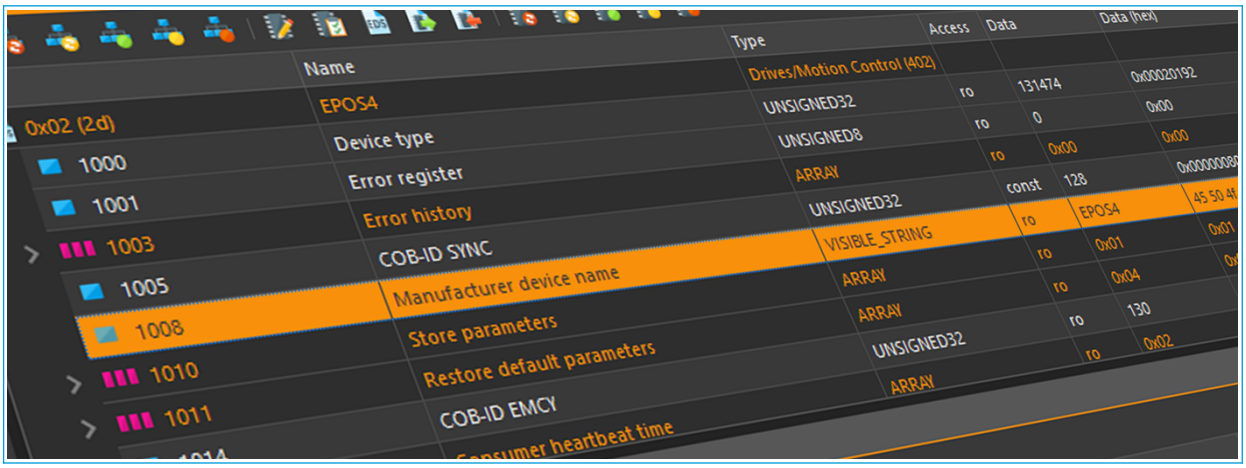
CETONI

Nemesys Syringe Pump M / S

Firmware Specification



ORIGINAL MANUAL - FEBRUARY 2024

CETONI GmbH
Wiesenring 6
07554 Korbussen
Germany

T +49 (0) 36602 338-0
F +49 (0) 36602 338-11
E info@cetoni.de

www.cetoni.de

1 Summaries and directories

1.1 Table of contents

1.2 Change history

REVISION	CHANGE
21.04.2021	Creation of document
15.02.2024	Fixed Object Index of Unit Velocity Object 0x60A9 Updated unit conversion section with right unit objects

2 About this Document

2.1 Intended Purpose

The purpose of the present document is to familiarize you with the described equipment and the tasks on safe and adequate installation and/or commissioning. Observing the described instructions in this document will help you:

- to avoid dangerous situations,
- to keep installation and/or commissioning time at a minimum and
- to increase reliability and service life of the described equipment.

Use for other and/or additional purposes is not permitted. cetoni, the manufacturer of the equipment described, does not assume any liability for loss or damage that may arise from any other and/or additional use than the intended purpose.

2.2 Target Audience

This document is meant for trained and skilled personnel working with the equipment described. It conveys information on how to understand and fulfill the respective work and duties. This document is a reference book. It does require particular knowledge and expertise specific to the equipment described.

2.3 Symbols and Signal Words Used

The following symbols are used in this manual and are designed to aid your navigation through this document:

HINT. Describes practical tips and useful information to facilitate the handling of the software.

IMPORTANT. Describes important information and other especially useful notes, in which no dangerous or damaging situations can arise.

ATTENTION. Indicates a potentially damaging situation. Failure to avoid this situation may result in damage to the product or anything nearby.

CAUTION. Describes a situation that may be dangerous. If this aspect is not avoided, light or minor injuries as well as damage to property could result.

3 System Overview

3.1 General Device Architecture

The device implements a CANopen slave device. CANopen is the internationally standardized (EN 50325-4) higher-layer protocol for embedded control system. The set of CANopen specification comprises the application layer and communication profile as well as application, device, and interface profiles. CANopen provides very flexible configuration capabilities. These specifications are developed and maintained by CiA members.

The communication interface of the device follows the CiA CANopen specifications as follows:

- *CiA 301* – Application Layer and Communication Profile
- *CiA 306* – Electronic Data Sheet Specification
- *CiA 303-2* – Representation of SI units and prefixes
- *CiA 303-3* – Indicator Specification

A CANopen device can be logically structured in three parts.

One part provides the communication interface (CAN, RS232) and another part provides the device's application, which controls e.g. the Input/Output (I/O) lines of the device in case of an I/O module.

The interface between the application and the CAN-interface is implemented in the object dictionary. The object dictionary is unique for any CANopen device. It is comparable to a parameter list and offers the access to the supported configuration- and process data.

The following section explains the basic concepts related to the CANopen protocol application layer. This document is intended as a basic overview only, and users are encouraged to review the CiA DS 301 specification for more information.

3.2 Object Dictionary

The most significant part of any CANopen device is the Object Dictionary. It is essentially a grouping of objects accessible via the network (via CAN or RS232) in an ordered, predefined fashion. The object dictionary is essentially a table, that stores configuration and process data. The figure below shows an example of an object dictionary. Each object within the dictionary is addressed using a 16-bit index 1 and an 8-bit subindex 2.

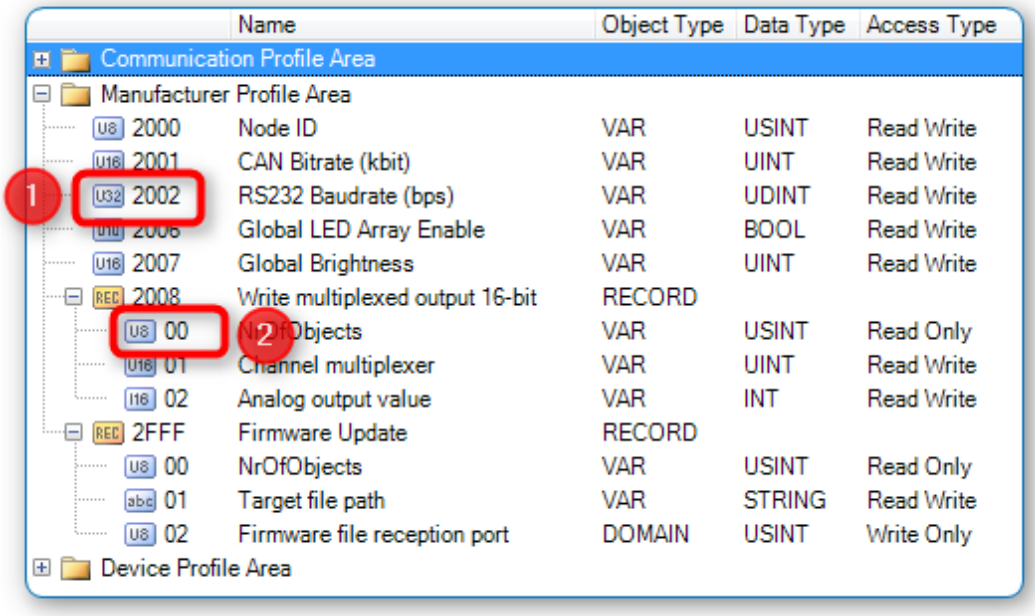


Figure 1: Object dictionary example

The 16-bit index 1 is used to address all entries within the Object Dictionary. In case of a simple variable, it references the value of this variable directly. In case of records and arrays however, the index addresses the entire data structure. The subindex 2 permits individual elements of a data structure to be accessed via the network.

- For single Object Dictionary entries (such as UNSIGNED8, BOOLEAN, INTEGER32, etc.), the subindex value is always zero.
- For complex Object Dictionary entries (such as arrays or records with multiple data fields), the subindex references fields within a data structure pointed to by the main index. This allows for up to 255 sub-entries at each index. Each entry can be variable in type and length.

The overall layout of the standard Object Dictionary conforms to other industrial field bus concepts.

Index	Description
0000_h	Reserved
0001_h-$009F_h$	Data types (not supported)
$00A0_h$-$0FFF_h$	Reserved
1000_h-$1FFF_h$	Communication Profile Area (CiA 301)
2000_h-$5FFF_h$	Manufacturer-specific Profile Area
6000_h-$9FFF_h$	Standardized Device Area (e.g. CiA 401 – I/O Modules)
$A000_h$-$FFFF_h$	Reserved

Table 1: Object dictionary layout

Access to each object dictionary entry is possible via SDO transfer (CAN) or via RS232 protocol by simply providing the index and sub index of the object dictionary entry to access.

4 CAN Communication

4.1 Introduction

This chapter provides general information about CAN communication and CANopen application layer. The information is relevant only for devices that support CAN communication via CAN interface. If your device only supports serial communication via RS232 CANopen Serial Interface (CSI), you can skip this chapter.

HINT. An excellent and easy to understand introduction to CAN and CANopen is available here:

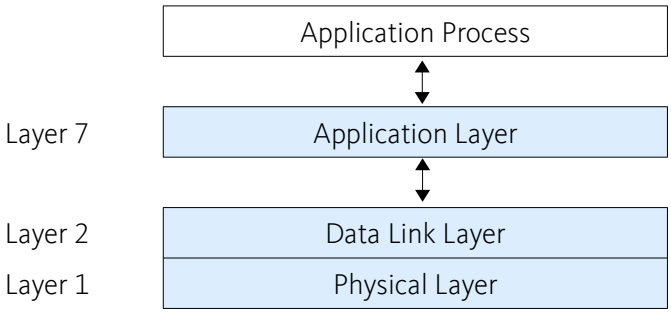
http://www.canopensolutions.com/english/about_canopen/CANopen-application-layer-basics.shtml

HINT. You can skip this chapter if your device does not support communication via CAN interface.

4.2 Reference Model of Data Communication

The Open Systems Interconnection Reference Model (OSI Reference Model) forms the basis for the description of communication systems today. The OSI model describes data communication systems in the form of a layer model, consisting of seven different layers, and assigns specific services to each layer. Simpler communication systems do not require all the functionalities of the OSI model. In general, only three layers (physical layer, data link layer and application layer) are relevant for data communication in the automation area.

The three layers shown in the figure implement the most important tasks of data communication in the fieldbus area.



4.3 CAN-Bus

4.3.1 CAN in the OSI reference model

The CAN protocol was specified by the company BOSCH. Regarding the OSI reference model, the CAN specification implements the data link layer completely and the physical layer partially. The physical signal representation is defined in the CAN protocol, while the form of the bus medium and the bus coupling was not specified.

4.3.2 Bus topology and data rate

The CAN bus uses a linear bus topology. The number of nodes is not limited by the CAN protocol, but depends on the performance of the driver chips used. Data rates up to 1 Mbit/s (network extension up to 40 m) and network extensions up to 1,000 m (at 80 Kbit/s) are possible. Two-wire lines with differential levels as well as fibre optic cables are possible as transmission medium.

4.3.3 Message transfer

The message receiver is not addressed, but the CAN messages are identified by a unique identifier – the CAN ID. Message transmission is based on the producer-consumer principle. This means that a message sent by one CAN node (producer) can be received by all other CAN nodes (consumers). On the basis of the message identifier, a subscriber decides whether a message is relevant for him or not.

4.3.4 Bus access

The identifier of a CAN message determines its priority. The message with the lowest CAN ID has the highest priority. Each message ID may only be sent from one CAN node in the system to avoid collisions. If several CAN nodes start sending a message at the same time, a collision occurs. This conflict is resolved by giving the message with the highest priority (with the lowest ID) bus access.

If the message with the highest priority has been sent, bus arbitration starts again for the remaining messages until all messages have been sent. This ensures that messages are not destroyed or lost.

4.3.5 Length of the payload data

The maximum data length of a CAN message is limited to 8 bytes. This enables fully functional data transmission in very difficult electromagnetic environments and guarantees short latency times for bus access of high priority messages.

