field of the carrier with the sensors and adding the cmCommutationAngle to this phase.
interpSinOffset	Offset of the sine signal in Volts (see fig. 3-18 "Sensor signal offsets (sine is red, cosine is blue)" on page 46)
interpCosOffset	Offset of the cosine signal in Volts (see fig. 3-18 "Sensor signal offsets (sine is red, cosine is blue)" on page 46)
interpSinAmplitude	Amplitude of the sine signal in Volts (see fig. 3-18 "Sensor signal offsets (sine is red, cosine is blue)" on page 46)
interpCosAmplitude	Amplitude of the cosine signal in Volts (see fig. 3-18 "Sensor signal offsets (sine is red, cosine is blue)" on page 46)
interpSinCosPhaseCorrection	Phase lag of sine relative to "cosine phase minus pi/2", in radians (see fig. 3-18 "Sensor signal offsets (sine is red, cosine is blue)" on page 46)
sensorDetectCrit	Amplitude of $\sqrt{\cos^2 + \sin^2}$ for detection. As the input signals are normalized to an amplitude of 1, a value between 0 and 1 can be chosen (see fig. 3-19 "Sensor signal criteria" on page 47)
sensorDetectHyst	Width of the symmetrical hysteresis band around sensorDetectCrit (see fig. 3-19 "Sensor signal criteria" on page 47)
sensorOrientation	The direction type (positive or negative) of the MR sensor
filterTimeConst ⓘ	Time constant for low pass filter, in seconds.
filterCriterion ⓘ	When the displacement filtered by the low pass filter is higher than the filter criterion, the carrier is considered as moving. This value is in position unit per sample.
maxSpeedCriterion ⓘ	The maximum displacement in position unit per sample.

Parameter	Description
maxSpeedDuration ①	The carrier is also considered moving when during the last maxSpeedDuration number of samples, one or more displacements exceeded maxSpeedCriterion.
sine folder	Contains the following entries • analogSlotId : defines the drive module slot to which the signal is sent • analogInputNr : defines the analog input (0 - 1 for NY4120/10, 0 - 3 for NY4150/10) to which the signal is sent • muxId : defines the input of the multiplexer on the I/O backplane of the NY4074 or NY4079 to which the signal is sent
cosine folder	Contains the following entries • analogSlotId : defines the drive module slot to which the signal is sent • analogInputNr : defines the analog input (0 - 1 for NY4120/10, 0 - 3 for NY4150/10) to which the signal is sent • muxId : defines the input of the multiplexer on the I/O backplane of the NY4074 or NY4079 to which the signal is sent

① The parameters marked with ① are used to determine whether a carrier is moving or not moving, used during homing.

Tab. 3-5: Sensor parameters

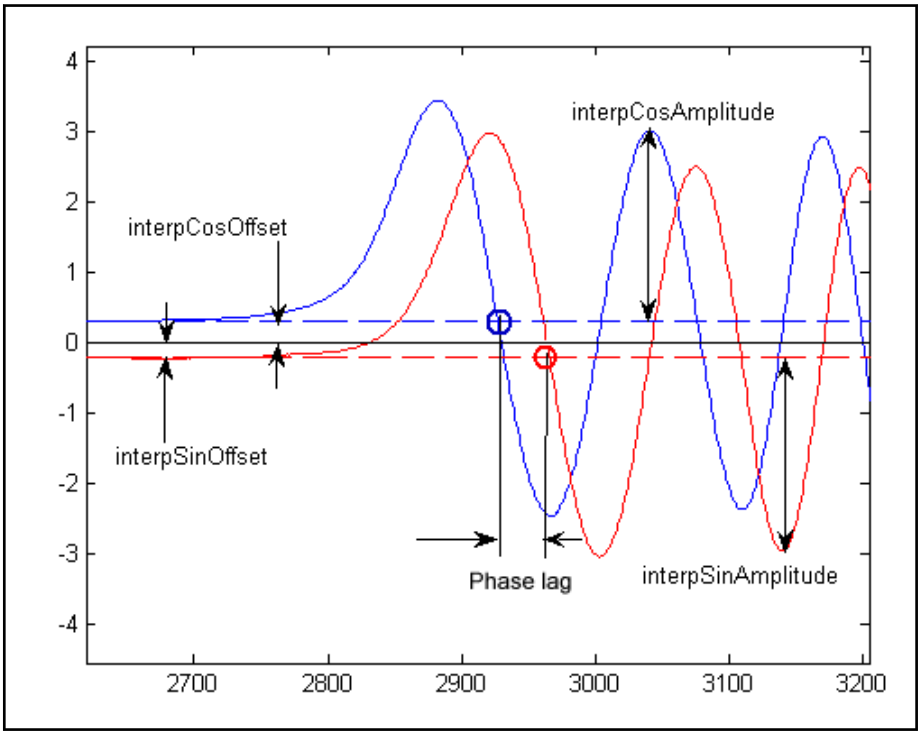


Fig. 3-18: Sensor signal offsets (sine is red, cosine is blue)

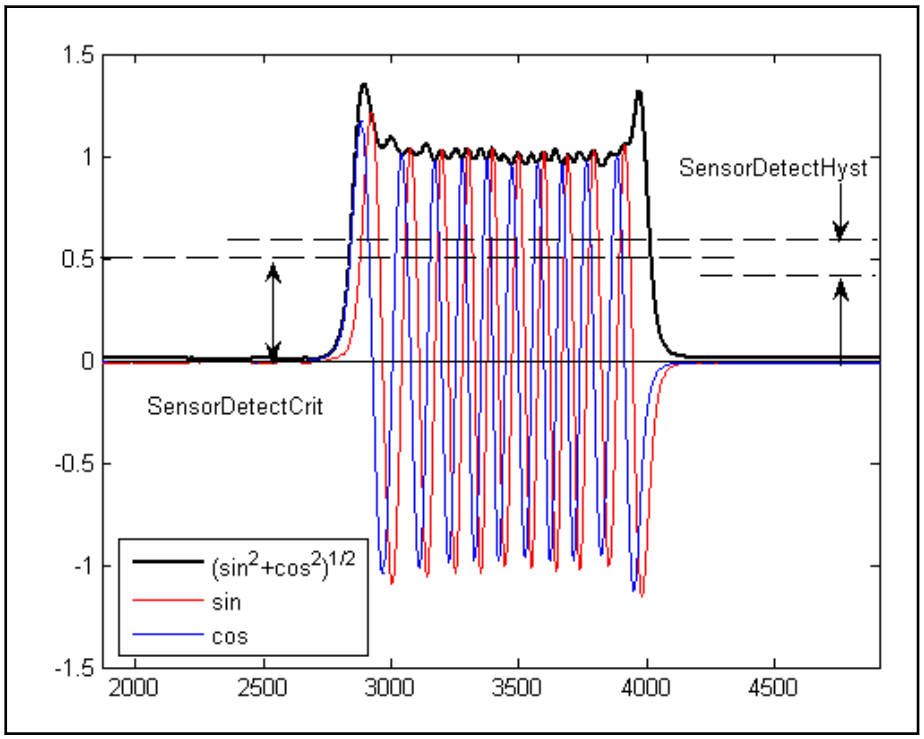


Fig. 3-19: Sensor signal criteria

4 CM - Carrier Management

4.1 Introduction

Carrier Management (CM) contains the functions available to drive the LMS. The functions are explained in chapter 4.7 "Carrier Management functions" on page 58. For a correct use of the CM functions, it is necessary to know the system state (chapter 4.2 "System state" on page 49), client state (chapter 4.3 "Client state" on page 50), the track states (chapter 4.4 "Track states" on page 52), carrier states (chapter 4.5 "Carrier states" on page 54), and carrier controller states (chapter 4.6 "Carrier controller states" on page 57).

4.2 System state

CM service starts automatically on the PC. CM service runs in the background. After the LMS client has a connection, the client can enroll for CM events. CM service supports the following events.

- State changes of the LMS system.
- State changes of the track and RTLC changes to the track layout.
- State changes of the carrier and moving carriers added or removed.
- State changes of the carrier controller.
- Error and warning notifications at the track level.

CM service uses certain TCP port numbers. See for detailed information chapter 2.4 "LMS software" on page 16.

The system state is initially `CM_SYSTEM_POWERUP`. After a client called `CmInitialize`, the system state changes to `CM_SYSTEM_INITIALIZATION_IN_PROGRESS`. When the system is initialized, the system state changes to `CM_SYSTEM_INITIALIZED`. Likewise, after a client called `CmShutdown`, the system state changes to `CM_SYSTEM_SHUTDOWN_IN_PROGRESS`. When the system is shutdown, the system state changes to `CM_SYSTEM_POWERUP`. See fig. 4-1 "State transition diagram for LMS system states" on page 49. A client must connect to CM service before the client can issue any CM command. CM service allows up to `CM_MAX_NR_OF_CLIENTS` (15) connections.

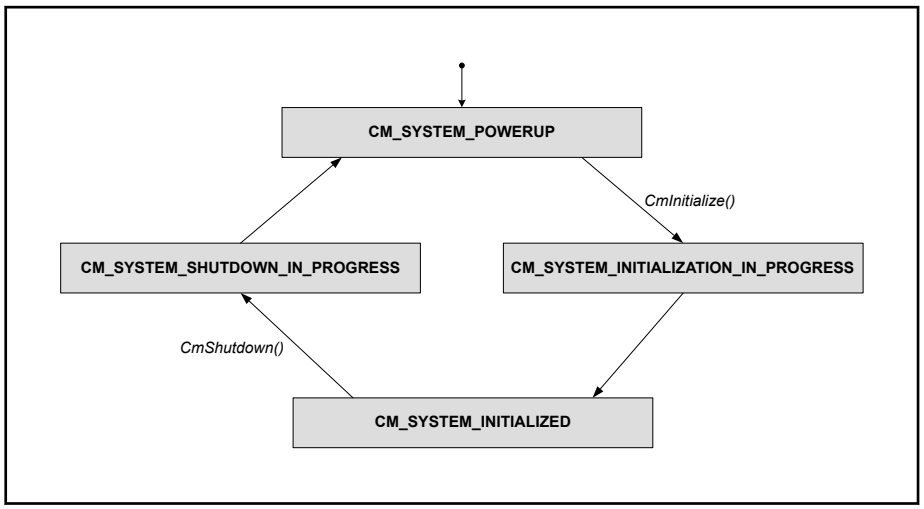


Fig. 4-1: State transition diagram for LMS system states

Initialization of the LMS is only allowed when the system state is CM_SYSTEM_POWERUP. This implies that between two system initializations a system shutdown must have been executed.

4.3 Client state

4.3.1 Introduction

When an LMS client application starts, its initial client state is CM_CLIENT_DISCONNECTED. The client cannot execute CM commands in this state, except the commands CmConnect and CmDisconnect. After the client called the function CmConnect, the client connects to CM, necessary internal connections are established, and the client state changes to CM_CLIENT_CONNECTED. When the client calls the function CmDisconnect the client is disconnected from CM, the necessary internal connections are removed, and the client state returns to CM_CLIENT_DISCONNECTED. See fig. 4-2 "STD for LMS client states" on page 50 for the client states. The functions are described in chapter 4.7.2 "Connect and disconnect an application" on page 61.

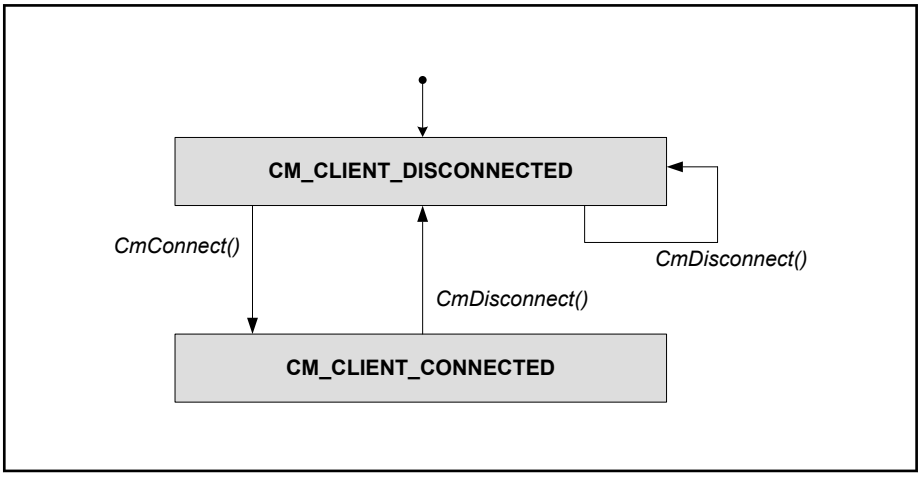


Fig. 4-2: STD for LMS client states

As every LMS client loads its own instance of N4kCm.dll and is only interested in its own state, the client state is not maintained in shared memory, but internally in N4kCm.dll. If the execution of a CM command causes a state change (of the track, carrier, etc.), this change is communicated to CM service, because CM service generates all CM events. CM service takes care that all interested clients are notified of the state change. CM service uses certain TCP port numbers. See for detailed information chapter 2.4 "LMS software" on page 16.

Every client is allowed to initialize the LMS with the function CmInitialize. The function CmInitialize is the only CM function that is allowed by any client when the system state is CM_SYSTEM_POWERUP and the client state is CM_CLIENT_CONNECTED.

Every client is allowed to shutdown the system with the function CmShutdown, even if the client that initiates the shutdown is not the client that initialized the system. However, the system state must be initialized and the client must be connected to execute CmShutdown. The system state changes to CM_SYSTEM_POWERUP after system shutdown and the shared memory contents is reset. The function CmShutdown can only be called in

the state CM_SYSTEM_INITIALIZED and results in the system state transition to CM_SYSTEM_POWERUP.

4.3.2 Application start

Every client must call CmConnect before any other CM function can be called. The following actions are executed when CmConnect is called.

- The internal connection with CM service is established. CM service uses certain TCP port numbers. See for detailed information chapter 2.4 "LMS software" on page 16.
- The client gets access to the data in the shared memory.
- The client state changes to CM_CLIENT_CONNECTED.

If any of these actions fails, the client state remains CM_CLIENT_DISCONNECTED and an error is returned to the application. All successfully completed actions are rolled back.

A typical sequence of an application start is the following pseudo code.

Program:

```
initializeApplication()                  // initialize the non-LMS part
. . .
. . .
CmConnect();                             // connect to LMS
CmReadSystemVariable(CM_VAR_SYSTEM_STATE, &systemState);
if (systemState != CM_SYSTEM_INITIALIZED)
   {
      CmInitialize(...)
      . . .
   }
. . .
```

Note that only the first client connection must execute the function CmInitialize. Subsequent clients only need to execute the function CmConnect.

4.3.3 Application stop

Several clients can be connected to the LMS. All allocated resources must be released before a client terminates. A client can release the allocated resources with the function CmDisconnect. The following actions are executed when CmDisconnect is called.

- The connection with CM service is closed.
- The access to the data in the shared memory is disabled.
- The client state changes to CM_CLIENT_DISCONNECTED.

If any of these actions fails, the client state still makes the transition to CM_CLIENT_DISCONNECTED, but an error is returned to the application.

The client cannot issue any other CM commands after CmDisconnect is called. This implies that if a shutdown of the system is needed, the shutdown must be done before CmDisconnect is called. No automatic system shutdown is initiated after the last client application has disconnected from CM service.

4.4 Track states

For each track of the LMS, CM maintains the state of the track. The main goal of the track state administration is to check whether commands can be accepted by the LMS.

The states in which a track can be are shown in fig. 4-3 "State Transition Diagram for tracks in CM" on page 52. The state diagram is entered in the state `CM_TRACK_POWERUP`, and the track state changes to `CM_TRACK_IDLE` after a call to the initialization function. The track state changes back to `CM_TRACK_POWERUP` after a call of the shutdown function from any state. See chapter 4.7 "Carrier Management functions" on page 58.

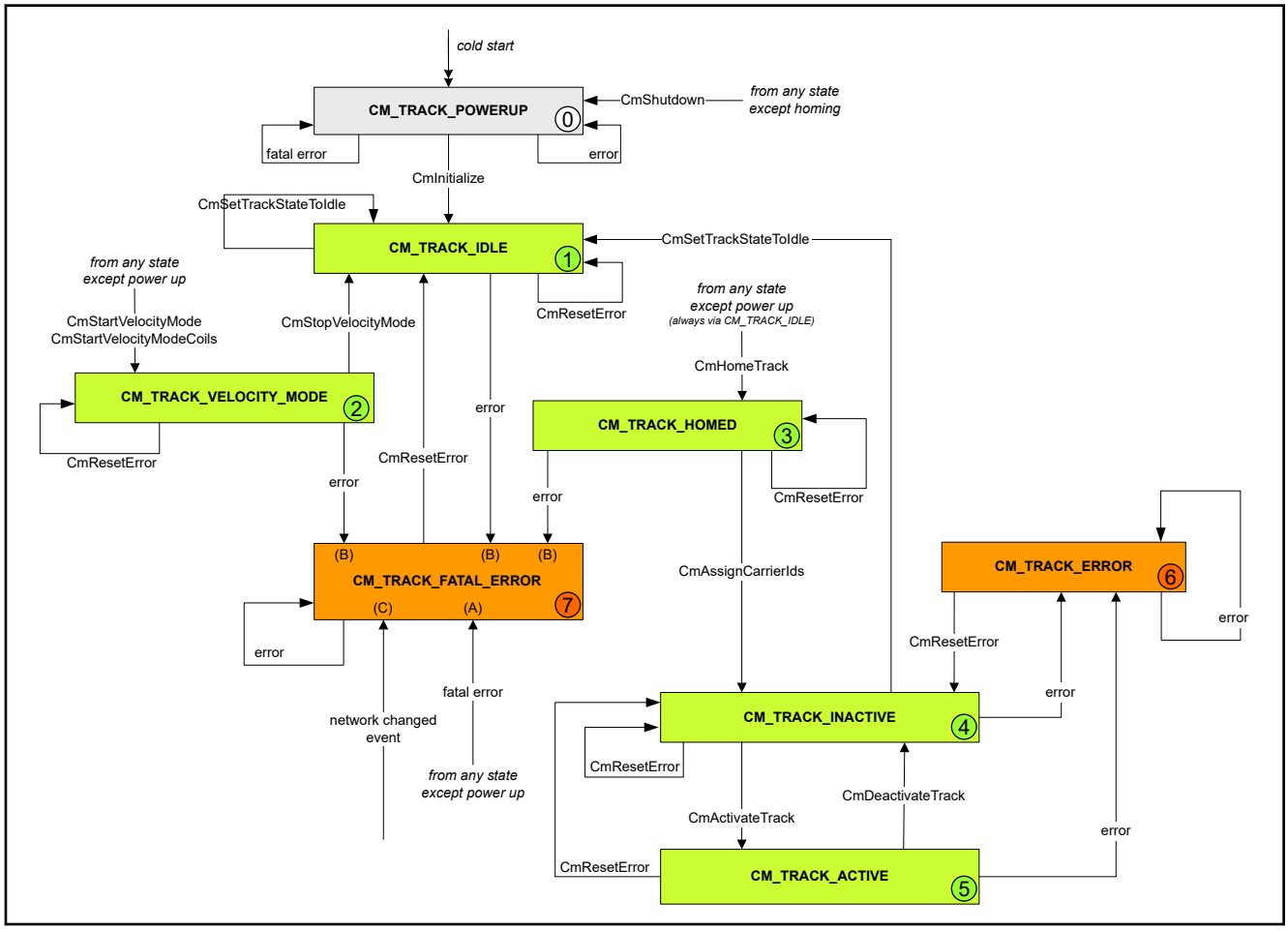


Fig. 4-3: State Transition Diagram for tracks in CM

The number in the circle is the enumerated value of the track state. The track administration has the following states.

State	Description
CM_TRACK_POWERUP	Initial state, also entered after `CmShutdown`.
CM_TRACK_IDLE	The track is defined: CM is initialized; all coils on the track are known and initialized.
CM_TRACK_VELOCITY_MODE	Each coil on the track operates in velocity mode. Carriers are moving with constant velocity.

State	Description
CM_TRACK_HOMED	The track is homed, that is, all coils know the absolute position of the carrier(s) in their sensor range, if any.
CM_TRACK_INACTIVE	The track is homed, and all carriers on the track have an ID.
CM_TRACK_ACTIVE	Each coil on the track operates in closed loop state when a carrier is in its coil control range. Moving carriers are moving in accordance with setpoint profiles, as requested by the application.
CM_TRACK_ERROR	An error has been detected in the state `CM_TRACK_INACTIVE` or `CM_TRACK_ACTIVE` by one or more of the coils on the track. All coils on the track complete the current movement, if any, as far as possible. New movement commands for carriers on the track are not accepted, except "stop" and "change feed override".
CM_TRACK_FATAL_ERROR	• Case A. A fatal error has been detected by one or more of the coils on the track. • Case B. An error has been detected in the state `CM_TRACK_IDLE`, `CM_TRACK_HOMED`, or `CM_TRACK_VELOCITY_MODE`. All coils on the track complete the current movement, if any, as far as possible. New movement commands for carriers on the track are not accepted, except "stop" and "change feed override". A fatal error requires the track to be homed again. • Case C. A network change event is received and processed. The LMS must be shutdown and initialized again.

Tab. 4-1: States in the track state transition diagram

The CM application can enroll for the event `SYS_EV_NETWORK_EVENT` with the function `SysDefineEventEnrolment`. When the network event `SYS_EV_NETWORK_EVENT` occurs, only with the following conditions, all moving carriers are stopped and all tracks go to the `CM_TRACK_FATAL_ERROR` state.

- `NYCE_NETWORK_EVENT_NODE_CONFIG_CHANGED`
- `NYCE_NETWORK_EVENT_NETWORK_CONFIG_CHANGED`
- `NYCE_NETWORK_EVENT_NETWORK_OUT_OF_SYNC`

To recover the application after an event `SYS_EV_NETWORK_EVENT` has occurred, you must shut down and initialize the LMS to re-establish synchronization between the nodes. CM movements, for example `CmMoveToPosition`, cannot function if nodes are not synchronized.

4.5 Carrier states

The carrier state transition diagram is shown in fig. 4-4 "State Transition Diagram for carriers in CM" on page 55. The state diagram is entered at the `CM_CARRIER_POWERUP` state when the track on which the carrier is situated is activated, or when the carrier is added to a track with state `CM_TRACK_ACTIVE`. The carrier state changes to `CM_CARRIER_POWERUP` when the carrier is removed from the track or the track state is no longer `CM_TRACK_ACTIVE` (shutdown, deactivate, error or fatal error; refer to the track state transition diagram).

Refer to the relevant functional descriptions for more details about the state transitions. Restrictions with respect to the usability of functions in different states are given in the function descriptions of chapter 4.7 "Carrier Management functions" on page 58.

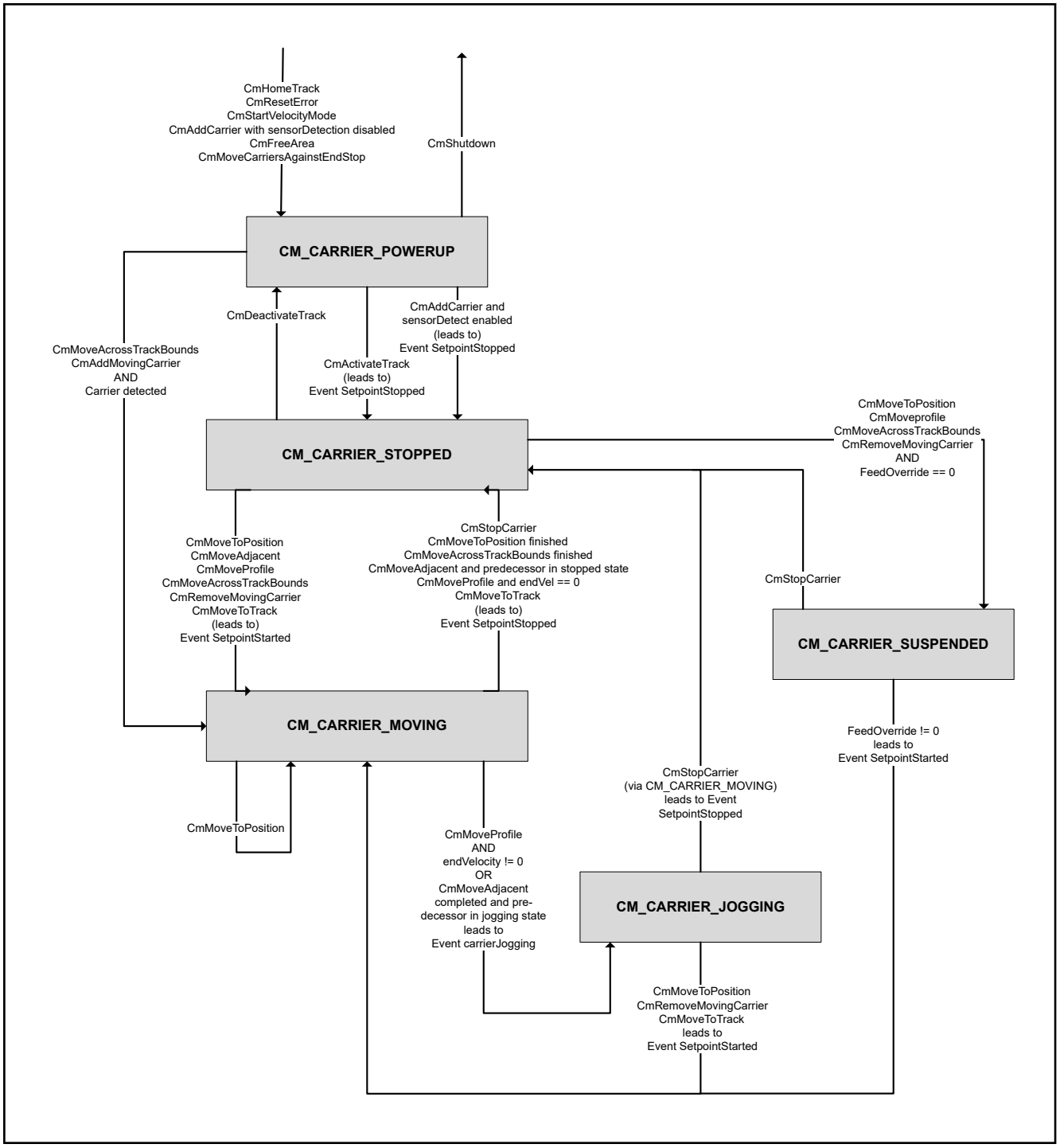


Fig. 4-4: State Transition Diagram for carriers in CM

The application can use the function `CmSynchronize` or the state of a carrier to check whether the carrier has completed its setpoint profile. The application can also enroll for an event to receive a notification when a profile is completed. Starting a position controlled movement is only possible if the track, on which the carrier is moving, is in the `CM_TRACK_ACTIVE` state. Normal completion of a setpoint is only guaranteed if the track is in the `CM_TRACK_ACTIVE` state.

In other track states, except the track error states, position controlled carrier movements are not possible.

In the track error states, CM_TRACK_ERROR and CM_TRACK_FATAL_ERROR, a carrier can only complete its current movement or execute a stop profile. However, ready detection implying that the carrier stopped in closed loop or is moving at constant velocity in closed loop is not always possible. Namely, it is not sure that the coil above which the carrier is moving or has stopped, operates properly. The carrier may stop in open loop above this coil, or continue in closed loop when it reaches the next coil. Therefore, the carrier state is not guaranteed in the track states CM_TRACK_ERROR and CM_TRACK_FATAL_ERROR.

For these reasons, the carrier state can only be well defined, when the track along which the carrier is moving, is in the state CM_TRACK_ACTIVE. The carrier can be in one of the following states:

State	Description
CM_CARRIER_POWERUP	A dummy state. This state is assigned when the track state is not CM_TRACK_ACTIVE.
CM_CARRIER_STOPPED	The carrier is stopped after completion of a setpoint profile with end velocity 0 (typically "move to end position", "move along profile" and "stop"; possibly "move adjacent", see chapter 4.7 "Carrier Management functions" on page 58). In addition, the state is entered after track activation and after physical addition of the carrier to a track.
CM_CARRIER_SUSPENDED	When the setpoint profile is not completed and the track feed override is constantly 0, or when a move command is issued while the feed override is 0. Note that feed override cannot be changed during a movement.
CM_CARRIER_JOGGING	When the carrier is moving at constant velocity, not equal to 0, after completion of a setpoint profile with nonzero end velocity (typically "move adjacent", possibly "move profile", see chapter 4.7 "Carrier Management functions" on page 58).
CM_CARRIER_MOVING	When the carrier is moving, in which moving at constant velocity after completion of a setpoint profile is excluded (typically "move adjacent", possibly "move profile", see function descriptions in chapter 4.7 "Carrier Management functions" on page 58).

Tab. 4-2: States in the carrier state transition diagram

When the carrier is moving, you can see in NYCeScope which type of movement is executed. Select the variable SAC_SPG_STATE in the Channels panel and apply the ruler in the Scope Time Domain Plot window to trace the variable. The following states are defined.

- Car MTC - executing carrier movement starting from current position
- Car MTI - executing carrier movement starting from initial position
- Car MEV - executing carrier movement starting from initial position with end velocity

- Car PRF - executing carrier movement starting from current position along profile
- Car STP - executing stop carrier movement
- Car QST - executing quick stop carrier movement

4.6 Carrier controller states

You can use `CmSynchronize`, enroll for events, or the carrier state `CM_CARRIER_STOPPED` to check when the setpoint profile of a movement with end velocity 0 is completed. However, the carrier state `CM_CARRIER_STOPPED` does not necessarily imply that the carrier has reached its actual end position. When the carrier is in the state `CM_CARRIER_STOPPED`, the coil that controls the carrier may still be busy moving the carrier to the actual end position. When that movement is completed, the carrier is settled on the end position. You can enroll for events to synchronize on carrier position settled. Further, the member `controllerState` in the structure `CM_CARRIER_DATA` holds the carrier controller state. You can read the structure `CM_CARRIER_DATA` with the functions `CmReadCarrierData` and `CmReadSingleCarrierData`. See NYCe 4000 Software User Manual, chapter 8, for more information.

The carrier controller can be one of the following states, see also fig. 4-5 "State Transition Diagram for carrier controller states in CM" on page 58.

- `CM_CARRIER_CONTROLLER_MOVING`

  The state `CM_CARRIER_CONTROLLER_MOVING` indicates that the controller generates setpoints to move the carrier to the end position.

- `CM_CARRIER_CONTROLLER_SETTLING`

  The state `CM_CARRIER_CONTROLLER_SETTLING` indicates that the controller is settling the carrier to the end position.

- `CM_CARRIER_CONTROLLER_STABILIZING`

  The state `CM_CARRIER_CONTROLLER_STABILIZING` indicates that the controller is stabilizing the carrier toward the end position.

- `CM_CARRIER_CONTROLLER_STEADY`

  The state `CM_CARRIER_CONTROLLER_STEADY` indicates that the carrier is settled on the end position.

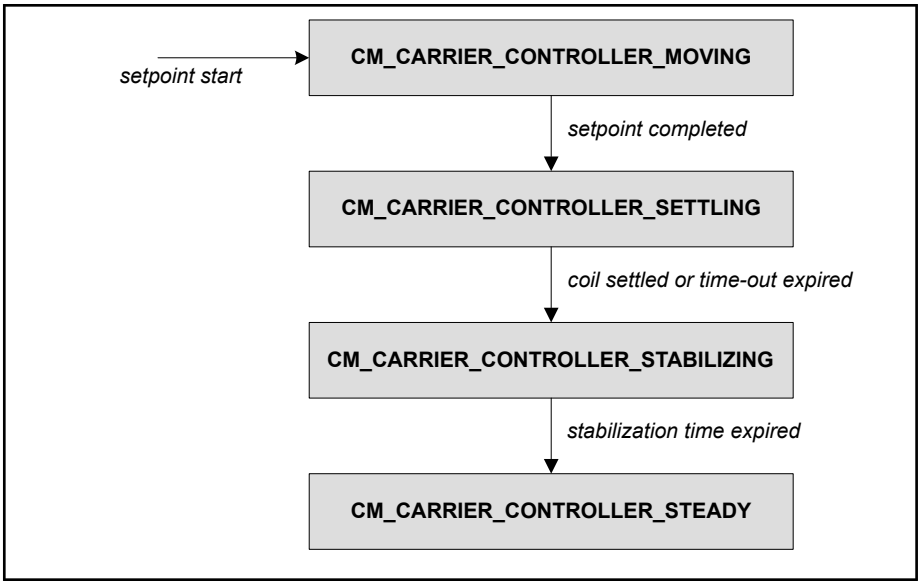


Fig. 4-5: State Transition Diagram for carrier controller states in CM

The carrier controller state is initialized to `CM_CARRIER_CONTROLLER_STEADY` when the track on which the carrier is situated is activated.

The event `CM_EV_CARRIER_CONTROLLER_MOVING_EN` is always generated. The event `CM_EV_CARRIER_CONTROLLER_STEADY_STATE_EN` is generated for every movement, except when a new movement is started. The events `CM_EV_CARRIER_CONTROLLER_SETTLING_EN` and `CM_EV_CARRIER_CONTROLLER_STABILIZING_EN` can always be missed, depending on parameter settings and control performance.

The events `CM_EV_CARRIER_CONTROLLER_xxx_LE` depend on the last known carrier controller state.

4.7 Carrier Management functions

4.7.1 Introduction

CM contains functions for the actions described in the following table. See the LMS Software Reference Manual for a detailed description.

All CM functions that need time to complete the requested action block the calling application, except the functions `CmHomeTrack` and `CmMoveCarriersAgainstEndstop`. These functions return immediately and a separate thread is started to execute the actual operation. The application can enroll for a specific event, so that the application receives a notification that the requested action of a called CM function is completed. The function `CmSynchronize` is available to synchronize on several asynchronous operations.

Action	Function(s)	More info
Connect and disconnect an application	CmConnect CmDisconnect	4.7.2
Initializing CM	CmInitialize	4.7.3

Action	Function(s)	More info
Activating a track	CmActivateTrack	4.7.4
Deactivating a track	CmDeactivateTrack	4.7.5
Set track to idle state	CmSetTrackStateToIdle	4.7.6
Runtime Track Layout Changes	CmAddFirstCoil CmAddLastCoil CmRemoveFirstCoil CmRemoveLastCoil CmSelectFirstCoilAlternative CmSelectLastCoilAlternative	4.7.7
Homing a track	CmHomeTrack CmWriteHomePars CmReadHomePars CmEnableBumper CmReadBumperPars	4.7.8
Get number of carriers and IDs	CmGetCarriersOnTrack	4.7.9
Carrier ID / carrier index conversion	CmGetCarrierId CmGetCarrierIndex	4.7.10
Get number of tracks	CmGetTracks	4.7.11
Get axis identifier of a coil	CmGetAxisIdOfCoil	4.7.12
Detect carrier presence in an area, create an area without carriers	CmCheckArea CmFreeArea	4.7.13
Carrier Position Adjustment	CmDownloadCpaTable CmLinkCpaTable CmUnlinkCpaTable CmEnableCpa	4.7.14
Assigning a carrier ID	CmAssignCarrierIds	4.7.15
Change the ID of a carrier	CmChangeCarrierId	4.7.16
Read and write the carrier parameters	CmReadCarrierPars CmWriteCarrierPars	4.7.17
Moving carriers against an end stop	CmMoveCarriersAgainstEndstop	4.7.18
Moving a carrier to a position	CmMoveToPosition	4.7.19
Moving a carrier adjacent to its predecessor	CmMoveAdjacent	4.7.20
Moving a carrier along a profile	CmMoveProfile	4.7.21
Moving a carrier at constant velocity	CmJog	4.7.22
Stopping a carrier	CmStopCarrier	4.7.23

Action	Function(s)	More info
Synchronized start of multiple carriers	CmDefineSyncGroup CmGetSyncGroup CmStartSyncGroup CmStopCarriers CmDeleteSyncGroup	4.7.24
Using feed override	CmSetFeedOverride	4.7.25
Removing a carrier Adding a carrier	CmRemoveCarrier CmAddCarrier	4.7.26
Move a carrier from track to track	CmMoveToTrack	4.7.27
Removing a moving carrier from an LMS Adding a moving carrier to an LMS Cancelling addition of a moving carrier to an LMS	CmRemoveMovingCarrier CmAddMovingCarrier CmCancelAddMovingCarrier	4.7.28
Moving a carrier at non-constant velocity across tracks Cancel a not yet started moving across tracks function	CmMoveAcrossTrackBounds CmCancelMoveAcrossTrackBounds	4.7.29
Moving a carrier with collision avoidance	CmCollAvoidMoveToPosition CmCollAvoidStopCarrier CmCollAvoidAddMovingCarrier CmCollAvoidCancelAddMovingCarrier CmCollAvoidRemoveMovingCarrier CmCollAvoidMoveBetweenTracks CmCollAvoidAddCarrierWithProfile CmCollAvoidCancelAddCarrierWithProfile CmCollAvoidRemoveCarrierWithProfile CmCollAvoidSynchronize CmCollAvoidReadCarrierState	4.7.30
Camming a carrier on a track	CmEcgLink CmEcgSetMasterSign CmEcgSetMasterModuloValue CmEcgSetMasterPos CmEcgLockCam CmEcgResetMasterModuloValue CmEcgUnlink	4.7.31
Markers	CmDefineSingleShotMarker CmDeleteSingleShotMarkers CmDefinePermanentMarker CmDeletePermanentMarker	4.7.32

Action	Function(s)	More info
Reading carrier data	CmReadSingleCarrierData CmReadCarrierData	4.7.33
Resetting errors	CmResetError	4.7.34
Clearing warnings	CmClearWarning	4.7.35
Handling special controller parameters for an area	CmSelectSpecialControllerPars CmWriteSpecialControllerPars CmReadSpecialControllerPars	4.7.36
Controlling carriers in velocity mode	CmStartVelocityMode CmStartVelocityModeCoils CmStopVelocityMode	4.7.37
Enable/disable sensor detection	CmEnableSensorDetection	4.7.38
Enable/disable start triggers	CmEnableStartTrigger	4.7.39
Synchronize on an asynchonous operation	CmSynchronize	4.7.40
Event handling	CmDefineEventEnrolment CmDeleteEventEnrolment	4.7.41
Reading a variable	CmReadSystemVariable CmReadTrackVariable CmReadCarrierVariable CmReadCoilVariable	4.7.42
Reading configuration parameters	CmReadSystemParameter CmReadTrackParameter CmReadPhysicalCoilParameter CmReadLogicalCoilParameter CmReadSensorParameter CmReadLogicalCoilName	4.7.43
Logging	CmSetLogFunction CmResetLogFunction	4.7.45
Shutting down the LMS	CmShutdown	4.7.46

Tab. 4-3: CM functions

4.7.2 Connect and disconnect an application

Function and explanation

A user application can only call CM commands after the application has become a client of CM. With the function `CmConnect` a user application can connect to CM. The function has no input or output parameters. After the successful execution of the `CmConnect` command the client state changes to `CM_CLIENT_CONNECTED`. If the application is already connected to CM, a warning is returned.

When a user application no longer uses the LMS, the application can disconnect from CM with the function `CmDisconnect`. The function has no

input or output parameters. The client state changes to CM_CLIENT_DISCONNECTED.

The error CM_ERR_HOME_TRACK_EXECUTING is returned if the client, that started a home procedure, calls CmDisconnect or is stopped before the home operation has finished. The client remains connected to CM. Use the function CmResetError to recover from this situation. If the application calls CmDisconnect but is already disconnected from CM, a warning is returned. The user application can no longer call CM commands, except CmConnect and CmDisconnect.

4.7.3 Initializing Carrier Management

Function and explanation

To initialize the Carrier Management the function CmInitialize is available. The input for this function is a pointer to a machine data file. Data, required for the initialization, are retrieved from this file and the axis (coil) initialization files. It is checked if there are any disabled sensors. If so, these sensors will be enabled again. The function CmInitialize is only allowed when the system state is CM_SYSTEM_POWERUP. After successful completion of the function CmInitialize the system state changes to CM_SYSTEM_INITIALIZED, and the state of all tracks in the system changes to CM_TRACK_IDLE.

During the execution of the CmInitialize function all possible existing defined track and carrier event enrolments are deleted. The entire operation of CmInitialize can take some time. All administrative actions are finished when control is returned to the application.

After CmInitialize, all axes of a track are in an error group. CmInitialize checks the defined minimum error handler and sets for all SAC_AX_ERR_PEER_X errors the minimum error handler to SAC_ERRH_SMOOTH_STOP (if the error handler was lower), except SAC_AX_ERR_PEER_WARNING. It is the responsibility of the application to set an appropriate peer error handler in the axis parameter XML file. "Quick stop open loop" is recommended. The minimum and default peer error handlers are as specified in the NYCe 4000 Software User Manual. See chapter 5 "Error handling and recovery" on page 155 in this manual for LMS specific errors.

4.7.4 Activating a track

Function

Track activation is realized by the function CmActivateTrack. The input for this function is the track ID.

Explanation

Activation of a track implies that for all coils on the track the power is enabled. In addition, for all coils which have one or two carriers in their coil control range, the control loop is closed. Other coils are put in a mode, in which they will close the control loop and eventually start setpoint generation as soon as a carrier appears in their coil control range. CmActivateTrack checks that all carriers are in the control range of a coil of the track. If this is not the case, the error CM_ERR_CARRIER_POSITION_ERROR is returned.

Activation of a track is only allowed if the track is in the state CM_TRACK_INACTIVE. The track state changes to CM_TRACK_ACTIVE and all carriers on the track are in the CM_CARRIER_STOPPED state.

4.7.5 Deactivating a track

Function

Track deactivation is realized by the function CmDeactivateTrack. The input for this function is the track ID.

Explanation

Deactivation of a track implies that for all coils on the track the control loop is opened and the power is disabled. Coils will no longer take action when a carrier appears in their coil control range.

Track deactivation is only possible if the track is in the state CM_TRACK_ACTIVE. After the track has been deactivated, the track state changes to CM_TRACK_INACTIVE and the carrier state changes to CM_CARRIER_POWERUP.

4.7.6 Set track to idle state

Function

With the function CmSetTrackStateToIdle the track state can be changed from CM_TRACK_INACTIVE to CM_TRACK_IDLE. The input for this function is the track ID.

Explanation

The function CmSetTrackStateToIdle is only accepted if the track state is CM_TRACK_INACTIVE or CM_TRACK_IDLE. In all other track states an error is returned. All carriers are removed from the administration. For certain functionality it may be necessary that the track state must be changed to CM_TRACK_IDLE. For example, with Runtime Track Layout Changes you can remove a carrier from one track and add that carrier to another track, but the precondition is that both tracks have the same track state. With the function CmSetTrackStateToIdle you can fulfil this precondition.

4.7.7 Runtime Track Layout Changes

Function

With the Runtime Track Layout Changes functionality you can dynamically add the first or last coil(s) to a track or remove the first or last coil(s) from a track while all tracks in the system remain operational. The track state where coils are added must be identical to the track state where coils are removed. You can also define so-called coil alternatives. Coil alternatives allocate or de-allocate physical coil(s) from a track.

CmAddFirstCoil, CmAddLastCoil

The function CmAddFirstCoil adds the first coil(s) at the beginning of the specified track and updates the begin position of the track. The function CmAddLastCoil adds the last coil(s) at the end of the specified track and updates the end position of the track. The number of coils that are involved is determined from the LMS configuration file. If more than one alternative physical coil is configured, you must select the correct alternative (CmSelectFirstCoilAlternative or CmSelectLastCoilAlternative) before calling CmAddFirstCoil or CmAddLastCoil. If the first or last coil(s) control one or more carriers, these

carriers are added to the track along with the coil(s). The coil(s) to be added and the coil axis or axes may not be in use by any track. The state of the specified track must be CM_TRACK_IDLE or CM_TRACK_ACTIVE. Further, the axis state of the coil(s) must match the state of the track. If the track state is CM_TRACK_IDLE, the axis state of the coil(s) has to be SAC_INACTIVE. If the track state is CM_TRACK_ACTIVE, the axis state of the coil(s) has to be SAC_FREE or SAC_READY. The axis state SAC_FREE means that the axis is not controlling a carrier. The axis state SAC_READY means that the axis is controlling a carrier.

CmRemoveFirstCoil, CmRemoveLastCoil

The function CmRemoveFirstCoil removes the first coil(s) from the beginning of the specified track and updates the begin position of the track. The function CmRemoveLastCoil removes the last coil(s) from the end of the specified track and updates the end position of the track. The number of coils that are involved is determined from the LMS configuration file. If the first or last coil(s) control one or more carriers, these carriers are removed from the track along with the coil(s). The coil(s) to be removed have to be in use by the specified track. The state of the specified track has to be CM_TRACK_IDLE or CM_TRACK_ACTIVE. Further, if the coil(s) control one or more carriers, the state of these carriers has to be CM_CARRIER_STOPPED.

CmSelectFirstCoilAlternative

The function CmSelectFirstCoilAlternative selects specific alternative physical coil(s) for use at the beginning of the track. The number of coils that are involved is determined from the LMS configuration file. The function CmSelectFirstCoilAlternative has to be called before CmAddFirstCoil if more than one alternative exists and you change the currently selected alternative. The involved coil(s) may not be in use by any track. If the involved coil(s) control one or more carriers, these carriers have to be in the state CM_CARRIER_STOPPED. The state of the specified track has to be CM_TRACK_IDLE or CM_TRACK_ACTIVE. When the function CmSelectFirstCoilAlternative is successfully finished, the specifed alternative is selected for use at the beginning of the track. Note that the coil(s) are not yet added to the track. If the RTLC implementation has only one physical coil alternative for all shared coil positions, it is not necessary to use CmSelectFirstCoilAlternative, because there is only one alternative.

CmSelectLastCoilAlternative

The function CmSelectLastCoilAlternative selects specific alternative physical coil(s) for use at the end of the track. The number of coils that are involved is determined from the LMS configuration file. The function CmSelectLastCoilAlternative has to be called before CmAddLastCoil if more than one alternative exists and you change the currently selected alternative. The involved coil(s) may not be in use by any track. If the involved coil(s) control one or more carriers, these carriers have to be in the state CM_CARRIER_STOPPED. The state of the specified track has to be CM_TRACK_IDLE or CM_TRACK_ACTIVE. When the function CmSelectLastCoilAlternative is successfully finished, the specifed alternative is selected for use at the end of the track. Note that the coil(s) are not yet added to the track. If the RTLC implementation has only one physical coill alternative for all shared coil positions, it is not necessary to use CmSelectLastCoilAlternative, because there is only one alternative.

No track (re-)initialization is required after a coil is added to a track or removed from a track. The begin position and end position of a track depend on which coils are added or removed. With NYCeLmsCommissioning you can specify alternative begin and end positions which are used when a coil is added to the track.

Explanation

An LMS track consists of one or more logical coils, but may also consist completely of RTLC coils. A logical coil can only be part of one LMS track. The coil represents in software a logical axis with two or three sensors. One single axis can be configured for coils in different tracks with the Runtime Track Layout Changes functionality. In fig. 4-6 "RTLC between two tracks: single axis, different coils" on page 65 you can see that axisC can be used by track0 when coil2 is added to track0. Likewise, you can remove coil2 from track0 and then add coil0 to track1 so that axisC can be used in track1.

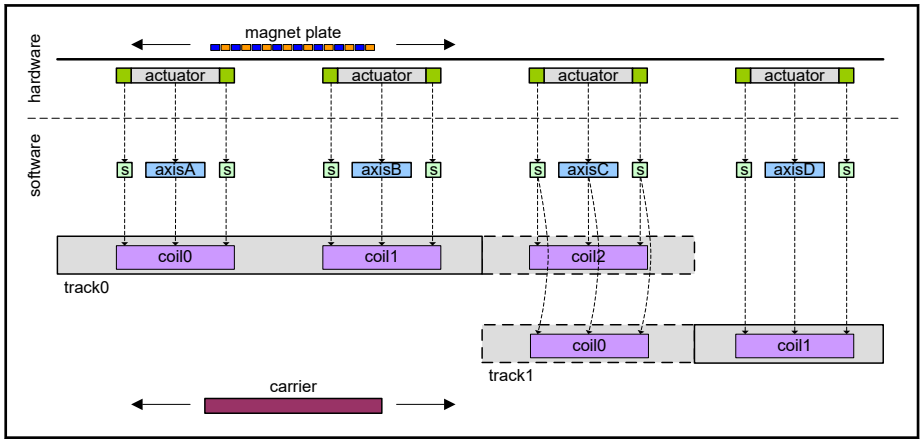


Fig. 4-6: RTLC between two tracks: single axis, different coils

The following actions are needed in the application to move axisC from track0 to track1 (see fig. 4-6 "RTLC between two tracks: single axis, different coils" on page 65).

1. `CmRemoveLastCoil(0)` – remove last coil from track0
2. `CmAddFirstCoil(1)` – add first coil to track1

If the last coil of track0 controls one or more carriers when the coil is removed from track0, these carriers are added to track1 when coil0 is added to that track.

The RTLC functionality also allows a single logical coil of a track to be configured for different axes, see fig. 4-7 "RTLC between two tracks: single coil, different axes" on page 66. Logical coil0 of track 1 can either point to AxisX or AxisY. Each axis can only be part of one track at any given moment. If AxisX must be used by track 1, logical coil1 must be removed from track 0 and logical coil0 (using AxisX) must be added to track 1. A typical application can be a machine where part of the track is shared by two separate tracks, for example two parallel processing tracks with a single return track. The coils of the two separate processing tracks take turns in moving back and forth to the return track.

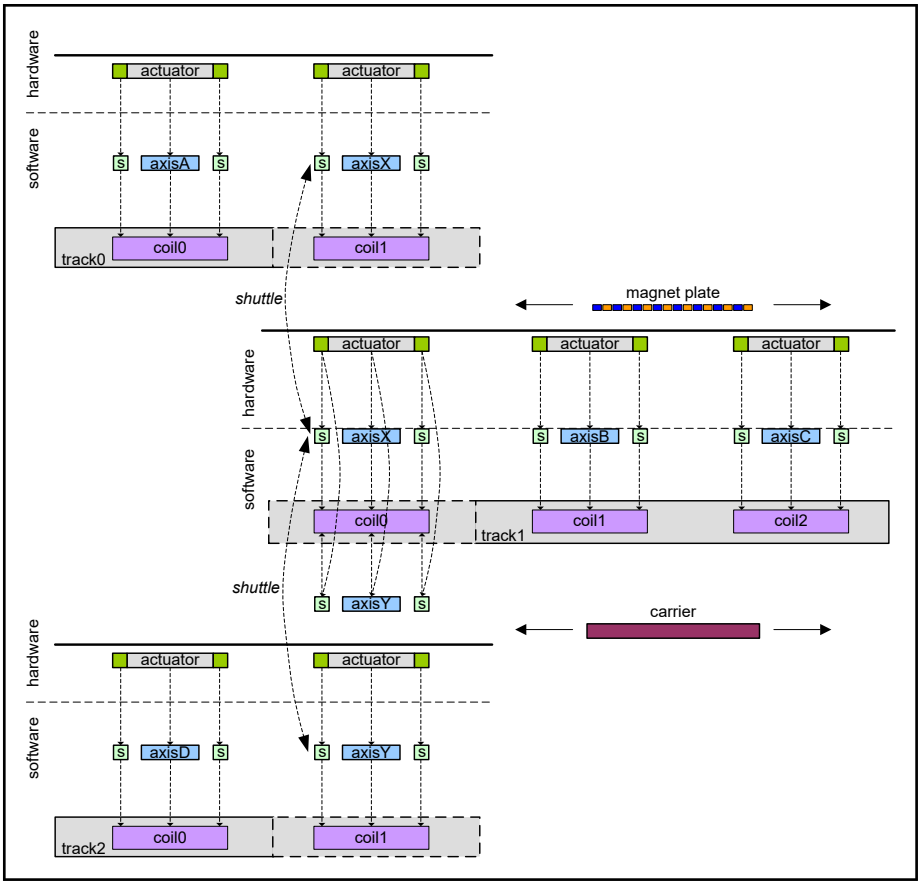


Fig. 4-7: RTLC between two tracks: single coil, different axes

The following actions are needed in the application to replace axisX in track1 with axisY from track2 (two alternatives) (see fig. 4-7 "RTLC between two tracks: single coil, different axes" on page 66).

1. `CmRemoveFirstCoil(1)` – remove first coil from track1
2. `CmRemoveLastCoil(2)` – remove last coil from track2
3. `CmSelectFirstCoilAlternative(1,1)` – select alternative 1 (axisY) for use in track1
4. `CmAddFirstCoil(1)` – add first coil to track1
5. `CmAddLastCoil(0)` – add axisX to track0 (optionally)

Note that it is not necessary to call `CmSelectLastCoilAlternative()` for track0, because there is only one alternative defined for the last coil of track0.

If the last coil of track0 controls one or more carriers when the coil is removed from track0, these carriers are added to track1 when coil0 is added to that track.

The maximum number of axes that can be configured for a single coil is limited to the maximum number of tracks.

If carriers with certain carrier identifiers are locked on track0 by axisX and coil1 is removed from track0, these carriers are also removed from track0. When coil0 is added to track1, these carriers are also added to track1 with the same carrier identifiers. During the configuration change these carriers remain locked by axisX.

Note that when coils with one or more carriers are removed from a track and not yet added to (another) track, the carrier identifiers of those carriers cannot be allocated to other carriers.

An axis can be used by zero or one track at any given moment. If coils share the same axis (with its sensors), the coils must be configured identically in the system XML file, except for the feedforward table, and coil positions.

If the track to which a coil is added has a different feed override setting than the track from which the coil is removed, the feed override setting for the axis is also changed when the coil is added to the track.

If the track to which the coil is added has a different "start on trigger" setting than the track from which the coil is removed, the "start on trigger" setting is also changed when the coil is added to the track.

General constraints

- A linear track must always include physical end stops to guarantee that carriers cannot move off the actual track-coil configuration.
- Shared LMS sensors cannot be shared between tracks. This means that the axis which can be used by multiple coils (axisX in fig. 4-6 "RTLC between two tracks: single axis, different coils" on page 65) must have its own LMS Sensors. Other axes in the track will not use these LMS sensors.
- A coil (or set of coils) that is added to a track or removed from a track can control at maximum `CM_MAX_NR_OF_CARRIERS` carriers. Carriers cannot be moved along a coil (or set of coils) that does not belong to any track (the coil is "in between tracks"). Likewise, carriers cannot be added to or removed from that coil (or set of coils). Carriers that are moved with a coil (or set of coils) must be detected by the LMS sensors of that coil (or set of coils), and by those LMS sensors only. LMS sensors of other coils may not detect those carriers while the coil(s) and carrier(s) are added to the track or removed from the track, else the error `CM_ERR_UNABLE_TO_REMOVE_COIL` is generated.
- If more than one coil is shared at the beginning or end of a track, these coils are always added and removed as a set. It is not possible to remove a set of, for example, two shared coils and then add just one shared coil to the same track. However, it is possible to add just one of these shared coils to another track.
- One physical coil can only be used in one position in any track. It is not possible to use a physical coil at the beginning of track 0, remove the coil, somehow physically transport the coil to the end of track 0, and then add the coil at the end of track 0.
- RTLC coils have the restriction that they can only be removed if no carrier is present above the coil, or the carrier is in the `CM_CARRIER_STOPPED` state.
- Coils with different pole configuration cannot be shared between tracks.
- Coils from a modulo track cannot be added or removed.
- Within one node at maximum 14 coil configurations can be defined. This means that a node can have 8 coil axes of which 2 can be used in 4 different tracks (6 + 2*4). It can also be used for example to define 4 coil configurations for 3 axes, 2 coil configurations for 7 axes, etc.

4.7.8 Homing a track

Function

To home a track, the function `CmHomeTrack` is available. The input for this function is a track ID and a movement direction. Other required parameters, for example home time out value and delay time between start home procedure and the check that carriers are actually moving, are retrieved from the XML configuration file.

With the function `CmSynchronize` with the synchronization request `CM_REQ_TRACK_HOMING_COMPLETED` it is possible to synchronize or poll on the completion of the homing procedure of a specified track, see chapter 4.7.40 "Synchronize on an asynchonous operation" on page 139. When homing is successfully finished the track state is `CM_TRACK_HOMED`, the function `CmGetCarriersOnTrack` returns the number of carriers on the track, see chapter 4.7.9 "Get number of carriers and IDs" on page 80. If the defined track type and the requested home mode combination are incompatible, the error `CM_ERR_NOT_SUPPORTED` is returned.

With the functions `CmWriteHomePars` and `CmReadHomePars` you can store and retrieve the homing parameters.

If the client application, that started the home movement, stops for whatever reason, you can stop the movement with the function `CmResetError` (issued by another client application).

Explanation

Homing is performed for a track to determine the number of carriers on the track and the absolute position of each of the carriers.

After the home movement, each carrier is added to the carrier administration with an "add carrier at position". If no sensor detects the carrier at this "add carrier at position", the update of the carrier administration fails, and the error `CM_ERR_CARRIER_OUTSIDE_SENSOR_RANGE` is generated. If sensors detect the carrier at this "add carrier at position" but other sensors inside the sensor range do not detect the carrier at this "add carrier at position", the update of the carrier administration is successful, but after the "add carrier at position" the state of the sensors is checked. If a sensor does not detect the carrier at this "add carrier at position", the error `CM_ERR_INACTIVE_SENSOR` is generated.

Several different homing modes are supported. In these modes the carriers are moved by activation of a velocity control loop.

Mode 0, "end stop homing", supported for linear tracks

This mode moves all carriers to one side of the track. When the movement is completed, the number of carriers can be determined, based on the sensors that are active. The approximate position of the carriers is then calculated from the track begin or end position and the carrier length.

To check if all carriers are at one side of the track, all sensors at that side must be active, while at the other side all sensors must be inactive. In addition, the number of carriers expected must be consistent with the ranges of the active sensors. If these conditions are met while all carriers are standing still, the homing procedure is successfully completed. Otherwise, the procedure is stopped with an error, as soon as no carrier motion is detected anymore or after a time-out.

Home mode 0 supports tracks with dual sensor coils, tracks with triple sensors coils and tracks with a mix of dual sensor coils and triple sensor coils. This mode requires that the size of the carrier is identical to the size of the magnet plate. If this requirement is not fulfilled, you must use mode 10.

Mode 1, "modulo homing", supported for modulo tracks

This mode moves all carriers on the track in one direction until all active sensors have valid, mutually consistent position information. To achieve this, all carriers must have left the sensor range of the sensor by which they were detected at the start of the homing procedure. Therefore, the homing procedure starts a movement of approximately 1.5 carrier length. When all active sensors have valid position information, the movement is stopped. If this situation is not established within a defined maximum time, the movement is also stopped by a time-out and the homing procedure returns an error message.

Home mode 1 only supports tracks with dual sensor coils and the length of the carriers must be equal to the length of the magnet plate.

Mode 5, "define home", supported for linear tracks

This mode is a fast homing method, but will only be successful if the following conditions are met.

- Each sensor in the system that detects a carrier has valid position information,
- has not been shut down (`CmShutdown` function),
- did not experience a power off condition.

Mode 5 is available to the user to define a user homing mode. The user defines homing movements (for example with the functions `CmStartVelocityMode` and `CmStopVelocityMode`) and then homes the track with mode 5 after these movements. The movements must be such that, when completed, each sensor that initially detects a carrier is left at least once. This means that all active sensors have detected a carrier entering the detection range and that the sensors have a valid position measurement before home mode 5 is called.

The user can develop movements for the homing method, but can also use home mode 0 or home mode 1 to make sure that all sensors have a valid position measurement.

Mode 5 checks without moving the carriers. Note that carriers may be moving when `CmHomeTrack` executing home mode 5 is started. `CmHomeTrack` checks during the specified timeout period that all carriers are at standstill. If the carriers are not at standstill when the timeout period expires, the error `CM_ERR_TIMEOUT` is returned. If all carriers are at standstill, `CmHomeTrack` checks whether each sensor that detects a carrier has valid position information. If the carriers are at a valid position detected by the sensors the track is homed, else the home procedure is stopped and the error `CM_ERR_INVALID_POSITION` is returned. Note that the specification of the movement direction is irrelevant.

If velocity mode is active (and the velocity is 0), the carriers are "held in position" during the homing procedure.

Home mode 5 supports tracks with dual sensor coils, tracks with triple sensors coils and also tracks with a mix of dual sensor coils and triple sensor coils. Supported are carriers that have a length equal to or larger than the length of the magnet plate.

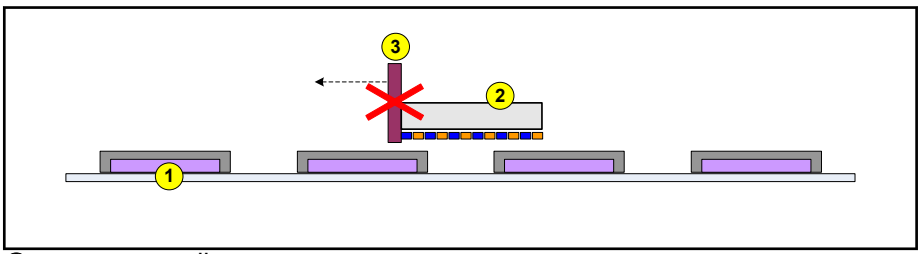
Mode 10, "multi-bumper homing", supported for linear tracks

A track can be divided in logical sections as required by a process plant. Such a section can, for example, be defined by a door which separates one process step from the other process step in the plant. The delimiting mechanical object, for example a door, can be used to home a carrier against. Such an object is called a "bumper". The bumper positions must be accurately defined (in the XML file). Thus, all positions in a track, the track begin position, track end position and all defined bumpers (upper and lower positions) must be calibrated. The maximum number of bumper positions in a track is defined by `CM_MAX_NR_OF_BUMPERS_PER_TRACK`.

The `upperSidePos` of a bumper is defined as the position of the center of the magnet plate when the carrier is against the upper side of the bumper. Likewise, the `lowerSidePos` of a bumper is defined as the position of the center of the magnet plate when the carrier is against the lower side of the bumper, see fig. 4-12 "Minimum distance between a sensor and a bumper" on page 74.

A wrong definition of the bumper position can lead to errors during the first movement after homing.

If a track uses MR sensors, in the homing prologue, the bumper position is determined as the average of the lower bumper position (which is the carrier position when pushing at the left side of the bumper) and the upper bumper position (which is the carrier position when pushing at the left side of the bumper). The bumper position is compared with the coil position to decide to which "section" a coil belongs. As the prologue uses different commutation phases within one section to move the carrier, you must make sure that the bumper is positioned between coils. This is explained as follows. If a bumper is placed above a coil, the coil is assigned to one section. However, above the other part of the coil there could also be a carrier, which is basically controlled by the coils of a second section. The first section can generate a current, which influences the carrier movement of the carrier, controlled by the second section. In the following figure the two leftmost coils belong to one section, and the rightmost to the other one. The movement of the carrier, which is in the second section is affected by the current applied in the first section.



① coil
② carrier
③ bumper

Fig. 4-8: Wrong bumper position on an MR sensor-based track

If only the lower or upper bumper position can be determined, and the other bumper position cannot be determined due to physical constraints, an incorrect bumper position is calculated which may cause a coil to be assigned to the wrong section. In that case the error `CM_ERR_MR_SENSOR_COMMUTATION_NOT_DETERMINED` may be reported during homing. You must set the other position in such a way, that the difference between upper and lower bumper position equals the carrier length plus bumper thickness.

MR track homing against a bumper

When multiple carriers are located against each other and the carrier length is not exactly a multiple of 24 mm, the prologue may occasionally fail on an MR sensor-based track. The prologue sends a current through the coil and checks whether the carrier moved a predefined distance. If the carrier is blocked by other carriers, this carrier cannot move to the commutation position based on the current, but the carrier may still have moved the predefined distance, because the carrier is pushed by the other carriers.

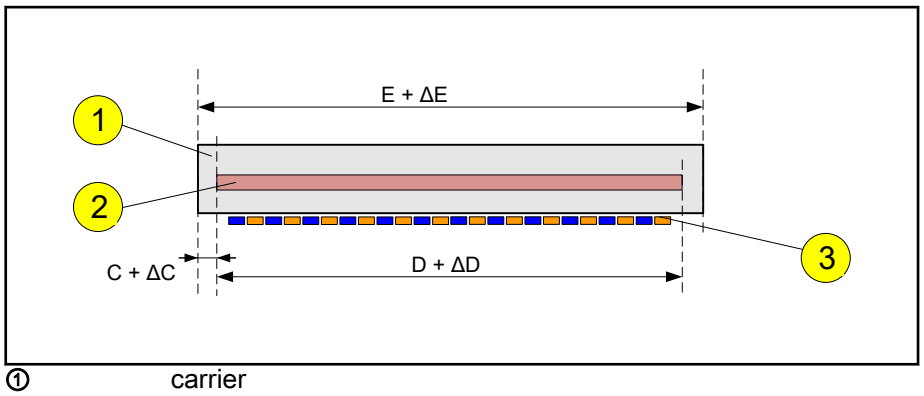
An other issue is that during homing the predefined carrier positions against the bumper must be accurate by half the pitch of the magnetic scale. Subsequently, the exact position of the carrier is determined by the signals from the sensor. This is the precondition as described for homing mode 14. In this context, a bumper can be a mechanical end stop or a bumper.

☞ Bumper homing is not recommended for MR sensor-based LMS systems.

If the defined approximate position is not sufficiently accurate, the carrier position is not correctly determined. The predefined carrier positions against the bumper are defined as follows. The first carrier position is taught, the second carrier position is that first carrier position plus one time the carrier length, the third carrier position is the first carrier position plus twice the carrier length, etc. The predefined carrier positions against the bumper are therefore subject to mismatches with the actual carrier positions, due to known tolerances and generic manufacturing tolerances. To prevent this homing issue, the number of carriers on a track must be limited. The maximum number of carriers that can be homed using bumper homing can be calculated, taking into account known tolerances and generic manufacturing tolerances. These generic tolerances are machine dependent. The tighter the tolerances, the larger the maximum number of carriers that can be homed with a homing against a bumper procedure. The chain of tolerances consists of tolerances of the carrier and tolerances of the track.

Carrier

A carrier is produced of a solid piece of metal with a bumper attached to it. A magnetic scale is attached on the carrier for position detection by the MR sensor. The following tolerances can be defined for the carrier.



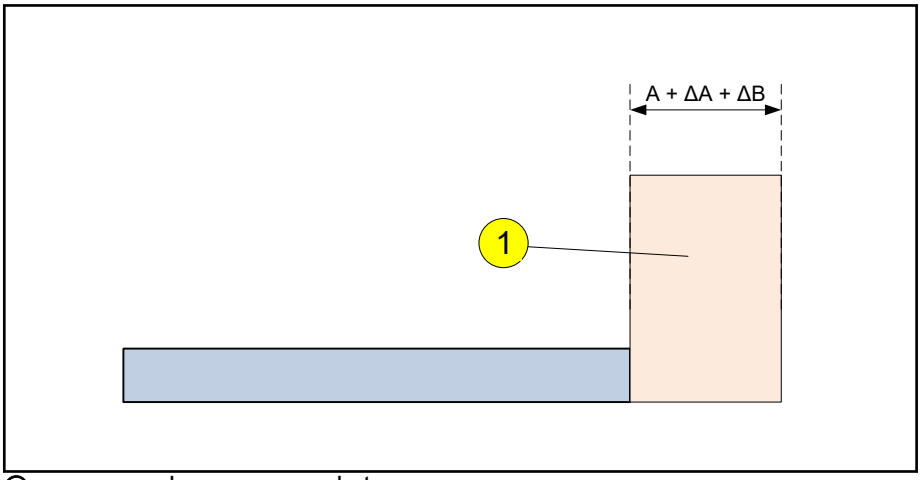
① carrier
② magnetic scale
③ magnet plate
ΔC Tolerance on the positioning of the magnetic scale
ΔD Tolerance on cutting the length of the magnetic scale
ΔE Milling tolerance on production of the carrier

Fig. 4-9: Carrier dimensions and tolerances

Application of elastic (rubber-like) material as bumpers on carriers is not recommended for MR sensor-based LMS systems, when bumper homing is used. If elastic bumpers are used on the carriers, the deformation of the bumper under force has to be included in the tolerance chain.

Track

A track is made up of a guidance system and a solid bumper, against which carriers can home. The following tolerances can be defined for the track.



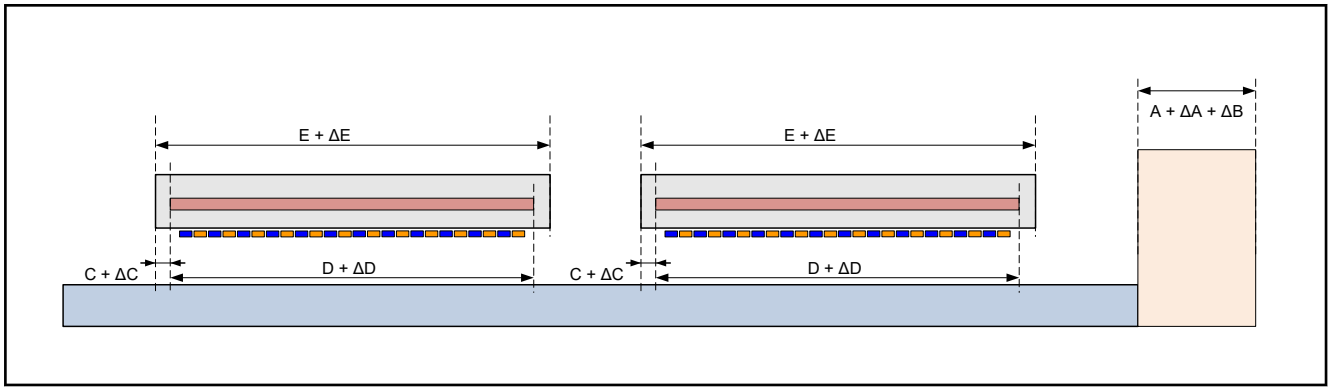
① bumper or end stop
ΔA Machining tolerance of the bumper
ΔB Elastic deformation of the bumper when carriers are pushing against it

Fig. 4-10: Track dimensions and tolerances

Application of elastic (rubber-like) material as bumper on the track is not recommended for MR sensor-based LMS systems, when bumper homing is used. If elastic bumpers are used on the track, the deformation of the bumper under force has to be included in the tolerance chain.

Tolerance chain

A system, taking all tolerances into account, can be represented as follows.



ΔA	Machining tolerance of the bumper
ΔB	Elastic deformation of the bumper when the carriers are pushing against it
ΔC	Tolerance on the positioning of the magnetic scale
ΔD	Tolerance on cutting the length of the magnetic scale
ΔE	Milling tolerance on production of the carrier

Fig. 4-11: System tolerances

The total allowed tolerance is ±2.5 mm for an MR sensor-based track. The following formula applies for n carriers.

```
±2.5 = ΔA + ΔB + ΔC + ΔD + n * ΔE
```

The tolerance on cutting the length of the magnetic scale has a fixed tolerance of ±0.4 mm, thus ΔD = 0.4. Further, 0.05 mm is subtracted of the total allowed tolerance to make sure that the total tolerance is not just over the limit. Thus, the following equation can be used.

```
±2.05 = ΔA + ΔB + ΔC + n * ΔE
```

Calculation example

The following assumptions make the calculation simpler.

- ΔA = ΔB = 0. These tolerances can be assumed 0, because when the carriers are pushed against the bumper, this is seen as the 0 position. Therefore, the milling tolerances on the bumper can be ignored.
- ΔC = ±0.5 = 1. Assumed is that all magnetic scales are glued within the tolerance of ±0.5 mm. If there is more margin between the gluing of the magnetic scales, this has to be taken into account. The system requirement specifies that all magnetic scales are glued within ±1 mm.
- ΔE = ±0.1 = 0.1. This tolerance depends on the accuracy with which the carriers can be produced. Here a milling tolerance of ±0.1 mm is assumed.

Maximum number of carriers in this example is

```
n = ( 2.05 - ΔC ) / ΔE  = 10
```

- Always round off the calculation to the lower integer number.
- The tolerance on the positioning of the magnetic scale is counted twice, because for the carrier with which the bumper position has been determined, the scale could be positioned with -x and for the last carrier in the row against the bumper during homing, the scale could be positioned with +x.

Homing of more carriers

Two different solutions are available to home more than 10 carriers on a track, as given by the example.

1. Decrease the milling tolerances of the carriers.
2. If decreasing milling tolerances is not possible, an other homing mode can be used. For example, home mode 12 is more suitable for homing a large number of carriers on a MR sensor-based track. However, some preconditions have to be taken into account. The number of carriers on the track has to be specified, and free sensors have to be available to move to when homing is started.

As the magnet plate does not need to be installed centered on the carrier, the offset is defined by the parameter motorMagnetEccentricity, see tab. 3-2 "Track parameters" on page 25.

Home mode 10 supports tracks with dual sensor coils, tracks with triple sensors coils and also tracks with a mix of dual sensor coils and triple sensor coils. Supported are carriers that have a length equal to or larger than the length of the magnet plate.

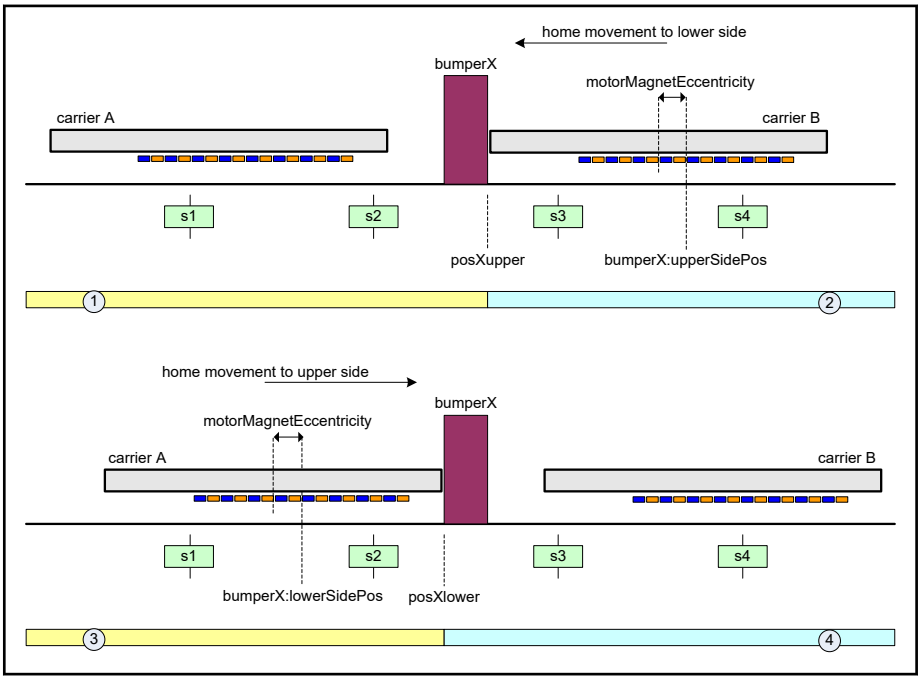


Fig. 4-12: Minimum distance between a sensor and a bumper

The following dimension constraints apply for the positioning of the sensors.

If the home movement is in the lower side direction, `posXupper` is defined as `upperSidePos - ((carrierLength/2) - motorMagnetEccentricity)`

- if `sensorPos > posXupper`
  - the sensor is allowed to detect a carrier in region 2.
  - the sensor must never be able to detect a carrier in region 1.
- if `sensorPos ≤ posXupper`
  - the sensor is allowed to detect a carrier in region 1.
  - the sensor must never be able to detect a carrier in region 2.

If the home movement is in the upper side direction, posXlower is defined as `lowerSidePos + ((carrierLength/2) + motorMagnetEccentricity)`

- if `sensorPos < posXlower`
  - the sensor is allowed to detect a carrier in region 3.
  - the sensor must never be able to detect a carrier in region 4.
- if `sensorPos ≥ posXlower`
  - the sensor is allowed to detect a carrier in region 4.
  - the sensor must never be able to detect a carrier in region 3.

Before the function `CmHomeTrack` is called, the user application must update the bumper active flags with the function `CmEnableBumper`. The function `CmEnableBumper` is not allowed while the track state is `CM_TRACK_HOMED`. The user application can check the actual bumper parameters with the function `CmReadBumperPars`. Note that after the function `CmInitialize` all bumper flags are set to `FALSE`, which means that the bumpers are not present.

After the function `CmHomeTrack` is called, all carriers move to one side of the track. When all carriers stand still against the track begin position, or track end position or one of the active bumpers, the carriers are added to the track administration.

Approximate position

The approximate position of the first carrier against the end stop or active bumper is determined with a predefined (taught) position. The approximate positions of the adjacent carriers are determined (calculated) based on the length of the carriers. The actual positions are subsequently determined using the sensor signals. The approximate positions must be within one sensor signal period accurate. If an approximate position is less accurate, the determined carrier position will be wrong which results in a blocking software error. The approximate position for a Hall sensor-based track must be accurate by half the pitch of the magnet plate (24 mm or +/-12 mm). The approximate position for an MR sensor-based track is one pole pitch of the magnetic scale (5 mm or +/-2.5 mm).

Given the method to determine the approximate positions, the home procedures using bumpers (home mode 10 and 11) are not recommended for MR sensor-based tracks. If the application wants to use a home procedure using bumpers, the magnetic scale must be positioned as accurately as possible, and the mechanical tolerance chain of magnetic scale length and carrier length must be taken into account to make sure that these home modes can determine the approximate positions with sufficient accuracy.

When all carriers are moved against a bumper, the carriers must stand still within a defined maximum time, else the movement is stopped by a time-out and the homing procedure returns an error message. The time-out is based on the actual length of the track and the home velocity.

No limitation exists for the distance that a carrier must or can move before a bumper is touched, however, note the following conditions.

- The mechanical layout of the track, especially the positioning of sensors and bumpers with respect to carriers and their magnet plates, must adhere to the mentioned constraints.
- The `sensorPos` (sensor position from the XML file) of all sensors in the system are not allowed to have a value between the physical minimum and maximum position of a bumper.

- Avoid the situation that sensors are "just" detected if all carriers push against bumpers.
- The reliability of home mode 10 is not guaranteed if two carriers activate the same sensor.

Mode 11, "homing against bumper", supported for modulo and linear tracks

This mode moves all carriers on the track in one direction against a bumper, and then positions every carrier above a coil. The coils above which a carrier is positioned do not need to be in successive order. Mode 11 supports tracks with dual sensor coils, tracks with triple sensors coils and also tracks with a mix of dual sensor coils and triple sensor coils. Supported are carriers that have a length equal to or larger than the length of the magnet plate.

Before homing mode 11 is started, all carriers are at random positions on the track, see fig. 4-13 "Carrier positions during homing mode 11" on page 76 situation A.

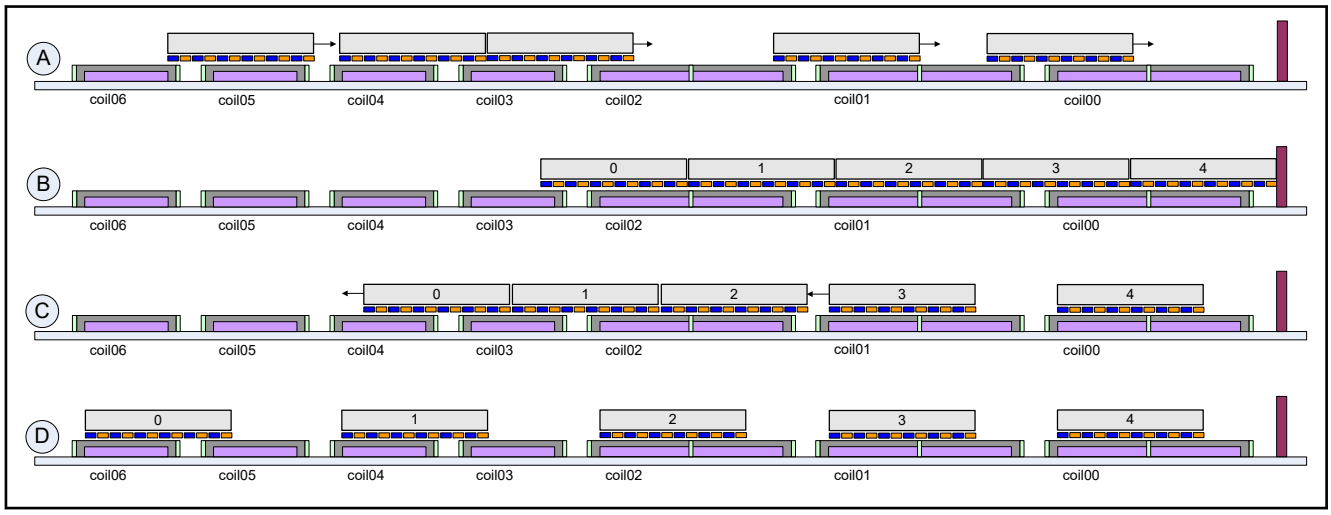


Fig. 4-13: Carrier positions during homing mode 11

During stage 1, all carriers are moved in velocity mode, against the upper or lower side of the bumper in the track, see fig. 4-13 "Carrier positions during homing mode 11" on page 76 situation B. When the movement is finished, the number of carriers on the track can be determined, each carrier is assigned a carrier index and the carriers are added to the track. The approximate position of each carrier is calculated using the upper or lower side position of the bumper. See home mode 10 for a description how the approximate carrier positions are determined.

During stage 2, all carriers are positioned above a coil. All carriers are moved in velocity mode, in the opposite direction as used in stage 1. When the carrier at the end of the carrier train is the only carrier above a coil, this carrier is stopped. The rest of the carrier train continues to move, see fig. 4-13 "Carrier positions during homing mode 11" on page 76 situation C.

When the carrier that is now at the end of the carrier train is the only carrier above a coil, that carrier is stopped. The rest of the carrier train continues to move. This procedure continues until all carriers are positioned above a coil. A carrier can be stopped while the carrier is above 2 coils, if the length of the magnet plate is larger than the distance between the 2 coils; see fig. 4-13 "Carrier positions during homing mode 11" on page 76 situation D. When all carriers are above a coil, the carriers are added to the track.

Mode 12, "bumpless homing", supported for linear tracks

This home mode is specifically designed for the situation that carriers transport sensitive products and for that reason carriers are not allowed to collide against each other.

Home mode 12 supports tracks with dual sensor coils, tracks with triple sensors coils and also tracks with a mix of dual sensor coils and triple sensor coils. Supported are carriers that have a length equal to or larger than the length of the magnet plate.

As the detection of the number of carriers on a track cannot be performed unambiguously, the number of carriers on a track must be specified.

Before the home mode is started, the following preconditions are checked and must be met.

1. The specified number of carriers must be identical to the number of detected carriers on the track.
2. The first carrier on the track must have at least two free sensors between the carrier and the begin stop of the track.

   -- or --

   The last carrier on the track must have at least two free sensors between the carrier and the end stop of the track.
3. On both sides of every carrier is at least one free coil with sensors in between.

Precondition 2 ensures that a gap exists between the carrier and an end stop to prevent a collision. Precondition 3 ensures that carriers do not collide against each other. It is the responsibility of the user that the carriers are positioned in such a way that all preconditions are met. With the function `CmFreeArea` you can fulfil the preconditions 2 and 3 (if possible), see chapter 4.7.13 "Detect carrier presence in an area, create an area without carriers" on page 82.

fig. 4-14 "Example of carrier positions when all preconditions are met" on page 77 shows possible situations of carrier positions where all preconditions are met.

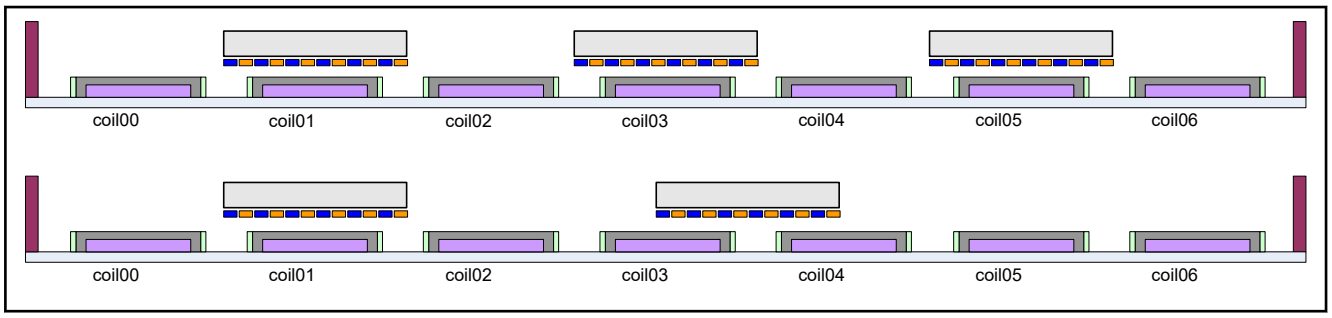


Fig. 4-14: Example of carrier positions when all preconditions are met

fig. 4-15 "Example of carrier positions when not all preconditions are met" on page 78 shows possible situations of carrier positions where one of the preconditions is not met. In example Ⓐ precondition 2 is not met. Examples in Ⓑ show several possible situations of carrier positions where precondition 3 is not met.