4.3.6 Structure of CAN Frames

The CAN specification distinguishes between two compatible message formats, the standard format with 11 bit identifier and the extended format with 29 bit identifier. CETONI devices only use messages with 11-bit identifiers. A CAN message in standard format is shown in figure below and consists of:

- The Data Frame begins with a dominant Start of Frame (**SOF**) bit for hard synchronization of all nodes.
- The SOF bit is followed by the Arbitration Field reflecting content and priority of the message.
- The next field – the Control Field – specifies mainly the number of bytes of data contained in the message.
- The Cyclic Redundancy Check (**CRC**) field is used to detect possible transmission errors. It consists of a 15-bit CRC sequence completed by the recessive CRC delimiter bit.

- During the Acknowledgment (**ACK**) field, the transmitting node sends out a recessive bit. Any node that has received an error-free frame acknowledges the correct reception of the frame by returning a dominant bit.
- The recessive bits of the End of Frame (**EOF**) terminate the Data Frame. Between two frames, a recessive 3-bit Intermission field must be present.

CETONI devices only use messages with 11-bit identifiers:

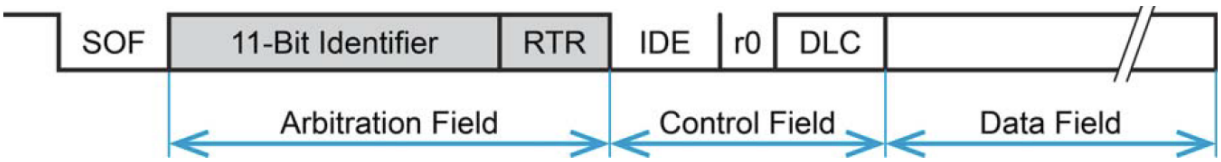


Figure 2: Standard frame format

- The Identifier's (COB-ID) length in the Standard Format is 11 bit.
- The Identifier is followed by the RTR (Remote Transmission Request) bit. In Data Frames, the RTR bit must be dominant, within a Remote Frame, the RTR bit must be recessive.
- The Base ID is followed by the IDE (Identifier Extension) bit transmitted dominant in the Standard Format (within the Control Field).
- The Control Field in Standard Format includes the Data Length Code (DLC), the IDE bit, which is transmitted dominant and the reserved bit r0, also transmitted dominant.
- The reserved bits must be sent dominant, but receivers accept dominant and recessive bits in all combinations.

4.3.7 Error Checking and Fault Confinement

The robustness of CAN may be attributed in part to its abundant error-checking procedures. The CAN protocol incorporates five methods of error checking: three at the message level and two at the bit level. If a message fails any one of these error detection methods, it is not accepted and an error frame is generated from the receiving node. This forces the transmitting node to resend the message until it is received correctly. However, if a faulty node hangs up a bus by continuously repeating an error, its transmit capability is removed by its controller after an error limit is reached. The following methods for error detection are used:

- Error checking at the message level is enforced by the CRC and the ACK slots. The 16-bit CRC

contains the checksum of the preceding application data for error detection with a 15-bit checksum and 1-bit delimiter. The ACK field is two bits long and consists of the acknowledge bit and an acknowledge delimiter bit.

- Also at the message level is a form check. This check looks for fields in the message which must always be recessive bits. If a dominant bit is detected, an error is generated. The bits checked are the SOF, EOF, ACK delimiter, and the CRC delimiter bits
- At the bit level, each bit transmitted is monitored by the transmitter of the message. If a data bit (not arbitration bit) is written onto the bus and its opposite is read, an error is generated. The only exceptions to this are with the message identifier field which is used for arbitration, and the acknowledge slot which requires a recessive bit to be overwritten by a dominant bit.
- The final method of error detection is with the bit-stuffing rule where after five consecutive bits of the same logic level, if the next bit is not a complement, an error is generated.

CAN uses the principle of error signalling. Detected errors are reported to the other network users by sending an error frame. This ensures that the communication with all functioning CAN nodes of a network is error-free and consistent and guarantees very short error response times.

4.4 CANopen Basics

4.4.1 Introduction

CANopen is a standardized application for distributed automation systems based on CAN (Controller Area Network) offering the following performance features:

- Transmission of time-critical process data according to the producer consumer principle
- Standardized device description (data, parameters, functions, programs) in the form of the so-called "object dictionary". Access to all "objects" of a device with standardized transmission protocol according to the client-server principle
- Standardized services for device monitoring (node guarding/heartbeat), error signalisation (emergency messages) and network coordination ("network management")
- Standardized system services for synchronous operations (synchronization message), central

time stamp message

- Standardized help functions for configuring baud rate and device identification number via the bus
- Standardized assignment pattern for message identifiers for simple system configurations in the form of the so-called "predefined connection set"

Subsequently described are the CANopen communication features most relevant to the CETONI CANopen devices. For more detailed information consult above mentioned CANopen documentation. The CANopen communication concept can be described similar to the ISO Open Systems Interconnection (OSI) Reference Model. CANopen represents a standardized application layer and communication profile

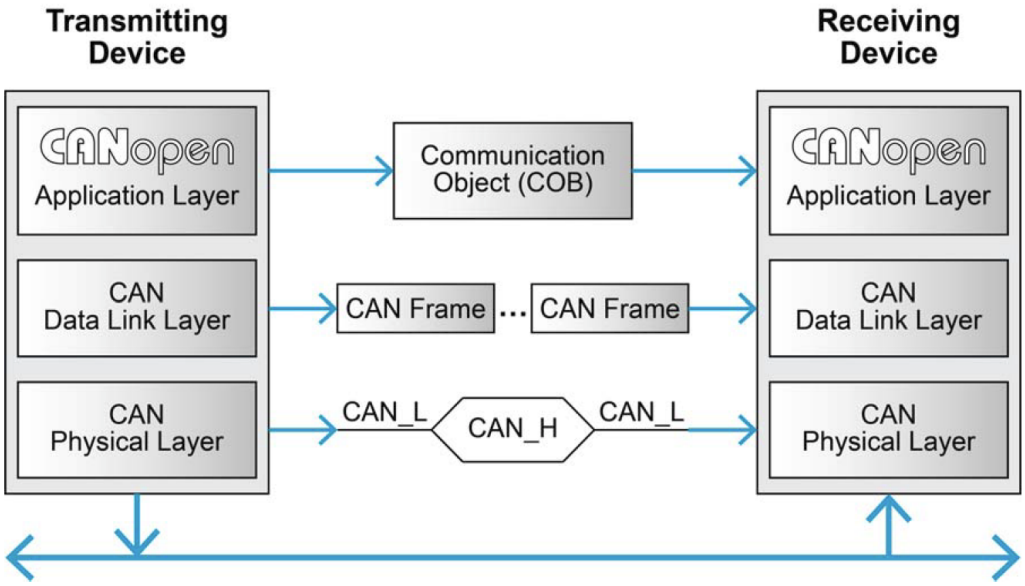


Figure 3: Protocol Layer Interactions

4.4.2 Physical Structure of the CAN Network

CANopen is a networking system based on the CAN serial bus. It assumes that the device's hardware features a CAN transceiver and a CAN controller as specified in ISO 11898. The physical medium is a differently driven 2-wire bus line with common return. The underlying CAN architecture defines the basic physical structure of the CANopen network. Therefore, a line (bus) topology is used. To avoid reflections of the signals, both ends of the network must be terminated. In addition, the maximum permissible branch line lengths for connection of the individual network nodes are to be observed.

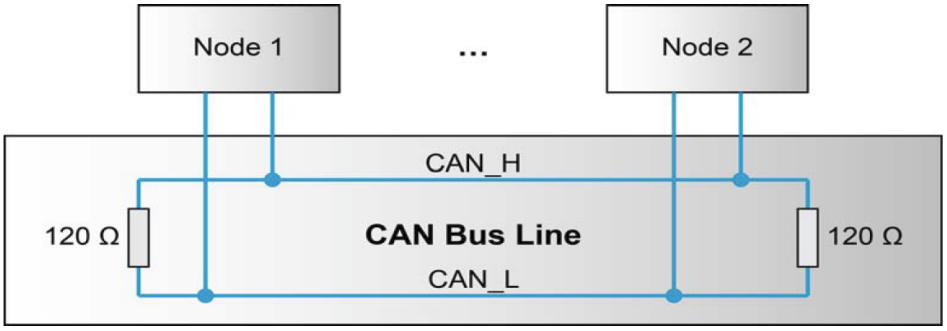


Figure 4: ISO 11898 basic network setup

The recommended permissible bit rates for a CANopen network are given in CiA 301: 10 kbps, 20 kbps, 50 kbps, 125 kbps, 250 kbps, 500 kbps, 800 kbps and 1000 kbps. In CiA 301 a recommendation for the configuration of the bit timing is also given.

Additionally, for CANopen, two additional conditions must be fulfilled:

- All nodes must be configured to the same bit rate and
- No node-ID may exist twice.

4.5 Communication Objects

CANopen uses communication objects for data transmission in the network. The following communication objects are specified by CANopen:

- Service data objects (**SDO**) are used to access the entries in the object dictionary.
- Process data objects (**PDO**) are used for fast transmission of process information
- Objects with special functions provide various system services (synchronization objects, time service objects, emergency objects)
- Network management objects (**NMT**) are necessary to start, stop and monitoring of network participants

In a CAN network, all objects refer to a specific message identifier. This means that each communication object has a unique CAN ID, and certain CAN message IDs are reserved for certain objects.

4.5.1 Service Data Objects – SDOs

With Service Data Objects (SDOs), the access to entries of a device Object Dictionary is provided. A SDO is mapped to two CAN Data Frames with different identifiers, because communication is confirmed. By means of a SDO, a peer-to-peer communication channel between two devices may be established. The owner of the accessed Object Dictionary is the server of the SDO. A device may support more than one SDO, one supported SDO is mandatory and the default case.

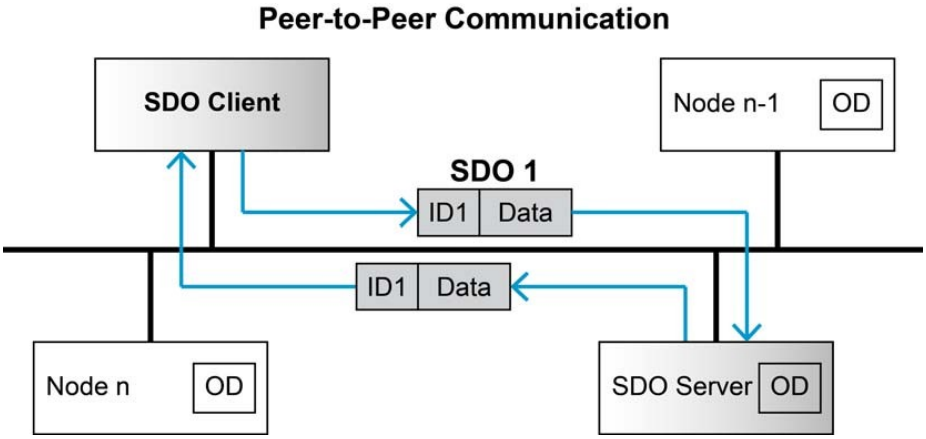


Abbildung 4.1: Service Data Object (SDO)

Read and write access to the CANopen Object Dictionary is performed by SDOs. The Client/Server Command Specifier contains the following information:

- download/upload
- request/response
- segmented/expedited transfer
- number of data bytes
- end indicator
- alternating toggle bit for each subsequent segment

SDOs are described by the communication parameter. The default Server SDO (S_SDO) is defined in the entry “1200h”. In a CANopen network, up to 256 SDO channels requiring two CAN identifiers each may be used.