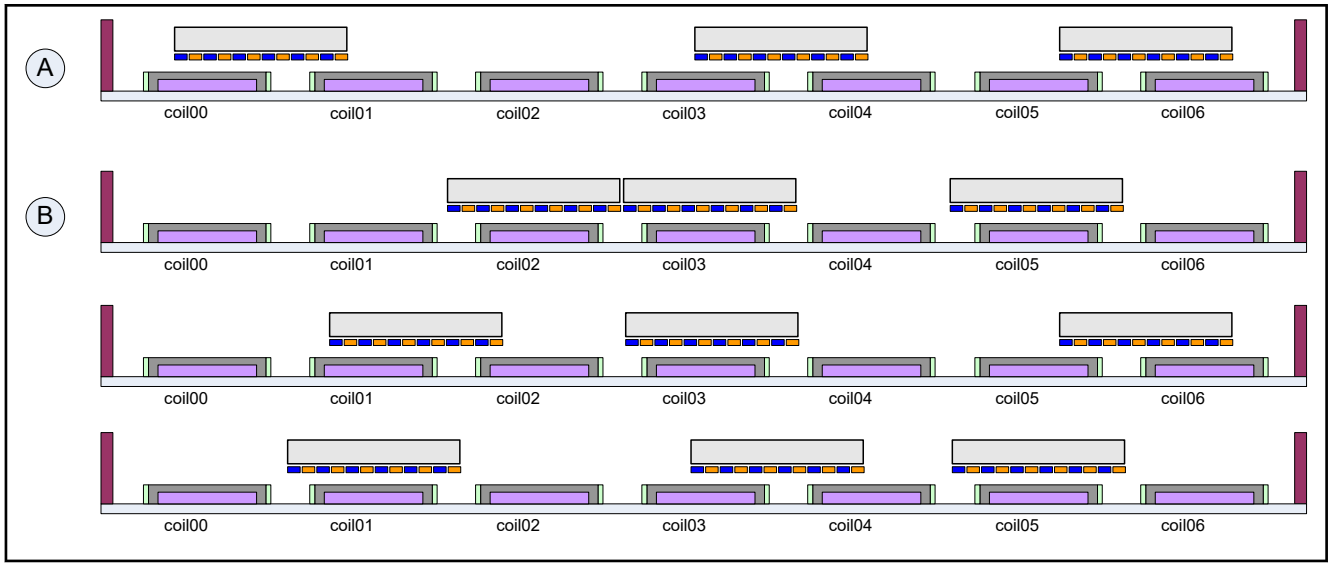


Fig. 4-15: Example of carrier positions when not all preconditions are met

Before home mode 12 is started the field `homeMode` in the structure `CM_HOME_PARS` must be set to 12, the field `homeVelocity` must specify the movement velocity of the carriers during home procedure and the field `homeNrOfCarriers` must specify the number of carriers on the track. Note that it is not required to define a home direction. The algorithm selects a home direction which may be the opposite of the direction specified in the structure `CM_HOME_PARS`. The number of free sensors at the begin and end of the track determines the initial movement direction. The movement direction is toward the end of the track that has the highest number of free sensors. If the number of free sensors at the begin and end of the track is equal, the initial movement direction is as specified with the parameter `direction` of the `CmHomeTrack` function.

If precondition 1 is not met, the home procedure is aborted and the error `CM_ERR_NR_OF_CARRIERS_INCONSISTENT` is returned.

If precondition 2 is not met, the home procedure is aborted and the error `CM_ERR_NO_FREE_SENSORS_TO_MOVE` is returned.

If precondition 3 is not met, the home procedure is aborted and the error `CM_ERR_NO_FREE_COIL_BETWEEN_CARRIERS` is returned.

After the function `CmHomeTrack` is called all carriers start to move at the same moment to maintain the gap between the carriers to prevent collisions. The movement of all carriers is stopped at the same moment when each carrier is detected by another sensor and the carrier position determined by this sensor is at least a half pole pitch larger or smaller, depending on the home direction, as the left or right sensor range boundary. During the movement the following conditions are checked.

- A carrier moves against an end stop.

  If this condition occurs, the home mode is aborted and the error `CM_ERR_CARRIER_AGAINST_ENDSTOP` is returned.

- The timeout period has expired.

  If this condition occurs, the home mode is aborted and the error `CM_ERR_TIMEOUT` is returned. Under normal conditions the carrier movement should be completed within the specified time. If, due to a

system failure, the normal movement cannot be realized, the timeout will occur.

- No carriers are moving.

  If this condition occurs, the home mode is aborted and the error `CM_ERR_CARRIERS_NOT_MOVING` is returned. If, due to a system failure or a mechanical obstruction, a collision may occur.

When the movement of the carriers is stopped, the software checks whether a coil has generated an asynchronous error. If this condition occurs, the home procedure is aborted and the error `CM_ERR_ASYNC_ERROR_OCCURRED` is returned.

When the movement of the carriers is stopped and no errors have occurred, a check for consistent sensor status for each sensor of the track is performed. Each sensor that detects a carrier must have a valid position and carrier index. If the carrier position is not valid, the error `CM_ERR_INVALID_POSITION` is returned. If the carrier index is not valid, the error `CM_ERR_INVALID_CARRIER_ID` is returned.

When the home procedure finishes successfully all carriers are added to the CM administration.

Mode 14, "assign position homing", supported for modulo and linear tracks

Like home mode 5, home mode 14 is a "define home" mode without carrier movement. However, home mode 5 has the precondition that all sensors of the track have detected a carrier and each sensor that detects a carrier has valid position information. With home mode 14, all carriers on the track may have a random position. The precondition for home mode 14 is that the position of all carriers must be known.

Home mode 14 can be used for linear tracks and modulo tracks. The track may have dual sensor coils, triple sensors coils, and a mix of these coil types. Carriers with a length equal to the magnet plate and carriers longer than the magnet plate are supported.

Before home mode 14 is started, the known position of the carriers is assigned to the struct member `homeCarrierPositions` of the structure `CM_HOME_PARS`. `homeCarrierPositions` is an array that contains the position of each carrier on the track. The struct member `homeNrOfCarriers` specifies the number of carriers on the track. The array `homeCarrierPositions` has `CM_MAX_NR_OF_CARRIERS` entries, but only the first entries (defined by `homeNrOfCarriers`) must be valid. The carrier positions in the array must be specified with increasing position value.

The function `CmInitialize` sets the number of carriers and all carrier positions to 0. With the function `CmWriteHomePars` the data specified in the `CM_HOME_PARS` structure is set. `CmHomeTrack` reads the actual home parameters and homes the track.

For home mode 14, the following additional checks are performed.

- The value specified by `homeNrOfCarriers` must be between 0 and `CM_MAX_NR_OF_CARRIERS`, else the error `CM_ERR_PARAMETER_ERROR` is returned.
- The defined entries of the array `homeCarrierPositions` hold position information of increasing value, else the error `CM_ERR_PARAMETER_ERROR` is returned.

During the execution of the home mode 14 procedure the following errors may be returned.

- CM_ERR_CARRIER_POSITION_OUTSIDE_SENSOR_RANGE if a carrier position is outside the range of any sensor.
- CM_ERR_INVALID_CARRIER_ID if a carrier has an invalid carrier index.
- CM_ERR_INVALID_POSITION if a carrier on the track does not have a valid position.

The carriers may have a random position on the track, but the approximate position of all carriers must be known. This defined approximate position must be accurate by half of the pitch of the magnet plate for a Hall sensor-based track (less than +/–6.0 mm) or half of the pitch of the magnetic scale for an MR sensor-based track (less than +/–1.25 mm). The exact position of the carrier is determined by the signals from the sensor. If the defined approximate position is not sufficiently accurate, the carrier position is not correctly determined.

Other modes

Other home modes are developed for a non-standard hardware configuration. For example, in the struct CM_HOME_PARS the members homeEdgeOffset and homeAllowedCollisionVelocity are defined. These parameters are only used in customer-specific homing modes. For more information, please contact Bosch Rexroth.

After homing of a track, the number of carriers and their positions on the track are known. To be able to perform the velocity controlled movement, the power of the coils is enabled. Homing is possible in every state of the track. When the track is homed successfully, the track is in the CM_TRACK_HOMED state.

4.7.9 Get number of carriers and IDs

Function

Depending on the track state, the function CmGetCarriersOnTrack returns information about the carriers on the track. If the track state is CM_TRACK_POWERUP or CM_TRACK_IDLE the error CM_ERR_NR_OF_CARRIERS_UNKNOWN is returned. If the track state is CM_TRACK_HOMED the number of carriers is returned and the warning CM_WRN_CARRIER_IDS_NOT_ASSIGNED. In the other track states this function returns the number of carriers on a specified track, and an array with carrier IDs which is sorted by increasing position.

Explanation

The function CmGetCarriersOnTrack retrieves the number of carriers on a track with the associated carrier identifiers. The input parameter to the function is the track identifier. The assigned carrier identifiers are stored in the output array ordered by increasing carrier position. The carrier identifier with the lowest carrier position is stored in array index 0. The size of the array is equal to the maximum number of carriers in a system (CM_MAX_NR_OF_CARRIERS). The identifier of a non-existing carrier or for a carrier for which no identifier is assigned is CM_NO_ID. The function is only allowed when the client state is CM_CLIENT_CONNECTED.

4.7.10 Convert carrier ID / carrier index

Function

In an LMS application you use CM functions. However, in some cases, data must be retrieved at SAC level instead of CM level. Likewise, in some cases, data at SAC level must be retrieved at CM level. With the function `CmGetCarrierIndex` you can convert the specified carrier ID to the associated carrier index. With the function `CmGetCarrierId` you can convert the specified carrier index to the associated carrier ID.

Explanation

For some specific application control you may use a UDSX implementation. As there is no CM functionality in UDSX, commands will operate at the SAC level. With the function `CmGetCarrierIndex` you can convert the carrier ID as known in the LMS application, to the carrier index for use in SAC functions. Likewise, if you retrieve carrier information from a coil, the data is associated with `SAC_VAR_LMS_CONTROL_CARRIER_INDEX` of that coil. With the function `CmGetCarrierId` you can convert the carrier index to the carrier ID.

4.7.11 Get number of tracks

Function

The function `CmGetTracks` returns the number of tracks in an LMS.

Explanation

The function `CmGetTracks` retrieves the number of tracks in an LMS with the associated track identifiers. The function has no input parameters. The assigned track identifiers are stored in the output array starting at index 0. The size of the array is equal to the maximum number of tracks in a system (`CM_MAX_NR_OF_TRACKS`). The identifier of a non-existing track is `CM_NO_ID`. The function is only allowed when the system state is `CM_SYSTEM_INITIALIZED` and the client state is `CM_CLIENT_CONNECTED`.

4.7.12 Get axis identifier of a coil

The function `CmGetAxisIdOfCoil` retrieves the axis identifier of a coil. The input parameters to the function are the track identifier `trackId` and the coil index `coilIndex`. The output parameter of the function is the axis identifier `pAxisId` of the specified coil. Only when an application is a CM client (used `CmConnect` to connect to CM), the application can retrieve the axis identifier of a coil, else the error `CM_ERR_CLIENT_NOT_CONNECTED` is returned. The function `CmGetAxisIdOfCoil` is only allowed when the system state is `CM_SYSTEM_INITIALIZED`, else the error `CM_ERR_SYSTEM_STATE_ERROR` is returned. Note that the axis identifier is only valid while the system state is `CM_SYSTEM_INITIALIZED`. The axis identifier becomes invalid after a call of the function `CmShutdown` and thus can no longer be used. With the system events `CM_EV_SYSTEM_INITIALIZED` and `CM_EV_SYSTEM_SHUTDOWN` the application can be notified when a CM client executes `CmInitialize` or `CmShutdown`. The axis identifier is also only valid as long as the application is a CM client. With the Runtime Track Layout Changes functionality a coil or coils can dynamically be added to or removed from the beginning or end of a track. The application must retrieve the axis identifier of a coil again after the coil is removed and an other coil is added. With the track event `CM_EV_TRACK_LAYOUT_CHANGED` the application can be notified when a CM client changes the layout of a track.

☞ You can use the axis identifier to call functions of the SAC and MAC subsystems. Note that SAC and MAC functions may influence the functionality of CM. Use the axis identifier for access to (coil) axis variables and parameters. Do **not** use the axis identifier for functions that may or will cause an axis state transition.

4.7.13 Detect carrier presence in an area, create an area without carriers

Function

The function `CmCheckArea` is available to check if any of the sensors, present in a specified area of the specified track, gives a valid signal. The input for this function is the track ID and a pointer to a `CM_AREA` structure, which specifies the area minimum and maximum position. The function `CmCheckArea` can only be called when the system state is `CM_SYSTEM_INITIALIZED` and the client state is `CM_CLIENT_CONNECTED`.

`CmCheckArea` executes the following area checks. If a check fails, the following errors are returned.

- For a linear track
  - Area minimum position > area maximum position. In case of error: `CM_ERR_INVALID_POSITIONS`
  - Area minimum position < (track begin position – ½ measurement scale length). In case of error: `CM_ERR_INVALID_MIN_POSITION`
  - Area maximum position > (track end position + ½ measurement scale length). In case of error: `CM_ERR_INVALID_MAX_POSITION`
  - At least 1 sensor is present in the area. In case of error: `CM_ERR_NO_SENSOR_IN_AREA`
- For a modulo track
  - Area minimum position < track begin position. In case of error: `CM_ERR_INVALID_MIN_POSITION`
  - Area minimum position > track end position. In case of error: `CM_ERR_INVALID_MIN_POSITION`
  - Area maximum position < track begin position. In case of error: `CM_ERR_INVALID_MAX_POSITION`
  - Area maximum position > track end position. In case of error: `CM_ERR_INVALID_MAX_POSITION`
  - At least 1 sensor is present in the area. In case of error: `CM_ERR_NO_SENSOR_IN_AREA`

☞ `CmCheckArea` cannot be used to detect a carrier in an area if the size of the carrier is larger than the magnet plate, because the sensors only detect the magnet plate.

With the function `CmFreeArea` carriers can be moved inside a specified area of the track until a part of that area is freed from carriers. The input for this function is the track ID and a pointer to a `CM_FREE_AREA_PARS` structure. The function can be used to fulfil preconditions for home mode 12. `CmFreeArea` can be used for linear tracks and modulo tracks.

☞ CmFreeArea may not completely free an area if the size of the carrier is larger than the magnet plate, because the sensors only detect the magnet plate. In this case the carrier may still cover a part of the area to be freed.

Explanation

To fulfil preconditions 2 and 3 of home mode 12, carriers must be moved within a specified area of a track until a part of that area no longer detects any carrier. The movement of carriers within the specified area is realized in velocity control mode with a non-zero velocity. Carriers on the track outside the specified area are not moved. To check that a part of the area no longer detects any carriers, all sensors inside that part of the area must detect no carrier.

The function CmFreeArea has the following parameters.

- trackId: identifies the track
- pointer to CM_FREE_AREA_PARS structure with the following members
  - beginPosition: specifies the begin position of the area
  - endPosition: specifies the end position of the area
  - freeAreaSize: defines with beginPosition the part of the area from which the carriers are moved away
  - velocity: specifies the velocity and direction of the movement

The following preconditions must be met for CmFreeArea.

- a coil must be present with left and right sensor inside the specified area beginPosition to endPosition. If this precondition is not met, the error CM_ERR_NO_COIL_IN_AREA is returned by CmFreeArea.
- a sensor must be present inside the area beginPosition to beginPosition + freeAreaSize if the velocity is a positive value, or a sensor must be present inside the area beginPosition to beginPosition - freeAreaSize if the velocity is a negative value. If this precondition is not met, the error CM_ERR_NO_SENSOR_IN_AREA is returned by CmFreeArea.
- a sensor must be present on the track after the endPosition if velocity is a positive value, or a sensor must be present on the track before the beginPosition if velocity is a negative value. If this precondition is not met, the error CM_ERR_NO_FREE_SENSOR_OUTSIDE_AREA is returned by CmFreeArea.

The function CmFreeArea sets the specified track in velocity control mode with velocity 0. The track state changes to CM_TRACK_VELOCITY_MODE and the state of all carriers on the track changes to CM_CARRIER_POWERUP. The coils of which the left and right sensors are located inside the specified area are set in velocity control with the specified velocity.

If no sensor within the specified beginPosition to endPosition detects a moving carrier, the velocity control is stopped, the track state changes to CM_TRACK_IDLE, and the error CM_ERR_NO_MOVING_CARRIER_IN_AREA is returned by the function CmFreeArea.

If all sensors inside the specified beginPosition to beginPosition + freeAreaSize if velocity is a positive value, or beginPosition to

`beginPosition` - `freeAreaSize` if `velocity` is a negative value do not detect a carrier, the velocity control is stopped, the track state changes to `CM_TRACK_IDLE`, and `NYCE_OK` is returned by the function `CmFreeArea`.

If the first sensor before `beginPosition` or the first sensor after `endPosition` detects a carrier, the velocity control is stopped, the track state changes to `CM_TRACK_IDLE`, and the error `CM_ERR_CARRIER_MOVED_OUTSIDE_AREA` is returned by the function `CmFreeArea`.

In the following figures "b" represents `beginPosition`, "f" `beginPosition` +/- `freeAreaSize`, and "e" `endPosition`.

fig. 4-16 "Create a free area with CmFreeArea" on page 84 shows a situation where you can successfully create a free area. Carrier ② is moved to coil02.

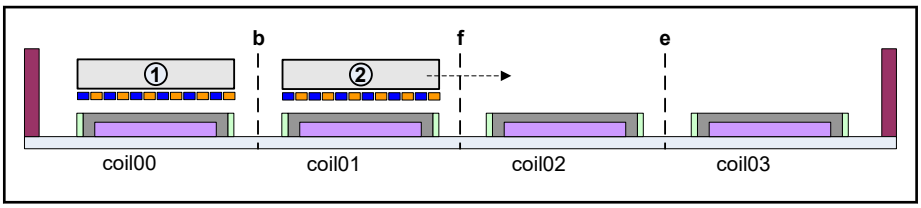


Fig. 4-16: Create a free area with CmFreeArea

fig. 4-17 "Not met preconditions for CmFreeArea" on page 84 shows preconditions that are not met. Conditions Ⓐ and Ⓑ result in the error `CM_ERR_NO_FREE_SENSOR_OUTSIDE_AREA`. Condition Ⓒ results in the error `CM_ERR_NO_SENSOR_IN_AREA`. Define the `beginPosition` and the `freeAreaSize` correctly to solve this condition.

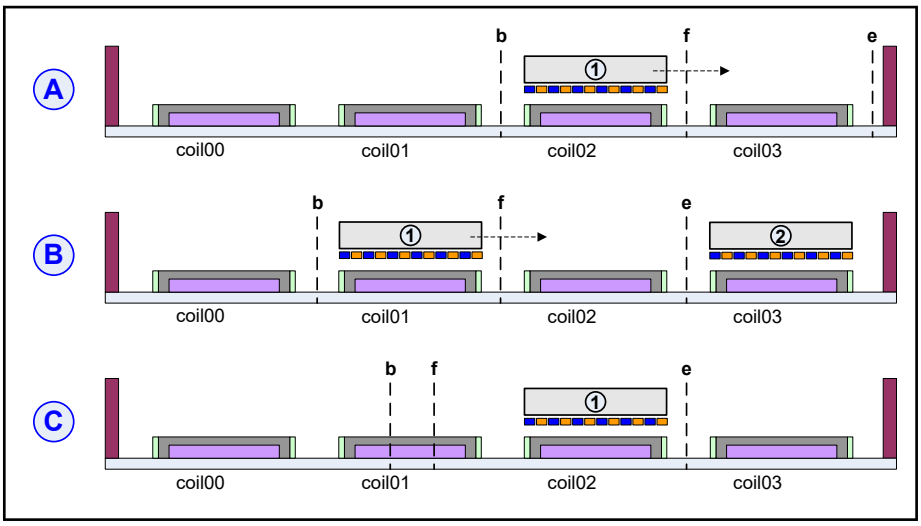


Fig. 4-17: Not met preconditions for CmFreeArea

fig. 4-18 "Not met postconditions of CmFreeArea" on page 85 shows postconditions that are not met. Condition Ⓓ results in the error `CM_ERR_NO_MOVING_CARRIER_IN_AREA`, because the carrier cannot be controlled by coil02. Condition Ⓔ results in the error `CM_ERR_NO_MOVING_CARRIER_IN_AREA`, because the carrier stops between coil02 and coil03. Move the `endPosition` to the left to solve this condition. Condition Ⓕ results in the error `CM_ERR_CARRIER_MOVED_OUTSIDE_AREA`, because the carrier is moved

to the left until the right sensor of coil00 detects the carrier. Move the `endPosition` to the left to solve this condition.

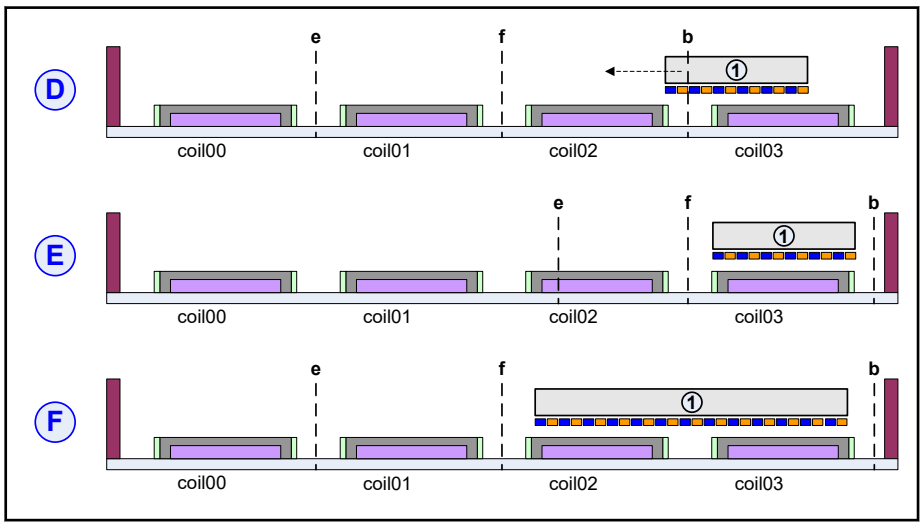


Fig. 4-18: Not met postconditions of CmFreeArea

If not enough space is available for the carriers between the specified `beginPosition` + / - `freeAreaSize` and the `endPosition`, carriers inside the area may bump against each other. See fig. 4-19 "Insufficient space causing a bump of carriers" on page 85. Carrier ① stops moving when this carrier enters the control range of coil02. Carrier ② bumps against carrier ① and pushes carrier ① toward the right sensor of coil02. When the right sensor of coil02 detects carrier ①, the movement of both carriers is stopped and the error `CM_ERR_CARRIER_MOVED_OUTSIDE_AREA` is generated. Move the `endPosition` to the right to solve this condition.

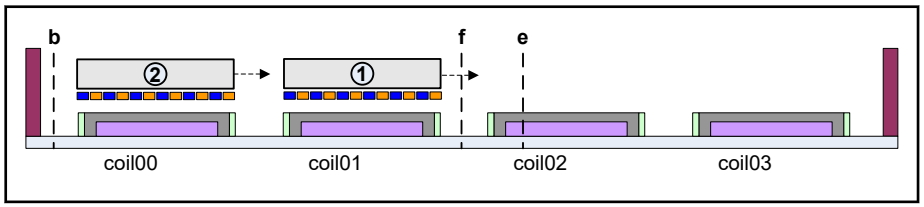


Fig. 4-19: Insufficient space causing a bump of carriers

4.7.14 Carrier Position Adjustment

Function

When a carrier with a magnet plate moves over a Hall sensor, the position of the carrier can be determined. However, due to inaccuracies in the production of a magnet plate, different carriers at the same position above the same Hall sensor result in slightly different carrier positions. The correct carrier position can be determined if the difference for that specific carrier is known. To this end, the carrier positions measured by the Hall sensor in the track can be compared with the carrier positions measured by an external encoder with higher accuracy. The differences are calculated and added as adjustments to the raw carrier positions measured by the Hall sensor.

With the function `CmDownloadCpaTable` you can download one Carrier Position Adjustment (CPA) table, identified by the table identifier, from the specified XML file that contains CPA tables. The function `CmLinkCpaTable` links a specified carrier to a specified CPA table. The function

`CmUnlinkCpaTable` disconnects the link between a specified carrier and its associated CPA table. The function `CmEnableCpa` enables or disables the CPA functionality for all carriers. With the function `CmReadSystemVariable` and the variable identifier `CM_VAR_SYSTEM_CPA_ENABLE` you can retrieve whether the CPA functionality is enabled (`TRUE`) or disabled (`FALSE`).

The CPA tool does not support the MR sensor; only the Hall sensor is supported.

Explanation

With the assumption that the measured carrier position from the Hall sensor and the external measurement system has the same value when the middle of the magnet plate is right above the Hall sensor, the adjustment of all other carrier positions of a specific carrier can be calculated. This is shown in tab. 4-4 "Example of carrier position adjustments" on page 86.

Index	measured carrier position	Difference	Adjustment
0	Spos - (5 * S)	99.90	-0.10
1	Spos - (4 * S)	100.10	0.10
2	Spos - (3 * S)	99.95	-0.05
3	Spos - (2 * S)	100.15	0.15
4	Spos - S	100.25	0.25
5	**Spos**	**100.00**	**0.00**
6	Spos + S	100.15	0.15
7	Spos + (2 * S)	100.20	0.20
8	Spos + (3 * S)	100.05	0.05
9	Spos + (4 * S)	99.90	-0.10
10	Spos + (5 * S)	100.10	0.10

Tab. 4-4: Example of carrier position adjustments

In this table, "Index" is the number of measured carrier positions, N = 11, with a range of 0 to N-1. "S" is the step size, and "Spos" is the Hall sensor position. The "Difference" column represents the absolute difference of carrier positions between the two measurement systems. The "Adjustment" column shows the calculated relative difference of carrier positions after subtracting "Difference" measured at "Spos". The adjustment values, per carrier, are stored in an XML file. This XML file can hold one or more CPA tables.

The CPA functionality has the following constraints.

- The number of adjustment entries in a CPA table must be an odd number.
- The adjustment value must be 0 when the measured position = sensor position (middle of table).
- The maximum number of adjustment values in a table is `CM_MAX_NR_OF_CPA_VALUES` (4001).
- The maximum number of CPA tables is `CM_MAX_NR_OF_CPA_TABLES` (128).

- The length of the file name of the XML file that holds the CPA tables is at maximum 256 characters long.
- It is the responsibility of the application that the correct CPA table is linked to each carrier.

If a link is created between a carrier and a CPA table and the CPA table functionality is enabled, the "raw" carrier positions measured by the left, middle and right Hall sensors are updated with the adjustment values in the CPA table for this carrier to actual positions.

The function `CmInitialize` disables the CPA functionality, unlinks carriers from CPA tables and removes any CPA table.

The function `CmDownloadCpaTable` is available to download a specified CPA table from a specified XML file. The return value of this function can be `NYCE_OK`, a common error code, `CTR_ERR_LMS_CPA_NOT_DISABLED` if CPA is not disabled, or `CTR_ERR_LMS_CARRIER_LINKED_TO_CPA_TABLE` if the specified CPA table is already linked to a carrier.

The function `CmLinkCpaTable` links a specified carrier to a specified CPA table. The return value of this function can be `NYCE_OK`, a common error code, `CTR_ERR_LMS_CPA_TABLE_NOT_AVAILABLE` if the specified CPA table is not available, or `CTR_ERR_LMS_CARRIER_LINKED_TO_CPA_TABLE` if the specified CPA table is already linked to a carrier.

The function `CmUnlinkCpaTable` disconnects the link between a specified carrier and its associated CPA table. The return value of this function can be `NYCE_OK`, a common error code, or the warning `CTR_WRN_LMS_CARRIER_NOT_LINKED_TO_CPA_TABLE` if no CPA table was linked to the specified carrier.

The function `CmEnableCpa` is available to enable and disable the CPA functionality for all carriers. The return value of this function can be `NYCE_OK` or a common error code.

The "raw" carrier positions of the left, middle and right sensor can be traced with the identifiers `SAC_VAR_LMS_LEFT_SENSOR_RAW_POSITION`, `SAC_VAR_LMS_MIDDLE_SENSOR_RAW_POSITION`, and `SAC_VAR_LMS_RIGHT_SENSOR_RAW_POSITION`. The "adjusted" carrier positions of the left, middle and right sensor can be traced with the identifiers `SAC_VAR_LMS_LEFT_SENSOR_POSITION`, `SAC_VAR_LMS_MIDDLE_SENSOR_POSITION`, and `SAC_VAR_LMS_RIGHT_SENSOR_POSITION`.

4.7.15 Assigning a carrier ID

Function

To assign IDs to carriers along the track, the function `CmAssignCarrierIds` is available. The inputs for this function are the track ID, the number of carriers, and an array with carrier IDs. The carrier with the lowest position value is assigned the first ID in the array, the carrier with the highest position value is assigned the last ID in the array.

Explanation

When the number of carriers along a track is known (after homing) an ID can be assigned to each of the carriers along the track. An ID is a number in the range from 0 to (`CM_MAX_NR_OF_CARRIERS-1`). Each ID within the entire LMS must be unique. The specified number of carriers (`nrOfCarriers`

parameter) defines the number of entries in the array of carrier IDs that will be assigned.

ID assignment is only allowed in state CM_TRACK_HOMED (typically) or CM_TRACK_INACTIVE. After ID assignment the track is in the state CM_TRACK_INACTIVE. Note that it is possible to call the function CmAssignCarrierIds again in the state CM_TRACK_INACTIVE after carrier IDs are already assigned by a previous call of CmAssignCarrierIds. This implies that new carrier IDs are assigned. The new carrier IDs must be unique, also compared with the previously assigned carrier IDs. After calling CmAssignCarrierIds again, the old carrier IDs are invalid.

4.7.16 Change the ID of a carrier

Function

After a carrier has been assigned an ID, it may be needed to change the carrier ID. This change can be done with the function CmChangeCarrierId. The input for this function is the current carrier ID and the new carrier ID.

Explanation

Initially, after power up, the track is homed and each carrier is assigned a carrier ID. However, the LMS may use carriers that are uniquely identified by some tag (for example an attached RFID or bar code strip). This unique identification may be used to identify a specific magnet plate (used by the CPA functionality) or a specific magnetic scale (used to set the MR sensor offset). In that case, you want to assign a specific carrier ID to the carrier. The change of a carrier ID is only allowed when the track state is CM_TRACK_INACTIVE or CM_TRACK_ACTIVE, and the carrier state is CM_CARRIER_POWERUP or CM_CARRIER_STOPPED. Note that when the carrier stopped with collision avoidance, the carrier may have stopped temporarily on a safe position. In that case the carrier state is CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS and CmChangeCarrierId returns CM_ERR_CARRIER_STATE_ERROR. If the specified current carrier ID is invalid, or the new carrier ID is not unique in the LMS, the error CM_ERR_INVALID_CARRIER_ID is returned.

4.7.17 Read and write carrier parameters

Function

With the functions CmReadCarrierPars and CmWriteCarrierPars you can read and write the carrier parameters of a specific carrier. Both functions have the same arguments. The first argument is the carrier ID, and the second argument is a pointer to a structure CM_CARRIER_PARS. The specified carrier ID does not yet have to be defined in the LMS, but it must be in the valid range.

The structure has 2 members, measurementScaleOffset and productOffset. The member measurementScaleOffset is only used in an LMS that uses MR sensors.

When the carrier with the specified carrier ID is present in the LMS, the function CmWriteCarrierPars can only be called when the carrier state of the specified carrier ID is CM_CARRIER_POWERUP or CM_CARRIER_STOPPED. The function CmWriteCarrierPars returns the error CM_ERR_TOO_MANY_CARRIER_PARAMETERS when more than CM_MAX_NR_OF_CARRIER_PARS carrier IDs are written.

Explanation

measurementScaleOffset The position of carriers on an LMS track that uses MR sensors is based on the signals that the MR sensors generate from the magnetic field of the magnetic scale that is attached to each carrier. As the position of this magnetic scale may have small differences between carriers, this difference must be corrected per carrier. One carrier in the LMS may be the "reference carrier" and this carrier may have an offset `measurementScaleOffset` 0. For all other carriers in the LMS the offset (of the magnetic scale) `measurementScaleOffset` is determined relative to the "reference carrier" and stored in the structure `CM_CARRIER_PARS`.

productOffset The position of a product on a carrier is usually accurately defined by some means of an alignment provision on the carrier. However, small differences of the placement of the alignment provision on different carriers are possible. These small differences between carriers must be corrected for each carrier. One carrier in the LMS may be the "reference carrier" and this carrier may have an offset `productOffset` 0. For all other carriers in the LMS, up to 127, the offset (of the alignment provision) `productOffset` is determined relative to the "reference carrier" and stored in the structure `CM_CARRIER_PARS`.

4.7.18 Moving carriers against an endstop

Function

You can move all carriers on a linear track against one of the end stops with the function `CmMoveCarriersAgainstEndstop`. The input for this function is the identification of the track `trackId`, movement direction `direction`, and a movement velocity `velocity` of the carriers. When `direction` is +1, the movement is in the positive direction. When `direction` is -1, the movement is in the negative direction. The function is only allowed when the track state is `CM_TRACK_IDLE` or `CM_TRACK_INACTIVE`.

Explanation

Moving the carriers against an end stop is an asynchronous operation. The carriers start moving toward the end stop when the function `CmMoveCarriersAgainstEndstop` returns. Synchronization on the completion of this function is possible with the function `CmSynchronize` with the `CM_REQ_TRACK_CARRIERS_AGAINST_ENDSTOP` synchronization request.

4.7.19 Moving a carrier to a position

Function

To move a carrier to a specific position, the function `CmMoveToPosition` is available. The input for this function is the carrier ID and a pointer to a structure `CM_MOVE_PARS`. This structure contains the required end position and the constraints for velocity, acceleration, and jerk.

Synchronization on the movement is possible with the function `CmSynchronize` with the synchronization request `CM_REQ_CARRIER_MOVEMENT_STARTED` or `CM_REQ_CARRIER_MOVEMENT_STOPPED`, or by reading the carrier state (carrier settled at end position).

Explanation

This function moves a carrier to the specified position. The initial velocity of the carrier is not necessarily 0. The CM software knows from its internal administration which coil or coils are currently controlling the carrier. With this information and the specified position the CM software determines which coils are involved in the movement. The coil(s) controlling the carrier start the execution of the movement immediately; the other coils, if any, start the generation of the 3rd order setpoint profile for the carrier when the carrier arrives in their control range.

- If the carrier is moving when the `CmMoveToPosition` function is called, the movement to the new position may require a direction change. If the movement in the opposite direction is required to stop the carrier(s) using a 3rd order setpoint profile or the stop position is beyond the track end position, a 2nd order setpoint profile with the maximum acceleration specified in the `CM_STOP_PARS` structure is used instead of the 3rd order setpoint profile.
- If the carrier is moving when the `CmMoveToPosition` function is called with a different end position, the carrier will always move to the new end position. The carrier will always move to the new end position, even if a reverse movement must be done when the end position cannot be reached with a single movement based on the specified movement profile. It is the responsibility of the application that carriers will not bump against each other or against an end stop. If the carrier is close to an end stop and the movement is changed "on the fly", it is advised to specify an acceleration that is equal or higher than the acceleration of the current movement to prevent a bump against an end stop.

The generated position profile is of third order. The generated profile is time optimal, taking into account constraints (maximum absolute values) for velocity, acceleration, and jerk. If jerk is set to `NYCE_INFINITE` the generated profile is of second order. Note that the constraints on velocity, acceleration and jerk set by this function are affected by the feed override if the feed override is not equal to 1.

Moving a carrier to a specified position is only allowed in the track state `CM_TRACK_ACTIVE`. There are no carrier state constraints. After the movement has been accepted, the carrier state is `CM_CARRIER_MOVING` if the feed override is not equal to 0 and `CM_CARRIER_SUSPENDED` otherwise. After completion of the movement, the carrier state is `CM_CARRIER_STOPPED`.

If a track controls more than one carrier, the following conditions can occur.

Assume the first situation as shown in fig. 4-20 "Condition causing SAC_AX_ERR_LMS_CARRIER_SWITCH_ERROR" on page 91. Carrier 1 is positioned above coil A and B, and carrier 2 is positioned above coil B and C when the track is activated. Because carrier 2 is further in the coil control range of coil B, carrier 2 is assigned to coil B and C, and carrier 1 is only assigned to coil A.

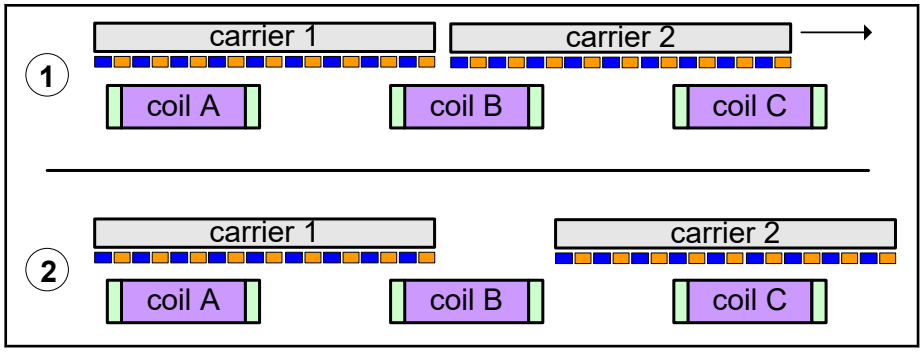


Fig. 4-20: Condition causing SAC_AX_ERR_LMS_CARRIER_SWITCH_ERROR

In situation 2, carrier 2 has moved due to a "move to position" command, and carrier 1 is held at its position by coil A. As carrier 2 moves away from coil B, carrier 1 is further in the coil control range of coil B, and coil B wants to control carrier 1. The error `SAC_AX_ERR_LMS_CARRIER_SWITCH_ERROR` is generated, because coil B does not have a setpoint for carrier 1.

fig. 4-21 "Condition causing SAC_AX_ERR_LMS_SETPOINT_ERROR" on page 91 shows another condition that can cause an error. Carrier 1 is positioned above coil A and coil B, see situation 1. In situation 2, carrier 1 is held at its position by coil A and carrier 2 is moved above coil B and is further in the coil control range than carrier 1. This means that coil B now controls carrier 2. When carrier 2 is moved toward coil C (situation 3), carrier 1 eventually will be further in the coil control range of coil B. At that moment the error `SAC_AX_ERR_LMS_SETPOINT_ERROR` is generated.

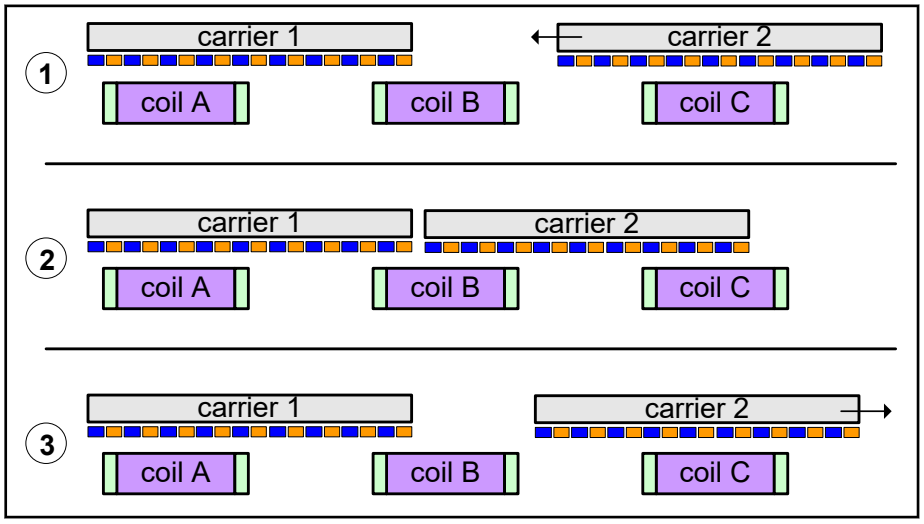


Fig. 4-21: Condition causing SAC_AX_ERR_LMS_SETPOINT_ERROR

A workaround is available for both conditions. First, use `CmMoveToPosition` with the current setpoint for the carrier that stays above the coil, then use `CmMoveToPosition` for the carrier that moves from the coil.

Modulo track

If the track is a modulo track, you can move from position A to position B either in positive direction or in negative direction.

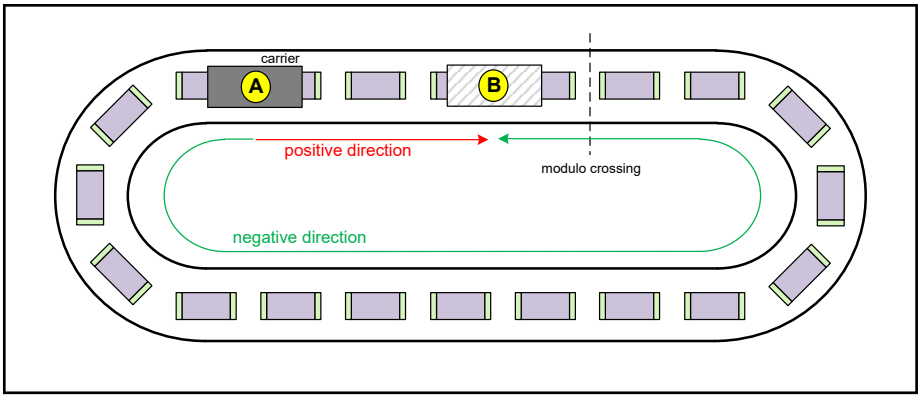


Fig. 4-22: Movement on a modulo track

Only if the track is a modulo track, the sign of the specified velocity determines the movement direction. If the sign of the velocity specification is positive, the movement is in the positive direction. If the sign of the velocity specification is negative, the movement is in the negative direction.

4.7.20 Moving a carrier adjacent to its predecessor

Function

For the start of this type of movement, the function `CmMoveAdjacent` is available. The inputs for this function are the following.

- a carrier ID
- a pointer to a `CM_MOVE_PARS` structure
- the required mutual distance between the carrier and its predecessor

Synchronization on the movement is possible with the function `CmSynchronize` with the synchronization request `CM_REQ_CARRIER_MOVEMENT_STARTED`, `CM_REQ_CARRIER_MOVEMENT_JOGGING`, `CM_REQ_CARRIER_MOVEMENT_STOPPED`, or by reading the carrier state variable.

Explanation

This function performs a movement with a carrier, which results in the carrier moving at a fixed distance of its preceding carrier (predecessor) on the track. The predecessor must be moving at a constant velocity when this command is executed. The fixed distance, `mutualDistance`, between the carriers is based on the length of the carriers, not the length of the magnet plates.

CM checks

- whether the actual setpoint velocity of the predecessor is constant at the moment the command is received (carrier state of the predecessor must be `CM_CARRIER_JOGGING`, `CM_CARRIER_MOVING`, or `CM_CARRIER_STOPPED`),
- whether the feed override is at a constant level,
- whether the member `maxJerk` of the argument `pCmMovePars` is set to `NYCE_INFINITE`, because `CmMoveAdjacent` only supports a second order movement profile,
- whether the magnet plates of the carrier and its predecessor will not move in the control range of the same coil, unless the magnet plates of

these carriers have a mutual distance which is a multiple of the magnetic period.

The carrier state of the carrier executing the "move adjacent" must be CM_CARRIER_STOPPED, else an error is returned. After the command has been accepted, the required setpoint profile for the carrier is calculated based on the actual position and velocity of the predecessor and the input parameters of the function (position at which the end velocity must equal the velocity of predecessor, velocity and acceleration constraints, and mutual distance between carrier and predecessor).

Once the movement is started, the carrier state changes to CM_CARRIER_MOVING.

The distance to be covered during a move adjacent before the end velocity is reached equals the difference between initial and desired mutual distance augmented by the distance covered in the mean time by the predecessor. This distance is referred to as the gap.

Bridging the gap is done, taking into account a velocity and acceleration constraint. The gap must be closed at a specified position. Bridging the gap is not a time-optimal setpoint profile. Instead, the deceleration during the profile is minimized, in order to reduce bumping. If the minimized deceleration is still greater than the maximum acceleration, the gap is closed with maximum acceleration and velocity, thus too late, and a warning is generated. When the gap is closed, the end velocity of the carrier remains equal to the velocity of the predecessor at the moment the function was called. The carrier state changes to CM_CARRIER_JOGGING or CM_CARRIER_STOPPED, depending on the velocity of the predecessor.

It is important that after calling CmMoveAdjacent the velocity of the predecessor is not changed. If the velocity is decreased, a collision may occur before the specified mutual distance is obtained. If the velocity of the predecessor is increased, the mutual distance is not correct after the "move adjacent" functionality.

Note that the constraints on velocity and acceleration, set by this function, are affected by the feed override if it is not equal to 1.

4.7.21 Moving a carrier along a profile

Function

To move a carrier along a profile, the function CmMoveProfile is available. The input for this function is the carrier ID, and a pointer to a structure CM_PROFILE_PARS. This structure contains the intermediate profile points (at most CM_MAX_NR_OF_PROFILE_POINTS may be specified).

Synchronization on the end of the movement is possible with the function CmSynchronize with the synchronization request CM_REQ_CARRIER_MOVEMENT_JOGGING or CM_REQ_CARRIER_MOVEMENT_STOPPED, or by reading the carrier state variable.

Explanation

This function makes it possible to move a carrier along a profile specified by means of an array of intermediate points. At each intermediate point, the position and velocity are specified. The initial position equals the actual setpoint position of the carrier and the initial velocity is equal to 0. The profile must be monotonous in position (that is, velocity always in the same direction or 0), otherwise the profile is not executed (parameter error). Note that the sign of the specified velocity value defines the direction of the movement. A

negative velocity value defines a profile with a movement in the negative direction (from high to low position). If the velocity at the last position in the profile array equals 0, the carrier stops at the specified end position. Otherwise, after completion of the profile, the carrier proceeds from the end position with a movement at constant velocity equal to the specified end velocity.

Between the initial position and velocity and the first profile point, as well as between each two successive profile points in the array (that is, on each segment of the profile), a position profile of second order is determined, which satisfies the initial and final conditions for position and velocity of each segment. By the boundary values of the segment (four in total), the duration and the (three) coefficients of the second order polynomial are uniquely determined (if existing). If no solution exists for one or more of the segments or if the acceleration on the segment exceeds the acceleration constraint of the track, an error is returned. Note that the constraints on velocity and acceleration for the track, used by this function for parameter validation, are affected by the feed override if it is not equal to 1.

The "move profile" command is only executed if the track on which the carrier is situated is in the state `CM_TRACK_ACTIVE`. The carrier state must be either `CM_CARRIER_STOPPED` or `CM_CARRIER_SUSPENDED`. During the execution of the profile, the carrier state changes to `CM_CARRIER_MOVING`. After completion of the profile, the carrier state is `CM_CARRIER_STOPPED` if the end velocity is 0, or `CM_CARRIER_JOGGING` otherwise.

4.7.22 Moving a carrier at constant velocity

Function

To move a carrier at a constant velocity, the function `CmJog` is available. The input for this function is the carrier ID, (end) velocity, and acceleration. The velocity can be a negative or a positive value.

Synchronization on the end of the acceleration is possible with the function `CmSynchronize` with the synchronization request `CM_REQ_CARRIER_MOVEMENT_JOGGING`, or by reading the carrier state variable.

Explanation

The function accelerates the specified carrier to the specified velocity using a 2nd order profile, unless the movement is suspended, because feed override is 0. After completion of this profile the carrier continues to move at constant velocity. The track state must be `CM_TRACK_ACTIVE`, and the carrier state must be `CM_CARRIER_STOPPED` or `CM_CARRIER_SUSPENDED`. If feed override is 0, the carrier state changes to (or remains) `CM_CARRIER_SUSPENDED`. If feed override is not 0, the carrier state changes from `CM_CARRIER_STOPPED` to `CM_CARRIER_MOVING`. During the acceleration movement the carrier state is `CM_CARRIER_MOVING`. After completion of the acceleration movement the carrier state is `CM_CARRIER_JOGGING`.

4.7.23 Stopping a carrier

Function

Stopping a carrier is possible with the function `CmStopCarrier`. The input for this function is the carrier ID and the data structure `CM_STOP_PARS` which specifies the maximum allowable deceleration `stopAcceleration` and jerk `stopJerk`.

Synchronization on the end of the movement (carrier stopped) is possible with the function `CmSynchronize` with the synchronization request `CM_REQ_CARRIER_MOVEMENT_STOPPED`, or by reading the carrier state variable.

Explanation

This function stops a carrier with a third order setpoint profile. The stop profile is time-optimal, taking into account the maximum allowable deceleration and jerk of the carrier. If the value for jerk is `NYCE_INFINITE`, a 2nd order setpoint profile is used.

Stopping a carrier is possible if the track is in the `CM_TRACK_ACTIVE` state, but also in the `CM_TRACK_ERROR` or `CM_TRACK_FATAL_ERROR` state. In the latter two conditions, the execution of the stop profile may or may not be possible, as at least one coil on the track is in an error state. For that reason, each coil on the track, which "sees" the carrier, appears in the coil control range, the initial value of the setpoint velocity is taken equal to the actual carrier velocity.

In contrary to the move functions discussed in the previous sections, the constraint on deceleration and jerk, set by this function, is **not** affected by the feed override (see chapter 4.7.25 "Using feed override" on page 96). A user-defined feed override setting is overruled and a feed override of 1.0 is used. The stop may be part of an error handling procedure at application level, as an attempt to avoid damage. If the track is in the `CM_TRACK_ACTIVE` state, the carrier state is always `CM_CARRIER_STOPPED` after completion of a possible stop profile.

4.7.24 Synchronized start of multiple carriers

Function

With the synchronized start of multiple carriers functionality it is possible to synchronize the start of motion commands for multiple carriers. The following steps are needed for a synchronized start of motion commands for multiple carriers.

1. Define a group of carriers for which the motion must start synchronized.
2. Send the motion command to each carrier of the sync group. Note that the motion does **not** start.
3. Start the motion commands for the sync group.

The prerequisite for correct synchronization is that all carriers in the group are at standstill. The consequence of this prerequisite is that only motion commands are supported for carriers in a group that support a movement from standstill. The following motion commands support motion from standstill.

- `CmMoveToPosition` (if carrier is initially at standstill)
- `CmMoveProfile`
- `CmMoveAdjacent`

Default, motion commands are executed at the moment coil(s), that control the carrier, receive the command if a carrier does **not** belong to a sync group. If the carrier belongs to a sync group, the execution of a motion command is postponed until the start for the sync group is issued.

Explanation

A group of carriers is defined with the function `CmDefineSyncGroup`. The function has 2 input parameters, an array of carrier identifiers and the length of the array, and one output parameter, the sync group identifier. A carrier can only belong to one sync group at a time. Up to 16 sync groups are supported. If all sync group identifiers are in use, the application must delete an existing sync group before a new sync group can be created. The carriers in the sync group may belong to different tracks. The carriers of the sync group are in the stopped state (`CM_CARRIER_STOPPED`) and the track they belong to is in the active state (`CM_TRACK_ACTIVE`).

With the function `CmGetSyncGroup` the carriers that belong to the specified sync group can be retrieved. The input parameter is the sync group identifier, and the output is 2 parameters, the number of carriers in the group and an array of carrier identifiers. The carrier identifiers in the array are ordered in the same way as they were when the sync group was defined.

To start the execution of the movement for all carriers in the sync group the functions `CmStartSyncGroup` and `CmEnableStartTrigger` are available. The state of the track(s) on which the carriers in the sync group are located must be `CM_TRACK_ACTIVE`. After the successful completion of the motion command the state of the carriers in the sync group changes from `CM_CARRIER_STOPPED` to `CM_CARRIER_MOVING`, and the sync group is automatically deleted.

With the function `CmDeleteSyncGroup` a sync group can be deleted. The only parameter for this function is the sync group identifier. If a motion command has been issued for a carrier (but has not yet been started) and the sync group is deleted, the motion command is cancelled. Likewise, if a carrier is removed from a sync group, any pending motion command for that carrier is cancelled.

Remarks

- When the function `CmInitialize` or `CmShutdown` is called, all defined sync groups are deleted.
- When the function `CmDisconnect` is called, all defined sync groups created by the client that calls the function `CmDisconnect` are deleted.
- When the function `CmRemoveCoil` is called and the removed coils control a carrier that belongs to a sync group, this carrier is removed from the sync group.
- When the function `CmRemoveCarrier` is called and the removed carrier belongs to a sync group, this carrier is removed from the sync group.
- When the feed override and start trigger functionality is also used, the carriers will not start a movement until all preconditions are valid.

With the function `CmStopCarriers` a stop of all carriers defined in the specified array is possible. The function has 3 parameters, the number of carriers, an array with carrier identifiers, and a structure `CM_STOP_PARS` that specifies the maximum allowed stop acceleration and jerk. This stop acceleration and jerk is used for all specified carriers. The initiation of the stop commands is possible for carriers at standstill and for carriers that execute a movement. The function `CmStopCarriers` is not allowed when a carrier state is `CM_CARRIER_POWERUP`.

4.7.25 Using feed override

Function

The feed override is changed by means of the function `CmSetFeedOverride`. The parameters of this function specify the track and the requested feed override value. The `CmSetFeedOverride` command is ignored if the feed override value to be set, equals the current feed override value, while the feed override is constant.

Explanation

By this function, the feed override of all carriers in control loop on a track is changed simultaneously. This allows all movements to be executed in slow motion or to speed up the motion. Effectively, a (constant) feed override value α implies that the setpoint velocity is multiplied by a factor α and the setpoint acceleration by a factor α^2. This way, the path traveled by a carrier is unchanged in relation to the path of other carriers, that is, mutual distances are not changed. Feed override has no effect on carriers in velocity loop mode. Thus, you cannot stop a home movement by setting the feed override to 0.0.

This function can be used to start all carriers in control loop mode simultaneously by changing the feed override from 0.0 to 1.0. When a new feed override value is specified, an immediate transition between the current and required feed override is performed.

Note that if the feed override is greater than 1.0, velocity and acceleration constraints, set by `CmMoveToPosition`, `CmMoveAdjacent` or `CmMoveProfile` may be violated. If the feed override is less than 1.0, these movements when said to be time optimal are no longer time optimal with respect to the original constraints specified.

Changing of the feed override is possible in all track states, but in `CM_TRACK_ERROR` or `CM_TRACK_FATAL_ERROR` state the feed override parameter must be 0.0. In these two states the execution of the feed override change may or may not be effective, as at least one coil on the track is in an error state. Nevertheless, changing feed override to 0.0 may be part of an error handling procedure at application level as an attempt to avoid damage.

If the track is in the `CM_TRACK_ACTIVE` state, all carriers on the track must be in the `CM_CARRIER_STOPPED` or `CM_CARRIER_SUSPENDED` state.

After successful completion of the `CmSetFeedOverride` command in the track state `CM_TRACK_ACTIVE`, the state of each carrier on the track may change from `CM_CARRIER_SUSPENDED` to `CM_CARRIER_MOVING` (if feed override changes from 0.0 to any other value). See fig. 4-4 "State Transition Diagram for carriers in CM" on page 55 for details.

4.7.26 Removing / Adding a carrier

Function

To remove a carrier from a track the function `CmRemoveCarrier` is available, which has the carrier ID as input.

To add a carrier on a track the function `CmAddCarrier` is used. This function requires the carrier ID and the approximate position where the carrier is added as input.

Removal and addition of a carrier is only allowed if the carriers already present on the track have been located and identified, thus when the track state is `CM_TRACK_INACTIVE` or `CM_TRACK_ACTIVE`.

When a carrier is removed from the system and a CPA table is linked to that carrier, the link to the CPA table is also removed. When that carrier is (again)

added to the system, the application must also re-establish the link to the CPA table.

Explanation

A carrier can be removed from a track. From that moment on, the carrier is no longer part of the LMS. A carrier can be removed from a track without any restrictions if the state of the track is `CM_TRACK_INACTIVE`. If the state of the track is `CM_TRACK_ACTIVE`, the velocity of the carrier to be removed must be 0, so the carrier must be in the state `CM_CARRIER_STOPPED` or `CM_CARRIER_SUSPENDED`. Do the following steps to remove a carrier from a track.

1. Call the function `CmRemoveCarrier`. The carrier ID and index is removed from the administration.
2. Remove the carrier physically from the track. The carrier is no longer detected by the sensors.

It is possible that the sensors detect more than one transition of "detected" to "not detected", because of multiple variations in the magnetic field from the carrier. The carrier removal may not be successful and sensor errors are reported. This problem can be avoided if the sensor detection is disabled. The sequence of actions is as follows.

1. Call the function `CmRemoveCarrier`.
2. Call the function `CmEnableSensorDetection` to disable detection for the sensors that must detect the carrier. The carrier is no longer detected by the sensors.
3. Remove the carrier physically from the track.
4. Call the function `CmEnableSensorDetection` to enable detection for the sensors that must detect the carrier.

A carrier can be added on a track. From that moment on, the carrier is part of the LMS. If a carrier is added, the approximate position at which the carrier is added must be known, accurate by half the pitch of the magnets. The exact position of the carrier is determined from the specified position and the actual phase of the sine and cosine signals from the LMS sensors. Do the following steps to add a carrier on a track.

1. Call the function `CmEnableSensorDetection` to disable detection for the sensors that must detect the carrier.
2. Call the function `CmAddCarrier`. The carrier ID and index is added to the administration.
3. Add the carrier physically on the track.
4. Call the function `CmEnableSensorDetection` to enable detection for the sensors. The carrier is detected by the sensors.

If a carrier is added on a track which is in the state `CM_TRACK_INACTIVE`, the carrier state is set to `CM_CARRIER_POWERUP`. If a carrier is added on a track which is in the `CM_TRACK_ACTIVE` state, the carrier state is set to `CM_CARRIER_STOPPED` if the sensor detection is enabled, or `CM_CARRIER_POWERUP` if the sensor detection is disabled.

If `CmAddCarrier` is called with a non-existing carrier ID and sensor detection is enabled, the error `SAC_AX_ERR_LMS_xxx_SENSOR_INACTIVE` is generated, where "`xxx`" is `LEFT`, `MIDDLE` or `RIGHT`.

If CmAddCarrier is called to add a carrier at a location which is already in use by another carrier, the error CM_ERR_PARAMETER_ERROR is returned.

You can use the function CmSynchronize with the synchronization request CM_REQ_CARRIER_MOVEMENT_STOPPED to synchronize on the completion of the CmAddCarrier operation. Note that the track state must be CM_TRACK_ACTIVE.

4.7.27 Move a carrier from track to track

Function

With the function CmMoveToTrack you can move a carrier from one track to an adjacent track in position mode. The carrier starts the movement on one track and stops the movement on the adjacent track. Both tracks are part of the same LMS. The functionality of CmMoveToTrack is basically identical to the functionality for moving a carrier to an end position, but with the addition that the carrier is transferred from one track to an adjacent track. During this movement the carrier identifier remains assigned to the carrier.

☞ CmRemoveMovingCarrier, CmAddMovingCarrier, and CmCancelAddMovingCarrier adds/removes a carrier from one track to another track from one LMS to another LMS, whereas CmMoveToTrack adds/removes a carrier from one track to another track in the same LMS.

Explanation

The movement of CmMoveToTrack can be started when the carrier is at standstill or already moving. The carrier moves to the specified side of the track, the track begin or track end. The movement is stopped when the carrier is at the specified (absolute value) end position. The generated position profile is of third order and is time-optimal taking into account the specified constraints (maximum absolute values) for velocity, acceleration and jerk. Track constraints of both tracks for the maximum velocity, acceleration and jerk of a carrier are also taken into account. The track constraint that has the lowest value is used. If the specified value for jerk is NYCE_INFINITE, a 2nd order setpoint profile is used.

Moving a carrier from track to track with CmMoveToTrack is only allowed when

- the two involved tracks are next to each other,
- the track state of both tracks is CM_TRACK_ACTIVE,
- feed override settings for both tracks is identical, and not 0,
- no carriers are between the carrier involved in the move to track operation and the end of the track toward which that carrier is moving.

There are no carrier state constraints.

After the command CmMoveToTrack is accepted, the carrier state changes to CM_CARRIER_MOVING. After the completion of the movement the carrier state changes to CM_CARRIER_STOPPED.

The track on which the carrier is at the start of the move to track operation is called the origin track. The track to which the carrier moves with the move to track operation is called the destination track. Besides the events that are generated when a carrier is moved to an end position, 2 additional events are generated when a carrier is moved from track to track.

1. The event `CM_EV_TRACK_MOVING_CARRIER_EN` indicates that a moving carrier enters the destination track. This event is generated when the outer sensor of the destination track detects the carrier. The event data contains the carrier identifier of the detected carrier.
2. The event `CM_EV_TRACK_MOVING_CARRIER_LE` indicates that a moving carrier leaves the origin track. This event is generated when the outer sensor of the origin track no longer detects the carrier. The event data contains the carrier identifier of the no longer detected carrier.

It is possible that the `CM_EV_TRACK_MOVING_CARRIER_LE` event occurs before the `CM_EV_TRACK_MOVING_CARRIER_EN` event is generated.

You can use the function `CmSynchronize` with the synchronization requests `CM_REQ_CARRIER_MOVEMENT_STARTED` and `CM_REQ_CARRIER_MOVEMENT_STOPPED` to detect asynchronous operation moments while a carrier is moved from track to track.

After the move to track function is started, the application is responsible not to make changes to the preconditions that apply for the destination track. If the feed override for the origin or destination track is changed while the carrier is still on the origin track, the error `CM_ERR_MOVE_TO_TRACK_EXECUTING` is returned. Other changes to the destination track, such as deactivate track or RTLC, while the carrier is still on the origin track also generate the error `CM_ERR_MOVE_TO_TRACK_EXECUTING`.

Conditions

When the `CmMoveToTrack` movement is started, the carrier is completely controlled by coils and only detected by sensors of the origin track. This implies that it is not possible to start this movement when the carrier is detected by sensors of more than one track. In this case the error `CM_ERR_CARRIER_POSITION_ERROR` is returned.

When the movement is stopped, the carrier must be completely controlled by coils and only detected by sensors of the destination track.

During a certain phase of the movement the carrier is controlled by coils of both tracks. During this phase one of the tracks controls the major part of the carrier. This track is called the controlling track, while the other track is called the detecting track. A track controls the major part of the carrier when the position of the carrier overlaps the control range of the outer coil more than the control range of the outer coil of the other track, see fig. 4-23 "Definition of controlling track and detecting track" on page 100.

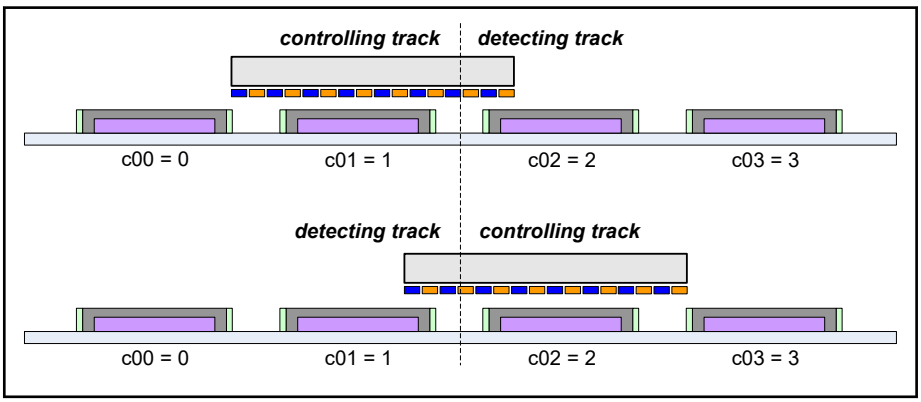


Fig. 4-23: Definition of controlling track and detecting track

When the carrier position, velocity, acceleration, jerk, carrier state and carrier controller state is retrieved with the functions `CmReadSingleCarrierData`, `CmReadCarrierData`, or `CmReadCarrierVariable` during this phase, the information returned is from the controlling track. You can use the variable `CM_VAR_CARRIER_POSITION_DETECTING_TRACK` to read the position of the carrier on the detecting track. With the variable `CM_VAR_CARRIER_DETECTING_TRACK_ID` you can read the track identifier of the detecting track.

Asynchronous error handling during CmMoveToTrack

If an asynchronous error occurs during the execution of the move to track function, the `CM_EV_TRACK_ASYNC_ERROR_OCCURRED` event is generated. CM checks whether a `CmMoveToTrack` operation is active for the involved carrier. If that is the case, the `CmStopCarrier` command is given for this carrier. The function `CmStopCarrier` issues a stop command for this carrier on both involved tracks, and this results in the following error situations, depending on the position of the carrier.

The following description of the error situations assumes that

- the physical order of the tracks in a multi-track system is track0 - track1 - track2,
- the current position of the carrier is above track0,
- and the application issues a `CmMoveToTrack` command for the carrier to move from track0 to track2.

This is an invalid command, because track2 is not adjacent to track0. However, CM has no knowledge of the physical track layout, thus CM cannot recognize that the command is invalid. At a certain moment the carrier enters track1 , because track2 is not adjacent to track0. Track1 detects an unexpected carrier on its outer sensor and the outer coil generates an asynchronous error. Track1 goes to the `CM_TRACK_ERROR` or `CM_TRACK_FATAL ERROR` state. Track1 initiates a stop for the carrier movement. The carrier can come to a standstill above the origin track and adjacent track or only above the adjacent track. Depending on the position of the carrier when the carrier movement stopped, the state of the origin track, and the state of the adjacent track, a specific recovery method is needed to position the carrier above a track with track state `CM_TRACK_ACTIVE`.

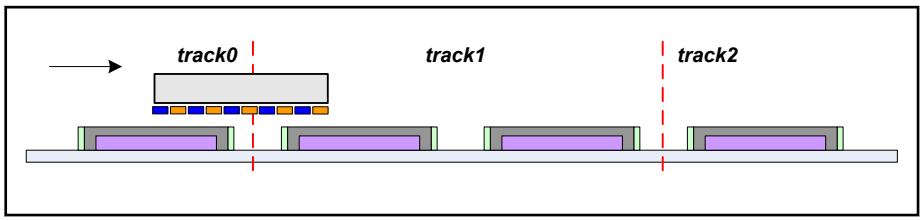


Fig. 4-24: Move to track - error situation 1

In error situation ①, the carrier comes to a standstill above the origin track and the adjacent track. The carrier can be positioned above one of the tracks with the velocity mode functionality. Call `CmStartVelocityModeCoils` for both tracks to activate the velocity control loop of the outer coils to move the carrier above one of the tracks. When the carrier is completely on one track call `CmStopVelocityMode` for both tracks. Subsequently, use CM commands to get the track(s) in the `CM_TRACK_ACTIVE` state. Use the default recovery method to get the track state of the destination track back to `CM_TRACK_ACTIVE`.

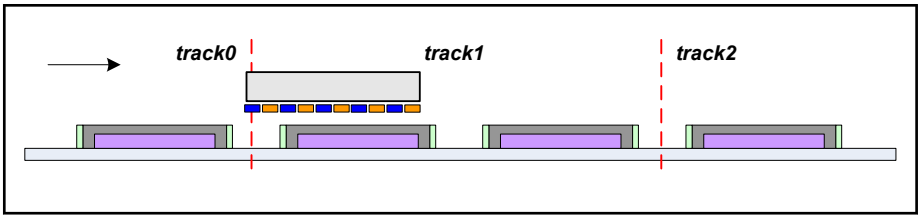


Fig. 4-25: Move to track - error situation 2

In error situation ②, the carrier has sufficient momentum to leave the coil control range of the last coil of track0, and comes to a standstill after the carrier has left the detection range of the last sensor of track0. As a result, the carrier is not detected by either of the tracks involved in the move to track command. Track1 will generate a fatal error but, as CM has no knowledge of the order of the tracks, CM is not aware that this error is related to the current execution of the move to track command. The carrier has left track0 but has not arrived at track2.

CM calculates at the start of a `CmMoveToTrack` command the time that the carrier needs to move past the outer sensor of the origin track. A margin is added to that time, and this time is stored in the carrier administration. During this time CM checks whether the carrier leaves the detection range of the last sensor of the origin track, and the carrier is detected by the outer sensor of the destination track. This is specified by the track layout constraints for sensor positions. When that time is elapsed and the check failed, `CmStopCarrier` is executed for that carrier and the asynchronous error `SAC_AX_ERR_MOVE_TO_TRACK_TIMEOUT` for the outer coils at the track side of the origin and destination track is activated. The default and minimum error handler for `SAC_AX_ERR_MOVE_TO_TRACK_TIMEOUT` is `SAC_ERRH_POS_CORRUPTED`. The track state of the origin and destination track goes to `CM_TRACK_FATAL_ERROR`. A home operation is required for all involved tracks.

CM functions allowed / not allowed during CmMoveToTrack

- A call to `CmStopCarrier` or `CmStopCarriers` is always allowed when a carrier is moving from track to track.
- A call to `CmMoveToTrack`, `CmMoveToPosition` or `CmRemoveMovingCarrier` is only allowed when a carrier is moving from track to track and the carrier is no longer above the origin track and has entered the destination track. In all other situations a call to `CmMoveToTrack`, `CmMoveToPosition` or `CmRemoveMovingCarrier` returns the error code `CM_ERR_MOVE_TO_TRACK_EXECUTING`.
- When a carrier is moving from track to track, it is not allowed to call `CmMoveToTrack` if the track and track side where the moving carrier should enter are the same. In this case `CmMoveToTrack` returns the error code `CM_ERR_MOVE_TO_TRACK_EXECUTING`.
- When a carrier is moving from track to track, it is not allowed to call `CmAddMovingCarrier` if the track side where the moving carrier is added is the same as the track side where the moving carrier enters. In this case `CmAddMovingCarrier` returns the error code `CM_ERR_MOVE_TO_TRACK_EXECUTING`.
- If `CmAddMovingCarrier` is already called for the same track side where the carrier should enter the track after a call to `CmMoveToTrack`,

the error CM_ERR_ADD_MOVING_CARRIER_ALREADY_PENDING is returned.

Recovery options if a carrier stops during track to track movement

While a carrier is moving from track to track, it is possible that the carrier movement stops.

This can occur when CmStopCarrier or CmStopCarriers is called, or when an asynchronous error occurs. When an asynchronous error occurs the carrier is stopped by the execution of the error handlers on the track that generated the asynchronous error and with a call to CmStopCarrier for the other track. If an asynchronous error occurs, the error CM_ERR_ASYNC_ERROR_OCCURRED is returned by CmSynchronize for the synchronization request CM_REQ_CARRIER_MOVEMENT_STOPPED.

- **Carrier stopped and completely above one track**

  If the carrier is at standstill completely above one of the tracks and the track state is CM_TRACK_ACTIVE, no recovery method is needed. The carrier can immediately be positioned on this track with CmMoveToPosition.

  If the track state is CM_TRACK_ERROR or CM_TRACK_FATAL_ERROR, the default error recovery method must be executed before the carrier can be positioned on this track.

- **Carrier stopped and partly above both tracks**

  If the carrier is at standstill above both tracks, a recovery method must be used to position the carrier on one of the tracks. A carrier is above two tracks when the carrier is detected by at least one sensor of the origin track and by at least one sensor of the destination track.

  fig. 4-26 "Carrier standstill above both tracks" on page 103 shows the condition where the carrier has entered the destination track, but has not completely left the origin track.

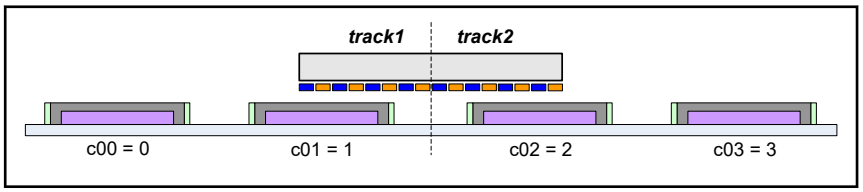


Fig. 4-26: Carrier standstill above both tracks

  – **Track state is CM_TRACK_ACTIVE or CM_TRACK_ERROR**

    If the carrier is at standstill above 2 tracks and both tracks are in the track state CM_TRACK_ACTIVE or CM_TRACK_ERROR, the carrier can be positioned on the controlling track with the most chance for success, because the controlling track can exert more force on the carrier than the detecting track. The recovery method to position the carrier above the controlling track in the case that track1 is the controlling track and track2 is the detecting track, is as follows.

    Recovery method

    1. Call CmResetError for the track(s) in CM_TRACK_ERROR state.

2. Call `CmRemoveCarrier`.
3. Call `CmActivateTrack` for the track(s) that were in the `CM_TRACK_ERROR` state.
4. Call `CmEnableSensorDetection` for track1. Disable the sensor detection of the sensors that detect the carrier.
5. Call `CmAddCarrier` for track1. Use for the add position the carrier position on track1 where the carrier stopped.
6. Call `CmEnableSensorDetection` for track1. Enable the sensor detection of the sensors that detect the carrier.
7. Call `CmMoveToPosition` to a position of track1.
   - If the carrier is not moving, call `CmStopCarrier`.
   - If the carrier is moving, the carrier will stop completely above track1.

If positioning the carrier on the controlling track fails, try to position the carrier on the detecting track. The recovery method to position above the detecting track uses the same steps, but the calls to the CM functions for track1 and track2 must be swapped.

If both recovery attempts do not work, position the carrier on one of the tracks with the velocity mode functionality. Call `CmStartVelocityModeCoils` for both tracks to activate the velocity control loop of the outer coils to move the carrier above one of the tracks. When the carrier is completely on one track call `CmStopVelocityMode` for both tracks. Subsequently, use CM commands to get the tracks operational again.

– **Track state is** `CM_TRACK_FATAL_ERROR`

  If the carrier is at standstill above 2 tracks and the track state of both tracks is `CM_TRACK_FATAL_ERROR`, the carrier can be positioned above one of the tracks with the velocity mode functionality. Call `CmStartVelocityModeCoils` for both tracks to activate the velocity control loop of the outer coils to move the carrier above one of the tracks. When the carrier is completely on one track call `CmStopVelocityMode` for both tracks. Subsequently, use CM commands to get the tracks operational again.

  If the carrier is at standstill above 2 tracks and the track state of one track is `CM_TRACK_FATAL_ERROR` and the track state of the other track is `CM_TRACK_ACTIVE` or `CM_TRACK_ERROR`, the carrier can be positioned with the most chance of success above the track that is not in the `CM_TRACK_FATAL_ERROR` state. The recovery method to position the carrier above the track which is not in the `CM_TRACK_FATAL_ERROR` state uses the same recovery steps if the state of track1 is `CM_TRACK_ACTIVE` or `CM_TRACK_ERROR` and the state of track2 is `CM_TRACK_FATAL_ERROR`.

  If the recovery method does not work, position the carrier on one of the tracks with the velocity mode functionality. Call `CmStartVelocityModeCoils` for both tracks to activate the velocity control loop of the outer coils to move the carrier above one of the tracks. When the carrier is completely on one track call `CmStopVelocityMode` for both tracks. Subsequently, use CM commands to get the tracks operational again.

Mechanical restriction

The mechanical guidelines for the distance between the centers of coils also applies for the outer coils of the two tracks involved with the track to track movement. With this (additional) guideline the carrier is always controlled by a coil when the carrier is at standstill covering both tracks. See also chapter 2.5 "Design guidelines for the NYCe 4000 LMS" on page 19.

For tracks used in the track to track movement functionality, a certain minimum distance between the tracks must exist. The distance is required, because it must be possible to move a carrier to the end (or begin) of the track while no sensor of the adjacent track can detect the carrier. This mechanical guideline must be respected for all the tracks of the system that use the track to track movement functionality. fig. 4-27 "Correct and wrong tracks layout" on page 105 shows a correct tracks layout and a wrong tracks layout.

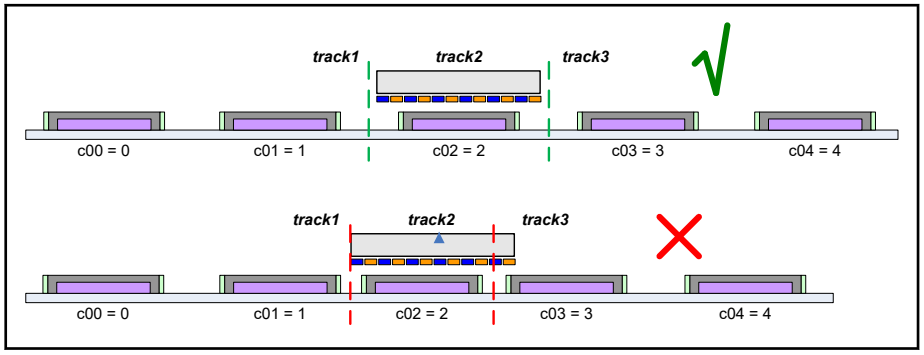


Fig. 4-27: Correct and wrong tracks layout

In the correct tracks layout the carrier can be moved to a position on track2 where no sensor of the adjacent tracks can detect the carrier. This is also possible for track1 and track3. The track to track movement functionality can be used in this tracks layout. In the wrong tracks layout the carrier cannot be moved to a position on track2 where no sensor of the adjacent tracks can detect the carrier. In this layout the track to track movement functionality cannot be used.

4.7.28 Removing / Adding / Cancelling a moving carrier from / to an LMS

Function

An LMS has the limitation of a maximum of 164 coils. A large track may need more than 164 coils and thus requires an additional LMS. However, the LMS systems do not share synchronization or communication channels, they are as such stand-alone systems, see fig. 4-28 "Moving a carrier from one LMS to an other LMS" on page 106. In such setup, to transfer a carrier from one LMS to another LMS (to support large tracks), the carrier movement must stop, then the carrier is removed from the LMS track, then the carrier is added to the other LMS track and then moved again. The drawbacks of this method is that the carrier movement must stop and that in the transition area the carrier must be completely covered by a coil of both LMS systems, which requires an extra coil (because coils cannot be shared between LMS systems).

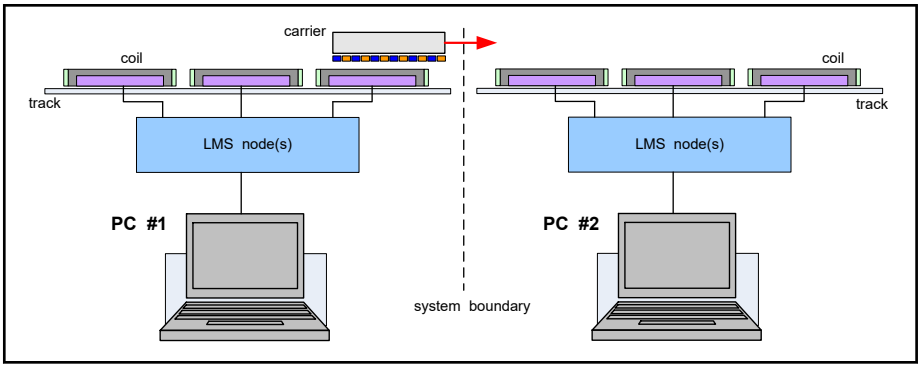


Fig. 4-28: Moving a carrier from one LMS to an other LMS

With the functions `CmRemoveMovingCarrier` and `CmAddMovingCarrier` you can remove a carrier from an LMS while the carrier is moving, and add a moving carrier to an (other) LMS. Further, you can cancel the addition of a moving carrier to an LMS with the function `CmCancelAddMovingCarrier`, as long as the carrier is not detected by the outer sensor of the track. You can use these functions also to combine a conveyor belt with an LMS. A carrier can be transfered from a moving conveyor belt to an LMS track, and a moving carrier can be transfered from an LMS track to a conveyor belt while the LMS track remains active.

⚠ CAUTION **Possible carrier out of control condition, causing possible machine damage.**

Make sure that the specified velocity for `CmRemoveMovingCarrier` and `CmAddMovingCarrier` is identical. This also applies if one of these 2 functions is used in combination with `CmStartVelocityMode`.

When a carrier is removed from the system and a CPA table is linked to that carrier, the link to the CPA table is also removed. When that carrier is (again) added to the system, the application must also re-establish the link to the CPA table.

Explanation

To move a carrier from LMS #1 (controlled by PC #1 in the figure) to LMS #2 (controlled by PC #2), LMS #1 must "shoot" the carrier in position mode, and LMS #2 must "catch" the carrier in position mode. Note that for LMS #2 it is not relevant whether the carrier is moved from an other LMS or from a conveyor belt.

With the function `CmRemoveMovingCarrier` you specify which carrier should leave the LMS, at which side of the track (`CM_TRACK_BEGIN` or `CM_TRACK_END`), and specify the acceleration, velocity and jerk of the carrier with the structure `CM_VELOCITY_PARS`. After the call of this function the carrier moves to the specified end of the track using a 3rd order setpoint profile, and continues to move until the carrier is no longer controlled by the last coil of the track. Of course, no error is generated when the carrier leaves the track. The event `CM_EV_CARRIER_MOVING_CARRIER_REMOVED` is generated when the carrier leaves the track. The carrier is removed from the carrier administration. The function `CmRemoveMovingCarrier` can be called while the carrier is at standstill or while the carrier is moving, the track state must be `CM_TRACK_ACTIVE`. If the value for jerk is `NYCE_INFINITE`, a 2nd order setpoint profile is used.

With the function CmAddMovingCarrier you can specify that a carrier will enter the LMS at the specified side of the track (CM_TRACK_BEGIN or CM_TRACK_END). The track state must be CM_TRACK_ACTIVE, and FeedOverride must not be 0. The 3rd order setpoint profile movement and end position are defined with the structure CM_MOVE_PARS. If the value for jerk is NYCE_INFINITE, a 2nd order setpoint profile is used. The end position must be such that the carrier is completely on the track. The carrier is added to the carrier administration when the carrier is detected.

The function CmAddMovingCarrier uses an endpoint. The event CM_EV_CARRIER_MOVING_CARRIER_ADDED is generated when the first sensor of the track detects the carrier and the carrier is added. The carrier state and carrier position data are valid after this event is generated. You probably want to move the carrier to an other position (with the function CmMoveToPosition) after the carrier is added. However, if you execute the CmMoveToPosition command after the event CM_EV_CARRIER_MOVING_CARRIER_ADDED, the error CM_ERR_ORPHAN_CARRIER is generated, because the carrier is not yet within the control range of the coil. You can use a time delay, or use the event CM_EV_CARRIER_CONTROLLER_MOVING_EN to prevent this error. The event CM_EV_CARRIER_CONTROLLER_MOVING_EN is generated when setpoints are generated for the coil, which implies that the carrier has entered the coil control range of that coil.

With the function CmCancelAddMovingCarrier it is possible to cancel the addition of a moving carrier to an LMS track. The cancellation is only possible as long as the carrier is not (yet) detected by the outer sensor of the track. The track can be in any state.

If the CmAddMovingCarrier command is cancelled with CmCancelAddMovingCarrier after the physical addition of the carrier is done, the error SAC_AX_ERR_LMS_LEFT_SENSOR_ACTIVE or SAC_AX_ERR_LMS_RIGHT_SENSOR_ACTIVE is generated after the carrier has entered the track, and the track state changes to CM_TRACK_FATAL_ERROR.

The function CmAddMovingCarrier is automatically cancelled at both sides of the track if the function CmHomeTrack, CmStartVelocityMode, CmDeactivateTrack, or CmResetError is called. The function CmAddMovingCarrier is automatically cancelled at that side where RTLC actions are executed that add or remove a coil.

General remarks

- A discontinuity in setpoints across the tracks occurs, because there is no synchronization. This will cause "velocity ripples".
- The specified velocity for the CmRemoveMovingCarrier and CmAddMovingCarrier must be identical.
- Only 1 CmAddMovingCarrier function per side of the track may be pending. Thus at maximum there can be 2 CmAddMovingCarrier functions pending (one at each side).
- If an error occurs during the carrier movement with CmAddMovingCarrier or CmRemoveMovingCarrier and the carrier is still in the control range of the coil, you can recover the error condition with the following steps.
  1. Call CmResetError for the track in the CM_TRACK_ERROR state.
  2. Call CmRemoveCarrier.

3. Call `CmEnableSensorDetection` to disable the sensors.
4. Physically move the carrier away.
5. Call `CmEnableSensorDetection` to enable the sensors.
6. Call `CmActivateTrack`.

See also chapter 5.6.3 "Specific recovery" on page 174 for `CmAddMovingCarrier`.

- Safety across LMS boundaries is the responsibility of the user. There is no synchronization between the involved LMS systems.
- The usage of RTLC functionality in combination with removal and addition of moving carriers is the responsibility of the user. Care must be taken that the correct coils are present in the track when a moving carrier is removed from or added to a track.

4.7.29 Moving a carrier across tracks

Function

If an LMS consists of multiple logical tracks across which a carrier must move, the application may have a short origin track where the carrier is positioned. The desired maximum velocity of the carrier cannot be reached on that short track. The functions `CmMoveToTrack` and `CmRemoveMovingCarrier` / `CmAddMovingCarrier` require a constant velocity, thus limiting the machine cycle time, because the movement on the destination track starts with this limited velocity and cannot be accelerated. The advantage of the mentioned functions is that no track layout information is needed.

With the function `CmMoveAcrossTrackBounds` you can move a carrier from one track to an other track at a non-constant velocity. Additionally, the origin and destination track may have one or more other logical tracks between them. However, track layout information must be specified for the function `CmMoveAcrossTrackBounds`. Normally, tracks are independent entities in CM and no information is needed about their location in the system. A physical track can be built from several logical tracks, and the machine layout may vary in time. If you want to use the function `CmMoveAcrossTrackBounds`, the application must have an administration of the logical tracks and know the distance between successive logical tracks on the physical track.