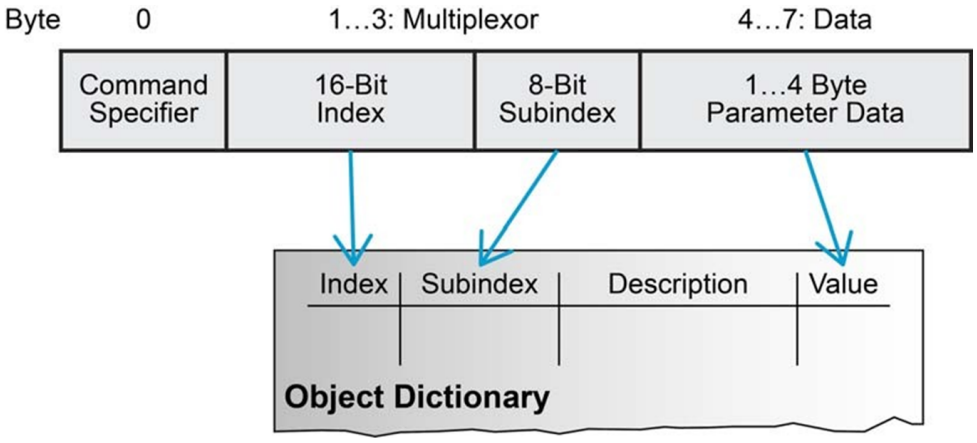


Abbildung 4.2: Object Dictionary Access

4.5.2 Process Data Objects – PDOs

Process data represents data that can be changing in time, such as the inputs (i.e. sensors) and outputs (i.e. motor drives) of the node controller. Process data is also stored in the object dictionary. However, since SDO communication only allows access to one object dictionary index at a time, there can be a lot of overhead for accessing continually changing data. In addition, the CANopen protocol has the requirement that a node must be able to send its own data, without needing to be polled by the CANopen master. Thus, a different method is used to transfer process data, using a communication method called Process Data Objects (**PDO**s).

PDO communication can be described by the producer/consumer model. Process data can be transmitted from one device (producer) to one another device (consumer) or to numerous other devices (broadcasting). PDOs are transmitted in a non-confirmed mode. The producer sends a Transmit PDO (TxPDO) with a specific identifier that corresponds to the identifier of the Receive PDO (RxPDO) of one or more consumers.

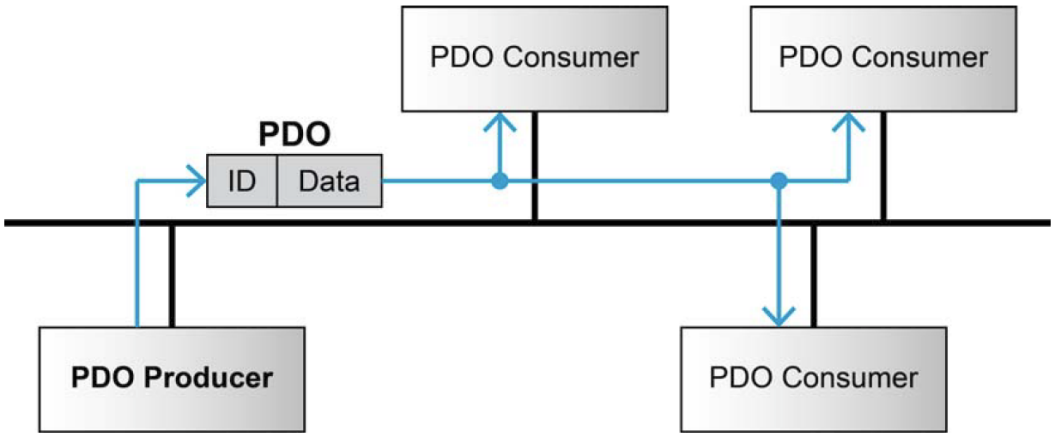


Figure 5: Process Data Object (PDO)

There are two types of PDOs: transfer PDOs (TPDOs) and receive PDOs (RPDOs). A TPDO is the data coming from the node (produced) and a RPDO is the data coming to the node (consumed). In addition, there are two types of parameters for a PDO: the configuration parameters and the mapping parameters. The section of the object dictionary reserved for PDO configuration and mapping information are indices 1400h-1BFFh.

IMPORTANT. PDO communication is not permitted in NMT state Pre-Operational. Switch to NMT Operational state to enable PDO transmission.

4.5.2.1 PDO CONFIGURATION PARAMETERS

The configuration parameters specify the COB-ID, the transmission type, inhibit time (TPDO only) and the event timer, which are explained in this section. There are different methods through which a PDO transfer can be initiated. These methods include event driven, time driven, individual polling and synchronized polling. The type of transmission is specified in the configuration parameters of the PDO. In event driven transmission, the PDO transfer is initiated when the process data in it changes.

- In time driven transmission, the PDO transfer occurs at a fixed time interval.
- In event-driven transmission a PDO transfer is triggered by the occurrence of an object-specific event or change of process data
- In individual polling, the PDO transfer is initiated using a mechanism called remote request, which is not commonly used.

- In synchronized polling, the PDO transfer is initiated using a SYNC signal. The sync signal is frequently used as a global timer. For example, if the CANopen master sends out a SYNC message, multiple nodes may be configured to see and respond to that SYNC. In this way, the master is able to get a "snapshot" of multiple process objects at the same time.

4.5.2.2 PDO MAPPING PARAMETERS

The mapping parameters specify which object dictionary values are sent by a single PDO message. For example, a single PDO message may contain data from object index 2001h, 2003h and 2005h.

Index	SubIndex	Example Value	Description
1801h	00h	100	Highest Subindex
	01h	00000181h	PDO COB ID
	02h	0	
	03h	1000	Inhibit Time
	04h	Unused	Unused
	05h	0 (disabled)	Event timer
1A01h	00h	3	Number of Entries
	01h	Index 2001h, Subindex 00h	Mapped OD item 1
	02h	Index 2003h, Subindex 00h	Mapped OD item 2
	03h	Index 2005h, Subindex 00h	Mapped OD item 3

Figure 6: TPDO 1 Communication Parameters (0x1801h) and Mapping Parameters (0x1A01h)

4.5.3 Sync Object

The SYNC producer provides the synchronization signal for the SYNC consumer.

As the SYNC consumers receive the signal, they will commence carrying out their synchronous tasks. In general, fixing of the transmission time of synchronous PDO messages coupled with the periodicity of the SYNC Object's transmission guarantees that sensors may arrange sampling of process variables and that actuators may apply their actuation in a coordinated manner. The identifier of the SYNC Object is available at index "1005h".

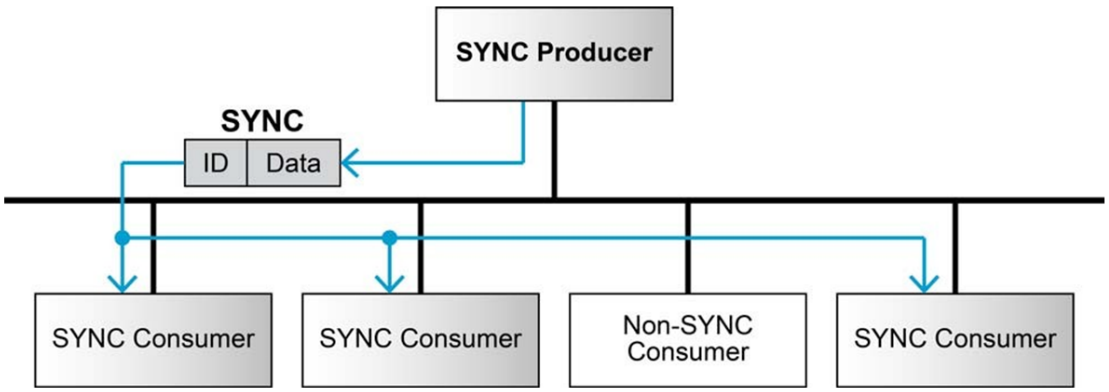


Figure 7

Figure 4.3: Synchronization Object (SYNC)

Synchronous transmission of a PDO means that the transmission is fixed in time with respect to the transmission of the SYNC Object. The synchronous PDO is transmitted within a given time window "synchronous window length" with respect to the SYNC transmission and, at the most, once for every period of the SYNC. The time period between SYNC objects is specified by the parameter "communication cycle period".

CANopen distinguishes the following transmission modes:

- synchronous transmission
- asynchronous transmission

Synchronous PDOs are transmitted within the synchronous window after the SYNC object. The priority of synchronous PDOs is higher than the priority of asynchronous PDOs.

Asynchronous PDOs and SDOs can be transmitted at every time with respect to their priority. Hence, they may also be transmitted within the synchronous window.

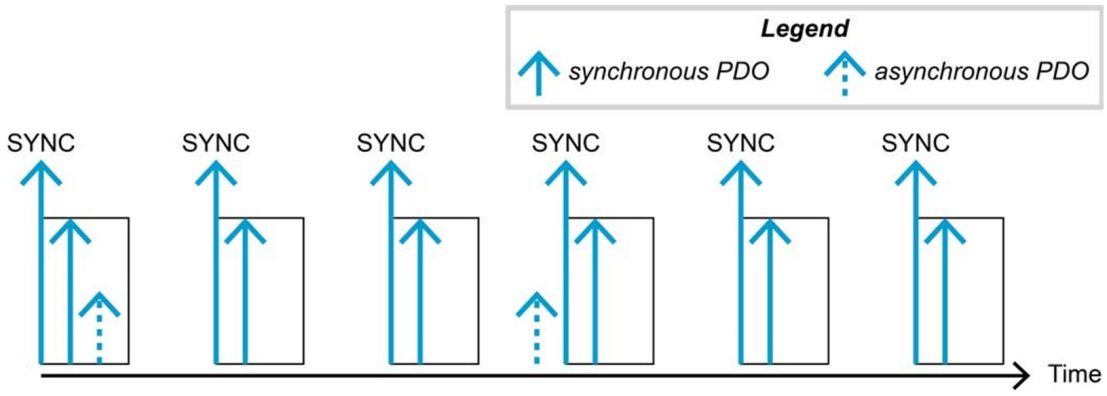


Figure 8: Synchronous PDO

4.6 Network Management – NMT

In addition to providing services and protocols for the transmission of process data and the configuration of devices, the operation of a system distributed over a network requires functions for the command control of the communication state of the individual network nodes. As data transmission by CANopen devices is in many cases event-controlled, continual monitoring of the communication ability of the network nodes is also required. CANopen provides so-called "network management" services and protocols for these tasks, namely:

- control of the communication state of network nodes and
- node monitoring.

4.6.1 NMT Services

The CANopen network management is node-oriented and follows a master/slave structure. It requires one device in the network that fulfils the function of the NMT Master. The other nodes are NMT Slaves.

Network management provides the following functionality groups:

- Module Control Services for initialization of NMT Slaves that want to take part in the distributed application.
- Error Control Services for supervision of nodes' and network's communication status.
- Configuration Control Services for up/downloading of configuration data from/to a network

module.

A NMT Slave represents that part of a node, which is responsible for the node's NMT functionality. It is uniquely identified by its module ID.

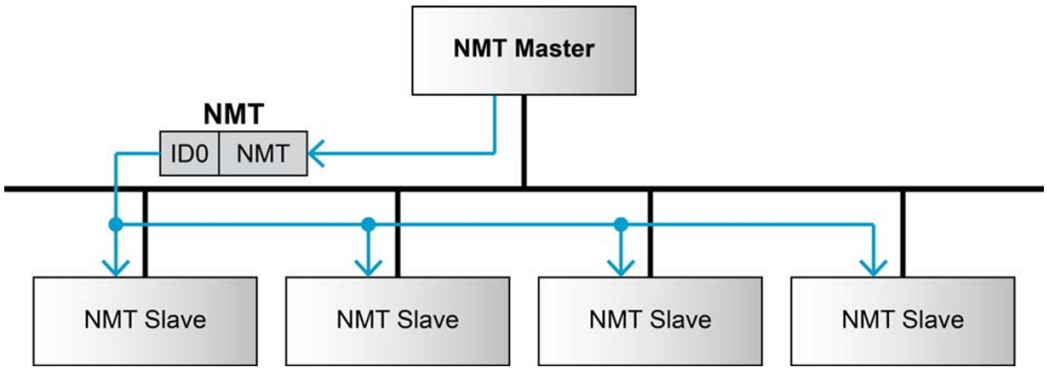


Figure 9: Network management (NMT)

The CANopen NMT Slave devices implement a state machine that automatically brings every device to "Pre-Operational" state, once powered and initialized.

The "Pre-Operational" state is primarily used for the configuration of CANopen devices. Therefore exchange of process data (via PDOs) is not possible in this state. The entries of the device object dictionaries can be accessed via "service data objects" (SDOs). By transmitting an SDO message, the object dictionary of a certain device can be modified, e.g. with a configuration tool.