Explanation

The function `CmMoveAcrossTrackBounds` has 4 arguments, `trackId`, `carrierId`, `beginPosition`, and a pointer to a structure `pCmMovePars`. The `trackId` specifies the track, `carrierId` specifies the carrier, `beginPosition` specifies the initial position which has to be the setpoint position, and the structure specifies the movement parameters which includes the end position. The begin and end position may be a position that is outside the range of the track. If the position is outside the range of the track, the position is called a virtual position.

In the following figures, the logical tracks are shown in different colors and the 0 position of each track is indicated. Note that the 0 position of track2 is a virtual position, because it is outside the range of the track, see fig. 4-30 "3 logical tracks, movement in same (positive) direction" on page 109. The distance d_{01} and d_{12} is positive in the direction of the movement. The initial position of the carrier is x_i, the end position is x_f. As x_i is defined on the blue

track, it is relative to the 0 position of that track. Likewise, as x_f is defined on the red track, it is relative to the 0 position of that track.

2 logical tracks

In coordinates of the blue track, the carrier must move from position x_i to position $d_{01} + x_f$. Position $d_{01} + x_f$ is a virtual end position for the carrier on the blue track, because that position is outside the range of that track. In coordinates of the red track, the carrier must move from position $x_i - d_{01}$ to position x_f. Position $x_i - d_{01}$ is a virtual begin position for the carrier on the red track, because that position is outside the range of that track. The distance that the carrier must move is $x_f - x_i + d_{01}$, which is of course the same in both coordinate systems.

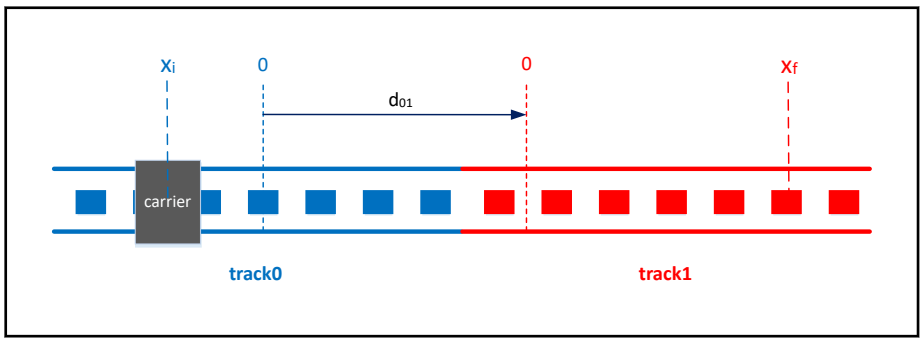


Fig. 4-29: 2 logical tracks, movement in same (positive) direction

3 logical tracks

In coordinates of the blue track, the carrier must move from position x_i to position $d_{01} + d_{12} + x_f$. Position $d_{01} + d_{12} + x_f$ is a virtual end position for the carrier on the blue track, because that position is outside the range of that track. In coordinates of the green track, the carrier must move from position $x_i - d_{01}$ to position $d_{12} + x_f$. Note that $x_i - d_{01}$ is a virtual begin position and $d_{12} + x_f$ is a virtual end position for the carrier on the green track, because those positions are outside the range of that track.

In coordinates of the red track, the carrier must move from position $x_i - d_{01} - d_{12}$ to position x_f. Position $x_i - d_{01} - d_{12}$ is a virtual begin position for the carrier on the red track, because that position is outside the range of that track. The distance that the carrier must move is $x_f - x_i + d_{01} + d_{12}$, which is of course the same in all 3 coordinate systems.

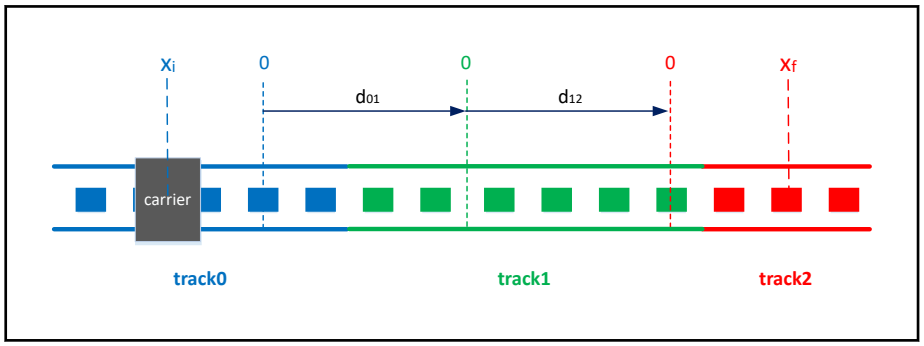


Fig. 4-30: 3 logical tracks, movement in same (positive) direction

different movement directions

It is possible that the positive direction of all involved tracks have different physical directions. An example of this situation is given in fig. 4-31 "2 logical tracks, movement in different directions" on page 110.

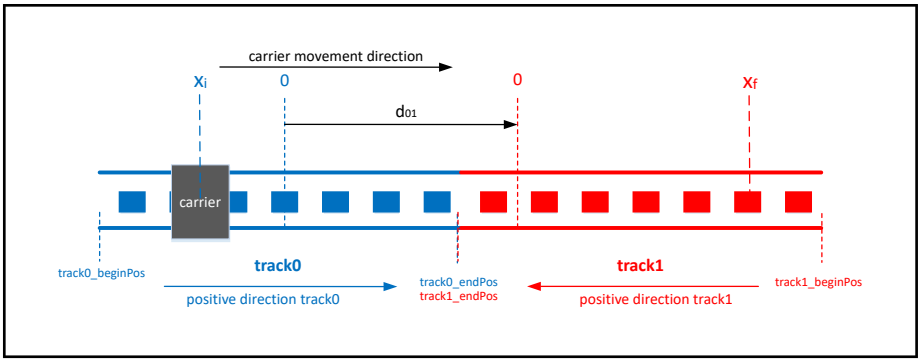


Fig. 4-31: 2 logical tracks, movement in different directions

In coordinates of the blue track, the carrier must move from position x_i to position d_{01} - x_f. Position d_{01} - x_f is a virtual end position for the carrier on the blue track, because that position is outside the range of that track. In coordinates of the red track, the carrier must move from position d_{01} - x_i to position x_f. Position d_{01} - x_i is a virtual begin position for the carrier on the red track, because that position is outside the range of that track. The distance that the carrier must move is d_{01} - x_f - x_i.

Constraints

The following constraints must be met before you can use the function `CmMoveAcrossTrackBounds`.

- All involved tracks must be in the state `CM_TRACK_ACTIVE`.
- The feed override of all involved tracks must be the same when the carrier moves over the tracks.
- The state of the carrier must be `CM_CARRIER_STOPPED`.
- The maximum velocity, acceleration and jerk of the move parameters must be the same for all involved tracks.
- The initial (virtual) position of the carrier is on one track, not between 2 tracks. Likewise, the (virtual) end position of the carrier is on one track, not between 2 tracks. Further, if the specified `beginPosition` argument of the function `CmMoveAcrossTrackBounds` is within the track range (between the begin and end position of the track), the carrier has to be present on that track, else the error `CM_ERR_CARRIER_NOT_ON_TRACK` is returned.
- No other carriers may block the movement when the carrier moves across several tracks.
- No RTLC is executed after the command `CmMoveAcrossTrackBounds` has been issued.

How to use

A call of the function `CmMoveAcrossTrackBounds` must be done for every track involved in the movement. The sequence order in which the function is called is important. The last call of `CmMoveAcrossTrackBounds` is for the track where the carrier starts the movement. The correct sequence is essential when the origin track and destination track are part of different Linear Motion Systems. Suppose a carrier is located on track10 in LMS-1, and has to move to track17 in LMS-2.

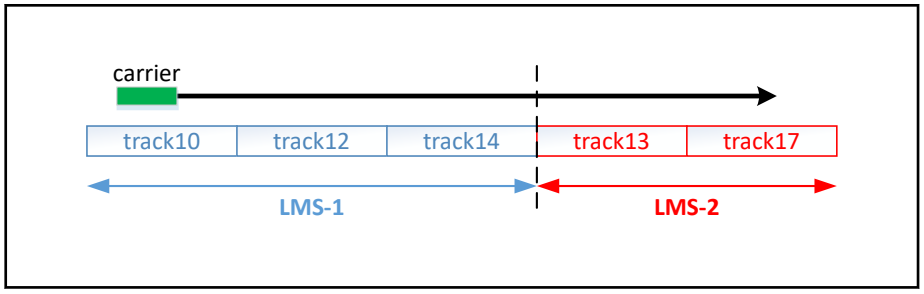


Fig. 4-32: Carrier movement across track bounds

The correct sequence of `CmMoveAcrossTrackBounds` commands to be issued is the following.

1. `CmMoveAcrossTrackBounds` for track12
2. `CmMoveAcrossTrackBounds` for track14
3. `CmMoveAcrossTrackBounds` for track13

   (first track of different LMS where the carrier will enter)
4. `CmMoveAcrossTrackBounds` for track17
5. `CmMoveAcrossTrackBounds` for track10

   start the movement of the carrier

The execution of the `CmMoveAcrossTrackBounds` commands in the specified sequence ensures

- that the track of a different LMS, where the carrier will enter the system, is always called first, making sure that all administration is correctly prepared to receive the carrier
- that the track where the carrier is currently located, always receives the command as last command, making it possible to reuse the same carrier ID when moving from track to track, because the receiving tracks are prepared for this carrier before it arrives.

The function executes the following checks before the movement starts.

- All constraints are met.
- If both the initial position and the end position are lower than the begin position of a track, or both are larger than the end position of a track, the error `CM_ERR_INVALID_POSITIONS` is generated.
- If the position of the carrier on the origin track is not the same as the specified `beginPosition` (the setpoint position), the error `CM_ERR_INITIAL_POSITION_MISMATCH` is generated.

If a `carrierId` is not present in the system, the function finds and reserves a free carrier index for the `carrierId`.

Events

The event `CM_EV_TRACK_MOVING_CARRIER_LE` is generated when the outer sensor of the last coil in the movement direction no longer detects the carrier. The event `CM_EV_TRACK_MOVING_CARRIER_EN` is generated when the outer sensor of the first coil in the movement direction detects the carrier. See chapter 4.7.27 "Move a carrier from track to track" on page 99 for more information. Further, the events related to the carrier (`CM_EV_CARRIER_STOPPED_EN`, `CM_EV_CARRIER_STOPPED_LE`, `CM_EV_CARRIER_MOVING_EN`, and `CM_EV_CARRIER_MOVING_LE`) are generated.

Performance If tracks are in different systems, or use a different `carrierId`, the setpoint data generated on the leaving track and the entering track may be discontinuous when the entered track starts to control the carrier. For this reason it is recommended to use the same `carrierId` if all involved tracks are in the same system.

Error recovery You can use the function `CmCancelMoveAcrossTrackBounds` to cancel a `CmMoveAcrossTrackBounds` command for the specified track and carrier if the movement has not yet started. If the carrier already started the movement, the error `CM_ERR_MOVEMENT_STARTED` is returned. If the function `CmCancelMoveAcrossTrackBounds` is called for a carrier that has not been issued the `CmMoveAcrossTrackBounds` command, the error `CM_ERR_NO_COMMAND_INITIATED` is returned. If the carrier movement stops, because an error occurred on one of the two tracks and the carrier is above two tracks, the recovery is similar to the recovery when the carrier stops between two tracks during the execution of `CmMoveToTrack`. If the carrier ID is the same in both tracks, the recovery is identical to the described method. If the carrier ID for both tracks is not the same, the recovery is similar but you must call `CmRemoveCarrier` twice, with the carrier ID of track1 and with the carrier ID of track2.

If the application sends the `CmMoveAcrossTrackBounds` command "simultaneously" to the origin and the destination track, it is possible that the destination track receives the command "too late". As a result, the destination track reports the error `SAC_AX_ERR_LMS_LEFT_SENSOR_ACTIVE` or `SAC_AX_ERR_LMS_RIGHT_SENSOR_ACTIVE`, which results in a fatal track error state. The carrier is at the end of its movement on the destination track. After the track error is reset and the track is homed again, the carrier ID used by `CmMoveAcrossTrackBounds` for the destination track cannot be used by the function `CmAssignCarrierIds`, because the carrier ID is already "in use". This situation can occur when the pending `CmMoveAcrossTrackBounds` is accepted after the track entered the fatal error state. The reset of the track does not cancel the pending `CmMoveAcrossTrackBounds`. The carrier ID can be released with the function `CmCancelMoveAcrossTrackBounds` for the destination track.

4.7.30 Moving a carrier with collision avoidance

Function

When you move carriers on a track, it is possible that movements may cross each other. This means that a carrier wants to move across some path of the track while another carrier is moving across that same path. If the movement of both carriers is executed it will result in a collision, and most likely the error `SAC_AX_ERR_POSITION_ERROR_EXCEEDED` is reported. The functionality "collision avoidance" has the following functions to move carriers on a track while avoiding a collision.

- `CmCollAvoidMoveToPosition`
- `CmCollAvoidStopCarrier`
- `CmCollAvoidAddMovingCarrier`
- `CmCollAvoidCancelAddMovingCarrier`
- `CmCollAvoidRemoveMovingCarrier`
- `CmCollAvoidMoveBetweenTracks`
- `CmCollAvoidAddCarrierWithProfile`
- `CmCollAvoidCancelAddCarrierWithProfile`
- `CmCollAvoidRemoveCarrierWithProfile`

- `CmCollAvoidDefineProcessArea`
- `CmCollAvoidDeleteProcessArea`
- `CmCollAvoidEnableProcessArea`
- `CmCollAvoidGetProcessAreaData`
- `CmCollAvoidSynchronize`
- `CmCollAvoidReadCarrierState`

The "collision avoidance" functionality ensures that when the initial path of a carrier (from current position to end position) crosses a path of another carrier, the initial path is divided into intermediate paths in such a way that no collision occurs with other carriers on the track. The movement of a carrier is temporarily stopped at a so-called "safe stop position" to avoid the collision.

If two carriers on a track both get a movement command in the same direction, it is not likely that they will collide. However, if two carriers are on a track and the specified movement has opposite direction such that they will move toward each other, depending on the end position of both carriers a collision may occur. Suppose a carier is at the begin of the track, and another carrier is at the end of the track, and both carriers get the move with collision avoidance command to move to the middle of the track. In this case, the carrier that receives the command first will start the movement. However, the other carrier cannot execute its move with collision avoidance command as this would cause a collision at the end of the movement. Instead of a movement to some safe stop position (which might cause issues later on, such as a dead lock condition), the command to move to a position with collision avoidance returns the error `CM_ERR_CARRIER_POSITION_ERROR`.

- The collision avoidance functionality is not supported for modulo tracks.
- The collision avoidance functionality does not support multi-client connections. If a second application invokes a collision avoidance function, `CM_ERR_SYSTEM_STATE_ERROR` is returned.
- It is strongly advised not to use a mix of "normal" CM and "collision avoidance" CM functions.

CM functions "normally" execute synchronously with the application. This means that when the function returns to the application, the movement of the requested command has been started. Possible errors are reported during the execution of the called function. However, the CM functions with collision avoidance functionality return immediately to the application. The execution of the requested movement still has to start, because of the determination whether the path is free for the entire movement or safe stop positions are needed. Only the functions `CmCollAvoidAddMovingCarrier` and `CmCollAvoidAddCarrierWithProfile` start their movement during the call of the function. A separate thread is started to control the completion of the requested movement. Therefore, asynchronous errors may be reported.

RTLC in combination with collision avoidance functionality is not supported. This does not imply that you cannot use RTLC and collision avoidance in the same application. It does, however, apply that it is not allowed to do a movement with collision avoidance to a section when this section is at the beginning or end of the track, and the track is shortened with RTLC during the movement in such way that the endpoint is now outside track limits. How to handle this is the responsibility of the application. For example, when a collision avoidance function is used to buffer in front of an elevator (with

RTLC functionality), the application must execute movements with collision avoidance to a position just before the elevator.

When movements with collision avoidance are used, the track parameter `safeDistance` is required. This parameter has to be specified in the XML configuration file. The default value is zero.

When the process area functionality is used, the parameters `minPosition` and `maxPosition`, which define a process area, have to be specified.

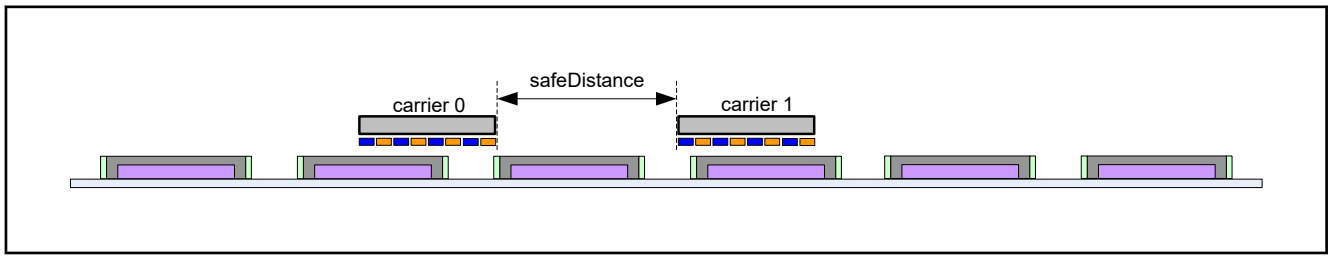
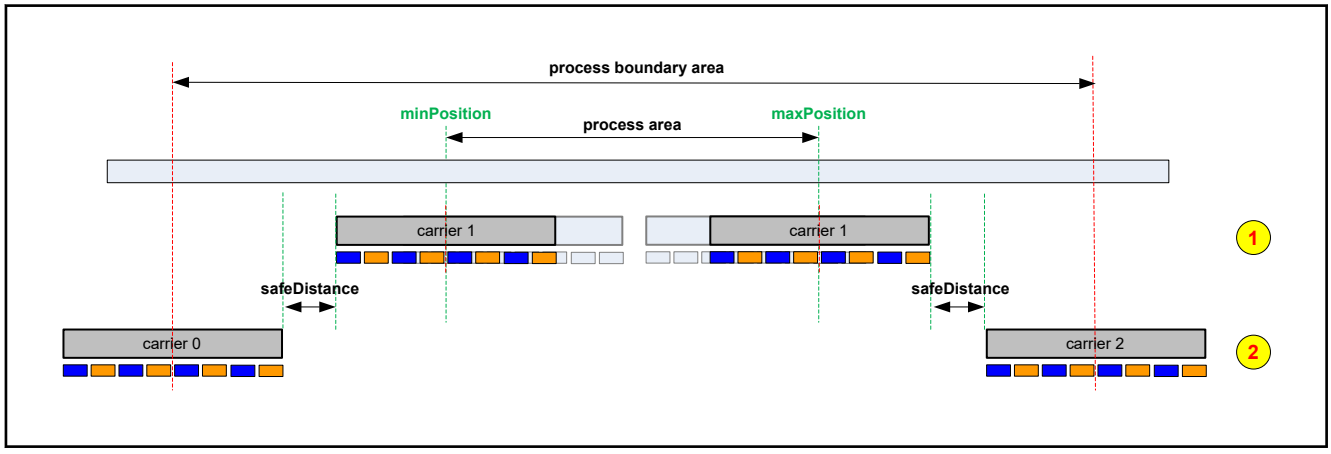


Fig. 4-33: Collision avoidance - safeDistance

safeDistance When a safe position is specified, it is checked that carriers are positioned in such a way that the distance between the carriers is never smaller than the defined `safeDistance`, see fig. 4-33 "Collision avoidance - safeDistance" on page 114. This means that when carrier 0 is moved, the carrier stops at or will not move to a position closer to carrier 1 than the defined `safeDistance`. Note that it is possible that carrier 0 stops at a position further away than `safeDistance` from carrier 1, because carrier 0 is not allowed to enter the control range of the coils covered by carrier 1.

process area A process area is an area on a track defined by a minimum and maximum position. Further, a process boundary area is defined by (process area minimum position - carrier length - safeDistance) and (process area maximum position + carrier length + safeDistance), see fig. 4-34 "Process area and process boundary area" on page 114. A process area can be disabled or enabled. A carrier in an enabled process area can move freely within the process area.



① carrier 1 can move freely from minPosition to maxPosition
② other carriers must be at a safe distance from the minimum and maximum position of carrier 1

Fig. 4-34: Process area and process boundary area

A process area that is defined and enabled is always involved in carrier movements with collision avoidance. If a carrier is moving inside an enabled process area, the safe positions of carriers that are moving toward this process area are taken into account. For example, when a carrier is crossing a process area with `CmCollAvoidMoveToPosition`, the begin and end position for this movement are outside the process area. If this carrier is moving inside this process area, the position of other carriers that are moving toward this process area are influenced. This is shown in fig. 4-35 "Collision avoidance - minimum and maximum positions" on page 115. Carrier 1 has an end position, where it is currently stopped, which is inside a defined and enabled process area. Carrier 1 can move freely within the process area defined by `minPosition` and `maxPosition` (as absolute positions in [pu]) without causing any collisions with other carriers on the track. As a result, carrier 0 will not keep the distance `safeDistance` to the actual or end position of carrier 1, but to the `minPosition` of the defined process area + `safeDistance`. This way, carrier 1 can move freely in negative direction up to `minPosition` without a violation of the `safeDistance` of carrier 0. Note that `minPosition` must be smaller than `maxPosition`.

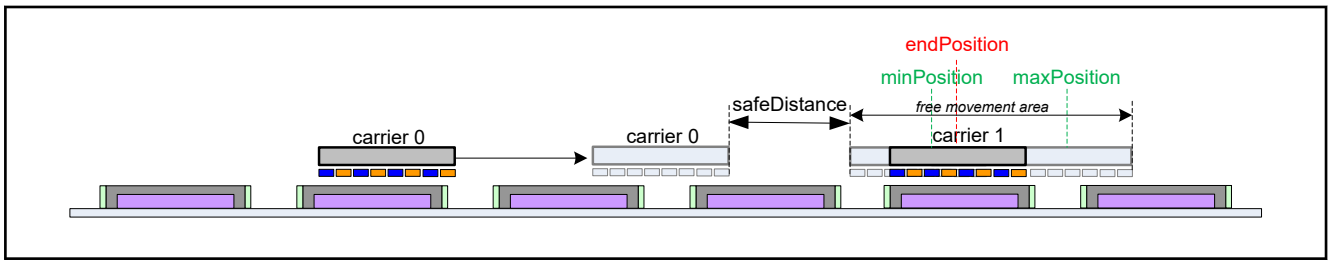


Fig. 4-35: Collision avoidance - minimum and maximum positions

A track can have up to `CM_COLL_AVOID_MAX_NR_OF_PROCESS_AREAS` process areas defined, which is also the maximum number of process areas that can be defined in a system. A process area can be statically defined in the XML configuration file, or dynamically with a CM function in the application. The static definition of a process area can be realized with the NYCeLmsCommissioning tool. This static definition is stored in the LMS configuration file. The dynamic definition of a process area can be realized with the function `CmCollAvoidDefineProcessArea`. A dynamically defined process area is not stored in the LMS configuration file. Every defined process area has a unique identifier in the range of 0 .. 32767. This identifier must be system-wide unique. The process area identifier is `CM_NO_ID` if the process area is not defined.

Besides the process area identifier, a process area definition (`minPosition` and `maxPosition`) and an implicitly defined process boundary area, a process area also has a defined `state`. The state of a process area can be the following, see fig. 4-36 "Process area state transitions" on page 116.

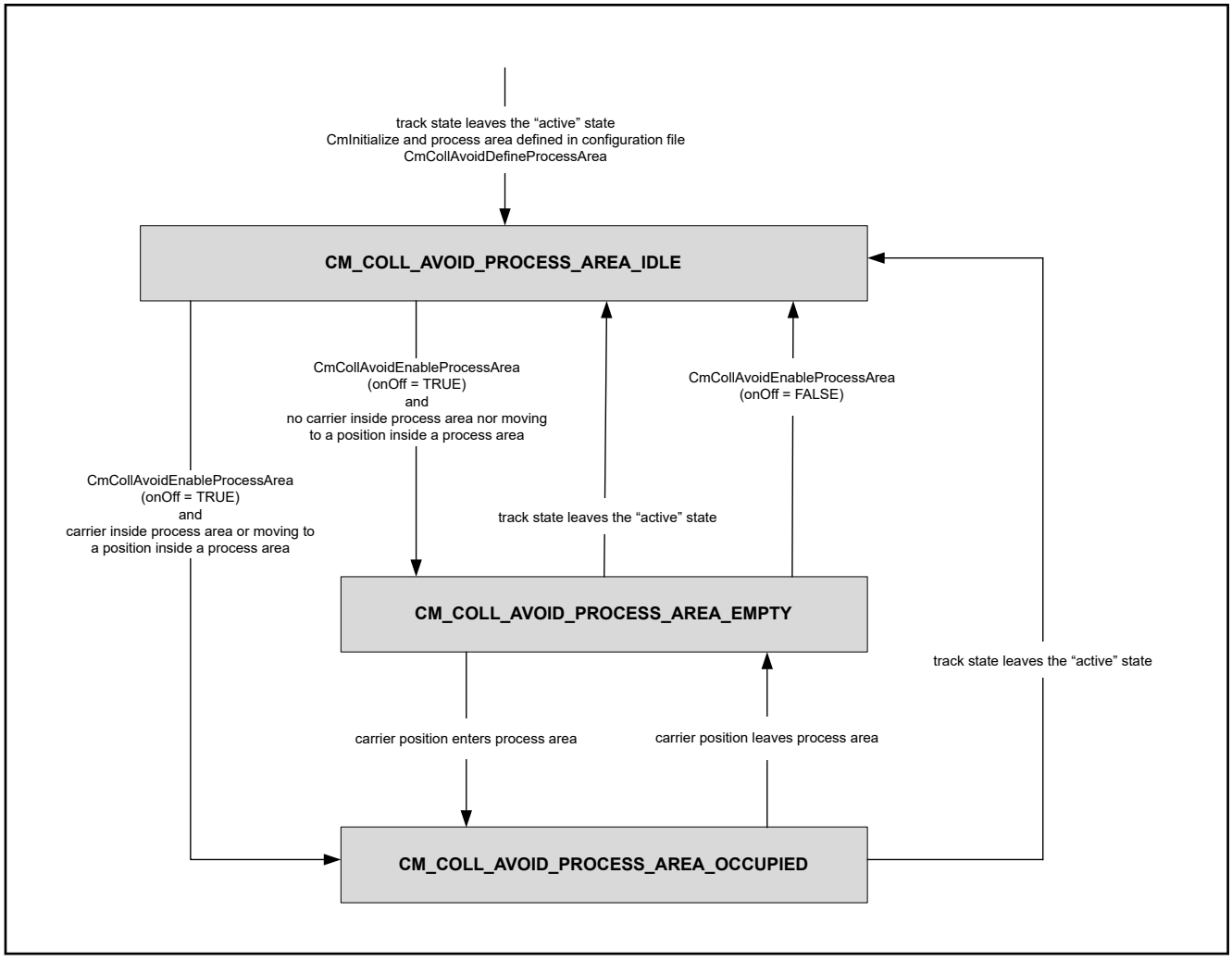


Fig. 4-36: Process area state transitions

- `CM_COLL_AVOID_PROCESS_AREA_IDLE`

  process area is disabled, no checks for this process area are executed. Transition to this process area state occurs when the track leaves the "active" state, or when a process area is defined in the XML configuration file and `CmInitialize` is called, or after the process area is defined with the function `CmCollAvoidDefineProcessArea`.

- `CM_COLL_AVOID_PROCESS_AREA_EMPTY`

  process area is enabled, no carriers are inside the process boundary area. This state is only possible when the track state is `CM_TRACK_ACTIVE`.

- `CM_COLL_AVOID_PROCESS_AREA_OCCUPIED`

  process area is enabled, a carrier is present inside the process boundary area. This state is only possible when the track state is `CM_TRACK_ACTIVE`.

The application must explicitly enable every process area with the function `CmCollAvoidEnableProcessArea`. When the track state leaves the state `CM_TRACK_ACTIVE`, the state of all process areas changes to `CM_COLL_AVOID_PROCESS_AREA_IDLE`.

A distance of at least one carrier length + `safeDistance` must be present between the minimum and maximum parameters of 2 process areas next to each other. The `state` of a process area changes to `CM_COLL_AVOID_PROCESS_AREA_OCCUPIED` when the carrier position of a carrier moves into the process boundary area. Likewise, the `state` of a process area changes to `CM_COLL_AVOID_PROCESS_AREA_EMPTY` when the carrier position of the carrier has moved out of the process boundary area. Note that a carrier that starts moving towards a position within the process area + `safeDistance` will not "reserve" the process area. The state of a process area only changes between "occupied" and "empty" when the actual setpoint position of the carrier moves across the above described boundaries.

track to track movement with collision avoidance

Two groups of functions with collision avoidance are available for the movement of a carrier from one track to an adjacent track.

1. Moving a carrier to an other track in the **same** LMS

   The function `CmCollAvoidMoveBetweenTracks` is used to move a carrier from one track to an other (adjacent) track in the **same** LMS. The specified profile parameters `velocity`, `acceleration`, and `jerk` define the profile for the movement. Moving a carrier from one track to an other track in a **different** LMS is not supported.

2. Moving a carrier to an other track in the **same or a different** LMS

   The functions `CmCollAvoidAddMovingCarrier`, `CmCollAvoidRemoveMovingCarrier`, `CmCollAvoidAddCarrierWithProfile`, `CmCollAvoidCancelAddCarrierWithProfile`, and `CmCollAvoidRemoveCarrierWithProfile` are used to move a carrier from one track to an other (adjacent) track in the **same or a different** LMS. Both LMSs have no knowledge of the other LMS and its tracks. The specified profile parameters `velocity`, `acceleration`, and `jerk` define the profile for the movement.

Explanation

CmCollAvoidMoveToPosition

With the function `CmCollAvoidMoveToPosition` you can start the movement of a carrier to a specified end position with collision avoidance. When the command returns successfully, the command has been accepted. If the path to the specified end position is clear, the carrier immediately moves to the specified end position. However, if the path is not clear, a safe stop position is calculated and the carrier stops at this intermediate position. It is possible that the carrier does not start a movement at all to avoid a collision, thus the start position is the intermediate stop position. At this position the carrier state changes to `CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`. When the path is cleared, the carrier starts to move (again). When the carrier stops at the end position, the carrier state changes to `CM_COLL_AVOID_CARRIER_STOPPED`.

The function `CmCollAvoidMoveToPosition` has 2 arguments. The first argument `carrierId` specifies the carrier. The second argument is the structure `pCmMovePars` which specifies the end position `endPosition` in [pu] of the movement and the movement parameters for maximum velocity, acceleration and jerk, respectively `maxVelocity` in [pu/s], `maxAcceleration` in [pu/s^2], and `maxJerk` in [pu/s^3].

Preconditions for the function `CmCollAvoidMoveToPosition` are that the track state must be `CM_TRACK_ACTIVE` and the feed override must be 1. The carrier state can be any carrier state, except `CM_COLL_AVOID_CARRIER_POWERUP`, `CM_COLL_AVOID_CARRIER_SUSPENDED`, or `CM_COLL_AVOID_CARRIER_JOGGING`.

The state of the carrier can have the following transitions.

`CM_COLL_AVOID_CARRIER_STOPPED`	→	`CM_COLL_AVOID_CARRIER_MOVING`
`CM_COLL_AVOID_CARRIER_MOVING`	→	`CM_COLL_AVOID_CARRIER_STOPPED` if the carrier stops at the end position
`CM_COLL_AVOID_CARRIER_MOVING`	→	`CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS` if the carrier stops at an intermediate position to avoid a collision
`CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`	→	`CM_COLL_AVOID_CARRIER_MOVING` resume the movement when it is possible without a collision
`CM_COLL_AVOID_CARRIER_STOPPED`	→	`CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS` if the carrier is at start position and a `CmCollAvoidMoveToPosition` command would cause a collision

CmCollAvoidStopCarrier

With the function `CmCollAvoidStopCarrier` you can initiate a stop profile for a carrier with collision avoidance. When the command returns successfully, the carrier has initiated the stop profile, or is already at standstill.

The function `CmCollAvoidStopCarrier` has 2 arguments. The first argument `carrierId` specifies the carrier. The second argument is the structure `pCmStopPars` which specifies the stop profile parameters for maximum acceleration and jerk, respectively `stopAcceleration` in [pu/s^2], and `stopJerk` in [pu/s^3].

Preconditions for the function `CmCollAvoidStopCarrier` are that the track state must be `CM_TRACK_ACTIVE`, or `CM_TRACK_ERROR`, or `CM_TRACK_FATAL_ERROR`. The carrier state can be any carrier state, except `CM_COLL_AVOID_CARRIER_POWERUP`, `CM_COLL_AVOID_CARRIER_SUSPENDED`, or `CM_COLL_AVOID_CARRIER_JOGGING`.

Only if the track state is `CM_TRACK_ACTIVE`, the state of the carrier can have the following transitions.

`CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`	→	`CM_COLL_AVOID_CARRIER_STOPPED`
`CM_COLL_AVOID_CARRIER_MOVING`	→	`CM_COLL_AVOID_CARRIER_STOPPED` after completion of the stop profile

CmCollAvoidAddMovingCarrier

With the function `CmCollAvoidAddMovingCarrier` you can add a moving carrier with collision avoidance to the LMS while the track is active. Ideally, the carrier is moving at a constant velocity. This same velocity must be specified in the movement parameters. The end position must be chosen such that the carrier is completely on the track. As soon as the carrier is detected, the carrier is added to the carrier administration. The carrier always

moves at least to the specified `firstSafePosition`. It is the responsibility of the application to make sure that velocity and deceleration is such that the carrier can stop at this `firstSafePosition`. If `firstSafePosition` is a position inside an occupied process area or a position beyond an occupied process area, the error `CM_ERR_TOO_MANY_CARRIERS_IN_PROCESS_AREA` is returned.

The `firstSafePosition` is a position where the carrier can stop given the specified deceleration. At the `firstSafePosition` the carrier state is `CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`. If the path to the end position is clear, the carrier will immediately move to the end position. If the path is not clear, a safe position is calculated and the carrier stops at this intermediate position. At this position the carrier state is `CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`. When the path is cleared, the carrier starts to move again. When the carrier stops at the end position the state changes to `CM_COLL_AVOID_CARRIER_STOPPED`.

The function `CmCollAvoidAddMovingCarrier` has 5 arguments. The first argument `trackId` specifies the track. The second argument `carrierId` specifies the carrier. The third argument `trackPositionId` specifies the beginning or end of the track. The fourth argument `firstSafePosition` specifies the position on the track where the carrier can move to without a collision. The last argument is the structure `pCmMovePars` which specifies the end position `endPosition` in [pu] of the movement and the movement parameters for maximum velocity, acceleration and jerk, respectively `maxVelocity` in [pu/s], `maxAcceleration` in [pu/s^2], and `maxJerk` in [pu/s^3].

Preconditions for the function `CmCollAvoidAddMovingCarrier` are that the track state must be `CM_TRACK_ACTIVE` and the feed override must be 1.

The state of the carrier can have the following transitions.

```
CM_COLL_AVOID_CARRIER_POWERUP          →  CM_COLL_AVOID_CARRIER_MOVING
                                          if the carrier is detected by the outer sensor

CM_COLL_AVOID_CARRIER_MOVING           →  CM_COLL_AVOID_CARRIER_STOPPED
                                          if the carrier stops at the end position

CM_COLL_AVOID_CARRIER_MOVING           →  CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS
                                          if the carrier stops at an intermediate position
                                          to avoid a collision

CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS →  CM_COLL_AVOID_CARRIER_MOVING
```

CmCollAvoidCancelAddMoving-Carrier

With the function `CmCollAvoidCancelAddMovingCarrier` you can cancel the addition of a moving carrier to the system with collision avoidance as long as the carrier is not detected by the outer sensor of the track.

The function `CmCollAvoidCancelAddMovingCarrier` has 3 arguments. The first argument `trackId` specifies the track. The second argument `carrierId` specifies the carrier. The last argument `trackPositionId` specifies the beginning or end of track.

CmCollAvoidRemoveMoving-Carrier

With the function `CmCollAvoidRemoveMovingCarrier` you can remove a moving carrier with collision avoidance from the LMS while the carrier is moving in position mode. The function specifies a velocity with which a carrier must move beyond the end of the track. When the carrier is no longer detected by the outer sensor of the last coil of the track, the carrier is removed from the carrier administration. If the path to the outer sensor of the

last coil of the track is clear, the carrier is removed from the track. If the path is not clear, a safe position is calculated and the carrier stops at this intermediate position. At this position the carrier state is CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS. When the path is cleared, the carrier starts to move again. If the last safe stop position is too close to the end of the track, given the specified acceleration, it might not be possible to reach the desired end velocity.

The function CmCollAvoidRemoveMovingCarrier has 3 arguments. The first argument carrierId specifies the carrier. The second argument trackPositionId specifies the beginning or end of the track. The third argument is the structure pCmVelocityPars which specifies the velocity parameters, the desired velocity velocity in [pu/s], maximum acceleration maxAcceleration in [pu/s^2], and maximum jerk maxJerk in [pu/s^3].

Preconditions for the function CmCollAvoidRemoveMovingCarrier are that the track state must be CM_TRACK_ACTIVE, and feed override must be 1. The carrier state can be any carrier state, except CM_COLL_AVOID_CARRIER_POWERUP, CM_COLL_AVOID_CARRIER_SUSPENDED, or CM_COLL_AVOID_CARRIER_JOGGING.

The state of the carrier can have the following transitions.

CM_COLL_AVOID_CARRIER_STOPPED	→	CM_COLL_AVOID_CARRIER_MOVING
CM_COLL_AVOID_CARRIER_STOPPED	→	CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS if the carrier is at the start position and a start of the movement will cause a collision
CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS	→	CM_COLL_AVOID_CARRIER_MOVING
CM_COLL_AVOID_CARRIER_MOVING	→	CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS if the carrier stops at an intermediate position to avoid a collision
CM_COLL_AVOID_CARRIER_MOVING	→	CM_COLL_AVOID_CARRIER_POWERUP if the carrier is no longer detected by the outer sensor

CmCollAvoidMoveBetweenTracks

With the function CmCollAvoidMoveBetweenTracks you can move a carrier from one track to an other (adjacent) track. Both tracks are in the **same** LMS. Depending on the specified profile, the carrier moves over the boundaries of the tracks with an accelerating, decelerating or constant velocity. The function provides collision avoidance during the entire movement of the carrier from the begin position on the origin track to the end position on the destination track. The carrier only starts the movement when the path from the begin position to the leaving side of the origin track and the path from the entering side of the destination track to the first safe position are clear. If initially any path is not clear, the begin position becomes a safe position. If both paths are clear, the carrier starts moving to the end position and stops at the end position. If the path on the destination track to the end position is not clear, the carrier moves to the first safe position or to a calculated safe position beyond the first safe position. From this moment on, the collision avoidance functionality executes the following specific movements.

- When during the movement to a safe position no further part of the path to the end position becomes clear, the carrier comes to a standstill at the calculated safe position.

- When during the movement to a safe position a further part of the path to the end position becomes clear, the movement continues to the next calculated safe position.
- When the carrier is at standstill at a safe position and a further part of the path to the end position becomes clear, the carrier starts automatically moving to the next calculated safe position.
- Eventually, the path to the end position becomes clear, and the carrier starts automatically moving to the end position and comes to a standstill.

The state of the carrier changes between `CM_COLL_AVOID_CARRIER_STOPPED`, `CM_COLL_AVOID_CARRIER_MOVING`, and `CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`. If the carrier is at standstill at a safe position, the carrier state is `CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`. If the carrier is at standstill at the end position, the carrier state is `CM_COLL_AVOID_CARRIER_STOPPED`.

With the initial carrier movement, from the begin position on the origin track to an end position on the destination track, both tracks execute the same movement with begin and end position in the track's own coordinate system. The physical distance between the tracks must be known to calculate the end position of the movement for the origin track and the begin position of the movement for the destination track. The physical distance `trackDistance` is the distance between the outer sensors of both tracks. This distance can be determined with the NYCeCommissioning tool.

The function `CmCollAvoidMoveBetweenTracks` has 7 arguments.

Argument	Description
`carrierId`	Identification of the carrier.
`trackOriginPositionId`	Identification of the track side, side where the carrier leaves the origin track.
`trackDestinationId`	Identification of the destination track.
`trackDestinationPositionId`	Identification of the track side, side where the carrier enters the destination track.
`trackDestinationFirstSafePosition`	Position on the destination track where the carrier can safely move to. If this is a position inside an occupied process area or a position beyond an occupied process area, the error `CM_ERR_TOO_MANY_CARRIERS_IN_PROCESS_AREA` is returned.
`trackDistance`	Physical distance between the outer sensors of the origin and destination track in [pu].
`pCmMovePars`	Structure of movement parameters `endPosition` in [pu] of the movement and the movement parameters for maximum velocity, acceleration and jerk, respectively `maxVelocity` in [pu/s], `maxAcceleration` in [pu/s^2], and `maxJerk` in [pu/s^3]. The end position is defined on the destination track.

Tab. 4-5: Arguments of CmCollAvoidMoveBetweenTracks

For each movement the specified constraints on maximum velocity, maximum acceleration, and maximum jerk are taken into account.

Preconditions for the function `CmCollAvoidMoveBetweenTracks` are that the state of both tracks must be `CM_TRACK_ACTIVE` and the feed override of both tracks must be 1. The carrier state must be `CM_COLL_AVOID_CARRIER_STOPPED`.

The state of the carrier can have the following transitions.

```
CM_COLL_AVOID_CARRIER_STOPPED          →  CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS
                                          if both paths for initial movement are not clear

CM_COLL_AVOID_CARRIER_STOPPED          →  CM_COLL_AVOID_CARRIER_MOVING
                                          if both paths for initial movement are clear

CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS →  CM_COLL_AVOID_CARRIER_MOVING
                                          resume the movement when it is possible without
                                          a collision

CM_COLL_AVOID_CARRIER_MOVING           →  CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS
                                          if the carrier stops at an intermediate position
                                          to avoid a collision

CM_COLL_AVOID_CARRIER_MOVING           →  CM_COLL_AVOID_CARRIER_STOPPED
                                          if the carrier is at the end position
```

CmCollAvoidAddCarrierWithProfile

With the function `CmCollAvoidAddCarrierWithProfile` you can add a moving carrier to a track. The carrier moves from a virtual begin position to an end position on the receiving track. Depending on the specified profile, the carrier enters the receiving track with an accelerating, decelerating or constant velocity. If `firstSafePosition` is a position inside an occupied process area or a position beyond an occupied process area, the error `CM_ERR_TOO_MANY_CARRIERS_IN_PROCESS_AREA` is returned. When the carrier enters the track, it starts moving to the first safe position. When the carrier moves beyond the handover position and the path to the end position is clear, the carrier automatically moves to the end position and comes to a standstill. The handover position `handoverPosition` is a position on the track where the carrier is only controlled by the receiving track. If the path to the end position is not clear, the carrier continues to move to the first safe position or a calculated safe position beyond the first safe position. From this moment on the collision avoidance functionality executes the movements as described in the explanation of `CmCollAvoidMoveBetweenTracks`.