IMPORTANT. PDO communication is not permitted in Pre-Operational state. Switch to Operational state to enable PDO transmission.

In addition to communication via SDO messages, emergency, synchronization, time stamp and of course NMT control messages can also be transmitted or received in the Pre-operational state. By transmitting a "Start-Remote-Node", a node switches to the "Operational" state.

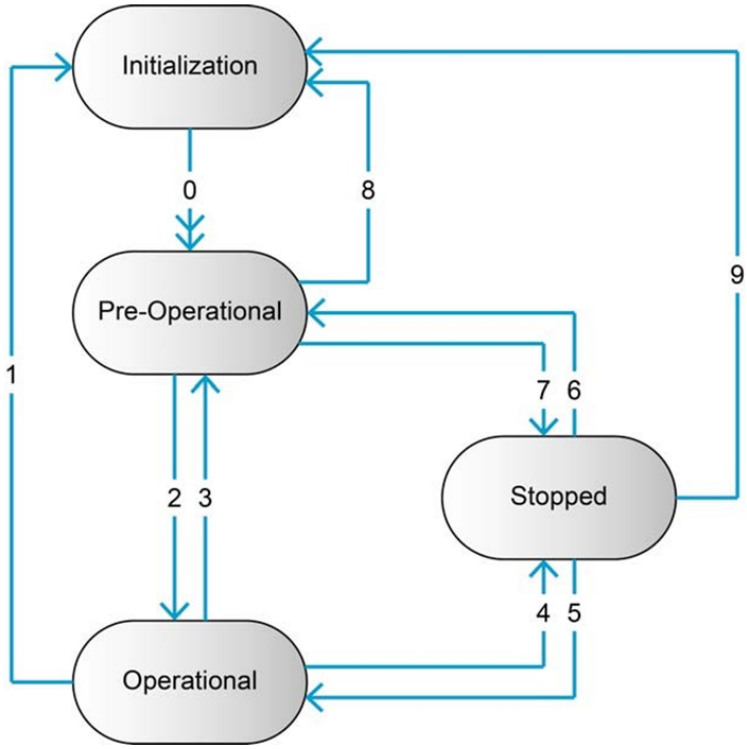


Figure 10: NMT slave states

In "Operational" state, PDO transfer is permitted. Furthermore, "Operational" can be used to achieve certain application behavior. The behavior's definition is part of the device profile's scope. In "Operational", all communication objects are active. Object Dictionary access via SDO is possible. However, implementation aspects or the application state machine may require to switching off or to read only certain application objects while being operational (e.g. an object may contain the application program, which cannot be changed during execution).

By switching a device into "Stopped" state it will be forced to stop PDO and SDO communication. Except for node guarding or heartbeat messages, a node cannot transmit or receive any other messages in this state.

4.7 CANopen Error Handling – EMCY

4.7.1 Principle

Emergency objects are triggered by the occurrence of a CANopen device internal error situation and are transmitted from an emergency producer on the CANopen device. They are assigned the highest possible priority to ensure that they get access to the bus without latency. Emergency objects are suitable for interrupt type error alerts. An emergency object is transmitted only once per 'error event'. No further emergency objects will be transmitted as long as no new errors occur on a CANopen device.

Zero or more emergency consumers may receive the emergency object.

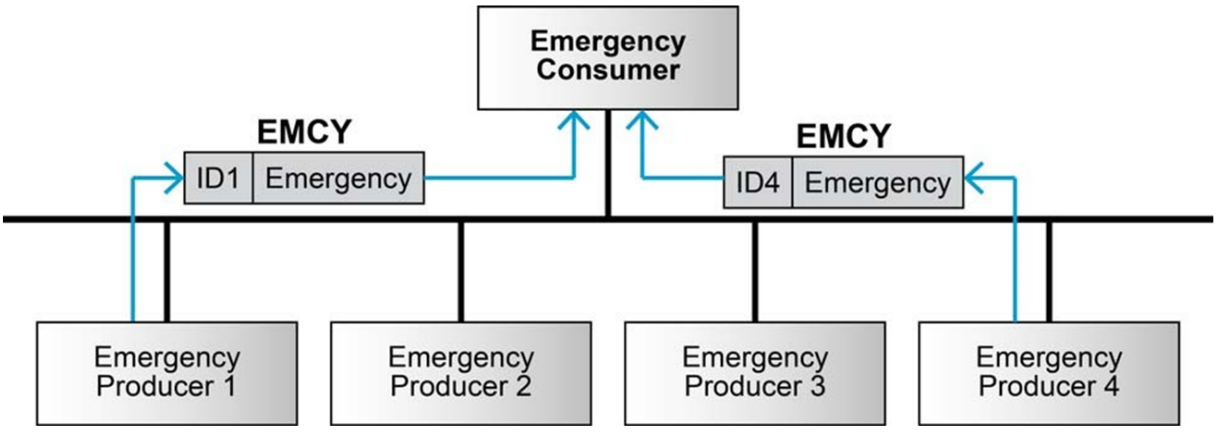


Figure 11: Emergency service (EMCY)

Simultaneously with transmission of the emergency message, the device writes the error code to [1003], where the error history is stored. The error register is content of the OD entry [1001] with bit-wise coding of the error cause

4.7.2 Emergency Message Frame

The device transmits emergency message frames over the CANopen network using *COB-ID EMCY (H1014)*. An emergency message consists of the error code with pre-defined error numbers and the actual state of the *Error Register (H1001)*.

Byte	0	1	2	3	4	5	6	7
Description	Error Code		Error	Manufacturer specific error code				

		Register	

Table 2: Emergency Message Frame

IMPORTANT. Emergency messages are only available for CAN bus communication and not for serial RS232 communication.

5 CANopen Serial Interface (CSI)

5.1 Overview

This section describes the CETONI CANopen based Serial Interface (CSI). This is a serial protocol that enables access to CANopen device object dictionaries via a RS232 serial interface. The CANopen based Serial Interface (CSI) supports an *Industrial RS232 Protocol with CRC checksum* for reliable RS232 connection of CETONI devices to control systems in industrial or laboratory environments. For a high degree of reliability in an electrically noisy environment, it features a checksum.

IMPORTANT. The protocol is a binary protocol with CRC checksum and handshaking. So it is not possible to simply access device parameters via serial terminal program.

5.2 Physical Layer

5.2.1 Electrical Standard

The CSI communication protocol uses the RS232 standard for transmitting data over a three wires cable, for the signals TxD, RxD and GND.

The RS232 standard can be used only for a point-to-point communication between a master and a single device slave. The standard uses negative, bipolar logic in which a negative voltage signal

represents a logic ‘1’, and positive voltage represents a logic ‘0’. Voltages of –3V to –25V with respect to signal ground (GND) are considered logic ‘1’, whereas voltages of +3V to 25V are considered logic ‘0’.

5.2.2 Medium

For the physical connection a 3 wire cable is required. It is recommended to install a shielded and twisted pair cable in order to have a good performance even in an electrically noisy environment. Depending on the bit rate used the cable length can range from 3 meters up to 15 meters. However we do not recommend RS232 cables longer than 5 meters.

6 Industrial RS232 Protocol with CRC checksum

6.1 Introduction

The serial EIA RS232 communication protocol is used to transmit and receive data over the CETONI device's RS232 serial port. Its principal task is to transmit data from a master (PC or any other central processing unit) to a single slave. The protocol is defined or point-to-point communication based on the EIA-RS232 standard.

The protocol can be used to implement the command set defined for CETONI devices. For a high degree of reliability in an electrically noisy environment, it features a checksum.

6.2 Protocol and Flow Control

The CETONI CANopen devices always communicates as a slave. A frame is only sent as an answer to a request. All commands send an answer. The master must always initiate communication by sending a packet structure.

Below described are the data flow while transmitting and receiving frame.

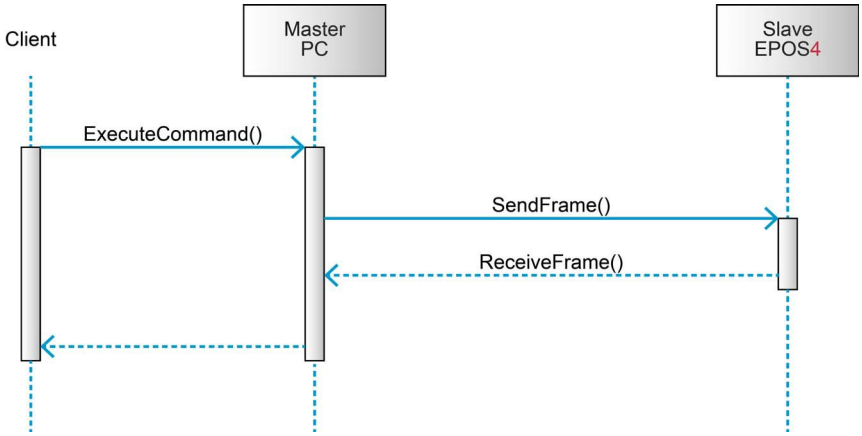


Figure 12: RS232 communication flow

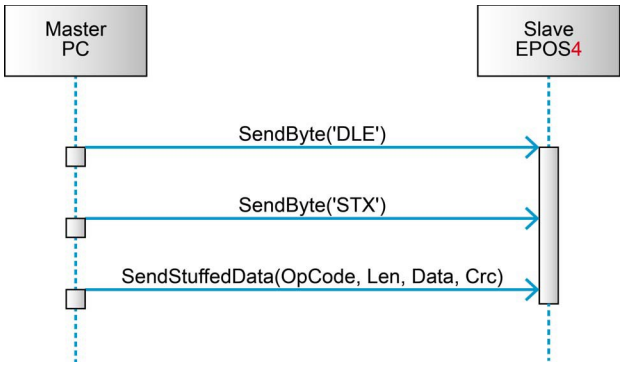


Figure 13: Sending a data frame to CETONI device

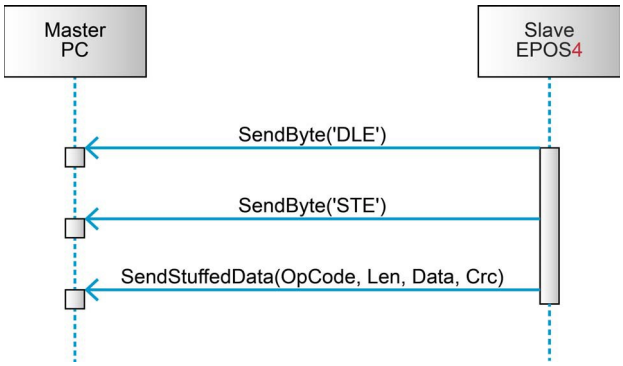


Figure 14: Receiving a response data frame from CETONI device

6.3 Frame Structure

The data bytes are sequentially transmitted in frames. A frame composes of...

- synchronization (and byte stuffing),
- header,
- variably long data field, and
- 16-bit long cyclic redundancy check (CRC) for verification of data integrity.

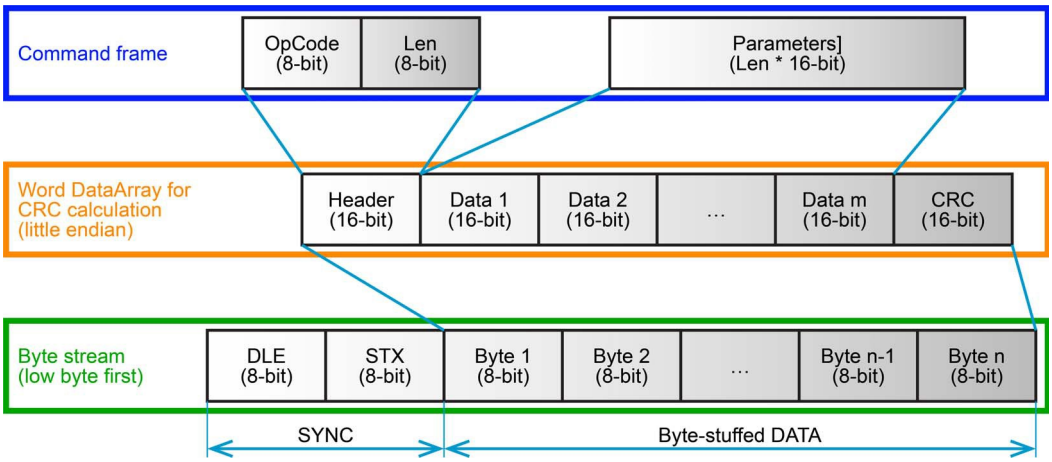


Figure 15: Frame structure

- SYNC - The first two bytes are used for frame synchronization.
  - DLE - Starting frame character “DLE” (Data Link Escape) = 0x90
  - STX - Starting frame character “STX” (Start of Text) = 0x02
- HEADER - The header consists of 2 bytes. The first field determines the type of data frame to be sent or received. The next field contains the length of the data fields.
  - OpCode - Operation command to be sent to the slave. For details on the command set
  - Len - Represents the number of words (16-bit value) in the data fields [0...143].

- DATA - The data fields contain the parameters of the message. The low byte of the word is transmitted first.
  - Data[i] The parameter word of the command. The low byte is transmitted first.
  - CRC 16-bit long cyclic redundancy check (CRC) for verification of data integrity.

IMPORTANT. As a reaction to a bad OpCode or CRC value, the slave sends a frame containing the corresponding error code.

6.4 CRC – Cyclic Redundancy Check

6.4.1 CRC Calculation

CRC is used for verification of data integrity.

IMPORTANT

- The 16-bit CRC checksum uses the algorithm CRC-CCITT.
- For calculation, the 16-bit generator polynomial “x16+x12+x5+x0” is used.
- The CRC is calculated before data stuffing and synchronization.
- Add a CRC value of “0” (zero) for CRC calculation.
- The data frame bytes must be calculated as a word.

6.4.2 CRC Algorithm

ArrayLength: Len + 2	WORD DataArray[ArrayLength]
Generator Polynom G(x):	10001000000100001 (= $x^{16}+x^{12}+x^5+x^0$)
DataArray[0]:	HighByte(Len) + LowByte(OpCode)
DataArray[1]:	Data[0]
DataArray[2]:	Data[1]
…	
DataArray[ArrayLength-1]:	0x0000 (ZeroWord)

The following C-Code shows how to calculate the CRC checksum:

```
uint16_t CalcFieldCRC(uint16_t *pDataArray, uint16_t ArrayLength)
{
  uint16_t shifter, c;
  uint16_t carry;
  uint16_t CRC = 0;
  //Calculate pDataArray Word by Word
  while(ArrayLength--)
  {
    shifter = 0x8000;           //Initialize BitX to Bit15
    c = *pDataArray++;          //Copy next DataWord to c
    do
    {
      carry = CRC & 0x8000;     //Check if Bit15 of CRC is set
      CRC <<= 1;                //CRC = CRC * 2
      if (c & shifter) CRC++;   //CRC = CRC + 1, if BitX is set in c
      if (carry) CRC ^= 0x1021; //CRC = CRC XOR G(x), if carry is true
      shifter >>= 1;            //Set BitX to next lower Bit, shifter = shifter/2
    } while (shifter);
  }
  return CRC
}
```

6.5 Byte Stuffing

The sequence “DLE” and “STX” are reserved for frame start synchronization. If the character “DLE” appears at a position between “OpCode” and “CRC” and is not a starting character, the byte must be doubled (byte stuffing). Otherwise, the protocol begins to synchronize for a new frame. The character “STX” needs not to be doubled.

Examples:

Sending Data	0x21, **0x90**, 0x45
Stuffed Data	0x21, **0x90, 0x90**, 0x45

Sending Data	0x21, **0x90**, **0x02**, 0x45

Stuffed Data	0x21, **0x90, 0x90**, **0x02**, 0x45

Sending Data	0x21, **0x90, 0x90**, 0x45
Stuffed Data	0x21, **0x90, 0x90, 0x90, 0x90**, 0x45

IMPORTANT. Byte stuffing is used for all bytes (CRC included) in the frame except the starting characters.

6.6 Transmission Byte Order

To send and receive a word (16-bit) via the serial port, the low byte will be transmitted first.

Multiple byte data (word = 2 bytes, long word = 4 bytes) are transmitted starting with the less significant byte (LSB) first.

A word will be transmitted in this order: byte0 (LSB), byte1 (MSB).

A long word will be transmitted in this order: byte0 (LSB), byte1, byte2, byte3 (MSB).

6.7 Data Format

Data is transmitted in an asynchronous way, thus each data byte is transmitted individually with its own start and stop bit. The format is

1 Start bit, 8 Data bits, No parity, 1 Stop bit (8N1)

Most serial communication chips (SCI, UART) can generate such data format.

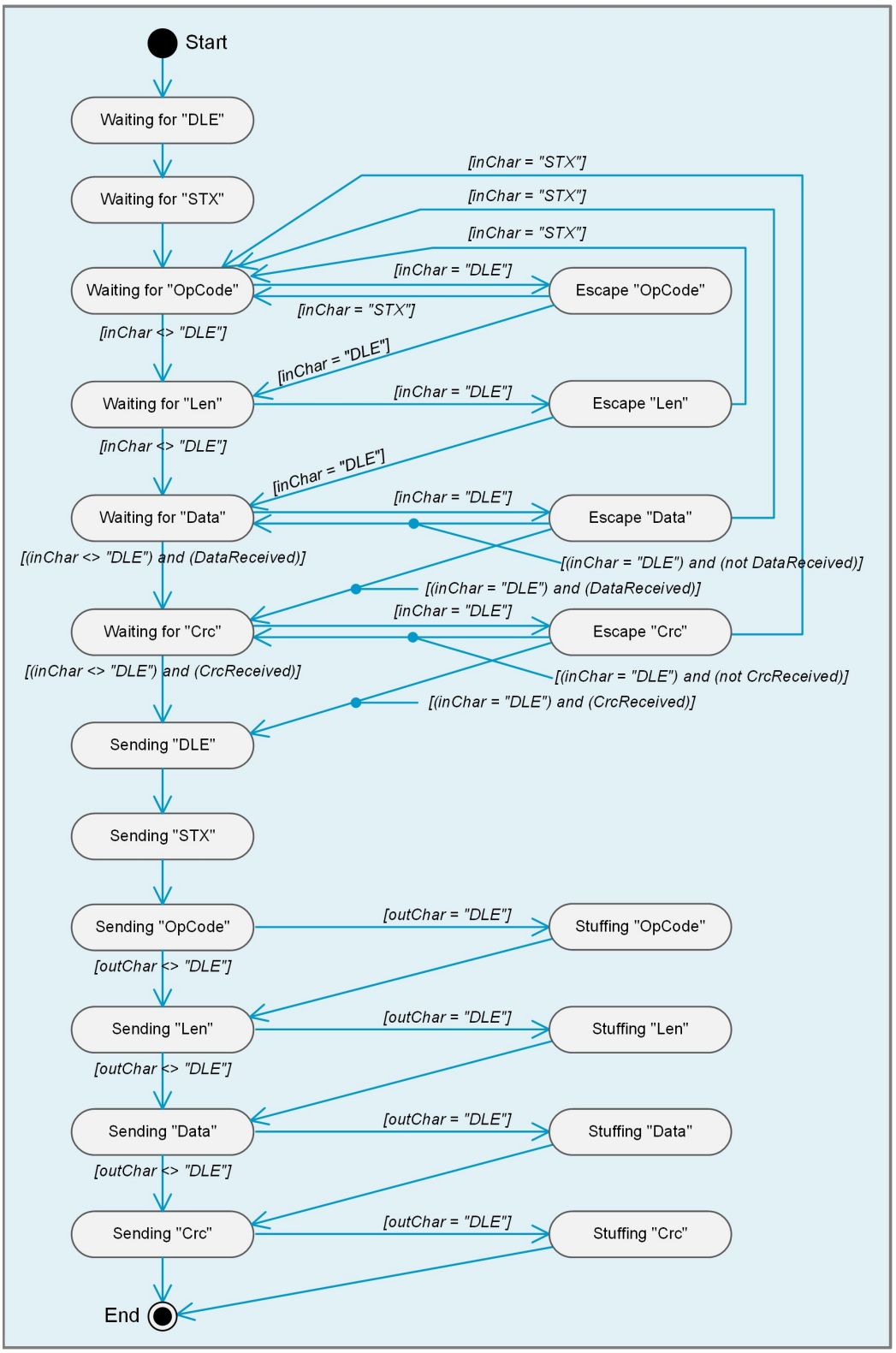
6.8 Timeout Handling

The timeout is handled over a complete frame. Hence, the timeout is evaluated over the sent data frame, the command processing procedure and the response data frame. For each frame (frames, data processing), the timer is reset and timeout handling will recommence.

Index	Subindex	Object	Default
0x2005	0x00	RS232 Frame Timeout	500 [ms]

HINT. To cover special requirements, the timeout may be changed by writing to the Object Dictionary!

6.9 Slave (device) state machine



6.10 Command Reference

6.10.1 Read Functions

6.10.1.1 READ OBJECT DICTIONARY ENTRY (4 DATA BYTES AND LESS)

Read an object value from the Object Dictionary of the device at the given Index and SubIndex.

REQUEST FRAME		
OpCode	BYTE	0x60
Len-1	BYTE	2 (number of words)
Parameters	BYTE	Node-ID
	WORD	Index of Object
	BYTE	Subindex of Object

The device responds with a data frame with 4 bytes of data.

RESPONSE FRAME		
OpCode	BYTE	0x00
Len-1	BYTE	4 (number of words)
Parameters	DWORD	Communication Error Code (see firmware spec.)
	BYTE [4]	Data Bytes read

6.10.2 Write Functions

6.10.2.1 WRITE OBJECT DICTIONARY ENTRY (4 DATA BYTES AND LESS)

Write an object value to the Object Dictionary at the given Index and SubIndex.

REQUEST FRAME		
OpCode	BYTE	0x68
Len-1	BYTE	4 (number of words)Node ID
Parameters	BYTE	Node ID
	WORD	Index of Object
	BYTE	SubIndex of Object
	BYTE Data[4]	Data Bytes to write

The device responds with a response frame without any data.

RESPONSE FRAME		
OpCode	BYTE	0x00
Len-1	BYTE	2 (number of words)
Parameters	DWORD	Communication Error Code (see firmware spec)

6.11 Example Frames

6.11.1 Reading Object 0x1000 – Device Type

Index	Sub Index	Name	Type	Access	Value
0x1000	0x00	Device Type	UInt32	RO	0x000200192

The following example shows, how to read the device type object. The device type object can be read via object dictionary index 0x1000 and sub-index 0. In the following example the object is read from a Nemesys pump and the value returned by the device is 0x00020192.

6.11.1.1 FRAME SETUP

REQUEST FRAME			
OpCode	BYTE	Read object	0x60
Len-1	BYTE	Number of words	0x02
Parameters	BYTE	Node-ID	0x02
	WORD	Index of Object	0x1000
	BYTE	Subindex of Object	0x00

6.11.1.2 CRC CALCULATION

Before you calculate the CRC checksum for the read request frame, you should have the following array of data words:

DataArray	
DataArray[0]	0x0260
DataArray[1]	0x0002
DataArray[2]	0x0010
DataArray[3]	0x0000 (use CRC value of 0 as placeholder)

Now you can calculate the checksum for the 4 words and insert the result 0xEECD into the DataArray[3] field in little endian order so that you get the following DataArray:

DataArray	
DataArray[0]	0x0260
DataArray[1]	0x0002
DataArray[2]	0x0010
DataArray[3]	0xEECD

IMPORTANT

- Make sure that the CRC is calculated correctly. If the CRC is not correct, the command will neither be accepted nor processed.
- CRC calculation includes all bytes of the data frame except synchronization bytes and byte stuffing.
- The data frame bytes must be calculated as a word.
- For calculation, use a CRC value of "0" (zero).