The function `CmCollAvoidAddCarrierWithProfile` has 6 arguments.

Argument	Description
`trackId`	Identification of the track.
`carrierId`	Identification of the carrier.
`beginPosition`	Virtual begin position.
`handoverPosition`	Position on the track where the carrier is completely controlled by the receiving track.
`firstSafePosition`	Position on the receiving track where the carrier can safely move to.
`pCmMovePars`	Structure of movement parameters `endPosition` in [pu] of the movement and the movement parameters for maximum velocity, acceleration and jerk, respectively `maxVelocity` in [pu/s], `maxAcceleration` in [pu/s^2], and `maxJerk` in [pu/s^3]. The end position is defined on the receiving track.

Tab. 4-6: Arguments of CmCollAvoidAddCarrierWithProfile

For each movement the specified constraints on maximum velocity, maximum acceleration, and maximum jerk are taken into account.

Preconditions for the function `CmCollAvoidAddCarrierWithProfile` are that the state of the receiving track must be `CM_TRACK_ACTIVE` and the feed override of that track must be 1. The initial state of the carrier is `CM_COLL_AVOID_CARRIER_POWERUP`. When the carrier enters the track, the carrier state changes to `CM_COLL_AVOID_CARRIER_MOVING`.

The state of the carrier can have the following transitions.

`CM_COLL_AVOID_CARRIER_POWERUP`	→	`CM_COLL_AVOID_CARRIER_MOVING` if the carrier is detected by the outer sensor of the receiving track
`CM_COLL_AVOID_CARRIER_MOVING`	→	`CM_COLL_AVOID_CARRIER_STOPPED` if the carrier stops at the end position
`CM_COLL_AVOID_CARRIER_MOVING`	→	`CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS` if the carrier stops at an intermediate position to avoid a collision
`CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS`	→	`CM_COLL_AVOID_CARRIER_MOVING` resume the movement when it is possible without a collision

CmCollAvoidCancelAddCarrierWithProfile

With the function `CmCollAvoidCancelAddCarrierWithProfile` you can cancel the adding of a moving carrier to a track if the carrier has not yet entered the track. The function has 2 arguments. The first argument `trackId` specifies the receiving track. The second argument `carrierId` specifies the carrier.

CmCollAvoidRemoveCarrierWithProfile

With the function `CmCollAvoidRemoveCarrierWithProfile` you can remove a moving carrier from a track. The carrier moves from a begin position on the leaving track to a virtual end position. The function provides collision avoidance during the entire path from the begin position to the leaving side of the track. The carrier only starts moving when the path from the begin position of the carrier to the leaving side of the track is clear. If the path is not clear, the begin position becomes a safe position. From this moment on the collision avoidance functionality executes the movements as described in the explanation of `CmCollAvoidMoveBetweenTracks`.

The function `CmCollAvoidRemoveCarrierWithProfile` has 2 arguments. The first argument `carrierId` specifies the carrier. The second argument `pCmMovePars` specifies the movement parameters `endPosition` in [pu] of the movement and the movement parameters for maximum velocity, acceleration and jerk, respectively `maxVelocity` in [pu/s], `maxAcceleration` in [pu/s^2], and `maxJerk` in [pu/s^3]. The end position is the virtual end position.

For each movement the specified constraints on maximum velocity, maximum acceleration, and maximum jerk are taken into account.

Preconditions for the function `CmCollAvoidRemoveCarrierWithProfile` are that the state of the track must be `CM_TRACK_ACTIVE` and the feed override of that track must be 1. The state of the carrier must be `CM_COLL_AVOID_CARRIER_STOPPED`.

The state of the carrier can have the following transitions.

```
CM_COLL_AVOID_CARRIER_STOPPED          →  CM_COLL_AVOID_CARRIER_MOVING
                                          if the path is clear

CM_COLL_AVOID_CARRIER_STOPPED          →  CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS
                                          if the path is not clear

CM_COLL_AVOID_CARRIER_STOPPED_SAFE_POS →  CM_COLL_AVOID_CARRIER_MOVING
                                          resume the movement when it is possible
                                          without a collision

CM_COLL_AVOID_CARRIER_MOVING           →  CM_COLL_AVOID_CARRIER_POWERUP
                                          when the carrier is no longer detected
                                          by the outer sensor
```

CmCollAvoidDefineProcessArea

With the function `CmCollAvoidDefineProcessArea` the application can define a process area with collision avoidance on a track, or redefine an existing process area. The distance between the outer bounds of 2 process areas next to each other must be at minimum one carrier length plus `safeDistance`. Further, the minimum and maximum position of the process area must be within the track boundaries. The function `CmCollAvoidDefineProcessArea` has 3 arguments. The first argument, `processAreaId`, specifies the process area identifier. The second argument specifies the track identifier. The third argument specifies the minimum position `minPosition` and maximum position `maxPosition` of the process area.

A process area can be defined in any track state. If a process area is redefined, the state of that process area must be `CM_COLL_AVOID_PROCESS_AREA_IDLE`, else the error `CM_ERR_PROCESS_AREA_STATE_ERROR` is returned.

CmCollAvoidDeleteProcessArea

With the function `CmCollAvoidDeleteProcessArea` the application can delete a process area with collision avoidance on a track. The function has one argument, `processAreaId`, which specifies the process area identifier.

A process area can be deleted in any track state. If a process area is deleted, the state of that process area must be `CM_COLL_AVOID_PROCESS_AREA_IDLE`, else the error `CM_ERR_PROCESS_AREA_STATE_ERROR` is returned.

CmCollAvoidEnableProcessArea

With the function `CmCollAvoidEnableProcessArea` the application can enable or disable a process area with collision avoidance on a track. The function has 2 arguments. The first argument, `processAreaId`, specifies the process area identifier. The second argument specifies the action `onOff`. To enable the specified process area, `onOff` has to be `TRUE`. To disable the specified process area, `onOff` has to be `FALSE`.

A process area can be disabled in any process area state. If a process area is to be enabled, the track state must be `CM_TRACK_ACTIVE`, else the error `CM_ERR_TRACK_STATE_ERROR` is returned.

A process area can be enabled in the following conditions.

- no carrier is positioned inside the process boundary area
- one carrier is positioned inside the process boundary area

In these situations the function `CmCollAvoidEnableProcessArea` returns `NYCE_OK`.

A process area can also be enabled in the following condition.

- one carrier is positioned inside the process area and one or more carriers are positioned in the area greater than the process area, but smaller than the process boundary area

In this situation the function `CmCollAvoidEnableProcessArea` returns `CM_WRN_TOO_MANY_CARRIERS_IN_PROCESS_AREA`. When the warning is returned, the carrier inside the process area can move freely in that process area, however, the safe distance for a carrier outside the process area but inside the process boundary area is not always respected. The state of the process area changes from `CM_COLL_AVOID_PROCESS_AREA_OCCUPIED` to `CM_COLL_AVOID_PROCESS_AREA_EMPTY` when all carriers have moved outside the process boundary area.

In other situations, where more than one carrier is positioned in the process boundary area, or moving to a position inside the process boundary area, the process area cannot be enabled, and the function `CmCollAvoidEnableProcessArea` returns `CM_ERR_TOO_MANY_CARRIERS_IN_PROCESS_AREA`.

The application can call `CmCollAvoidEnableProcessArea` to enable or disable a process area which is already enabled or disabled, respectively.

If the process area is defined, and the function `CmCollAvoidEnableProcessArea` is called for that process area, the state of the process area changes to

- `CM_COLL_AVOID_PROCESS_AREA_EMPTY` when no carriers are positioned inside the process boundary area
- `CM_COLL_AVOID_PROCESS_AREA_OCCUPIED` when only one carrier is positioned inside the process area and none, one or more carriers are positioned inside the process boundary area.

The state of the process area can have the following transitions.

`CM_COLL_AVOID_PROCESS_AREA_IDLE`	→	`CM_COLL_AVOID_PROCESS_AREA_EMPTY` if the process area is enabled and no carrier is present in the process boundary area
`CM_COLL_AVOID_PROCESS_AREA_IDLE`	→	`CM_COLL_AVOID_PROCESS_AREA_OCCUPIED` if the process area is enabled and one carrier is present in the process boundary area
`CM_COLL_AVOID_PROCESS_AREA_IDLE`	→	`CM_COLL_AVOID_PROCESS_AREA_OCCUPIED` if the process area is enabled and one carrier is present in the process area and one or more carriers are in the process boundary area
`CM_COLL_AVOID_PROCESS_AREA_EMPTY`	→	`CM_COLL_AVOID_PROCESS_AREA_IDLE` if the process area is disabled and no carrier is present in the process boundary area
`CM_COLL_AVOID_PROCESS_AREA_OCCUPIED`	→	`CM_COLL_AVOID_PROCESS_AREA_IDLE` if the process area is disabled and one carrier is present in the process area
`CM_COLL_AVOID_PROCESS_AREA_OCCUPIED`	→	`CM_COLL_AVOID_PROCESS_AREA_IDLE` if the process area is disabled and one carrier is present in the process area and one or more carriers are in the process boundary area

CmCollAvoidGetProcessAreaData

With the function `CmCollAvoidGetProcessAreaData` the application can read all (`CM_COLL_AVOID_MAX_NR_OF_PROCESS_AREAS`) process area identifiers, the area (minimum and maximum position) of each process area, and the state of all process areas of a track. The function has 4 arguments.

The first argument, `trackId`, specifies the track of which the process area information is to be read. The second argument is the array `processAreaData` which holds the retrieved information of all process areas. The third argument, `processAreaDataLength`, specifies the allocated size of the array `processAreaData` in number of elements. The last argument, `pNrOfProcessAreas`, specifies the number of valid process area identifiers in the array `processAreaData`. The data of each process area is stored in the array. Non-existing process area entries in the array are identified with the value of `processAreaId` set to `CM_NO_ID`, the minimum and maximum positions of the area are set to 0, and the state of that process area is set to `CM_COLL_AVOID_PROCESS_AREA_IDLE`.

CmCollAvoidSynchronize

With the function `CmCollAvoidSynchronize` you can wait until the carrier has stopped at the end position with the synchronization request `CM_REQ_CARRIER_MOVEMENT_STOPPED`, or wait until the specified timeout has expired. You can also synchronize on removing a carrier (`CM_REQ_CARRIER_MOVING_CARRIER_REMOVED`) or adding a carrier (`CM_REQ_CARRIER_MOVING_CARRIER_ADDED_AND_CONTROLLED`). When the carrier stops at an intermediate position to avoid a collision the synchronization requests `CM_REQ_CARRIER_MOVEMENT_STOPPED`, `CM_REQ_CARRIER_MOVING_CARRIER_ADDED_AND_CONTROLLED` and `CM_REQ_CARRIER_MOVING_CARRIER_REMOVED` are not fulfilled.

The function `CmCollAvoidSynchronize` has 3 arguments. The first argument `cmId` specifies a CM object which refers to a track or a carrier. The second argument `syncRequest` specifies the synchronization request. The last argument `timeout` specifies the timeout in [s].

When `timeout` is 0, the function `CmCollAvoidSynchronize` checks on completion of synchronization request. When `timeout` is `NYCE_INFINITE`, the function `CmCollAvoidSynchronize` waits indefinitely on completion of the synchronization request.

With the synchronization request `CM_REQ_CARRIER_MOVEMENT_STOPPED`, `CmCollAvoidSynchronize` checks or waits until either the timeout has expired or the carrier has stopped at the end position.

CmCollAvoidReadCarrierState

With the function `CmCollAvoidReadCarrierState` you can retrieve the state of a carrier involved in a collision avoidance movement. The function has 2 arguments. The first argument `carrierId` specifies the carrier. The second argument `pCarrierState` returns the collision avoidance carrier state, which is enumerated in the `CM_COLL_AVOID_CARRIER_STATE` enumeration.

4.7.31 Camming a carrier on a track

Function

Just as you can execute a camming profile on a rotary axis based on the position of a master axis, you can also execute a camming profile on a carrier on a track based on the position of a master axis. The functionality is virtually the same. Camming on rotary axes uses NHI and SAC functions, camming on carriers uses NHI and CM functions.

The NHI functions used for camming on rotary axes and camming on carriers are the same. With the functions `NhiEcgDefineCamMemory`, `NhiEcgDeleteCamMemory`, `NhiEcgDownloadCamTable`, and `NhiEcgDownloadCamTableFromFile` you define (allocate) and delete (release) the memory for cam tables in the firmware of all involved nodes and enable downloading of a cam table from a structure defined by the

application or from an XML formatted file to all involved nodes. The modulo track is implemented analogous to the modulo axis. The modulo range of an axis is defined by the minimum and maximum position. One period equals (maximum position - minimum position + 1 increment). A track is defined by the track begin position and the track end position. The track period is the difference between track end position and track begin position plus 1 increment (0.5 pu). The size of a cam table for a modulo track is 1 period. The maximum (end) position of the cam table is the minimum (begin) position plus 1 period. See for more information NYCe 4000 Software User Manual, chapter "Node Hardware Interface (NHI)".

With the function `CmEcgLink` a track is linked to a master axis. If the node sample frequency is higher than 1 kHz, `CmEcgLink` returns the error code `CM_ERR_ECG_SAMPLE_FREQUENCY_HIGHER_THAN_1KHZ`. Note that when a track is linked to a master axis, the carriers on the track are not (yet) moving according to a cam profile. With the function `CmEcgLockCam` carriers start moving according to the specified cam table based on the position of the master axis. Camming is stopped with the function `CmEcgUnlink`.

After a master axis is linked to a track, the displacement of the master axis is available to all coils of that track. The displacements are accumulated to obtain a master position. The sign of the displacement can be inverted with the function `CmEcgSetMasterSign`. The default sign is positive (1). With the function `CmEcgSetMasterPos` a (new) value for the master axis position as maintained by all coils on the track can be set. This function is typically used when the master axis is at standstill.

If the master axis rotates in one direction, in due time, the error `SAC_AX_ERR_MASTER_POSITION_OVERFLOW` occurs. To avoid this error, the master position as maintained by the coils is calculated using a modulo value. The modulo value is retrieved from the cam table with the function `CmEcgSetMasterModuloValue`. This function uses the minimum master position `minMasterPos` and the master range `masterPeriod` from the specified cam table. The value of `minMasterPos` and `masterPeriod` can be reset to 0 with the function `CmEcgResetMasterModuloValue`.

A carrier can be put in camming mode with the function `CmEcgLockCam`. The specified struct `SAC_ECG_LOCK_CAM_PARS` contains the cam table which specifies pairs of master and slave positions for a given interval of the cam function, the master and slave end position and the lock mode. A carrier can be unlocked from the cam profile using, for example, the function `CmStopCarrier`.

- Use of a master axis with a high-resolution encoder (for example 20-bit per revolution) is recommended.
- If a cam table is defined for a modulo track in the full range of the track, it is recommended to start the "lock cam" of the carriers in a sync group.
- A safe method to start camming is with the master axis initially at standstill.
- A safe method to start camming when the master axis rotates, is executing a `CmEcgLockCam` for multiple carriers in a sync group.
- At CM level, only the lock mode `SAC_ECG_LOCK_MAS_REL_SLAVE_REL` is supported, and `masterEndPos` and `slaveEndPos` value must be 0.

The following table lists the properties of the camming carrier functionality.

Number of nodes on which cam tables are downloaded	≤ 18
Number of coils involved in ECG functionality	≤ 72
Number of carriers involved in ECG functionality	≤ 20
Number of tracks linked to a master axis	≤ 2
Number of cam tables loaded per node	1
Node frequency	1 kHz
CM on the node support	No

Tab. 4-7: Supported camming carrier functionality

Explanation

CmEcgLink and CmEcgUnlink

With the function `CmEcgLink` a track is linked to a master axis. A track is a collection of coils. As different carriers can move over the same coil it is not possible to link a specific carrier to a master axis. The master axis must be `SAC_VIRTUAL_AXIS`, `SAC_SENSOR_AXIS`, `SAC_STEPPER_AXIS`, `SAC_SENSING_STEPPER_AXIS`, or `SAC_SERVO_AXIS`. A track can be linked to only one master axis. The function `CmEcgLink` links all coils of the specified track to the specified master axis. If an error is detected, for example when a coil is already linked to an other master axis, an error is generated and all coils that are already linked are unlinked. The function `CmEcgUnlink` unlinks all coils of the specified track from the master axis.

Master position

With the function `CmEcgSetMasterSign` the sign of the master displacements can be changed to result in a master position that moves in the correct direction. With the function `CmEcgSetMasterPos` the value for the master axis position as maintained by all coils on the track is set. This makes the master position and the position maintained by the coils consistent.

With the function `CmEcgSetMasterModuloValue` the master position as maintained by the coils uses a modulo value. The modulo value is retrieved from the minimum master position `minMasterPos` and the master range `masterPeriod` from the specified cam table. `minMasterPos` is the first master position in the cam table, and `masterPeriod` is the difference between the last and the first master position in the cam table. The default value of `minMasterPos` and `masterPeriod` is 0. This implies that it is possible that the error `SAC_AX_ERR_MASTER_POSITION_OVERFLOW` may occur. You can call the function `CmEcgSetMasterModuloValue` with a different cam table to change the values of `minMasterPos` and `masterPeriod`. The value of `minMasterPos` and `masterPeriod` can also be reset to 0 with the function `CmEcgResetMasterModuloValue`. The modulo parameters can only be changed when a master axis is linked, the specified cam table is loaded, and the track has no camming carriers.

Camming

Carriers can be put in camming mode with the function `CmEcgLockCam`. The state of the carrier(s) must be `CM_CARRIER_STOPPED`. `CmEcgLockCam` has two parameters, `carrierId` which specifies the carrier, and `pCamPars` which specifies a struct `SAC_ECG_LOCK_CAM_PARS`.

Struct member	explanation
`tableId`	identifier of the cam table that the carrier must follow.
`masterTablePos`	the master position in the cam table from which the table must be followed.

Struct member	explanation
masterEndPos	the master position from which the table must be followed (end of transition).
slaveEndPos	the slave position from which the table must be followed (end of transition).
lockMode	indicates how masterEndPos and slaveEndPos are specified (absolute, relative, or (for slave only) automatically.

Tab. 4-8: members of the struct SAC_ECG_LOCK_CAM_PARS

The meaning of the members in the struct SAC_ECG_LOCK_CAM_PARS is shown in fig. 4-37 "Lock cam parameters" on page 129.

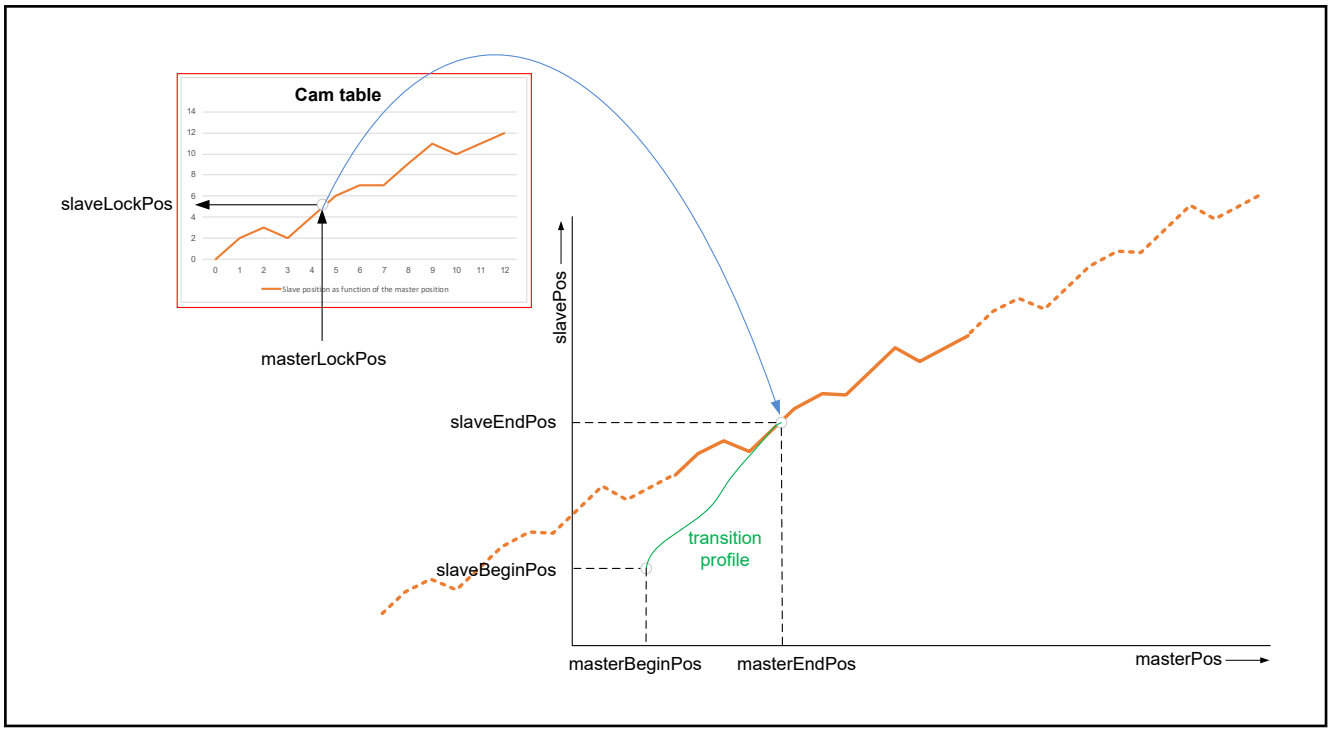
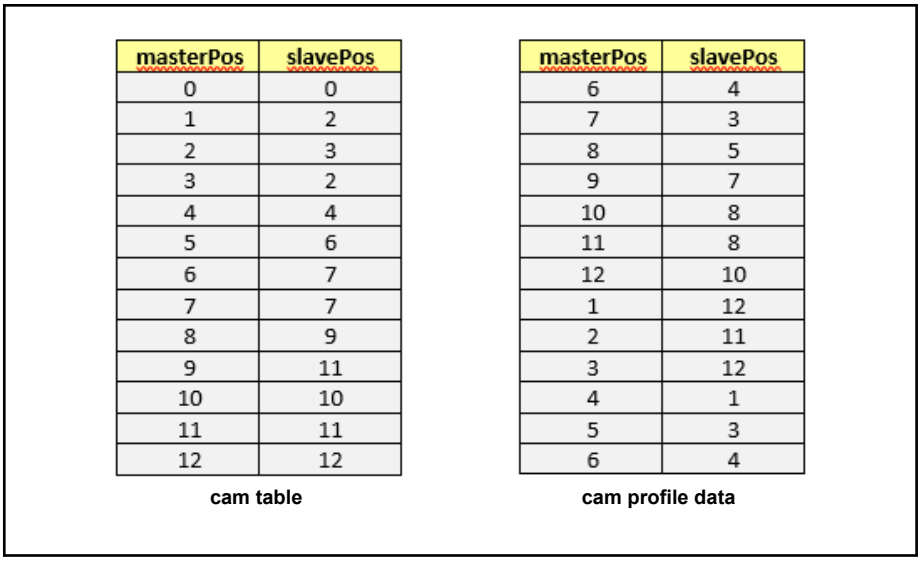


Fig. 4-37: Lock cam parameters

The cam table and the masterLockPos are shown in the upper left part of the figure. This is the master position in the cam table coordinate system, at which camming really started. Given the masterLockPos, the slaveLockPos can be retrieved from the cam table.

The master position / slave (carrier) position relation is shown in the lower right part of the figure. This figure shows the masterEndPos and slaveEndPos in this coordinate frame. These are the positions where the master and slave should be when the camming is really starting. From that point on the slave position is a function of the master position, in accordance with the cam table. As you can see, the cam table is periodically repeated (the dashed graph in the figure).

Example



masterPos	slavePos
0	0
1	2
2	3
3	2
4	4
5	6
6	7
7	7
8	9
9	11
10	10
11	11
12	12

cam table

masterPos	slavePos
6	4
7	3
8	5
9	7
10	8
11	8
12	10
1	12
2	11
3	12
4	1
5	3
6	4

cam profile data

Fig. 4-38: Example of a cam table and the cam profile

The table at the left side shows a cam table. Both master axis and slave axis are modulo axes counting mod 12. Thus "12" and "0" represent the same position. The following calculations are needed to generate the cam profile data, which is shown in the table at the right side.

Program:

```
"lock cam" with
   masterLockPos = 2  →  slaveLockPos = 3
   masterEndPos  = 6
   slaveEndPos   = 4

   masterOffset = masterEndPos - masterLockPos
   slaveOffset  = slaveEndPos - slaveLockPos

   slavePos = tableValue(masterPos - masterOffset) + slaveOffset
```

At the moment the "lock cam" is started, the master and the slave are possibly not at the position `masterEndPos` and `slaveEndPos` respectively, but at the `masterBeginPos` and `slaveBeginPos` as shown in the figure. In that case, a transition profile is generated, while the master moves from `masterBeginPos` to `masterEndPos`.

An additional precondition regarding the "lock cam" parameters applies when the master position calculations use a modulo value. If the master axis is a modulo axis, the `minMasterPos` and `masterPeriod` must match the values as can be retrieved from the specified cam table in the struct `SAC_ECG_LOCK_CAM_PARS`.

If the track, on which the carrier is located, is a modulo track, carrier positions at the coil level may have to exceed the modulo boundaries of the track without being modulo-ed. At coil level, modulo boundaries of the track are unknown. Further, modulo transitions of the master axis possibly do not coincide with modulo transitions of the slave. For these reasons, a restriction applies on the cam table for modulo tracks. The cam table defines a function `f` of the master position for the range $X_{m,min}$ (that is `minMasterPos`) until $X_{m,max}$ (that is `minMasterPos` + `masterPeriod`). This function must satify the following conditions.

- $f(X_{m,max}) - f(X_{m,min}) =$ `trackPeriod`
- for all X_m in the range $[X_{m,min} : X_{m,max}]$: $f(X_{m,min}) \leq f(X_m) \leq f(X_{m,max})$

No transition profile before the actual lock to the cam table implies that only a relative `masterEndPos` value 0 is allowed, or an absolute `masterEndPos` value equal to the current master position as known by the slave. To lock a carrier to a moving master axis it is most convenient to specify a relative `masterEndPos` value 0. To avoid discontinuities in the slave setpoint position, the relative `slaveEndPos` value must also be 0, or the absolute `slaveEndPos` must be equal to the actual slave setpoint position. For simplicity, when there is no transition profile, at CM level, only the lock mode `SAC_ECG_LOCK_MAS_REL_SLAVE_REL` is supported, and `masterEndPos` and `slaveEndPos` must be 0.

4.7.32 Markers

Function

Using the LMS marker functionality, you can activate a digital output when a carrier passes a specified position on a track. The digital output has to be configured before you can use the LMS marker functionality. The action on the digital output can be set, clear, or toggle the state of the digital output. Further, an event can be generated when a marker is detected. The LMS marker functionality has the following functions to define and delete markers.

- CmDefineSingleShotMarker
- CmDeleteSingleShotMarkers
- CmDefinePermanentMarker
- CmDeletePermanentMarker

☞ The type of the digital output, on which the output action is executed, must be configured on the NY4120/10 module. If the digital output type is not configured when the marker is defined, the error `CML_ERR_INVALID_DIGOUT_TYPE` is generated.

☞ The carrier position is used to detect whether a marker position is reached. When carriers have different product offset values, the carrier position is adjusted, resulting in a different physical position on the track for each carrier. As a result, one carrier may reach the marker position, and an other carrier may not reach that marker position.

☞ Based on the specified marker position, it is calculated which coil controls this position best. The node connecting this coil has to be identified by the parameter `outputNodeName`. If the coil is not connected to the specified node, an error is generated.

See NYCe 4000 Software User Manual, chapter 8 "Single Axis Control (SAC)", subchapter "Markers", for more information about the marker functionality in general. This chapter specifically describes the LMS marker functionality.

On one node up to 8 marker calculations can be executed per drive unit per MCU sample period. If more than one marker occurs within the same MCU sample period, the order of the generated marker events sent to the PC is undefined.

Explanation

CmDefineSingleShotMarker

The single shot marker defines a position on a track. When a carrier passes this position, a digital output is activated. When the marker is detected, the marker is automatically deleted. If a digital output must be activated again on this position, the single shot marker has to be defined again.

The single shot marker is carrier-specific. When a single shot marker is defined, the digital output is only activated when the specified carrier moves over the marker position. When an other carrier moves over the marker position, the digital output is not activated and the single shot marker is not deleted.

The single shot marker is defined with the function `CmDefineSingleShotMarker`. The function has 3 arguments, `trackId` specifies the track, `carrierId` specifies the carrier, and `pCmMarkerPars` is a pointer to a structure of type `CM_MARKER_PARS`. This structure contains the following specific marker information.

Structure member	Description
`markerId`	specifies the marker identifier
`position`	defines the marker position
`direction`	specifies whether the marker is detected when passing the position while moving in the negative direction, positive direction, or any direction
`outputAction`	specifies the output action, set, clear or toggle
`outputNodeName`	specifies the name of the node where the coil is connected controlling the specified marker position
`outputId`	specifies the slot identifier and I/O number of the digital output that is activated by the marker
`eventFlag`	specifies whether a marker event has to be generated when the marker is detected

Tab. 4-9: CM_MARKER_PARS structure members

The identifier used for a single shot marker does not have to be unique. The same marker identifier can be used for multiple marker definitions. The marker position can be any position on the track between track begin and track end position. If the `eventFlag` is set, the event `CM_EV_MARKER_SINGLE_SHOT` is generated when the marker is detected. The event data contains the marker identifier.

With the function `CmDefineSingleShotMarker` up to 128 single shot markers per coil can be defined when the track state is `CM_TRACK_INACTIVE`, `CM_TRACK_ACTIVE`, and `CM_TRACK_ERROR`, because, in these track states, carriers have an identifier assigned to them. However, a single shot marker will only execute an output action and/or generate an event when the track state is `CM_TRACK_ACTIVE`. Per coil on the track, single shot markers are detected in the order in which they are defined.

CmDeleteSingleShotMarkers

With the function `CmDeleteSingleShotMarkers` all defined single shot markers on a track are deleted. The function has one argument, `trackId`

which specifies the track for which all defined markers have to be deleted. The function can be called in any track state, except CM_TRACK_POWERUP.

All defined single shot markers on a track are also deleted in the following situations.

- When the track makes a state transition to an other state than CM_TRACK_INACTIVE, CM_TRACK_ACTIVE, or CM_TRACK_ERROR. This means that calling the function CmShutdown, CmSetTrackStateToIdle, CmHomeTrack, CmStartVelocityMode, or CmStartVelocityModeCoils deletes all single shot markers defined on the specified track.
- When the track enters the state CM_TRACK_FATAL_ERROR.
- When coils are removed from a track using the RTLC (Runtime Track Layout Changes) functionality, all single shot markers on positions controlled by the removed coils are deleted.

CmDefinePermanentMarker

The permanent marker defines a position on a track. When a carrier passes this position, a digital output is activated. When the marker is detected, the marker is not deleted. The permanent marker is not carrier-specific. When a permanent marker is defined, the digital output is activated by any carrier that moves over the marker position.

The permanent marker is defined with the function CmDefinePermanentMarker. The function has 2 arguments, trackId specifies the track, and pCmMarkerPars is a pointer to a structure of type CM_MARKER_PARS. This structure is described in the explanation of the function CmDefineSingleShotMarker.

The identifier used for a permanent marker has to be unique. The marker position can be any position on the track between track begin and track end position. If the eventFlag is set, the event CM_EV_MARKER_PERMANENT is generated when the marker is detected. The event data contains the marker identifier.

The function CmDefinePermanentMarker can be called after the function CmInitialize has been called. Up to 16 permanent markers per coil can be defined. The track may be in any track state, except CM_TRACK_POWERUP. However, a permanent marker will only execute an output action and/or generate an event when the track state is CM_TRACK_ACTIVE.

CmDeletePermanentMarker

With the function CmDeletePermanentMarker a defined permanent marker is deleted. The function has one argument, markerId which specifies the permanent marker that has to be deleted. The function can be called in any track state, except CM_TRACK_POWERUP.

All defined permanent markers on a track are deleted in the following situations.

- When the function CmShutdown is called.
- When coils are removed from a track using the RTLC (Runtime Track Layout Changes) functionality, all permanent markers on positions controlled by the removed coils are deleted.

4.7.33 Reading carrier data

Functions and explanation

With the function CmReadCarrierVariable you can retrieve information of a carrier, see tab. 4-13 "Carrier variables" on page 147. It is convenient to have a function available to retrieve data of interest of a single carrier or all

carriers at a time. To this end, the functions CmReadSingleCarrierData and CmReadCarrierData are available.

The input parameter for the function CmReadSingleCarrierData is a carrier ID, and the function returns a structure of type CM_CARRIER_DATA.

The function CmReadCarrierData has no parameters, and the function returns an array with CM_MAX_NR_OF_CARRIERS elements of structure CM_CARRIER_DATA. The structure CM_CARRIER_DATA contains the carrier identifier, position of the carrier, the velocity, acceleration and jerk of the carrier, the carrier state, and the carrier controller state. The returned data is only valid if the carrier state is not CM_CARRIER_POWERUP. To obtain the data of a specific carrier, you must search in the returned carrierData array the member carrierId of each entry for its carrier identifier. If the carrierId member is CM_NO_ID the array entry represents a non-existing carrier.

The client state must be CM_CLIENT_CONNECTED. There are no restrictions regarding track and carrier state, although useful information for the carrier state and carrier controller state can only be expected if the track state is CM_TRACK_ACTIVE.

4.7.34 Resetting errors

Function

Resetting errors is possible with the function CmResetError. This function has the track ID as input.

This function can be used in any track state.

Explanation

This function resets the errors and warnings detected by the coils of a track, and it changes the state of the track as follows.

- in case of a warning or no error

  from CM_TRACK_ACTIVE to CM_TRACK_INACTIVE
- in case of an error

  from CM_TRACK_ERROR to CM_TRACK_INACTIVE
- in case of a fatal error

  from CM_TRACK_FATAL_ERROR to CM_TRACK_IDLE

See fig. 4-3 "State Transition Diagram for tracks in CM" on page 52.

Presuming the error cause has been removed, the track is now ready to be made operational again, either by just activating the track (in case of a warning or a non-fatal error), or by homing, re-assigning IDs and activating the track (if the error was fatal).

If CmResetError is called while the cause of an error is still present, the track state is changed to CM_TRACK_FATAL_ERROR or CM_TRACK_ERROR by the CM error handling.

You can use CmResetError to recover from certain errors. For example, if a client application is stopped while a home operation, or a movement in velocity mode is started by that client application, the movement is not stopped. You can issue a CmResetError command from another client application to stop the movement.

4.7.35 Clearing warnings

Function

Clearing warnings is possible with the function `CmClearWarning`. This function has the track ID as input.

This function can be used in any track state.

Explanation

A separate function is available which only clears all warnings on a specific track. This allows you to clear warnings without modification of the track state.

4.7.36 Handling special controller parameters for an area

Function

Selecting if the special controller parameters must be used for an area of a track is done with the function `CmSelectSpecialControllerPars`. This function has the track ID, a pointer to a `CM_AREA` structure, which specifies the area begin and end position, and a boolean, indicating if the special controller parameters must be used, as input.

With the functions `CmWriteSpecialControllerPars` and `CmReadSpecialControllerPars` you can store and retrieve the special controller parameters.

Explanation

A set of special controller parameters can be defined for an area of a track. These special controller parameters are valid for the coils of which the coil position is within the boundaries of the area. For an area of a track it is possible to select if the normal or special controller parameters must be used. The normal controller parameters of a coil are specified in the configuration file.

When multiple areas are defined they may not overlap. Default the set of special controller parameters for an area of a track are the same as the normal controller parameters.

Writing a set of special controller parameters for an area of a track is done with the function `CmWriteSpecialControllerPars`. The function has the track ID, a pointer to a `CM_AREA` structure and a pointer to the `CM_CONTROLLER_PARS` structure as input. The `CM_AREA` structure specifies the area begin and end position. The `CM_CONTROLLER_PARS` structure specifies the controller parameters.

Reading a set of special controller parameters for an area of a track is done with the function `CmReadSpecialControllerPars`. The function has the track ID, and a pointer to a `CM_AREA` structure. The `CM_AREA` structure specifies the area begin and end position. The function returns a pointer to a `CM_CONTROLLER_PARS` structure which contains the special controller parameters, and a boolean that indicates whether the special controller parameters are used. The information of the first coil in the area detected in positive direction is returned.

Activation of the special controller parameters requires the use of both functions `CmWriteSpecialControllerPars` and `CmSelectSpecialControllerPars`. If a special controller parameter value is invalid, the error `CM_ERR_PARAMETER` is returned. The check if the parameters are valid is done when both functions have been called. Thus,

the order in which the 2 functions are called defines which function will return the error code if an invalid special controller parameter value is specified.

- If the function `CmSelectSpecialControllerPars` (with the boolean set to `TRUE`) was called before `CmWriteSpecialControllerPars`, the function `CmWriteSpecialControllerPars` returns the error if a special controller parameter value is invalid.
- If the function `CmWriteSpecialControllerPars` was called (with an invalid parameter value) before the function `CmSelectSpecialControllerPars` (with the boolean set to `TRUE`) is called, the function `CmSelectSpecialControllerPars` returns the error if a special controller parameter value is invalid.

Reading the set of special controller parameters of an area of a track and if they are used is done with the function `CmReadSpecialControllerPars`. This function has the track ID and a pointer to a `CM_AREA` structure, which specifies the area begin and end position, as input. This function outputs a pointer to the `CM_CONTROLLER_PARS` structure, containing the special controller parameters, and a boolean, indicating if the special controller parameters are used. The information of the first coil in the area seen in positive direction is returned.

4.7.37 Controlling carriers in velocity mode

Function

You can move carriers on a track with a constant velocity by activation of the velocity control loop. Two CM functions are available to activate the velocity control loop, `CmStartVelocityMode` and `CmStartVelocityModeCoils`. The velocity control loop is stopped with the CM function `CmStopVelocityMode`. The function `CmStartVelocityMode` can be used to move all carriers on a track with a given constant velocity, or hold all carriers in place (velocity set to 0). The function `CmStartVelocityModeCoils` activates the velocity control loop of a range of successive coils of a track and can be used to implement a user-specific home mode.

Explanation

If the velocity mode is started with the function `CmStartVelocityMode`, the velocity control loop is activated with a specific velocity for all coils of the track. All carriers on the track move with the specified constant velocity. The function `CmStartVelocityMode` has 2 arguments, the track identifier and the carrier velocity. The sign of the velocity indicates the direction. If the specified velocity is 0, all carriers are held in their current position. The function `CmStartVelocityMode` can be started from any state of the track. After calling the function `CmStartVelocityMode`, the track state changes to `CM_TRACK_VELOCITY_MODE`. When `CmStartVelocityMode` is called for the track shown in fig. 4-39 "Moving all carriers on a track in velocity mode" on page 137 with velocity 1000, the velocity control loop of all coils is activated with velocity 1000. All carriers are moving in positive direction with velocity 1000. When `CmStartVelocityMode` is called with velocity -2000, the velocity control loop of all coils is activated with velocity -2000. All carriers are moving in negative direction with velocity 2000.

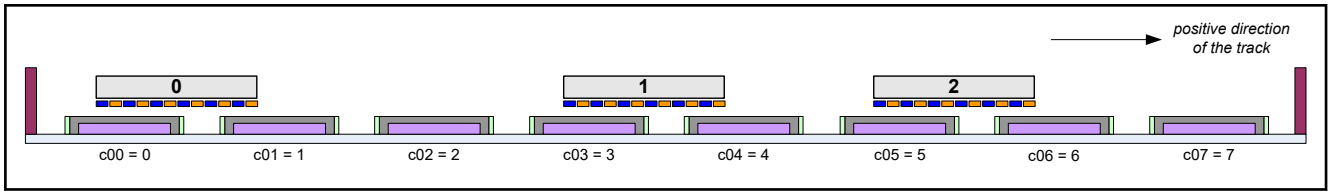


Fig. 4-39: Moving all carriers on a track in velocity mode

You can also activate the velocity control loop with the function `CmStartVelocityModeCoils` for a specific range of (successive) coils on a track with a specific velocity. All coils outside the specified range are also activated, but with a velocity of 0. This enables the movement of carriers that are above the range of successive coils with a constant velocity, and hold carriers that are above the other coils in place. The function `CmStartVelocityModeCoils` has 4 arguments, the track identifier, the coil index that specifies the start coil of the range of successive coils, the coil index that specifies the end coil of that range, and the carrier velocity. The sign of the velocity indicates the direction. The function `CmStartVelocityModeCoils` can be started from any state of the track. After calling the function `CmStartVelocityModeCoils`, the track state changes to `CM_TRACK_VELOCITY_MODE`.

When `CmStartVelocityModeCoils` is called for the track shown in fig. 4-40 "Moving specific carriers on a linear track in velocity mode" on page 137 with start coil index 2, end coil index 4, and velocity 1000, the velocity control loop of coils c02, c03, and c04 is activated with velocity 1000 and the velocity control loop of the coils c00, c01, c05, c06, and c07 is activated with velocity 0. As result, the carriers 0 and 2 are held in place, and the carrier 1 moves in positive direction with velocity 1000. Swapping the values of the start coil index and end coil index results in the same behavior.

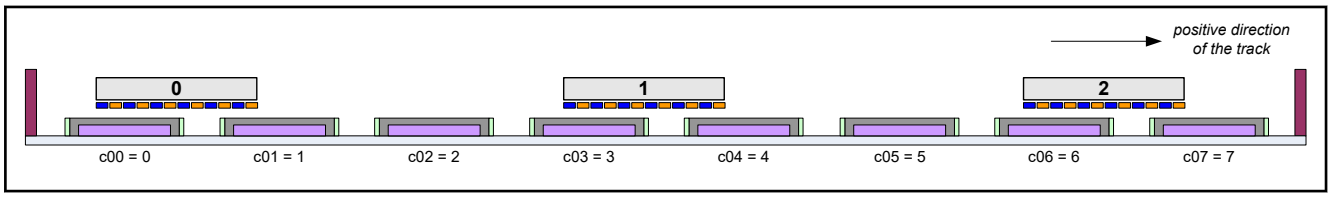


Fig. 4-40: Moving specific carriers on a linear track in velocity mode

When `CmStartVelocityModeCoils` is called for the track shown in fig. 4-41 "Moving specific carriers on a modulo track in velocity mode" on page 138 with start coil index 2, end coil index 4, and velocity 1000, the velocity control loop of coils c02, c03, and c04 is activated with velocity 1000 and the velocity control loop of the coils c00, c01, c05, ..., c18, c19, and c20 is activated with velocity 0. As result, the carriers 0 and 1 are held in place, and the carrier 2 moves in positive direction with velocity 1000. Swapping the values of the start coil index and end coil index results in the same behavior. However, if the modulo crossing is between the start coil index and the end coil index, it does matter which coil index is specified as start coil index, and which coil index is specified as end coil index. If start coil index is 19 and end coil index is 2, the velocity control loop of coils c19, c20, c00, c01, and c02 is activated with velocity 1000, and the velocity control loop of the coils c03 .. c18 is activated with velocity 0. If start coil index is 2 and end coil index is 19, the velocity control loop of coils c02 .. c19 is activated with velocity 1000, and the velocity control loop of the coils c20, c00, c01 is activated with velocity 0.