6.11.1.3 CREATE BYTE STREAM AND SEND DATA

- Pack the DataArray to a byte stream (low byte first): 60 02 02 00 10 00 CD EE
- Add sync bytes: 90 02 60 02 02 00 10 00 CD EE
- Add byte stuffing: 90 02 60 02 02 00 10 00 CD EE (because the frame does not contain 0x90 data bytes, nothing changes)
- Send stuffed data (low byte first)

6.11.1.4 WAIT FOR THE RECEIVE FRAME

The device will answer to the command "ReadObject" with an answer frame and the returned parameters in the data block as follows (Reception order low byte first). The first two bytes are the sync bytes:

90 02 00 04 00 00 00 00 92 01 02 00 9A ED

IMPORTANT. Do not send any data before the receive frame or a timeout is present. EPOS4 cannot process data simultaneously.

6.11.1.5 REMOVE BYTE STUFFING IN SYNCHRONIZATION ELEMENTS

Byte stream without stuffing and synchronization elements:

00 04 00 00 00 00 92 01 02 00 9A ED

6.11.1.6 CRC CHECK

Now you should have the following array of data words for CRC calculation.

DataArray	
DataArray[0]	0x0400
DataArray[1]	0x0000
DataArray[2]	0x0000
DataArray[3]	0x0192
DataArray[4]	0x0002
DataArray[5]	0xED9A

Now you can calculate the checksum for the 6 words. The result of the CRC calculation should be 0. Only if the result is 0, you have received a valid data frame.

6.11.1.7 CHECK THE RECEIVED DATA

Now you have the following valid response frame:

RESPONSE FRAME			
OpCode	BYTE	OpCode	0x00
Len-1	BYTE	Number of words	0x04
Parameters	DWORD	Communication Error Code	0x00000000 (no error)
	BYTE [4]	Data Bytes read	0x00020192

Now check the received communication error and readout the received data.

IMPORTANT

- If the error code is unequal to “0” (zero), the command was not processed!
- Check Communication Error Code Definition for error details.
- Fix the error before attempting to resend the data frame.

6.11.2 Writing Object 0x1017 – Producer Heartbeat Time

Index	Sub Index	Name	Type	Access	Value
0x1017	0x00	Producer Heartbeat Time	UInt32	RW	400

The following example shows, how to write a producer heartbeat time of 400 milliseconds into the object 0x1017 sub-index 0. The hex value for 400 milliseconds is 0x190.

6.11.2.1 FRAME SETUP

REQUEST FRAME			
OpCode	BYTE	Write object	0x68
Len-1	BYTE	Number of words	0x04
Parameters	BYTE	Node-ID	0x02
	WORD	Index of Object	0x1017
	BYTE	Subindex of Object	0x00
	BYTE Data[4]	Data Bytes to write	0x00000190

6.11.2.2 CRC CALCULATION

Before you calculate the CRC checksum for the write request frame, you should have the following array of data words:

DataArray	
DataArray[0]	0x0468
DataArray[1]	0x1702
DataArray[2]	0x0010
DataArray[3]	0x0190
DataArray[4]	0x0000
DataArray[5]	0x0000 (use CRC value of 0 as placeholder)

Now you can calculate the checksum for the 4 words and insert the result 0xEC77 into the **DataArray[5]** field in little endian order so that you get the following DataArray:

DataArray	
DataArray[0]	0x0468
DataArray[1]	0x1702
DataArray[2]	0x0010
DataArray[3]	0x0190
DataArray[4]	0x0000
DataArray[5]	0xEC77

IMPORTANT

- Make sure that the CRC is calculated correctly. If the CRC is not correct, the command will neither be accepted nor processed.
- CRC calculation includes all bytes of the data frame except synchronization bytes and byte stuffing.
- The data frame bytes must be calculated as a word.
- For calculation, use a CRC value of "0" (zero).

6.11.2.3 CREATE BYTE STREAM AND SEND DATA

- Pack the DataArray to a byte stream (low byte first): 68 04 02 17 10 00 90 01 00 00 77 EC
- Add sync bytes: **90 02** 68 04 02 17 10 00 90 01 00 00 77 EC
- Add byte stuffing: 90 02 68 04 02 17 10 00 **90 90** 01 00 00 77 EC
- Send stuffed data (low byte first)

6.11.2.4 WAIT FOR THE RECEIVE FRAME

The device will answer to the command "WriteObject" with an answer frame and the returned parameters in the data block as follows (Reception order low byte first). The first two bytes are the sync bytes:

90 02 00 02 00 00 00 00 40 8B

IMPORTANT. Do not send any data before the receive frame or a timeout is present. EPOS4 cannot process data simultaneously.

6.11.2.5 REMOVE BYTE STUFFING IN SYNCHRONIZATION ELEMENTS

Byte stream without stuffing and synchronization elements:

00 02 00 00 00 00 40 8B

6.11.2.6 CRC CHECK

Now you should have the following array of data words for CRC calculation.

DataArray	
DataArray[0]	0x0200
DataArray[1]	0x0000
DataArray[2]	0x0000
DataArray[3]	0x8B40

Now you can calculate the checksum for the 4 words. The result of the CRC calculation should be 0. Only if the result is 0, you have received a valid data frame.

6.11.2.7 CHECK THE RECEIVED DATA

Now you have the following valid response frame:

RESPONSE FRAME			
OpCode	BYTE	OpCode	0x00
Len-1	BYTE	Number of words	0x02
Parameters	DWORD	Communication Error Code	0x00000000 (no error)

Now check the received communication error.

IMPORTANT

- If the error code is unequal to “0” (zero), the command was not processed!
- Check Communication Error Code Definition for error details.
- Fix the error before attempting to resend the data frame.

6.12 Communication Error Code Definition

ABORT CODE	NAME	CAUSE
0x0000 0000	No abort	Communication successful
0x0503 0000	Toggle error	Toggle bit not alternated
0x0504 0000	SDO timeout	SDO protocol timed out
0x0504 0001	Command unknown	Command specifier unknown
0x0504 0004	CRC error	CRC check failed
0x0601 0000	Access error	Unsupported access to an object
0x0601 0001	Write only error	Read command to a write only object
0x0601 0002	Read only error	Write command to a read only object
0x0601 0003	Subindex cannot be written	Subindex cannot be written, subindex 0 must be “0” (zero) for write access
0x0601 0004	SDO complete access not supported	The object can not be accessed via complete access
0x0602 0000	Object does not exist error	Last read or write command had wrong object index or subindex
0x0604 0041	PDO mapping error	Object is not mappable to the PDO
0x0604 0042	PDO length error	Number and length of objects to be mapped would exceed PDO length
0x0604 0043	General parameter error	General parameter incompatibility
0x0604 0047	General internal incompatibility error	General internal incompatibility in device
0x0606 0000	Hardware error	Access failed due to hardware error
0x0607 0010	Service parameter error	Data type does not match, length or service parameter do not match
0x0607 0013	Service parameter too short error	Data type does not match, length of service parameter too low
0x0609 0011	Subindex error	Last read or write command had wrong object subindex
0x0609 0030	Value range error	Value range of parameter exceeded
0x0800 0000	General error	General error
0x0800 0020	Transfer or store error	Data cannot be transferred or stored
0x0800 0022	Wrong device state error	Data cannot be transferred or stored to application because of present device state
0x0F00 FFBE	Password error	Password is incorrect
0x0F00 FFBF	Illegal command error	Command code is illegal (does not exist)
0x0F00 FFC0	Wrong NMT state error	Device is in wrong NMT state

7 Pump Control

7.1 Object Dictionary

Internally the pump uses a EPOS4 CANopen DS402 servo drive to move the pusher and the syringe piston. A detailed description of the EPOS4 CANopen drives is provided with the document EPOS4-Firmware-Specification.pdf. You can control the drive by reading and writing the object dictionary entries of the device. The controller has an extensive object directory (see section 6 *Object Dictionary* in the *EPOS4 firmware specification*), but only a few entries are relevant for the control of the Nemesys pumps. The following table list all object dictionary entries that are required for pump control.

Index	Name
0x1001	Error Register
0x1003	Error History (Predefined Error Field)
0x2005	RS232 Frame Timeout
0x210C	Custom persistent memory
0x3000	Axis configuration
0x3003	Gear configuration
0x3141	Digital input properties
0x30D3	Velocity Actual Values
0x3160	Analog input properties
0x3182	Analog output general purpose
0x6040	Controlword
0x6041	Statusword
0x6060	Modes of operation
0x6061	Modes of operation display
0x6064	Position actual value
0x606B	Velocity demand
0x607A	Target position
0x607D	Software position limit
0x607F	Max profile velocity
0x6081	Profile velocity

0x60A9	Si unit velocity
0x60FD	Digital inputs
0x60FE	Digital outputs

7.2 Operating Modes

The EPOS4 CANopen drive supports a number of operating modes (see section 3 *Operating Modes* in the EPOS4-Firmware-Specification.pdf). For pump control, only one of these operation modes is required.

- **MODE 1 – PROFILE POSITION:** this mode is required for normal pumps tasks like aspirating or dispensing

To activate a mode, you simply need to write the mode index 1 for Profile Position Mode into the object dictionary entry *0x6060 Modes Of Operation*. To read out the active operation mode, you simple need to read the current value of *0x6061 Modes Of Operation Display*.

Index	Subindex	Object	Description	Type
0x6060	0	Modes of Operation	Write into this object to switch operating mode	INTEGER8
0x6061	0	Modes of Operation Display	Read current operating mode from this object	INTEGER8

7.3 Translation of volume / flow units

7.3.1 Introduction

Whenever you call a function that requires a volume (position) value or a flow (speed) value, the value is given in device internal units like increments for position values and mrpm (millirevolutions per minute) for velocity values. These technical units are not well suited for dosing tasks (for volumes and flow rates) and need to be translated by the application to implement pump control.

This translation depends on several parameters like mechanical configuration of the single dosing units (gear) and it also depends on the syringes used for dosing. The following two sections will show you, how you can convert the internal device units for device control into units for volume and flow values.

7.3.2 Reading out device parameters

7.3.2.1 OVERVIEW

For the calculation of flow rates and volumes the following device parameters are required:

Index	Subindex	Object	Description	Type
0x60A9	0	SI unit velocity	Contains the velocity SI unit	UNSIGED32
0x3003	1	Gear reduction numerator	Read gear numerator	UNSIGED32
0x3003	2	Gear reduction denominator	Read gear denominator	UNSIGED32
0x3000	5	Main sensor resolution	Read the encoder resolution from this object	UNSIGED32

7.3.2.2 SI UNIT VELOCITY

The Object 0x60A9 defines the velocity unit.

Bit 31..24	Bit 23..16	Bit 15..8	Bit 7..0
Prefix	Numerator	Denominator	Reserved (0)

All velocity values are given and returned in internal velocity units. The following values are possible:

Value	Velocity unit	Symbol
0x00B44700	revolutions/minute	rev/min (rpm)
0xFFB44700	0.1 revolutions/minute	deci rev/min (drpm)
0xFEB44700	0.01 revolutions/minute	centi rev/min (crpm)
0xFDB44700	**0.001 revolutions/minute**	**milli rev/min (mrpm) Default**
0xFCB44700	0.0001 revolutions/minute	10^{-4} rev/min (10^{-4} rpm)
0xFBB44700	0.00001 revolutions/minute	10^{-5} rev/min (10^{-5} rpm)
0xFAB44700	0.000001 revolutions/minute	micro rev/min (µrpm)

The default unit is **mrpm – millirevolutions** per minute.

This is how the prefix is given:

Prefix	Factor	Symbol	Notion I
-	10 ^ 0	-	0x00
Deci	10 ^ -1	d	0xFF
Centi	10 ^ -2	c	0xFE
Milli	10 ^ -3	m	0xFD

7.3.2.3 ENCODER RESOLUTION

The encoder resolution defines the number of increments per motor revolution. All internal position values are given in increments. To read out the encoder pulse number you need to read the object dictionary entry *0x3000, Subindex 5 – Main sensor resolution*.

7.3.2.4 GEAR FACTOR

The gear factor defines the factor for the conversion of motor revolutions into the moved pusher distance in mm. The gear factor consists of a gear nominator and a gear denominator. You need to read the following two object dictionary entries, to get the gear factor.

Index	Subindex	Object	Description	Type
0x3003	1	Gear reduction numerator	Read gear numerator	UNSIGED32
0x3003	2	Gear reduction denominator	Read gear denominator	UNSIGED32

From these to values, you can calculate the gear factor:

Gear Factor (rev/mm) = Gear numerator / Gear denominator

7.3.3 Position value conversion

7.3.3.1 CALCULATING THE POSITION CONVERSION FACTOR

With the values read from the device, you can calculate a position conversion factor for conversion between internal device units (increments) and millimetres. First we can convert the increment value into motor revolutions with the help of the encoder resolution value:

Motor revolutions =Increments / Encoder Resolution

Then you have to translate these motor rotations into the distance of the pusher with the help of the gear factor:

Distance in mm = Motor revolutions / Gear factor

So the final calculation is:

$$mm = \frac{Increments}{Encoder\ Resolution(inc/rev) \times Gear\ factor(rev/mm)}$$

From this formula we can extract the position conversion factor:

Position conversion factor (inc/mm) = Encoder Resolution (inc/rev) x Gear Factor (rev/mm)

7.3.3.2 CONVERSION OF POSITION VALUES

To convert from increments into a distance in mm, you just need to de divide the increments value by the position conversion factor:

mm = Increments / Position conversion factor

To convert a distance in mm into an increments value, you just need to multiply the distance with the conversion factor:

Increments = mm * Position conversion factor

7.3.3.3 EXAMPLE POSITION CONVERSION

The following example shows how to convert a distance in millimetres into internal position units:.

Encoder resolution:	8192 inc/rev
Gear factor:	21,78 rev/mm
Position conversion factor	8192 inc/rev * 21,78 rev/mm = 178.421,76 inc/mm
Distance in mm:	10 mm
Position value in increments:	10 mm * 178.421,76 inc/mm = 1.784.217,6 inc ≈ 1.784.218 inc

7.3.4 Velocity value conversion

7.3.4.1 CALCULATING THE VELOCITY CONVERSION FACTOR

With the values read from the device, you can calculate a velocity conversion factor for conversion between internal device units and millimetres/second (mm/s). First we convert the internal velocity unit into into revolutions/minute.

$$rev/min = device\ velocity \times 10^{velocity\ unit\ prefix}$$

If the velocity unit prefix in object 0x60A0 is 0xFD, this would mean rev/min = device velocity x 10 ^ -3.

Then we can convert the revolutions/minute into revolutions per second by dividing by 60.

rev/s = rev/min / 60 s/min

Finally we can calculate the velocity in millimetres/second with the help of the gear factor:

$$mm/s = \frac{rev/s}{Gear\ factor\ (rev/mm)}$$

So the final calculations is:

$$mm/s = \frac{device\ velocity \times 10^{velocity\ notation\ index}}{60\,s/min \times Gear\ factor\ (rev/mm)}$$

From this formula we can extract the velocity conversion factor:

$$Velocity\ conversion\ factor = \frac{60\,s/min \times Gear\ factor\ (rev/mm)}{10^{velocity\ notation\ index}}$$

7.3.4.2 CONVERSION OF VELOCITY VALUES

To convert from device velocity into a velocity in mm/s, you just need to divide device velocity values by the velocity conversion factor:

mm/s = Device velocity / Velocity conversion factor

To convert a velocity in mm/s into a device velocity value, you just need to multiply the velocity with the conversion factor:

Device velocity = mm/s x Velocity conversion factor

7.3.4.3 EXAMPLE VELOCITY CONVERSIONS

Velocity unit :	0xFDB44700 = millirevolutions per minute, Prefix is 0xFD = -3
Gear factor:	21,78 rev/mm
Velocity conversion factor:	60s/min x 21,78 rev/mm / 10^{-3} = 1.306.800
Velocity value:	2 mm/s
Device velocity:	2 mm/s * 1.306.800 = 2.613.600 mrev/min

7.3.5 Volume value conversions

7.3.5.1 CALCULATION

Section Position value conversion shows, how to convert internal device position into millimetres. This section shows, how to convert a position value in millimetres into a volume in millilitres. To convert a pusher movement in millimetres into a volume value in millilitres, you need to know the inner diameter of the syringe mounted on the device. With the help of the inner syringe diameter and a length in millimetres, you can calculate the cylinder volume in mm^3.

$$Volume(mm^3) = \frac{\pi}{4} d(mm)^2 \cdot length(mm)$$

One millilitre is equal to 1000 mm^3. So you can calculate millilitres directly with the following formula:

$$Volume(ml) = \frac{\pi}{4} d(mm)^2 \cdot length(mm) / 1000$$

From a given value in millilitres you can calculate the pusher distance with the following formula:

$$mm = \frac{Volume(ml) \cdot 1000 \cdot 4}{\pi d^2}$$

With then help of the Position conversion factor you can now convert millimetres into internal device position units (increments).

7.3.5.2 EXAMPLE VOLUME CONVERSION

The following example shows, how to convert a volume value in millilitres into internal device position units:

Volume:	10 ml
Inner syringe diameter:	14,5673 mm
Distance in mm:	10 ml * 1000 mm^3/ml * 4 / π / (14,5673 mm)^2 = 60 mm
Position conversion factor:	178.421,76 inc/mm
Position value in increments:	60 mm * 178.421,76 inc/mm = 10.705.305,6 inc ≈ 10.705.306 inc

7.3.6 Flow value conversions

7.3.6.1 CALCULATION

Section Velocity value conversion shows, how to convert internal device velocity into millimetres/second (mm/s) and vice versa. This section shows, how to convert a velocity value in millimetres/second (mm/s) into a flow value in millilitres/second (ml/s). To convert a pusher movement in millimetres/second into a flow value in millilitres/second, you need to know the inner diameter of the syringe mounted on the device. With the help of the inner syringe diameter and a length in millimetres, you can calculate the cylinder volume in mm^3.

$$Volume(mm^3) = \frac{\pi}{4} d(mm)^2 \cdot length(mm)$$

One millilitre is equal to 1000 mm^3. So you can calculate millilitres directly with the following formula:

$$Volume(ml) = \frac{\pi}{4} d(mm)^2 \cdot length(mm) / 1000$$

Now we can easily create the formula for conversion of velocity values in mm/s into flow values in ml/s

$$Flow(ml/s) = \frac{Volume(ml)}{s} = \frac{\pi\, d(mm)^2}{4 \times 1000} \times Velocity(mm/s)$$

and the formula for conversion of flow values in ml/s into velocity values in mm/s

$$Velocity(mm/s) = \frac{Flow(ml/s) \cdot 1000 \cdot 4}{\pi\, d(mm)^2}$$

With the help of the Velocity conversion factor you can now convert mm/s into internal device velocity.

7.3.6.2 EXAMPLE FLOW CONVERSION

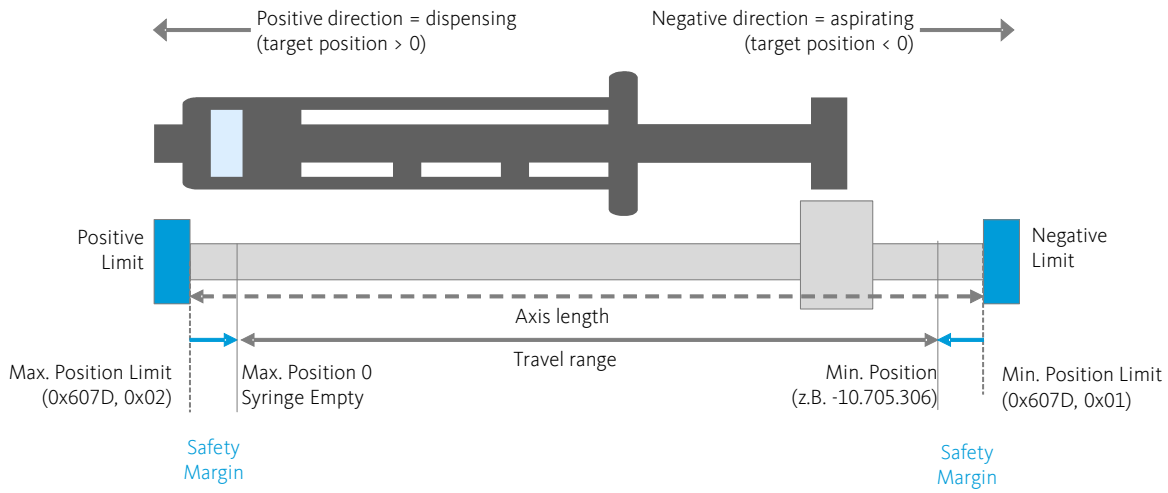
The following example shows, how to convert a flow value in millilitres/second (ml/s) into internal device velocity units:

Flow:	1,054814 ml/s
Inner syringe diameter:	14,5673 mm
Velocity in mm/s:	1,054814 ml/s * 1000 mm3/ml * 4 / π / (14,5673 mm)2 = 6,328 mm/s
Velocity conversion factor:	1.306.800
Device velocity:	6,328 mm/s * 1.306.800 = 8.269.430,4 mrev/min

7.4 Reading out device configuration

7.4.1 Device Overview

The following picture shows the device hardware:



The *axis length* is the distance between the *Positive Limit (Max. Position Limit)* and the *Negative Limit (Min. Position Limit)*. During normal dosing operations, the pusher should not reach these limits. Therefore the real *travel range* is limited by two safety margins.

7.4.2 Calculating the travel range

The object *0x607D Software Position Limit* contains information about the travel range and limits the position range. Using position values outside of this range will cause an error and triggers sending of an emergency message.

Index	Subindex	Object	Description	Type
0x607D	1	Min. position limit	Negative limit	INTEGER32
0x607D	2	Max. position limit	Positive limit	INTEGER32

To calculate the travel range, you first need to read the value of the *0x0607D Subindex 2 Max. position limit*. This value is the safety margin from the zero position.

Safety Margin = 0x0607D Subindex 2 - Max. position limit

The maximum position value and the syringe empty position is the position 0:

Max. Position = 0

To calculate the Min. Position value, you need to read the object *0x0607D Subindex 1 Min. position limit*. The *Min. position limit* is a negative value and therefore you need to add the safety margin:

Min. Position = 0x0607D Subindex 1 Min. position limit + Safety Margin

You now have the minimum and maximum position values and you can calculate the travel range from these values:

Travel range (inc) = Max. Position – Min. Position

The typical ravel range for the Nemesys S and Nemesys M syringe pumps is from 0 to -10.705.306 increments.

If you command a dosing operation, then you should never use position values outside of the travel range (e.g. 0 to -10.705.306). That means, the *Max. Position* and *Min. Position* values limit your travel range. The position counter value increases if the drive moves in positive direction towards the positive limit and decreases if it moves in negative direction towards the negative limit. If you would like to empty the syringe or aspirate, you need to move towards the positive limit. If you would like to aspirate or refill, then you need to move toward the negative limit.

Here is the C code from the Nemesys RS232 library for reading out and calculating the minimum and maximum position values:

```
CsiDevReadObject(&Nemesys->Device, NemV4_OD_H607D_SW_POS_LIMITS,
  2, (uint32_t*)&Nemesys->MaxPos);
CsiDevReadObject(&Nemesys->Device, NemV4_OD_H607D_SW_POS_LIMITS,
  1, (uint32_t*)&Nemesys->MinPos);
int SafetyMargin = Nemesys->MaxPos;
Nemesys->MaxPos -= SafetyMargin;
Nemesys->MinPos += SafetyMargin;
```

7.4.3 Calculating the maximum flow rate

The object *0x607F Subindex 0* provides the maximum possible velocity in device units.

Index	Subindex	Object	Description	Type
0x607F	0	Max profile velocity	Maximum allowed velocity value	UNSIGNED32

You can convert this value into the maximum possible flow rate using the calculations from section Translation of volume / flow units.

7.4.4 Reading out the device type

To get additional information about the type of device, you can read object *0x210C Subindex 3 Custom persistent memory 3*.

Index	Subindex	Object	Description	Type
0x210C	3	Custom Persistent Memory 3	Pump configuratopm	UNSIGNED32

The object contains a bitfield with additional information about various pump configuration parameters:

Field	Value	Description
Product Type	6	Nemesys M
	7	Nemesys S

Here is the Nemesys RS232 library C code for reading out the product type:

```
CsiDevReadObject(&Nemesys->Device, NemV4_OD_H210C_CUSTOM_PERSISTENT_MEM, 3, &AxisConf);
Nemesys->ProductType = (AxisConf >> 10) & 0x7F;
```

7.5 Initializing

Here is your checklist what you should do, to properly initialize your software parameters and to prepare your software and the pump for the first dosage command:

(1) Read the encoder resolution, the gear factor and the velocity SI unit for calculation of the velocity and position conversion factors to convert between device units (inc, mrpm) and SI units (mm, mm/min).