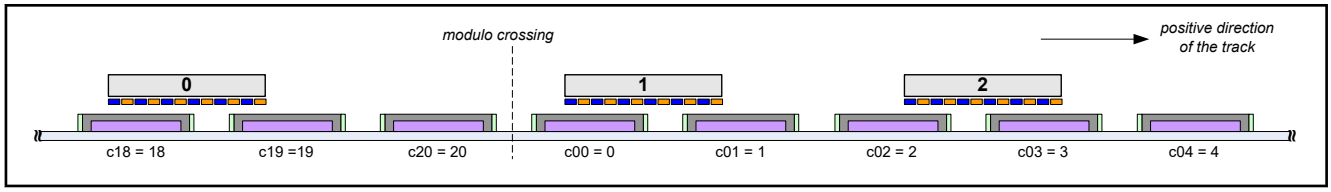


Fig. 4-41: Moving specific carriers on a modulo track in velocity mode

The velocity control loop is deactivated for all coils of the track with the function `CmStopVelocityMode`. The only argument for this function is the track identifier. The function `CmStopVelocityMode` is only allowed in the track state `CM_TRACK_VELOCITY_MODE`. When the function `CmStopVelocityMode` returns, all carriers are removed from the track administration and the track state changes to `CM_TRACK_IDLE`.

Remarks

- When `CmStartVelocityMode` or `CmStartVelocityModeCoils` is called, the track stays in velocity control mode until the function `CmStopVelocityMode` is called.
- With a movement in velocity mode no errors are generated when a carrier comes to standstill because the coils (of which the control loop is active with a velocity not equal to 0) cannot generate sufficient force to move the carrier. This implies that no error is generated when a carrier pushes against an endstop or against a carrier that is held in place.
- You can use the functions `CmStartVelocityMode`, `CmStartVelocityModeCoils` and `CmStopVelocityMode` in combination with the function `CmHomeTrack` for home mode 5 or 14 to implement a customer-specific home mode. With the functions `CmStartVelocityMode`, `CmStartVelocityModeCoils` and `CmStopVelocityMode` carriers can be moved to free coils and sensors of a track. You can use the function `CmReadCoilVariable` for variable `CM_VAR_COIL_xxx_CARRIER_DETECTED`, `CM_VAR_COIL_xxx_CARRIER_POS_VALID` and `CM_VAR_COIL_xxx_CARRIER_MOVING` to determine which sensor detects a carrier, which sensor has valid position information, or which sensor detects a moving carrier respectively. "`xxx`" can be `LEFT_SENSOR`, `MIDDLE_SENSOR`, or `RIGHT_SENSOR`. If the sensor has valid position information, the carrier position that is determined by the sensor can be read with the function `CmReadCoilVariable` for variable `CM_VAR_COIL_xxx_CARRIER_POSITION`. With the function `CmHomeTrack` for mode 5 or 14 the carriers can be added to the track. This is even possible when the carriers are held in place in velocity mode.
- It is the responsibility of the application to stop a movement with the function `CmStopVelocityMode`. If the client application, that started the movement, stops for whatever reason, you can stop the movement with the function `CmResetError` (issued by another client application).

4.7.38 Enabling / Disabling sensor detection

Function

To realize the sensor enable/disable detection functionality, the function `CmEnableSensorDetection` is available. The inputs for this function are:

- the track identifier
- a pointer to a `CM_AREA` structure

- a boolean indicating whether the sensor detection is enabled or disabled

Explanation

When a carrier is added to a track or removed from a track, the magnetic field sensed by the LMS sensors is subject to variations. This may cause a gradual change of the sine/cosine amplitude, but variations on the way are possible. This may lead to a repeated detected/not-detected condition of the added or removed carrier, in its turn possibly cause sensor errors, as if carriers enter from nowhere. To avoid this, it is desirable that the application can disable/enable the detection of carriers which are logically not anymore or not yet present in the system.

Example

Suppose a carrier must move from one track to another track. This could be done with the following sequence:

1. Remove the carrier from the first track.

   After `CmRemoveCarrier(carrierId)`, the specified carrier is logically no longer present in the system.
2. Disable the detection of the sensors that are sensing the removed carrier.
3. Disable the detection of the relevant sensors of the other track.
4. Add the carrier to the other track.

   The carrier is added by `CmAddCarrier(carrierId, trackId, addPosition)`.
5. Move the carrier to the other track.

   Wait until the carrier is above the receiving coil.
6. Enable the detection of the previously disabled sensors of both tracks.

4.7.39 Enabling / Disabling start triggers

Function

To start movement of carriers on the change of an input, the function `CmEnableStartTrigger` is available. This function simultaneously enables or disables the start on trigger for all coils of the track.

This function has the track identifier and a boolean, indicating whether the start on trigger is enabled or disabled, as input parameters.

Explanation

This functionality is intended for cases where all carriers on a track must start moving synchronously after the change of state of a digital input. The carriers that must start synchronously are grouped with a `CmDefineSyncGroup` command. The input is typically set by a PLC.

The input must be changed on all coils of the track. Therefore the start trigger must be defined in the axis (coil) configuration section of the NYCe 4000 XML file of each coil. Also the edge (rising, falling or any) must be specified in the XML file for each coil.

4.7.40 Synchronize on an asynchonous operation

Function

An asynchronous operation occurs when a CM command starts the requested operation and immediately returns to the application while the operation continues. For example, the movement of a carrier is an asynchronous operation. The carrier starts the movement when the CM movement command returns to the application. An application can wait or poll for the start of a carrier movement, the jogging of a carrier, and the stop of a carrier movement, with the function `CmSynchronize`. The function `CmSynchronize` can only be called when the client state is `CM_CLIENT_CONNECTED`.

Explanation

With the function `CmSynchronize` an application can wait or poll for a specific moment in an asynchronous operation. The parameters for the function `CmSynchronize` are a CM object, a synchronization request and a timeout period specified in seconds.

The CM object identifies for which CM object the synchronization is requested. CM object can be the LMS system as a single entity, a track, a carrier, or a coil.

The available synchronization requests and the behavior of `CmSynchronize` is as follows.

- `CM_REQ_CARRIER_MOVEMENT_STARTED`

  This synchronization request checks for the start of a movement of a carrier. `CmSynchronize` acts as follows.

  – If the carrier is not in the state `CM_CARRIER_STOPPED` or `CM_CARRIER_SUSPENDED` or no movement is pending, the error code `CM_CARRIER_STATE_ERROR` is returned.

  – If the carrier is in the `CM_CARRIER_MOVING` state, `NYCE_OK` is returned.

  – If the two mentioned conditions do not apply, `CmSynchronize` waits for the specified timeout time.

- `CM_REQ_CARRIER_MOVEMENT_STOPPED`

  This synchronization request checks for the stop of a movement of a carrier. `CmSynchronize` acts as follows.

  – If the carrier is not in the state `CM_CARRIER_POWERUP`, `CM_CARRIER_STOPPED`, `CM_CARRIER_MOVING`, `CM_CARRIER_JOGGING`, or `CM_CARRIER_SUSPENDED`, the error code `CM_CARRIER_STATE_ERROR` is returned.

  – If the carrier is in the `CM_CARRIER_STOPPED` state or no movement is pending, `NYCE_OK` is returned.

  – If the two mentioned conditions do not apply, `CmSynchronize` waits for the specified timeout time.

- `CM_REQ_CARRIER_MOVEMENT_JOGGING`

  This synchronization request checks for that the carrier is jogging. `CmSynchronize` acts as follows.

  – If the carrier is not in the state `CM_CARRIER_STOPPED`, `CM_CARRIER_MOVING`, or `CM_CARRIER_SUSPENDED`, or no movement is pending, the error code `CM_CARRIER_STATE_ERROR` is returned.

  – If the carrier is in the `CM_CARRIER_JOGGING` state, `NYCE_OK` is returned.

  - If the two mentioned conditions do not apply, `CmSynchronize` waits for the specified timeout time.

- `CM_REQ_TRACK_HOMING_COMPLETED`

  This synchronization request checks that the track is homed. `CmSynchronize` acts as follows.

  - If the function `CmHomeTrack` is not called, the error code `CM_TRACK_STATE_ERROR` is returned.
  - If the home operation for the track is completed, `NYCE_OK` is returned.
  - If the two mentioned conditions do not apply, `CmSynchronize` waits for the specified timeout time.

  Homing of a track is an asynchronous operation. The home procedure is started when the function `CmHomeTrack` returns.

- `CM_REQ_TRACK_CARRIERS_AGAINST_ENDSTOP`

  This synchronization request checks that all carriers on the track are against the end stop. `CmSynchronize` acts as follows.

  - If the track state is `CM_TRACK_IDLE` or `CM_TRACK_INACTIVE`, or the function `CmMoveCarriersAgainstEndstop` is not called, the error code `CM_TRACK_STATE_ERROR` is returned.
  - If all carriers are against the end stop, `NYCE_OK` is returned.
  - If the two mentioned conditions do not apply, `CmSynchronize` waits for the specified timeout time.

  Moving all carriers against an end stop of a track is an asynchronous operation. The carriers start to move when the function `CmMoveCarriersAgainstEndstop` returns.

- `CM_REQ_CARRIER_MOVING_CARRIER_REMOVED`

  This synchronization request checks that a carrier, for which the command `CmRemoveMovingCarrier` is issued, has left the track. `CmSynchronize` acts as follows.

  - If the specified carrier does not exist, the error code `CM_ERR_INVALID_CARRIER_ID` is returned.
  - If the specified carrier exists, but no `CmRemoveMovingCarrier` is issued for that carrier, the error code `CM_ERR_CARRIER_NOT_MARKED_FOR_REMOVAL` is returned.
  - If the specified carrier exists and that carrier is marked for removal (`CmRemoveMovingCarrier` issued), `CmSynchronize` waits until the event `CM_EV_CARRIER_MOVING_CARRIER_REMOVED` occurs for the specified carrier, or until the specified timeout expires.
  - If the application uses polling (timeout = 0), `CmSynchronize` returns the error code `CM_ERR_REQUEST_TIMEOUT` if the specified carrier is marked for removal. The application must poll until the returned code is `CM_ERR_INVALID_CARRIER_ID`. Note that in case the specified carrier does not exist, the error code `CM_ERR_INVALID_CARRIER_ID` is always returned.

- `CM_REQ_CARRIER_MOVING_CARRIER_ADDED_AND_CONTROLLED`

  This synchronization request checks that for the specified carrier the command `CmAddMovingCarrier` is issued. `CmSynchronize` acts as follows.

- If the specified carrier does not exist, the error code `CM_ERR_INVALID_CARRIER_ID` is returned.
- If the specified carrier exists, but no `CmAddMovingCarrier` is issued for that carrier, the error code `CM_ERR_CARRIER_NOT_MARKED_FOR_ADDITION` is returned.
- If the specified carrier exists and that carrier is marked for addition (`CmAddMovingCarrier` issued), `CmSynchronize` waits until the events `CM_EV_CARRIER_CONTROLLER_MOVING_EN` occurs for the specified carrier, or until the specified timeout expires.
- If the application uses polling (timeout = 0), `CmSynchronize` returns the error code `CM_ERR_REQUEST_TIMEOUT` if the specified carrier is marked for addition. The application must poll until the returned code is `CM_ERR_CARRIER_NOT_MARKED_FOR_ADDITION`.
- If the application calls `CmCancelAddMovingCarrier`, the synchronization request is cancelled, and `CmSynchronize` returns the error `CM_ERR_REQUEST_ABORTED`. In case the application uses polling, this returned error must also be checked.

Pending `CmSynchronize` requests are aborted if the function `CmInitialize` or `CmShutdown` is called, or if an error occurs.

The timeout period specifies how long `CmSynchronize` waits for the completion of the synchronization request. If the specified timeout period is 0, the status of the synchronization request is polled. If the specified timeout period is `CM_INDEFINITE`, the function `CmSynchronize` waits indefinitely for the completion of the synchronization request.

When an asynchronous error occurs while waiting for a `CmSynchronize` request completion, the status `CM_ERR_ASYNC_ERROR_OCCURRED` is returned. The status `CM_ERR_ASYNC_ERROR_OCCURRED` can also be returned from the functions `CmMoveCarriersAgainstEndstop`, `CmHomeTrack`, and `CmFreeArea`. The actual error that occurred can be retrieved from the logging or via the function `CmReadTrackVariable` with parameter `CM_VAR_TRACK_FIRST_ERROR_COIL` which returns an identifier of the coil that detected the error. With the identifier you can use the function `CmReadCoilVariable` to retrieve the error code.

When a system shutdown is initiated by a client application, all pending `CmSynchronize` requests are aborted and all defined event enrolments are deleted.

4.7.41 Event handling

The event mechanism enables the execution of a specified enrolment function when the specified event occurs. CM service supports the following events.

- System events
  - `CM_EV_SYSTEM_INITIALIZED`
  - `CM_EV_SYSTEM_SHUTDOWN`
- State changes of a track. An "entered" and "left" event is available for each track state. The events are generated when a track makes a state transition. See fig. 4-3 "State Transition Diagram for tracks in CM" on page 52 for the track states. The event identifiers are the following.
  - `CM_EV_TRACK_IDLE_EN`
  - `CM_EV_TRACK_IDLE_LE`

- CM_EV_TRACK_VELOCITY_MODE_EN
- CM_EV_TRACK_VELOCITY_MODE_LE
- CM_EV_TRACK_HOMED_EN
- CM_EV_TRACK_HOMED_LE
- CM_EV_TRACK_INACTIVE_EN
- CM_EV_TRACK_INACTIVE_LE
- CM_EV_TRACK_ACTIVE_EN
- CM_EV_TRACK_ACTIVE_LE
- CM_EV_TRACK_MOVING_CARRIER_EN
- CM_EV_TRACK_MOVING_CARRIER_LE
- CM_EV_TRACK_ERROR_EN
- CM_EV_TRACK_ERROR_LE
- CM_EV_TRACK_FATAL_ERROR_EN
- CM_EV_TRACK_FATAL_ERROR_LE

Note that the track state CM_TRACK_POWERUP has no "entered" or "left" event defined, because these state transitions occur with the functions CmShutdown and CmInitialize. During the execution of these functions all defined events are deleted.

- State changes of a carrier. An "entered" and "left" event is available for each carrier state. The events are generated when a carrier makes a state transition. See fig. 4-4 "State Transition Diagram for carriers in CM" on page 55 for the carrier states. The event identifiers are the following.
  - CM_EV_CARRIER_POWERUP_EN
  - CM_EV_CARRIER_POWERUP_LE
  - CM_EV_CARRIER_STOPPED_EN
  - CM_EV_CARRIER_STOPPED_LE
  - CM_EV_CARRIER_SUSPENDED_EN
  - CM_EV_CARRIER_SUSPENDED_LE
  - CM_EV_CARRIER_MOVING_EN
  - CM_EV_CARRIER_MOVING_LE
  - CM_EV_CARRIER_JOGGING_EN
  - CM_EV_CARRIER_JOGGING_LE
  - CM_EV_CARRIER_MOVING_CARRIER_ADDED
  - CM_EV_CARRIER_MOVING_CARRIER_REMOVED
- State changes of the carrier controller. An "entered" and "left" event is available for each carrier controller state. The events are generated when a carrier controller makes a state transition. See fig. 4-5 "State Transition Diagram for carrier controller states in CM" on page 58 for the carrier controller states. The event identifiers are the following.
  - CM_EV_CARRIER_CONTROLLER_MOVING_EN
  - CM_EV_CARRIER_CONTROLLER_MOVING_LE
  - CM_EV_CARRIER_CONTROLLER_SETTLING_EN
  - CM_EV_CARRIER_CONTROLLER_SETTLING_LE
  - CM_EV_CARRIER_CONTROLLER_STABILIZING_EN

  - CM_EV_CARRIER_CONTROLLER_STABILIZING_LE
  - CM_EV_CARRIER_CONTROLLER_STEADY_EN
  - CM_EV_CARRIER_CONTROLLER_STEADY_LE
- Error notification. One error event is available, CM_EV_TRACK_ASYNC_ERROR_OCCURRED. The event is generated when a coil executes an error handler with severity 2 to 9. See chapter 5.4 "Error handling at coil level" on page 170 for more information.
- Warning notification. One warning event is available, CM_EV_TRACK_ASYNC_WARNING_OCCURRED. The event is generated when a coil executes an error handler with severity 1. See chapter 5.4 "Error handling at coil level" on page 170 for more information.
- Track layout changed notification. One track layout changed event is available, CM_EV_TRACK_LAYOUT_CHANGED. The event is generated when a track layout changes due to adding or removing coils.
- Single shot and permanent markers. When a marker is detected and the marker event generation is enabled, the event CM_EV_MARKER_SINGLE_SHOT, or CM_EV_MARKER_PERMANENT, respectively, is generated.

With the function CmDefineEventEnrolment a user application can enroll for an event. To enroll for any track or carrier event, the system state must be CM_SYSTEM_INITIALIZED. You can enroll for a state change of a track or carrier when the track id or carrier id is not (yet) valid. For example, you can enroll for carrier events for a carrier id that is not valid, because the carrier is added to the track later on. Enrolment for the system events can be done in any system state. The function CmDefineEventEnrolment has 4 parameters.

1. cmId - the identification of the CM object, and is a combination of an object type and an object identifier.
2. eventId - the event identifier.
3. handler - the event enrolment function.
4. UserData - specific user data that can be used to differentiate between successive enrolment function for the same event.

The event enrolment function is of data type CM_EVENT_HANDLER. The event enrolment function has 4 parameters.

1. cmId - the identification of the CM object that generated the event.
2. eventId - the identifier of the event that occurred.
3. EventData - the event specific data, if valid.
4. UserData - the specific user data supplied when the enrolment was defined.

The following event data is returned for a specific event identifier.

CM event identifier	Returned event data
All track state "entered" events	int32_t eventLong track state that is left.
All track state "left" events	int32_t eventLong track state that is entered.

CM event identifier	Returned event data
All carrier state "entered" events	`int32_t eventLong` carrier state that is left.
All carrier state "left" events	`int32_t eventLong` carrier state that is entered.
`CM_EV_TRACK_MOVING_CARRIER_EN` (note: `CmMoveToTrack` and `CmMoveAcrossTrackBounds` only)	`int32_t eventLong` carrier identifier of the detected carrier.
`CM_EV_TRACK_MOVING_CARRIER_LE` (note: `CmMoveToTrack` and `CmMoveAcrossTrackBounds` only)	`int32_t eventLong` carrier identifier of the no longer detected carrier.
`CM_EV_CARRIER_MOVING_CARRIER_ADDED` `CM_EV_CARRIER_MOVING_CARRIER_REMOVED`	`int32_t eventLong` trackId where carrier is added or removed.
All carrier controller state "entered" events	`int32_t eventLong` carrier controller state that is left.
All carrier controller state "left" events	`int32_t eventLong` carrier controller state that is entered.
`CM_EV_TRACK_ASYNC_ERROR_OCCURRED`	`int32_t eventLong` coil index of the coil that generated the asynchronous error.
`CM_EV_TRACK_ASYNC_WARNING_OCCURRED`	`int32_t eventLong` coil index of the coil that generated the asynchronous warning.
`CM_EV_TRACK_LAYOUT_CHANGED`	`int16_t eventShortArray` `[0]` : coil index of the first coil. `[1]` : coil index of the last coil. `[2]` : alternative at track begin. `[3]` : alternative at track end.
`CM_EV_MARKER_SINGLE_SHOT` `CM_EV_MARKER_PERMANENT`	`uint32_t eventUlong` marker identifier of the single shot or permanent marker that is detected.

Tab. 4-10: Event data returned by an event

Note that event data is only available if the CM object identifier is valid. For example, when a carrier is added to a track an event is generated, but the event data is set to -1, because there is no "carrier state that is left".

With the function `CmDeleteEventEnrolment` a user application can delete the enrolment for an event. The function `CmDeleteEventEnrolment` has the same 4 parameters as the function `CmDefineEventEnrolment`.

If several CM clients define a link between an enrolment function and an event, the event enrolment function defined by each CM client is automatically executed when the event occurs. Each CM client has a separate thread to execute the enrolment function. This thread has a queue for the received events. The event that is received first is also processed first. The thread can only execute one enrolment function at a time. It is possible to link more enrolment functions to one event. If this is the case, the enrolment functions are executed in the order that they were defined. It is also possible to link the same enrolment function to several events.

☞ It is the responsibility of the programmer to make sure that the execution of event handlers is performed faster than the application is receiving events. Otherwise it is possible that an application loses events when the queue overflows, which can occur if the application receives more events than it processes up to the time that the queue of received events is full.

Remarks

- CM service uses certain TCP port numbers. See for detailed information chapter 2.4 "LMS software" on page 16.
- When the function `CmInitialize` or `CmShutdown` is called, all links between track and carrier enrolment functions and events for all CM clients are deleted.
- When the function `CmDisconnect` is called, all links between all enrolment functions and events for the calling client are deleted.

4.7.42 Reading a variable

With the variable access functionality you can read actual system, track, carrier, and coil information. Each variable has a unique identifier. The client state must be `CM_CLIENT_CONNECTED` to read a variable.

System variables

To read a system variable the function `CmReadSystemVariable` is available. The input parameter for this function is the variable identifier, and returns the value of that variable. The following system variables are supported.

Variable identifier	Variable information
`CM_VAR_SYSTEM_STATE`	System state. The data type is `CM_SYSTEM_STATE`.
`CM_VAR_SYSTEM_CPA_ENABLE`	CPA state. The data type is `CM_BOOL`.

Tab. 4-11: System variables

Track variables

To read a track variable the function `CmReadTrackVariable` is available. The input parameters for this function are a track identifier and the variable identifier, and returns the value of that variable. The following track variables are supported.

Variable identifier	Variable information
CM_VAR_TRACK_STATE	Track state. The data type is CM_TRACK_STATE.
CM_VAR_TRACK_BEGIN_POS	Track begin position [pu]. The data type is CM_LREAL.
CM_VAR_TRACK_END_POS	Track end position [pu]. The data type is CM_LREAL.
CM_VAR_TRACK_FIRST_COIL_INDEX	Index of the first coil. The data type is CM_INT.
CM_VAR_TRACK_LAST_COIL_INDEX	Index of the last coil. The data type is CM_INT.
CM_VAR_TRACK_BEGIN_ALTERNATE	Selected alternative at track begin. The data type is CM_INT.
CM_VAR_TRACK_END_ALTERNATE	Selected alternative at track end. The data type is CM_INT.
CM_VAR_TRACK_FEED_OVERRIDE_VALUE	Current feed override value. The data type is CM_LREAL.
CM_VAR_TRACK_FEED_OVERRIDE_CONSTANT	Boolean indicating whether feed override is constant. The data type is CM_BOOL.
CM_VAR_TRACK_FIRST_ERROR_COIL	Index of the first coil where an error has occurred. The data type is CM_INT.

Tab. 4-12: Track variables

Carrier variables

To read a carrier variable the function CmReadCarrierVariable is available. The input parameters for this function are a carrier identifier and the variable identifier, and returns the value of that variable. The following carrier variables are supported.

Variable identifier	Variable information
CM_VAR_CARRIER_STATE	Carrier state. The data type is CM_CARRIER_STATE.
CM_VAR_CARRIER_POSITION	Carrier position [pu]. The data type is CM_LREAL.
CM_VAR_CARRIER_VELOCITY	Carrier velocity [pu/s]. The data type is CM_LREAL.
CM_VAR_CARRIER_ACCELERATION	Carrier acceleration [pu/s^2]. The data type is CM_LREAL.
CM_VAR_CARRIER_JERK	Carrier jerk [pu/s^3]. The data type is CM_LREAL.
CM_VAR_CARRIER_SETPOINT_STOPPED_POSITION	Carrier setpoint position [pu]. The data type is CM_LREAL. This variable can only be read when the carrier state is CM_CARRIER_STOPPED.
CM_VAR_CARRIER_CONTROLLER_STATE	Carrier controller state. The data type is CM_CARRIER_CONTROLLER_STATE.
CM_VAR_CARRIER_TRACK_ID	Track ID on which the carrier is. The data type is CM_INT.
CM_VAR_CARRIER_CPA_TABLE_ID	CPA table ID to which the carrier is linked. The data type is CM_INT.
CM_VAR_CARRIER_POSITION_DETECTING_TRACK	Position of the carrier on the detecting track. The data type is CM_LREAL.
CM_VAR_CARRIER_DETECTING_TRACK_ID	The track identifier of the detecting track.

Tab. 4-13: Carrier variables

Coil variables

To read a coil variable the function `CmReadCoilVariable` is available. The input parameters for this function are a track identifier, coil index, and the variable identifier, and returns the value of that variable. The following coil variables are supported.

Variable identifier	Variable information
`CM_VAR_COIL_POSITION`	Coil position [pu]. The data type is `CM_LREAL`.
`CM_VAR_COIL_ERROR_CODE`	Coil error code. The data type is `NYCE_ERROR_CODE`.

`CM_VAR_COIL_LEFT_SENSOR_ERROR`
`CM_VAR_COIL_MIDDLE_SENSOR_ERROR`
`CM_VAR_COIL_RIGHT_SENSOR_ERROR`
	Error status of the left, middle, right sensor of the coil. The value `FALSE` indicates no error, `TRUE` indicates that an error is detected by the sensor.
	The data type is `CM_BOOL`.

`CM_VAR_COIL_LEFT_SENSOR_DISABLED`
`CM_VAR_COIL_MIDDLE_SENSOR_DISABLED`
`CM_VAR_COIL_RIGHT_SENSOR_DISABLED`
	State of a sensor.
	The data type is `CM_BOOL`.

`CM_VAR_COIL_LEFT_SENSOR_CARRIER_DETECTED`
`CM_VAR_COIL_MIDDLE_SENSOR_CARRIER_DETECTED`
`CM_VAR_COIL_RIGHT_SENSOR_CARRIER_DETECTED`
	Carrier detected status of the left, middle, right sensor of the coil. The value `FALSE` indicates no carrier detected, `TRUE` indicates that a carrier is detected by the sensor.
	The data type is `CM_BOOL`.

`CM_VAR_COIL_LEFT_SENSOR_CARRIER_POS_VALID`
`CM_VAR_COIL_MIDDLE_SENSOR_CARRIER_POS_VALID`
`CM_VAR_COIL_RIGHT_SENSOR_CARRIER_POS_VALID`
	Carrier position valid status of the left, middle, right sensor of the coil. The value `FALSE` indicates no valid position, `TRUE` indicates that a valid position is generated by the sensor.
	The data type is `CM_BOOL`.

`CM_VAR_COIL_LEFT_SENSOR_CARRIER_MOVING`
`CM_VAR_COIL_MIDDLE_SENSOR_CARRIER_MOVING`
`CM_VAR_COIL_RIGHT_SENSOR_CARRIER_MOVING`
	Carrier moving status of the left, middle, right sensor of the coil. The value `FALSE` indicates carrier at standstill or no carrier detected by the sensor, `TRUE` indicates that a moving carrier is detected by the sensor.
	The data type is `CM_BOOL`.

`CM_VAR_COIL_LEFT_SENSOR_CARRIER_POSITION`
`CM_VAR_COIL_MIDDLE_SENSOR_CARRIER_POSITION`
`CM_VAR_COIL_RIGHT_SENSOR_CARRIER_POSITION`
	Carrier position [pu] of the left, middle, right sensor of the coil.
	The data type is `CM_LREAL`.

Tab. 4-14: Coil variables

4.7.43 Reading configuration parameters

Function

With the read configuration parameter functionality, an application can read the value of the configuration parameters which are configured during `CmInitialize`. The NYCe 4000 parameter functionality supports reading and writing. However, the LMS parameter functionality only supports reading. Changing LMS configuration parameters can be done with the LMS Commissioning tool, thus safe-guarding parameter consistency.

The parameters of the LMS configuration are grouped in the following configuration sections.

- System – the parameters belonging to the system
- Track – the parameters belonging to a track
- Physical coil – the parameters belonging to a physical coil of the system

- Logical coil – the parameters belonging to a logical coil of a track
- Sensor – the parameters belonging to a sensor of a physical coil
- Bumper – the parameters belonging to a bumper of a track
- CollAvoid – the parameters belonging to a track or a process area
- Carrier – the parameters belonging to a carrier

A separate read function is available for each configuration section. The `CmReadXxxParameter` functions return the value of a configuration parameter as a double. The configuration parameter is specified by an identifier. The supported configuration parameter identifiers are defined in the header file cmparameters.h. The configuration section is identified by the enumeration `CM_PAR_SECTION_TYPE` in the header file cmtypes.h. Separate read functions, `CmReadXxxName`, are available for configuration parameters defined by a string. These functions return the value of a configuration parameter as a string.

The following read functions are available.

All mentioned read functions are only allowed when the system state is `CM_SYSTEM_INITIALIZED`.

CmReadSystemParameter With the function `CmReadSystemParameter` you can read a system configuration parameter. Function input argument is a system configuration parameter identifier. Function output argument is the value of the parameter.

CmReadTrackParameter With the function `CmReadTrackParameter` you can read a track configuration parameter. Function input arguments are a track identifier and a track configuration parameter identifier. Function output argument is the value of the parameter.

CmReadPhysicalCoilParameter With the function `CmReadPhysicalCoilParameter` you can read a physical coil configuration parameter. Function input arguments are a coil name and a physical coil configuration parameter identifier. Function output argument is the value of the parameter.

CmReadLogicalCoilParameter With the function `CmReadLogicalCoilParameter` you can read a logical coil configuration parameter. Function input arguments are a track identifier, a coil index, an alternative indication, and a logical coil configuration parameter identifier. Function output argument is the value of the parameter.

CmReadSensorParameter With the function `CmReadSensorParameter` you can read a sensor configuration parameter. Function input arguments are a coil name, a sensor identifier, and a sensor configuration parameter identifier. Function output argument is the value of the parameter.

CmReadLogicalCoilName With the function `CmReadLogicalCoilName` you can read a logical coil name. Function input arguments are a track identifier, a coil index, an alternative indication, and the number of characters allocated for the name. Function output argument is the name of the coil.

Explanation

Configuration parameter identifiers Each configuration parameter definition consists of the prefix "`CM_PAR_`", followed by a configuration section identification, and ends with the name of the parameter. All characters are written in uppercase.

No configuration parameters are defined for the configuration section "bumper". The value of bumper section configuration parameters can be read with the function `CmReadBumperPars`.

No configuration parameters are defined for the configuration section "carrier". The value of carrier section configuration parameters can be read with the function `CmReadCarrierPars`.

No configuration parameters are defined for the configuration section "CollAvoid". The value of process area configuration parameters can be read with the function `CmCollAvoidGetProcessAreaData`.

Parameter data types

Each parameter has a specific data type. The following table lists the data types used in CM, and the mapping data type, as defined by the configuration parameter identifier, see cmparameters.h. These definitions are used for proper data conversion.

CM_TYPE_ID	CM data type
CM_TYPE_BOOL	CM_BOOL
CM_TYPE_INT	CM_INT
CM_TYPE_UINT	CM_UINT
CM_TYPE_DINT	CM_DINT
CM_TYPE_UDINT	CM_UDINT
CM_TYPE_REAL	CM_REAL
CM_TYPE_LREAL	CM_LREAL
CM_TYPE_ENUM	Any enumeration

Tab. 4-15: CM data types and data type mapping to CM_TYPE_ID

4.7.44 Conversion functions

Conversion functions are available that convert parameter, variable, enumeration and event identifiers to a printable text string and vice versa, or check that a specified identifier is valid. The conversion functions all have an identical naming scheme. The first part of the function name is for parameters "`CmParId`", for variables "`CmVarId`", for enumeration types "`Cm<enum>`" (where `<enum>` is the name of the enumeration type), and for events "`CmEvent`". The last part of the function name represents the actual performed conversion.

Parameters and variables

The header file cm_par_conversion.h defines the functions that convert the `CM_PAR_ID` definition to a printable text string and vice versa, and a function to check that the specified parameter identifier is valid.

The header file cm_var_conversion.h defines functions that convert the `CM_VAR_ID` definition to a printable text string and vice versa, and a function to check that the specified variable identifier is valid.

The following conversions are available, where "`Xxx`" is either "`Par`" or "`Var`".

- `CmXxxIdToString` - identifier to exact string representation.
- `CmXxxIdToUserString` - identifier to human readable string.
- `CmXxxIdToDescription` - identifier to descriptive string.
- `CmXxxIdFromString` - specified string to value.
- `CmXxxIdIsValid` - check validity of specified value of the identifier.
- `CmXxxIdNrOfIds` - number of defined CM parameters/variables.
- `CmXxxIdByIndex` - retrieve the n-th CM parameter/variable.

- `CmXxxIdGetUnit` - retrieve the unit of the CM parameter/variable.
- `CmXxxIdGetDataType` - retrieve the data type of the CM parameter/ variable.
- `CmXxxIdGetCategory` - retrieve the configuration section of the CM parameter/variable.

Enumerations

The header file cmenums.h defines the functions that convert the enumeration definition to a printable text string and vice versa, and a function to check that the specified enumeration identifier is valid. The following conversions are available, where "`Xxx`" stands for the CM enumeration type.

- `XxxToString` - value of enum to its exact string representation.
- `XxxToUserString` - value of enum to a human-readable string.
- `XxxToDescription` - value of enum to a descriptive string.
- `XxxFromString` - string to value of enum.
- `XxxIsValid` - check validity of specified value of enum.

Events

The header file cm_event_conversion.h defines the functions that convert the `CM_EVENT` definition to a printable text string and vice versa, and a function to check that the specified event is valid. The following conversions are available.

- `CmEventToString` - name of the CM event.
- `CmEventToDescription` - CM event to descriptive string.
- `CmEventFromString` - specified string to CM event value.
- `CmEventIsValid` - check validity of specified CM event.
- `CmEventNrOfEvents` - number of defined CM events.
- `CmEventByIndex` - retrieve the n-th CM event.
- `CmEventGetCategory` - retrieve the category of the CM event.

4.7.45 Logging

All generated log data is always stored in shared memory. With the function `CmSetLogFunction` an application can additionally specify a user-defined log function. The prototype of the log function is `(*DEH_LOG_FUNC)`. The parameters of the user-defined log function specify the subsystem (log data source defined by `NYCE_SUBSYSTEM`), the log type (defined by `DEH_LOG_TYPE`), and the message. When `CmSetLogFunction` is called, the log function pointer in the DEH subsystem is set to the user-defined log function and an internal connection is set up to include the logging from CM service. If the connection cannot be set up the error `CM_ERR_LOG_SOCKET_SETUP_ERROR` is returned. CM service uses certain TCP port numbers. See for detailed information chapter 2.4 "LMS software" on page 16. With the functions `DehSetLoggingMask` and `DehSetSSLoggingMask` you set the logging mask and process identifier. Note that you can only get log data of the process that you own. Further, when you activate event logging (with the function `DehSetLogEventSetting`), you only receive events logged of your own process for which an event enrolment is registered with the function `XxxDefineEventEnrolment` or `CmDefineEventEnrolment`).

With the function `CmResetLogFunction` the log function pointer in the DEH subsystem is set to the default log function `DehDefaultLogFunction` and the internal log connection to CM service is closed. If the internal connection cannot be closed the warning `CM_WRN_LOG_SOCKET_CLOSE_FAILED` is returned. The function `CmResetLogFunction` has no parameters.

Logging from CM service is always stored in shared memory and logging is also included in case of a user-defined log function.

☞ A registered user-defined log function may affect LMS performance.

4.7.46 Shutting down the LMS

Function

The LMS is shut down with the function `CmShutdown`. This function has no input or output. Any connected client can execute the function `CmShutdown`. The system state changes to `CM_SYSTEM_POWERUP`, and the state of all tracks in the system changes to `CM_TRACK_POWERUP` after `CmShutdown`.

Explanation

With this function the LMS software is shut down. All connected clients receive a notification that the system is shut down and as a result will delete all defined track and carrier event enrolments after the shut down operation is started. The entire operation of `CmShutdown` can take some time. All administrative actions are finished when control is returned to the application. As all defined track and carrier event enrolments are automatically deleted after shut down, the client does not need to execute any track or carrier `CmDeleteEventEnrolment` functions. However, if a client executes a track or carrier `CmDeleteEventEnrolment` while a shut down operation is in progress, the return status may be `NYCE_OK` or `CM_ERR_ENROLMENT_NOT_DEFINED`, depending on the progress of the system shut down operation. Defined system enrolments are not deleted. In addition, all claimed resources of the operating system are released. The function `CmShutdown` is only allowed when the client state is `CM_CLIENT_CONNECTED` and the system state is `CM_SYSTEM_INITIALIZED`.

☞ You can execute `CmShutdown` in any track state, except when homing is active. It is the responsibility of the application to make sure that no carriers are moving when `CmShutdown` is executed.

4.8 Tuning of a coil axis

You can use the NYCeTuner to tune the current loop and position velocity loop of a coil axis. However, you must do the proper initialization of the LMS track first. Do the following the steps to tune the current loop and position velocity loop of a coil axis.

Tune the current loop of a coil axis without a carrier

1. `CmInitialize`.
2. Move carriers away from the coil to be tuned (carriers outside the sensor range).
3. Start NYCeTuner and select the coil.
4. Click the "Open Loop" button in the NYCeTuner "Axis Tuning Control" section.

5. Select test signal properties, start test signal. Do the normal tuning actions, etc.

Tune the position velocity loop of a coil axis in open or closed loop with a carrier

1. `CmInitialize`.
2. `CmHomeTrack`.
3. `CmAssignCarrierIds`.
4. `CmActivateTrack` and `CmMoveToPosition` to move a carrier above the coil, or move manually a carrier above the coil and then `CmActivateTrack`.
5. Start NYCeTuner and select the coil.
6. Use NYCeTuner, but take care that the carrier remains in the control range.

   If the carrier leaves the control range, `CmDeactivateTrack` and restart from 4.

   If a fatal error occurs (for example sensor error), restart from 2.

5 Error handling and recovery

5.1 Introduction

The NYCe 4000 LMS software is able to detect and handle many errors synchronously and asynchronously. Synchronous errors are errors returned from called CM functions (type `NYCE_STATUS`) as return value or return parameter. Asynchronous errors are errors that can occur at any moment, caused by for example, an overvoltage condition, temperature too high detection, or hardware issues etc.

Check this chapter for the error code explanations and possible solutions. If the error code is not listed in this chapter, see the NYCe 4000 Software User Manual, chapter "Diagnostics support in NYCe 4000".

5.2 Synchronous errors and warnings

Synchronous errors are errors which are the result of the execution of a function of the CM API level. Errors caused by a CM API call can be detected in NYCe 4000 at any level of the host and node software. Some of the errors, detected directly at CM API level, are returned immediately as a return value or return parameter of type `NYCE_STATUS` by the CM API function.

Synchronous errors may originate from the following causes.

- Wrong use of the CM functions by the application, for example the specification of invalid parameters.
- Unexpected behavior of the LMS, caused by an asynchronous error.

All synchronous errors must be completely handled in the application. tab. 5-1 "Synchronous errors and warnings" on page 155 lists all synchronous errors and warnings with a brief explanation.

Error	Description
CML_ERR_INVALID_AXIS_TYPE	This error is reported if in an LMS node (the MCU executes the LMS firmware) a configuration file is loaded which is not for LMS axes, or a non-LMS drive module is installed in the node. ⇒ Check that the axis configuration file is for LMS axes. ⇒ Check that the installed drive modules in the node are NY4120/10 (the letters "LMS" are printed on the front of these drive modules).
CM_ERR_ADD_MOVING_CARRIER_ALREADY_PENDING	The function `CmAddMovingCarrier` has already been called to add a moving carrier to the same side of the track.
CM_ERR_INITIAL_POSITION_MISMATCH	The actual carrier position differs from the specified carrier position when the function `CmMoveAcrossTrackBounds` is called. Note that the specified begin position has to be the setpoint position.

Error	Description
CM_ERR_CARRIER_AGAINST_END_STOP	This error is reported when home mode 12 is started and a carrier moves against a mechanical end stop. ⇒ Position the carriers correctly on the track.
CM_ERR_CARRIER_ALREADY_BELONGS_TO_A_SYNC_GROUP	The specified carrier is already added to a sync group. ⇒ Make sure that a carrier is only added to one sync group.
CM_ERR_CARRIER_ALREADY_DETECTED	The function `CmCancelAddMovingCarrier` was called but the carrier was already detected so the cancel operation is not possible.
CM_ERR_CARRIER_BUMPER_INTERSECTION	Position of a carrier detected during the homing procedure is in the area of an active bumper. ⇒ Check for an active bumper defined in the software, but physically not present.
CM_ERR_CARRIER_ID_NOT_FOUND	This error can occur in combination with the function `CmMoveAdjacent` when there is no "predecessor" carrier present in the system, or in combination with the function `CmRemoveCarrier` when the specified carrier identifier is valid, but not present in the system. The function `CmReadCarrierPars` can also return this error. ⇒ Check the specified carrier identifier. The error is also returned when both members of the struct `CM_CARRIER_PARS` (`measurementScaleOffset` and `productOffset`) are set to 0.
CM_ERR_CARRIER_MOVED_OUTSIDE_AREA	This error is reported from `CmFreeArea` if the first sensor before the `beginPosition` or the first sensor after the `endPosition` detects a carrier. ⇒ Check and correct the specified `beginPosition`, `endPosition` and `freeAreaSize`.
CM_ERR_CARRIER_NOT_MARKED_FOR_ADDITION	This error is returned when a `CmSynchronize` command is given with the synchronization request `CM_REQ_CARRIER_MOVING_CARRIER_ADDED_AND_CONTROLLED` for an existing carrier which is not marked for addition.
CM_ERR_CARRIER_NOT_MARKED_FOR_REMOVAL	This error is returned when a `CmSynchronize` command is given with the synchronization request `CM_REQ_CARRIER_MOVING_CARRIER_REMOVED` for an existing carrier which is not marked for removal.
CM_ERR_CARRIER_NOT_ON_TRACK	This error is reported when a carrier is not detected on the origin track when a `CmMoveAcrossTrackBounds` is executed.

Error	Description
CM_ERR_CARRIER_POSITION_ERROR	This error is reported when the `CmActivateTrack` command detects a carrier that is not in the control range of a coil of the track. This error is also reported when a `CmMoveToTrack` command is given while the involved carrier is detected by sensors of more than one track. The error is also returned by `CmSynchronize` with the synchronization request `CM_REQ_CARRIER_MOVEMENT_STOPPED` if after the `CmMoveToTrack` movement is stopped, the involved carrier is not completely controlled by the coils of the second track or not only detected by sensors of the second track.
CM_ERR_CARRIER_POSITION_OUTSIDE_SENSOR_RANGE	A carrier is added to the track, but the detected position where the carrier is added is outside the range of the expected sensor. ⇒ Check that the position specified in the `CmAddCarrier` command is correct. If this error occurs after a homing procedure, the carriers are not in the expected positions. ⇒ Sensor not powered (check for cable break). ⇒ Coil not powered (check for cable break). ⇒ Check for mechanical obstructions. ⇒ Check for a bumper physically set, but not defined or set in the application. ⇒ Check for defective sensor. ⇒ Check coil and sensor positions with the NYCeLmsCommissioning tool.
CM_ERR_CARRIER_STATE_ERROR	The carrier is in the wrong state for the requested function.
CM_ERR_CLIENTS_CONNECTED	This error is returned when `CmDeactivateCmService` is called while there are still clients connected to CM. ⇒ Make sure that all CM clients are disconnected before you call `CmDeactivateCmService`.
CM_ERR_CLIENT_NOT_CONNECTED	This error is returned if • a CM function (not `CmConnect` or `CmDisconnect`) is called when the client state is `CM_CLIENT_DISCONNECTED`. ⇒ Make sure that the client is connected before you call any CM function beside the 2 mentioned CM functions. • any CM function is called and the CM socket server is not running. ⇒ Check that the CM socket server is running.

Error	Description
CM_ERR_CM_SERVICE_NOT_AVAILABLE	This error is returned in the following cases. • When a CM function is called, but the CM service is not active in the system. ⇒ Make sure that the service `N4kCmService` is running. You need administrator rights to start this service.
CM_ERR_COIL_ALREADY_ADDED	`CmAddFirstCoil` or `CmAddLastCoil` is called, but the first/last coil of the specified track is already added. ⇒ Check the track configuration.
CM_ERR_COIL_ALREADY_REMOVED	`CmRemoveFirstCoil` or `CmRemoveLastCoil` is called, but the first/last coil of the specified track is already removed. ⇒ Check the track configuration.
CM_ERR_DUPLICATE_CARRIER_ID	This error can occur when using RTLC functionality where the logical track configuration does not match the physical track configuration, and a carrier is manually pushed onto a neighboring track while a Cm movement was not completed. To recover from this error the track has to be homed.
CM_ERR_ECG_LOCK_MODE_NOT_SUPPORTED	This error is returned when the specified lock mode for camming carriers is not supported.
CM_ERR_ECG_SAMPLE_FREQUENCY_HIGHER_THAN_1KHZ	The CM camming functionality is only supported at a sample frequency of 1 kHz.
CM_ERR_ENROLMENT_DEFINED_FOR_TOO_MANY_CARRIERS	This error is returned when the function `CmDefineEventEnrolment` is called when already event enrolments are defined for `CM_MAX_NR_OF_CARRIERS` (64) carriers.
CM_ERR_ENROLMENT_NOT_DEFINED	This error is returned when the function `CmDeleteEventEnrolment` is called with an event identifier for which no event enrolment is defined.
CM_ERR_FILENAME_TOO_LONG	The specified file name has too many characters.
CM_ERR_FILE_CONTENTS_ERROR	The data in the specified file is not correct.
CM_ERR_FILE_OPEN_ERROR	The specified file could not be opened. Check if the file exists.
CM_ERR_HOME_MODE_10_FAILED	This error is reported if not all detected sensors have a valid position after the home operation with home mode 10. ⇒ Check that the preconditions for home mode 10 are met, see chapter 4.7.8 "Homing a track" on page 67, "Mode 10". ⇒ Check the specified bumper positions. ⇒ Check for mechanical obstructions.

Error	Description
CM_ERR_HOME_TRACK_EXECUTING	Called function is not allowed while a homing procedure is active. This error is also reported if the application that started a home operation is stopped before the home operation has finished. ⇒ Use `CmResetError` to recover from this situation.
CM_ERR_INACTIVE_SENSOR	Sensor is inactive, however, based on neighbor sensor position information the sensor is expected to be active. ⇒ Check the signal levels of the sensors. When this error occurs during a homing procedure, the positions of the sensors are not correct, or the position of the carrier is wrong. ⇒ Check coil and sensor positions with the NYCeLmsCommissioning tool.
CM_ERR_INVALID_BUMPER_ID	Specified bumper identifier on function call is invalid.
CM_ERR_INVALID_CARRIER_ID	Specified carrier identifier in function call is invalid. ⇒ Check that the carrier identifier has a valid value.
CM_ERR_INVALID_CARRIER_INDEX	Specified carrier index in function call is invalid. ⇒ Check that the carrier index has a valid value.
CM_ERR_INVALID_COIL_ALTERNATIVE	This error is returned if the specified alternative coil for the function `CmSelectFirstCoilAlternative` or `CmSelectLastCoilAlternative` is not correct.
CM_ERR_INVALID_COIL_INDEX	The specified coil index is not correct.
CM_ERR_INVALID_COIL_NAME	The specified coil name is not correct.
CM_ERR_INVALID_EVENT_ID	The specified event identifier is not correct.
CM_ERR_INVALID_MAX_POSITION	This error is returned by the function `CmCheckArea` when the carriers are out of the area maximum position.
CM_ERR_INVALID_MIN_POSITION	This error is returned by the function `CmCheckArea` when the carriers are out of the area minimum position.
CM_ERR_INVALID_PARAMETER	The specified parameter is not correct.
CM_ERR_INVALID_PAR_ID	The specified parameter identifier is not correct.
CM_ERR_INVALID_POSITION	The carriers are not in the expected positions, for example during a homing procedure, or an error occurred during a homing movement which requires the movement to be stopped. ⇒ Check the carrier positions after the homing procedure.

Error	Description
CM_ERR_INVALID_POSITIONS	This error is returned by the function `CmCheckArea` or `CmMoveAcrossTrackBounds` when the carriers are not in the expected positions.
CM_ERR_INVALID_PROCESS_AREA_ID	Invalid process area identifier specified. The functions `CmCollAvoidDefineProcessArea`, `CmCollAvoidDeleteProcessArea`, and `CmCollAvoidEnableProcessArea` may return this error.
CM_ERR_INVALID_SENSOR_ID	The specified sensor identifier is not correct.
CM_ERR_INVALID_STRING	An invalid string was supplied to a function to convert a string to an enumeration value.
CM_ERR_INVALID_SYNC_GROUP_ID	The specified sync group identifier is not correct.
CM_ERR_INVALID_TRACK_ID	Specified track identifier in function call is invalid.
CM_ERR_INVALID_VAR_ID	The specified variable identifier in a call to `CmReadXxxVariable` is not correct.
CM_ERR_LOG_SOCKET_SETUP_ERROR	This error is returned when the function `CmSetLogFunction` is called to set up a user-defined log function, and the creation of the log socket has failed.
CM_ERR_MASTER_AXIS_TYPE_NOT_SUPPORTED	This error is returned when the specified axis type of the master axis for camming carriers is not supported.
CM_ERR_MR_SENSOR_COMMUTATION_NOT_DETERMINED	This error is returned when the commutation angle cannot be determined during a homing procedure, when a carrier is moving in velocity mode (CmStartVelocityMode or CmStartVelocityModeCoils), moving against an end stop (CmMoveCarriersAgainstEndstop), or a carrier is moved to a specified area (CmFreeArea). ⇒ Start the movement again.
CM_ERR_MOVEMENT_STARTED	`CmCancelMoveAcrossTrackBounds` returns this error when the cancel command is given for a carrier that already has started the movement.
CM_ERR_MOVE_TO_ENDSTOP_EXECUTING	This error is returned when the function `CmHomeTrack` or `CmMoveCarriersAgainstEndstop` is called while a movement to an end stop is already active.
CM_ERR_MOVE_TO_TRACK_EXECUTING	Called function is not allowed while a `CmMoveToTrack` procedure is active.
CM_ERR_NOT_END_OF_STROKE	The carriers are not in the expected positions after a homing procedure, or an error occurred during a homing movement which requires the movement to be stopped. ⇒ Check the carrier positions after the homing procedure.

Error	Description
CM_ERR_NOT_IMPLEMENTED	Requested functionality is not available in this software release. Home mode 2 and home mode 4 are not supported any longer. ⇒ Check that the application does not use home mode 2 or home mode 4. ⇒ Check for a linear track that the home mode is mode 0, mode 5 or mode 10. ⇒ Check for a modulo track that the home mode is mode 1.
CM_ERR_NOT_SUPPORTED	This error is returned when function is called which is not supported for the configuration. For example a home mode for a defined track type which is not compatible.
CM_ERR_NO_COIL_IN_AREA	This error is reported from `CmFreeArea` if no coil with left and right sensor is inside the area specified by `beginPosition` to `endPosition`. ⇒ Check and correct the specified `beginPosition` and `endPosition`.
CM_ERR_NO_COILS_IN_TRACK	This error is returned when `CmCancelAddMovingCarrier` was called but there are no coils in the specified track so the cancel operation is not possible.
CM_ERR_NO_COMMAND_INITIATED	`CmCancelMoveAcrossTrackBounds` returns this error when the cancel command is given for a carrier that has not been issued a `CmMoveAcrossTrackBounds` command.
CM_ERR_NO_FREE_COIL_BETWEEN_CARRIERS	This error is reported when home mode 12 is started and there is not at least one free coil between two carriers. ⇒ Position the carriers correctly on the track.
CM_ERR_NO_FREE_SENSORS_TO_MOVE	This error is reported when home mode 12 is started and there are not sufficient free sensors at the begin and end of the track. ⇒ Position the carriers correctly on the track.
CM_ERR_NO_FREE_SENSOR_OUTSIDE_AREA	This error is reported from `CmFreeArea` if no sensor is present before the `beginPosition`, or no sensor is present after the `endPosition`, or this sensor detects a carrier. ⇒ Check and correct the specified `beginPosition` or `endPosition`.
CM_ERR_NO_MOVING_CARRIER_IN_AREA	This error is reported from `CmFreeArea` if no sensor within the area specified by `beginPosition` to `endPosition` detects a moving carrier. ⇒ Check and correct the specified `beginPosition` or `endPosition`.

Error	Description
CM_ERR_NO_SENSOR_IN_AREA	This error is reported from `CmCheckArea` or `CmFreeArea` if the area between `beginPosition` and `beginPosition` +/- `freeAreaSize` (depending on `velocity` value) does not contain a sensor. ⇒ Check and correct the specified `beginPosition` and `freeAreaSize`.
CM_ERR_NR_OF_CARRIERS_INCONSISTENT	This error is reported when home mode 12 is started and the specified number of carriers does not match the detected number of carriers on the track. ⇒ Make sure that the specified number of carriers on the track matches the actual physical number of carriers on the track.
CM_ERR_NR_OF_CARRIERS_UNKNOWN	This error is returned when the function `CmGetCarriersOnTrack` is called when the track state is `CM_TRACK_POWERUP` or `CM_TRACK_IDLE`.
CM_ERR_OBSTRUCTED_BY_CARRIER	This error is reported when a `CmMoveToTrack` command is given and there are carrier(s) in between the carrier involved in the move to track operation and the end of the track where the carrier is heading to.
CM_ERR_ORPHAN_CARRIER	The carrier is no longer controlled by a coil. This condition can occur during the execution of `CmMoveToPosition` while the carrier is moving. During the execution of `CmMoveToPosition` while the carrier is moving, the software detects above which coils the carrier is positioned. The error is reported if the movement of the carrier is so fast that the carrier is already above other coils, or conversely, the host software updates the positions too slow. ⇒ Check that no other programs are running on the PC. ⇒ Move the carrier at a lower velocity.
CM_ERR_PARAMETER_ERROR	Parameters specified for a function are for example outside the valid value range.
CM_ERR_PARAMETER_OUT_OF_RANGE	When retrieving an enumeration value based on an index value, the specified index parameter is not in the valid range. For example the function `CmVarIdByIndex`.
CM_ERR_PRECONDITION_NOT_OK	Precondition(s) for the `CmHomeTrack` function are not correct to execute the homing movement.
CM_ERR_PRF_ACCELERATION_LIMIT_EXCEEDED	This error is returned by `CmMoveProfile` when the specified parameters exceed the maximum allowed track acceleration.
CM_ERR_PRF_ALL_VELOCITIES_ZERO	This error is returned by `CmMoveProfile` when all specified points contain a velocity setting of 0.

Error	Description
CM_ERR_PRF_INVALID_NR_OF_PROFILE_POINTS	This error is returned by `CmMoveProfile` when the number of points is 0, or higher than the maximum supported number of points.
CM_ERR_PRF_INVALID_POSITION	This error is returned by `CmMoveProfile` when one (or more) of the specified positions is outside the track boundaries.
CM_ERR_PRF_MOVEMENT_DIRECTION_NOT_CONSISTENT	This error is returned by `CmMoveProfile` when the specified parameters require movements in the positive and negative direction.
CM_ERR_PRF_VELOCITY_LIMIT_EXCEEDED	This error is returned by `CmMoveProfile` when the specified parameters exceed the maximum allowed track velocity.
CM_ERR_PROCESS_AREA_STATE_ERROR	`CmCollAvoidDefineProcessArea` returns this error when the specified process area identifier exists and the process area is enabled. `CmCollAvoidDeleteProcessArea` returns this error when the specified process area identifier exists, but the state of the process area is not `CM_COLL_AVOID_PROCESS_AREA_IDLE`.
CM_ERR_REQUEST_ABORTED	This error is returned by `CmSynchronize` when the synchronization request has been cancelled.
CM_ERR_REQUEST_TIMEOUT	This error is returned by `CmSynchronize` when the specified timeout on the synchronization request has elapsed.
CM_ERR_RPC_NOT_IMPLEMENTED	This error is returned when a CM function is called while CM service is active on an other device, and the CM function is not supported remotely.
CM_ERR_SENSOR_DETECTION_NOT_DISABLED	This error is returned by `CmAddCarrier` when the detection of the sensors in the area where the carrier will be added are not disabled. ⇒ Make sure that the sensors in the area where the carrier will be added are disabled before `CmAddCarrier` is called.
CM_ERR_SENSOR_STATUS_CRITERION	This error is returned by the `CmHomeTrack` function when a sensor detects a carrier but the position is not valid.
CM_ERR_SHUTDOWN_IN_PROGRESS	This error is returned to a CM client that calls a CM function, while an other CM client has called the function `CmShutdown`. ⇒ Make sure that all CM clients are "in sync", if multiple CM clients are used in the LMS application.
CM_ERR_SERVICE_COMMANDSOCKET_ALREADY_RUNNING	This error is returned when during setting up the connection between the application and the CM service it is detected that the TCP/IP socket server is already running.

Error	Description
CM_ERR_SERVICE_COMMANDSOCKET_NOT_RUNNING	This error is returned when during closing the connection between the application and the CM service it is detected that the TCP/IP socket server is already stopped.
CM_ERR_SERVICE_COMMUNICATION_ERROR	This error is returned when an error occurred in the communication between the application and CM service.
CM_ERR_SYSTEM_ERROR	This error may occur on the PC. ⇒ Check messages in NYCeLogger. ⇒ Possible cause: internal software error. Contact Bosch Rexroth.
CM_ERR_SYSTEM_STATE_ERROR	A CM function is called while the system state is such that the called CM function is not allowed.
CM_ERR_TIMEOUT	A requested action takes too much time. When this error occurs during a homing procedure, the homing operation took too much time. ⇒ Check for possible friction on the track. ⇒ Positions of coils or sensors specified in the configuration file are not correct, check coil and sensor positions with the NYCeLmsCommissioning tool.
CM_ERR_TOO_MANY_CARRIERS_IN_PROCESS_AREA	If a carrier is moved onto a track, where the "firstSafePosition" is within or beyond a process area which is occupied, this error is returned. The functions `CmCollAvoidAddMovingCarrier`, `CmCollAvoidMoveBetweenTracks`, and `CmCollAvoidAddCarrierWithProfile` may return this error. Further, the function `CmCollAvoidEnableProcessArea` returns this error when more than one carrier is inside the process area when the process area is enabled.
CM_ERR_TOO_MANY_CARRIERS_ON_TRACK	More than the maximum number of carriers is detected on the track. ⇒ Check the number of carriers physically on the track.
CM_ERR_TOO_MANY_CARRIER_PARAMETERS	This error is returned when more than `CM_MAX_NR_OF_CARRIER_PARS` carriers are specified with the function `CmWriteCarrierPars`.
CM_ERR_TOO_MANY_CLIENTS	Maximum number of clients (15) connected to CM service reached. ⇒ Terminate a client, then connect the other client.
CM_ERR_TOO_MANY_COILS_IN_SYNC_GROUP	The maximum number of coils added to a sync group is reached.
CM_ERR_TOO_MANY_PROCESS_AREAS	`CmCollAvoidDefineProcessArea` returns this error when too many process areas are defined.
CM_ERR_TOO_MANY_SYNC_GROUPS	The maximum number of sync groups is defined.

Error	Description
CM_ERR_TRACK_INVALID_FEED_OVERRIDE	This error is reported when a `CmMoveToTrack` command is given and the feed override settings of the involved tracks is not identical, or equals 0.
CM_ERR_TRACK_STATE_ERROR	The track is in the wrong state for the requested function.
CM_ERR_UNABLE_TO_REMOVE_COIL	`CmRemoveFirstCoil` or `CmRemoveLastCoil` is called, but the track is not configured for Runtime Track Layout Changes. ⇒ Check the track configuration. This error is also generated if a carrier is above an RTLC coil and a non-RTLC coil and the RTLC coil is removed.
CM_ERR_UNKNOWN_ENUM_STRING	An attempt was made to convert an unknown enumeration value to string.
CM_ERR_WRONG_NR_OF_CARRIERS	This error is returned when the number of carrier identifiers to be assigned is lower than the actual number of carriers on the track.
CM_WRN_ALREADY_CONNECTED	This warning is returned when the function `CmConnect` is called while the client already has connected to CM service.
CM_WRN_ALREADY_DISCONNECTED	This warning is returned when the function `CmDisconnect` is called while the client already has disconnected from CM service.
CM_WRN_CARRIER_IDS_NOT_ASSIGNED	This warning is returned when the function `CmGetCarriersOnTrack` is called when the track state is `CM_TRACK_HOMED`.
CM_WRN_END_POSITION_NOT_REACHABLE	This warning is returned when the carrier end position is not reachable.
CM_WRN_ENROLMENT_ALREADY_DEFINED	This warning is returned when the specified combination of event ID and enrolment function is already defined when the function `CmDefineEventEnrolment` is called.
CM_WRN_LOG_SOCKET_CLOSE_FAILED	This warning is returned when the removal of the log socket failed after calling the function `CmResetLogFunction`.
CM_WRN_PARTIALLY_CONNECTED	This warning is returned when the number of nodes or coils is not as expected. A node or coil has been removed from the LMS.
CTR_ERR_LMS_CARRIER_ALREADY_DETECTED	This error is returned when during adding a moving carrier to the system, the operation is cancelled. ⇒ Do not use the function `CmCancelAddMovingCarrier` when the carrier is already added to the system with the function `CmAddMovingCarrier`.

Error	Description
CTR_ERR_LMS_CARRIER_LINKED_TO_CPA_TABLE	This error is reported if a carrier is linked to a CPA table when `CmDownloadCpaTable` or `CmLinkCpaTable` is called. ⇒ Check the application.
CTR_ERR_LMS_COIL_CONFIGURATION_DOES_NOT_EXIST	At startup of the NYCe 4000 Software LMS (`CmInitialize`) or when a coil is added (`CmAddFirstCoil` or `CmAddLastCoil`) an axis is detected for which no coil configuration is defined. This error should not be returned under normal conditions. ⇒ Contact Bosch Rexroth if the error persists.
CTR_ERR_LMS_CPA_NOT_DISABLED	This error is reported if CPA functionality is not disabled when `CmDownloadCpaTable` is called. ⇒ Check the application.
CTR_ERR_LMS_CPA_TABLE_NOT_AVAILABLE	This error is reported if the specified CPA table is not available when `CmLinkCpaTable` is called. ⇒ Check the application.
CTR_ERR_LMS_INVALID_CPA_TABLE_SIZE	This error is reported if the size of the CPA table is not correct when `CmDownloadCpaTable` is called. ⇒ Check the application.
CTR_ERR_LMS_TOO_MANY_COIL_CONFIGURATIONS	A new coil configuration is defined but there are already 14 coil configurations defined. The maximum number of coil configurations is limited to 14 per node. ⇒ Check the track configuration.
CTR_ERR_STATE_ERROR	When this error occurs during the initialization of LMS, one of the coils may have an asynchronous error in the LMS software. Check messages in NYCeLogger. Check the state and error code of that coil (start NYCeConfigurator or NYCeTuner and select the concerned coil). ⇒ Error input. When the digital input is configured as temperature sensor input - coil temperature error. - check if polarity of this input is correct (must be active low). After an MCU replacement, make sure that the NHI settings are reloaded from flash. The default setting of the digital input is active high. After replacement of the MCU, do "initialize from file" and then "save to flash". Then the correct settings are loaded from flash. Else: check that the device is connected to this input. ⇒ Sensor supply error.

Error	Description
CTR_WRN_LMS_CARRIER_NOT_BETTER_IN_RANGE	This warning is returned when a sensor detects 2 carriers, and the current carrier is further in the sensor range than the new carrier. ⇒ Define a better initial carrier position which will be used during the home procedure.
CTR_WRN_LMS_CARRIER_NOT_IN_SENSOR_RANGE	This warning is returned when a calculated exact carrier position is not within the sensor range. ⇒ Define a better initial carrier position which will be used during the home procedure.
CTR_WRN_LMS_CARRIER_NOT_LINKED_TO_CPA_TABLE	This warning is reported if no CPA table is linked to the specified carrier when `CmUnlinkCpaTable` is called. ⇒ Check the application.
CTR_WRN_LMS_COIL_CONFIGURATION_ALREADY_DEFINED	A new coil configuration is defined for the specified axis, but the configuration is already defined. This error should not be returned under normal conditions. ⇒ Contact Bosch Rexroth if the error persists.
CTR_WRN_LMS_SENSOR_NOT_DETECTED	This warning is returned when a carrier is added to a sensor and this sensor does not detect a carrier. ⇒ Define a better initial carrier position which will be used during the home procedure.
EDH_ERR_GROUP_ALREADY_DEFINED	This error is reported when you start the application and the application was closed a previous time without a call of `CmShutdown`. ⇒ Close all LMS programs. Restart the PC and power cycle the nodes.
NCS_ERR_NET_SERVICE_NOT_AVAILABLE	The NetService is not started or started a little later (by the Operating System on the PC) than the application. ⇒ Check that the NetService is started. ⇒ Delay the startup of the application or check that the NetService is running before the application is started.
RPC_ERR_COMMUNICATION_ERROR	This error is returned when a CM function is called while CM service is active on an other device, and the communication mechanism fails.

Tab. 5-1: Synchronous errors and warnings

5.3 Asynchronous errors

In addition to the synchronous errors which are detected immediately at the function call, asynchronous errors are detected at the firmware level.

The detection of asynchronous errors is done via the standard NYCe 4000 mechanism for error configuration, detection and handling (see NYCe 4000 Software User Manual).

The asynchronous errors are usually the result of an unexpected failure in the system. tab. 5-2 "Asynchronous errors" on page 168 lists the specific asynchronous errors of the NYCe 4000 LMS software.

Check this chapter for the error code explanations and possible solutions. If the error code is not listed in this chapter, see the NYCe 4000 Software User Manual, chapter "Diagnostics support in NYCe 4000".

Error	Description
CM_ERR_ASYNC_ERROR_OCCURRED	This error is returned when an asynchronous error occurs during the execution of a function. The error can be returned from the functions `CmSynchronize`, `CmHomeTrack`, `CmFreeArea`, and `CmMoveCarriersAgainstEndstop`. The error that occurred can be retrieved from the logging or via the function `CmReadTrackVariable` with parameter `CM_VAR_TRACK_FIRST_ERROR_COIL` which returns an identifier of the coil that detected the error. With the identifier you can use the function `CmReadCoilVariable` to retrieve the error code.
CM_ERR_CARRIERS_NOT_MOVING	This error is returned when no carriers are moving during the home procedure.
SAC_AX_ERR_LMS_LEFT_SENSOR_INACTIVE SAC_AX_ERR_LMS_MIDDLE_SENSOR_INACTIVE SAC_AX_ERR_LMS_RIGHT_SENSOR_INACTIVE	Sensor signal amplitude of the left, middle, or right sensor on the coil too low. This error can only occur after a sensor previously detected a carrier, but the signal level drops while according to the magnet plate length the sensor still "expects to see" a carrier. ⇒ Check the sensor signals with NYCeScope (too low or not present). ⇒ If the measured amplitude is lower than the settings of `Sine amplitude` or `Cosine amplitude` in the NYCeLmsCommissioning tool, the signals have become weaker than during commissioning. ⇒ Check distance between coil and magnet plate. ⇒ Check hardware, possible cabling problem. ⇒ Check for a problem with the carrier. ⇒ Check magnet plate is OK (check the magnetic strength and polarity). When the error is reported during the first move command after homing to an end stop, the physical end stop or bumper position may not match with the position specified in the XML configuration file. ⇒ Check the positions with the NYCeLmsCommissioning tool.

Error	Description
SAC_AX_ERR_LMS_LEFT_SENSOR_ACTIVE SAC_AX_ERR_LMS_MIDDLE_SENSOR_ACTIVE SAC_AX_ERR_LMS_RIGHT_SENSOR_ACTIVE	These errors are reported when the signals of the sensor pair (sum of squared signals) are unexpectedly high, which means that the sensor is detected when it is not expected. This is the case whenever a carrier moves onto a track from an external track, without adding the carrier. Other possible cases: The sensor signals are of bad quality. ⇒ Check correct positioning of the sensors. ⇒ Check that the air gap is correct. Configuration is not properly defined. ⇒ Check correct sensor orientation. ⇒ Check the axis resolution. ⇒ Check correctly defined detection criteria. Magnet plate of the carrier installed in the wrong direction. ⇒ Check for correctly mounted magnet plate.
SAC_AX_ERR_LMS_SETPOINT_ERROR	This error is reported when a carrier moves into the control range of a coil, but the carrier is not expected to come within the control range of that coil (the move command was not to the position range of that coil). ⇒ Prevent overshoot of the carrier so that the next coil will not try to control the carrier without having a setpoint. ⇒ Prevent that 2 carriers are above the same coil and one carrier will receive a move command.
SAC_AX_ERR_LMS_CARRIER_SWITCH_ERROR	Like `SAC_AX_ERR_LMS_SETPOINT_ERROR`, but with this error the coil did not receive a setpoint from a previous carrier. When this error occurs, it will be typically at the first move command after `CmActivateTrack`. ⇒ Prevent overshoot of the carrier so that the next coil will not try to control the carrier without having a setpoint. ⇒ Prevent that 2 carriers are above the same coil and one carrier will receive a move command.
SAC_AX_ERR_LMS_COMMUTATION_ANGLE_ERROR	The software is unable to determine which sensor to use for commutation.
SAC_AX_ERR_LMS_GAP_ERROR	This error is detected when a `CmMoveAdjacent` is initiated, but it turns out to be impossible to close the gap between the carrier and its predecessor at the appropriate position, taking into account the constraints on velocity and acceleration. The function `CmMoveAdjacent` performs calculations during the movement, in contrast to all other CM move commands. ⇒ Check timing, or redefine the velocity and acceleration constraints.

Error	Description
SAC_AX_ERR_LMS_SENSOR_SUPPLY_ERROR	The positive and negative supply voltage of the sensors can be read for each drive module via the AnIn1 input of the drive module. If either voltage drops below or rises above a defined level in absolute value (`SAC_PAR_LMS_SENSOR_SUPPLY_VALUE` +/- 10%), the error is generated. The positive and negative supply voltages can be read and traced by the SAC variables `SAC_VAR_LMS_SENSOR_SUPPLY_POS_VALUE` and `SAC_VAR_LMS_SENSOR_SUPPLY_NEG_VALUE`. You must configure for each coil that the sensor supply voltages are read back via the parameter `SAC_PAR_LMS_SENSOR_SUPPLY_SLOT_ID`. If the parameter is set to `NO_SLOT`, the sensor supply voltages are not checked. If the `slotID` is valid, the parameter `SAC_PAR_LMS_SENSOR_SUPPLY_ANIN_ID` indicates the analog input, which must be "1" (that is, AnIn1, analog input 1) to read the sensor supply voltages.
SAC_AX_ERR_MOVE_TO_TRACK_TIMEOUT	This error is generated when the timeout is detected during the `CmMoveToTrack` command execution.

Tab. 5-2: Asynchronous errors

5.4 Error handling at coil level

When the motion controller detects an error condition for a coil, the motion controller automatically and immediately executes (without any intervention of the user application possible) the defined error handler. For every asynchronous error a default error handler and a minimum error handler is defined. See the NYCe 4000 Software User Manual for a detailed description of error and safety handling. The following error handlers are defined, ordered in increasing severity.

Severity	Error Handler	Description
0	SAC_ERRH_NONE	No handling
1	SAC_ERRH_WARNING	A warning is signaled
2	SAC_ERRH_SMOOTH_STOP	Smooth Stop is executed
3	SAC_ERRH_QUICK_STOP	Quick Stop is executed
4	SAC_ERRH_FULL_STOP	Full Stop is executed
5	SAC_ERRH_QSTOP_OPEN_LOOP	Quick Stop is executed and the controller switches to open loop
6	SAC_ERRH_OPEN_LOOP	Controller switches to open loop and velocity control with target velocity 0 is executed
7	SAC_ERRH_DISABLE_DRIVE	Controller and drive are disabled
8	SAC_ERRH_POS_CORRUPTED	Controller and drive are disabled, and position is lost (corrupted)
9	SAC_ERRH_COMMUNICATION_LOST	Controller and drive are disabled, and position is lost (corrupted). Recovery only possible by execution of SacShutdown

Tab. 5-3: NYCe 4000 Error Handler descriptions

Error handler considerations

If a stop profile is executed in an error condition, the feed override setting is overruled and a feed override value of 1.0 is used.

It is not desirable to switch a coil directly to open loop, because the carrier controlled by that coil can move uncontrolled and a following carrier will not be controlled anymore by the coil for which the error was detected. So, the handler "Open loop" (handler 6) is not recommended for coils. Only if it is necessary to disable the drive or the measurement system is corrupted, this open loop behavior cannot be avoided.

An error detected by a coil that results in a smooth stop, quick stop or full stop (handlers 2, 3, 4 and 5) or disable drive (handlers 7 and 8) causes all carriers on the track to stop. Of course, a coil with drive disabled cannot stop the carriers, but all other coils can.

If an error is detected by a coil, an appropriate so-called peer error with associated error handler is executed for all other coils on the track.

Special error conditions

If an error occurs on a track with carriers that move with virtually no friction, for example carriers with magnetic bearings, it must be possible to brake the carriers in as many possible error conditions. Normally, in error conditions the LMS stops the carrier movement via an error handler. However, if the error severity is higher than quick stop open loop, the LMS no longer controls the carrier movement. In this case, low-friction carriers will continue to move. For the following special error conditions the LMS attempts to brake the carriers using velocity control with target velocity 0.

- defective LMS sensor
- LMS sensor signals too low
- LMS sensor power supply failure
- unexpected carrier entry on a track
- broken magnet plate
- detection magnetic scale failure

If the `SAC_ERRH_OPEN_LOOP` error handler is defined for the errors

- `SAC_AX_ERR_LMS_LEFT_SENSOR_ACTIVE`
- `SAC_AX_ERR_LMS_MIDDLE_SENSOR_ACTIVE`
- `SAC_AX_ERR_LMS_RIGHT_SENSOR_ACTIVE`
- `SAC_AX_ERR_LMS_LEFT_SENSOR_INACTIVE`
- `SAC_AX_ERR_LMS_MIDDLE_SENSOR_INACTIVE`
- `SAC_AX_ERR_LMS_RIGHT_SENSOR_INACTIVE`
- `SAC_AX_ERR_LMS_SENSOR_SUPPLY_ERROR`

and the `PEER_OPEN_LOOP` error handler is set to `SAC_ERRH_OPEN_LOOP`, then, in case of the mentioned special error conditions, velocity control with target velocity 0 is executed on all coils of the track that detect a carrier.

The errors `SAC_AX_ERR_LMS_XXX_SENSOR_ACTIVE`, `SAC_AX_ERR_LMS_XXX_SENSOR_INACTIVE`, and `SAC_AX_ERR_LMS_SENSOR_SUPPLY_ERROR` cannot be configured for error handler `SAC_ERRH_DISABLE_DRIVE` (where `XXX` can be `LEFT`, `MIDDLE`, or `RIGHT`).

The default and minimum error handler for the mentioned errors are set to `SAC_ERRH_OPEN_LOOP`. The coils of the track go to the state `SAC_FREE` and

the track state changes to CM_TRACK_FATAL_ERROR, because the carrier position is lost. The state of all carriers on the track is not changed. With the coil parameter controllerProportionalGainInError you can set the proportional gain used in the mentioned error conditions. The default value is equal to the value of the parameter controllerProportionalGain.

Notes

If an error occurs during homing of a track, a "stop homing" procedure is started. This "stop homing" procedure resets the coils and allows the carriers to move freely. However, if one of the special error conditions occurs during homing of a track, the "stop homing" procedure is not started for the coils involved in the SAC_ERRH_OPEN_LOOP error handler. Thus, these coils remain in velocity mode with target velocity 0.

Suppose you set for the sensor errors the error handler for a coil to SAC_ERRH_OPEN_LOOP, but for the neighbor coil you change the peer error handler SAC_ERRH_OPEN_LOOP to SAC_ERRH_QUICK_STOP. In that case, the neighbor coil will not execute the velocity mode with target velocity 0, but will execute the quick stop (if possible).

Three error handlers cause a track state change to CM_TRACK_FATAL_ERROR.

- SAC_ERRH_OPEN_LOOP
- SAC_ERRH_POS_CORRUPTED
- SAC_ERRH_COMMUNICATION_LOST

These 3 error handlers do **not** work for the following peer errors.

- SAC_AX_ERR_PEER_SMOOTH_STOP
- SAC_AX_ERR_PEER_QUICK_STOP
- SAC_AX_ERR_PEER_FULL_STOP
- SAC_AX_ERR_PEER_QSTOP_OPEN_LOOP

Although you can configure these error handlers for these peer errors, they will **not** work.

The default error handlers are listed in the NYCe 4000 Software User Manual, chapter "Single Axis Control", subchapter "Error and Safety Handling". You can change a default handler to an other error handler, but the new error handler must have a severity level equal to or higher than the listed minimum severity error handler, as listed in the same table. The selection of a different error handler can be done via the coil initialization XML files. For a peer error, except the warning, any handler defined with a severity level lower than smooth stop is overwritten by smooth stop at the initialization of the LMS software.

5.5 Error handling at CM level

After an error of severity level 1 (warning) occurred, the "warning occurred" event is generated. The coil state and the track state are not affected. This implies that such an error does not affect the behavior of the LMS motion, as far as the CM layer is concerned. However, the application may decide that some action is required for the motion part (for example CmStopCarrier, CmSetFeedOverride with feed override 0.0, or deactivation of the track). Possibly, the motion is not affected by the warning, but some action has to be taken in another part of the machine.

When an error handler of severity 2 thru 9 is executed by a coil, Carrier Management takes care of the state transitions of track and carriers in accordance with fig. 4-3 "State Transition Diagram for tracks in CM" on page 52 and fig. 4-4 "State Transition Diagram for carriers in CM" on page 55.

Depending on the track state, the error handlers of severity 2, 3, 4, 5 and 7 cause the following track state change and generate the "error occurred" event. See tab. 5-4 "Track state changes for error handlers 2, 3, 4, 5 and 7" on page 173.

Track state before error handler	Track state change by error handler	Event is generated
CM_TRACK_IDLE	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_VELOCITY_MODE	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_HOMED	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_INACTIVE	CM_TRACK_ERROR	yes
CM_TRACK_ACTIVE	CM_TRACK_ERROR	yes
CM_TRACK_ERROR	no track state change	yes
CM_TRACK_FATAL_ERROR	no track state change	yes
CM_TRACK_POWERUP	no track state change	no

Tab. 5-4: Track state changes for error handlers 2, 3, 4, 5 and 7

Depending on the track state, the error handlers of severity 6, 8, and 9 cause the following track state change and generate the "error occurred" event. See tab. 5-5 "Track state changes for error handlers 6, 8, and 9" on page 173.

Track state before error handler	Track state change by error handler	Event is generated
CM_TRACK_IDLE	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_VELOCITY_MODE	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_HOMED	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_INACTIVE	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_ACTIVE	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_ERROR	CM_TRACK_FATAL_ERROR	yes
CM_TRACK_FATAL_ERROR	no track state change	yes
CM_TRACK_POWERUP	no track state change	no

Tab. 5-5: Track state changes for error handlers 6, 8, and 9

You can also influence the state of the coils of the track by configuring the error handlers per error, in particular the "peer" errors, in the XML file of the coil. Further, you can enroll for the CM error event CM_EV_TRACK_ASYNC_ERROR_OCCURRED and the CM warning event CM_EV_TRACK_ASYNC_WARNING_OCCURRED, see chapter 4.7.41 "Event handling" on page 142.

The carrier state is only defined if the track state is CM_TRACK_ACTIVE. Therefore, the state of the carriers on a track controlled by a coil which has handled an error with any of the handlers 2 thru 9 is not defined.

5.6 Error recovery

5.6.1 Default recovery

If an error is detected by a coil of a track, you can use the function CmReadTrackVariable to retrieve the coil index of the coil that detected the error. The track is specified by the parameter trackId. The variable identifier is set to CM_VAR_TRACK_FIRST_ERROR_COIL. The parameter *pValue returns the coil index of the coil that detected the error. Note that the event data of the event CM_EV_TRACK_ASYNC_ERROR_OCCURRED also returns the coil index. Subsequently, you can retrieve the error that the coil detected with the function CmReadCoilVariable with the parameters trackId for the track identifier, coilIndex for the coil that detected the error, and the variable identifier CM_VAR_COIL_ERROR_CODE. The parameter *pValue returns the error code. The application can use the error code to execute required error handling.

If there are only warnings, the warnings can be cleared by CmClearWarning.

If there are more severe errors, the track must be reset. This means that all errors per coil are cleared, and the drives of all coils are disabled. In addition, the track state changes from CM_TRACK_ERROR to CM_TRACK_INACTIVE or from CM_FATAL_FATAL_ERROR to CM_TRACK_IDLE.

If the track state is CM_TRACK_INACTIVE, a CmActivateTrack command is sufficient to get the track operational again. When the track state is CM_TRACK_IDLE, the track must be homed again with the CmHomeTrack command and the carrier IDs have to be re-assigned with the CmAssignCarrierIds command before the track can be activated again.

5.6.2 Subsystem error recovery

If an error is reported from a subsystem, for example SAC_ERR_AXIS_REMOVED, the LMS must be restarted. The SAC errors are listed in the NYCe 4000 Software User Manual.

- If the application executes on another device than where CmService is activated, the application must do the following actions.
  1. CmDisconnect
  2. CmConnect
  3. CmShutdown
  4. CmInitialize
- If the application executes on the same device where CmService is activated, the application must do the following actions.
  1. CmShutdown
  2. CmInitialize

All other connected clients must execute CmDisconnect and CmConnect.

5.6.3 Specific recovery

Depending on the action(s) before an error occurred or specific conditions, a specific error recovery may be applicable.

CmMoveToTrack

If the function `CmSynchronize` with the synchronization request `CM_REQ_CARRIER_MOVEMENT_STOPPED` returns the error code `CM_ERR_CARRIER_POSITION_ERROR` see chapter 4.7.27 "Move a carrier from track to track" on page 99 for an applicable error recovery procedure.

CmAddMovingCarrier

If a carrier gets stuck during `CmAddMovingCarrier` at the first coil of the destination track, for example due to too much mechanical friction, the recovery depends on the condition whether the carrier is in the control range of the first coil, or the carrier is detected by the first sensor but not in the control range of the coil.

1. Carrier gets stuck in the control range of the coil.

   The recovery procedure is described in chapter 4.7.28 "Removing / Adding / Cancelling a moving carrier from / to an LMS" on page 105.

2. Carrier gets stuck at the first sensor but not in the control range of the first coil.

   In general, for this situation a synchronous error can be detected using `CmSynchronize` with the synchronization request `CM_REQ_CARRIER_MOVING_CARRIER_ADDED_AND_CONTROLLED`. If the carrier gets stuck, this synchronization request will run into a timeout.

   If an asynchronous error occurs, two cases can be distinguished.

   2.1 The system has an origin track.

   The origin track can go to an error state. As the carrier is not in the control range of the destination track, it is assumed that the carrier is still in the control range of the origin track, and that the carrier is stuck on the origin track. In this condition, the setpoint of the carrier on the origin track is increasing, while the actual position is not. With the detection of a "maximum following error exceeded", this can be solved. The origin track will go to an error state. As the carrier is still in the control range of the last coil of the origin track, you can use the recovery procedure described in chapter 4.7.28 "Removing / Adding / Cancelling a moving carrier from / to an LMS" on page 105. It is advised that in a system with much friction (and possibly coils at a slightly larger distance between them), the detection of a "maximum following error exceeded", or an other error detection should be enabled.

   2.2 The system uses an external transport system, such as a conveyor belt.

   In this case, no error can be detected by the LMS. For the LMS, it is unknown why the carrier does not move. The "origin track" or conveyor belt may have stopped moving. An automatic recovery is not possible without manual intervention, because the LMS is not able to move the carrier. Suggested actions are the following.

   - Reset the track.
   - Push the carrier manually.
   - Activate the track again, unless the manual action causes a fatal error. In that case, the track must be homed again.

   Basically, the system has a mechanical issue. The conveyor belt does not have enough friction to push the carrier far enough

onto the destination track, while the track itself has too much friction and possibly other disturbance forces such as cogging. Due to these conditions, the problem of a stuck carrier will occur frequently. Improvements of the system to resolve this problem can be one or more of the following.

- Move the coil closer to the edge.
- Reduce friction of the LMS system.
- Increase friction of the conveyor belt.

Notes

rexroth
A Bosch Company

Bosch Rexroth AG
Bgm.-Dr.-Nebel-Str. 2
97816 Lohr a.Main
Germany
Tel. +49 9352 18 0
Fax +49 9352 18 8400
www.boschrexroth.com/electrics

R911425761

DOK-NY4000-LMS*USR*V54-RE02-EN-E