(2) Initialize your local syringe parameters in your software to properly convert volumes and flow rates (ml, ml/min) into positions and velocities (mm, mm/min).

(3) Read the software position limits from the device to properly calculate your travel range and your position limits for volume and flow commands

(4) Read the maximum profile velocity from the device for proper calculation of the maximum flow rate.

(5) Read the product type from the device to initialize your force sensor and force limit conversion factors to convert between mV and force values.

(6) Ensure, that the operation mode is profile position mode

(7) Ensure, that force monitoring is enabled

(8) Check safety stop input – it should be off.

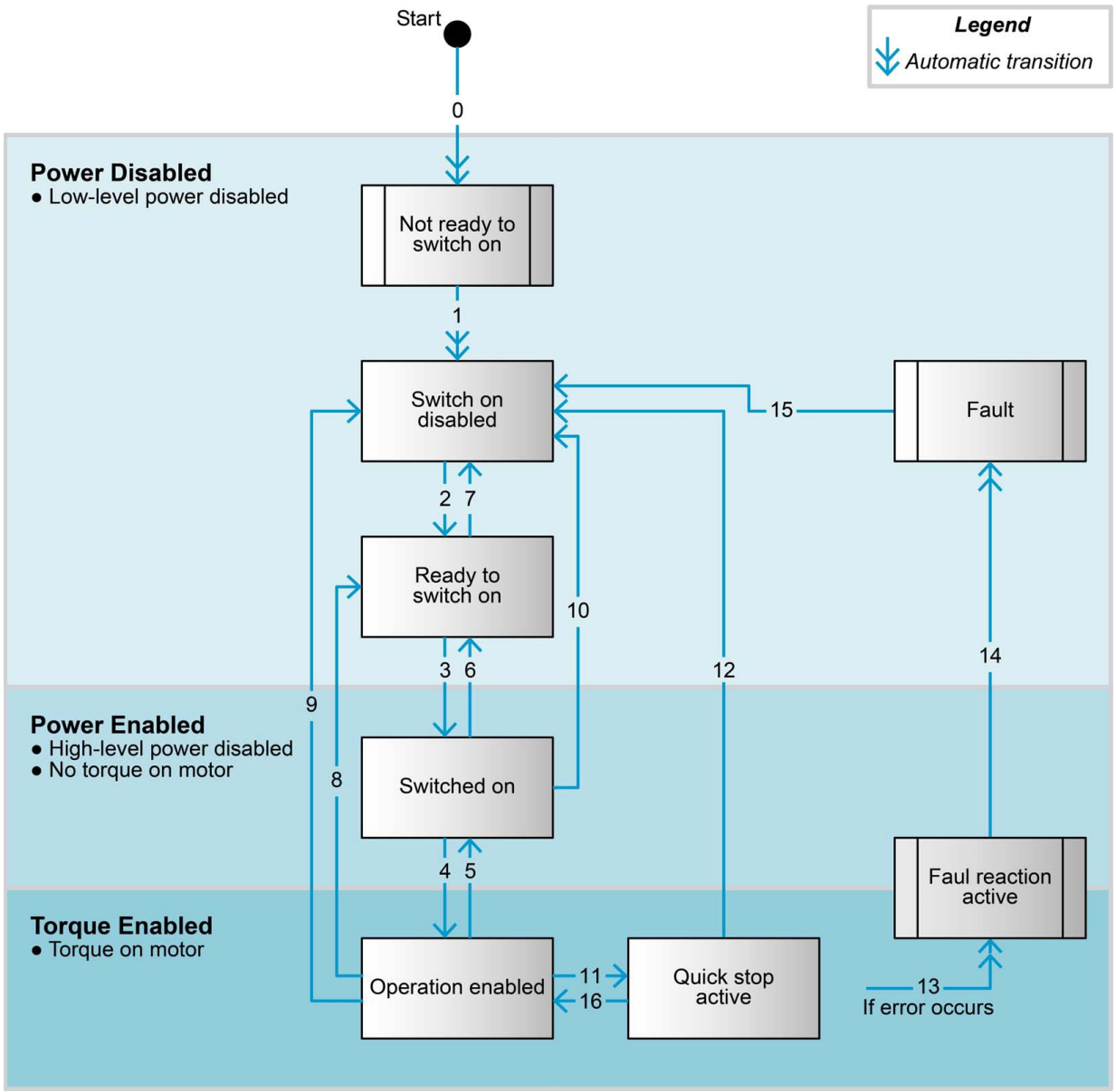
(9) Check if the device is in fault state and clear fault if this is required

(10) Set the pump drive into enabled state

7.6 Pump Drive Control

7.6.1 Drive State Machine

The pump drive uses the state machine below. States may be changes by using the *Controlword* (Object 0x6040). The actual state can be read using the *Statusword* (Object 0x6041). A new state transition must not be initiated before the previous one is completed and the *Statusword* is changed accordingly.



7.6.2 Reading State of Drive

You need to read the Statusword to get the state of the drive:

Index	Subindex	Object	Description	Type
0x6041	0	Statusword	Read current device state and status information from this object	UNSIGNED16

Below are the Statusword bits for the profile position mode:

Bit	Name	Decription
15	Position referenced to home position	Not required - ignore
14	reserved (0)	
13	Following error	• 0 – No following error • 1 - The required force is higher than the maximum drive force (high pressure) or the drive is blocked somehow
12	Setpoint acknowledge	• 0 - Positioning to the previous setpoint is ongoing and a new setpoint may be accepted • The previous setpoint has been assumed and no additional setpoint may be accepted
11	Internal limit active	
10	Target reached	A value of 1 indicates, that target position is reached / drive is stopped
9	Remote	Not required - ignore
8	reserved (0)	
7	Warning	
6	Switch on disabled	Actual drive state (see Table below)
5	Quick stop	
4	Voltage enabled (power stage on)	
3	Fault	
2	Operation enabled	
1	Switched on	
0	Ready to switch on	

The Statusword bits 0- 6 (see table above) indicate the actual state of the drive as shown in the state diagramm

State	Statusword [binary]	Description
Not ready to switch on	xxxx xxxx x00x 0000	Drive function is disabled
Switch on disabled	xxxx xxxx x10x 0000	Drive initialization is complete. Drive parameters may be changed. Drive function is disabled.
Ready to switch on	xxxx xxxx x01x 0001	Drive parameters may be changed. Drive function is disabled.
Switched on	xxxx xxxx x01x 0011	Drive function is disabled. Current offset calibration done.
Operation enabled	xxxx xxxx x01x 0111	No faults have been detected. Drive function is enabled and power is applied to the motor.
Quick stop active	xxxx xxxx x00x 0111	«Quick stop» function is being executed. Drive function is enabled and power is applied to the motor.
Fault reaction active	xxxx xxxx x00x 1111	A fault has occurred in the drive. Selected fault reaction is being executed.
Fault	xxxx xxxx x00x 1000	A fault has occurred in the drive. Drive parameters may have changed. Drive function is disabled.

The following example code from the Nemesys RS232 C-Library shows the test macros to test the received Statusword for a certain state:

```
// common bits in status word
#define SWBIT_SWON_RDY      0x0001 ///< ready to switch on
#define SWBIT_SWON          0x0002 ///< switched on
#define SWBIT_OP_EN         0x0004 ///< operation enabled
#define SWBIT_FAULT         0x0008 ///< fault
#define SWBIT_VOLT_EN       0x0010 ///< voltage enabled
#define SWBIT_QUICK_STOP    0x0020 ///< quick stop
#define SWBIT_SWON_DIS      0x0040 ///< switch on disabled
#define SWBIT_WARNING       0x0080 ///< warning
#define SWBIT_REMOTE        0x0200 ///< remote
#define SWBIT_TARG_REACHED  0x0400 ///< target reached
#define SWBIT_INT_LIM       0x0800 ///< internal limit active


/// Status words of object dictionary entry OD_H6041_STATUS_WORD

typedef enum eStatusWord
{
  SW_SWON_NOT_RDY  = 0x00, ///< not ready to switch on
  SW_SWON_DIS      = 0x40, ///< switch on disabled
  SW_SWON_RDY      = 0x21, ///< ready to switch on
  SW_SWON          = 0x23, ///< switched on
  SW_OP_EN         = 0x27, ///< operation enabled
  SW_QUICK_STOP    = 0x07, ///< quick stop active
  SW_FAULT_REAC    = 0x0F, ///< fault reaction active
  SW_FAULT         = 0x08, ///< fault
  SW_UNDEFINED     = 0xFF  ///< undefined state
} TStatusWord;


// statusword test macros
#define STATE_IS_FAULT(_wState_)        (((_wState_) & 0x004F) == SW_FAULT)
#define STATE_IS_FAULT_REAC(_wState_)   (((_wState_) & 0x004F) == SW_FAULT_REAC)
#define STATE_IS_QUICK_STOP(_wState_)   (((_wState_) & 0x006F) == SW_QUICK_STOP)
#define STATE_IS_SWON_NOT_RDY(_wState_) (((_wState_) & 0x004F) == SW_SWON_NOT_RDY)
#define STATE_IS_SWON_DIS(_wState_)     (((_wState_) & 0x004F) == SW_SWON_DIS)
#define STATE_IS_SWON_RDY(_wState_)     (((_wState_) & 0x006F) == SW_SWON_RDY)
#define STATE_IS_SWON(_wState_)         (((_wState_) & 0x006F) == SW_SWON)
#define STATE_IS_OP_EN(_wState_)        (((_wState_) & 0x006F) == SW_OP_EN)
```

State transitions of the drive state machine are caused by internal events in the drive or by commands from the host via Controlword. The transition numbers in the table below are the number shown in the drive state diagram.

Transition	Event	Action
0	Reset	Initialize drive
1	Drive has initialized successfully	Activate communication
2	«Shutdown» command received	
3	«Switched on» command received	Initialize current sensor. Current offset calibration.
4	«Enable operation» command received	Enable drive function (enable current controller and, if needed, position or velocity controller)
5	«Disable operation» command received	Stop movement according to «Disable operation option code». Disable drive function.
6	«Shutdown» command received	Disable power section
7	«Quick stop» or «Disable voltage» command received	
8	«Shutdown» command received	Stop movement according to «Shutdown option code». Disable drive function and power section.
9	«Disable voltage» command received	Stop movement according to «Shutdown option code». Disable drive function and power section.
10	«Quick stop» or «Disable voltage» command received	
11	«Quick stop» command received	Stop movement according to «Quick stop option code»
12	«Disable voltage» command received	Disable drive function and power section
13	A fault has occurred	Start fault reaction
14	The fault reaction is completed	Disable drive function and power section
15	«Fault reset» command received	Reset fault condition if no fault is present
16	«Enable operation» command received	

7.6.3 Device Control via Controlword

State transitions of the drive state machine are caused by internal events in the drive or by commands from the host via Controlword:

Index	Subindex	Object	Description	Type
0x6040	0	Controlword	Write to this object to control the internal device state machine	UNSIGNED16

Below are the profile position mode specific Controlword bits:

Bit	Name	Desription
15	Endless movement	Not required – set to 0
14...9	reserved	
8	Halt	0 – Execute or continue positioning 1 – Stop drive
7	Fault reset	A rising edge from 0 to 1 triggers a fault reset
6	Abs / rel	0 – Target position is an absolute value 1 - Target position is a relative value
5	Change set immediately	• 0 - Finish actual positioning, then start next positioning. The actual positioning is considered as completed as soon as the position demand value reaches the target position • 1 - Abort actual positioning and start next positioning
4	New setpoint	A rising edge from 0 to 1 indicates a new setpoint
3	Enable operation	Bits to trigger drive state machine changes
2	Quick stop	
1	Enable voltage	
0	Switched on	

State machine changes are triggered by the following bit patterns in the Controlword

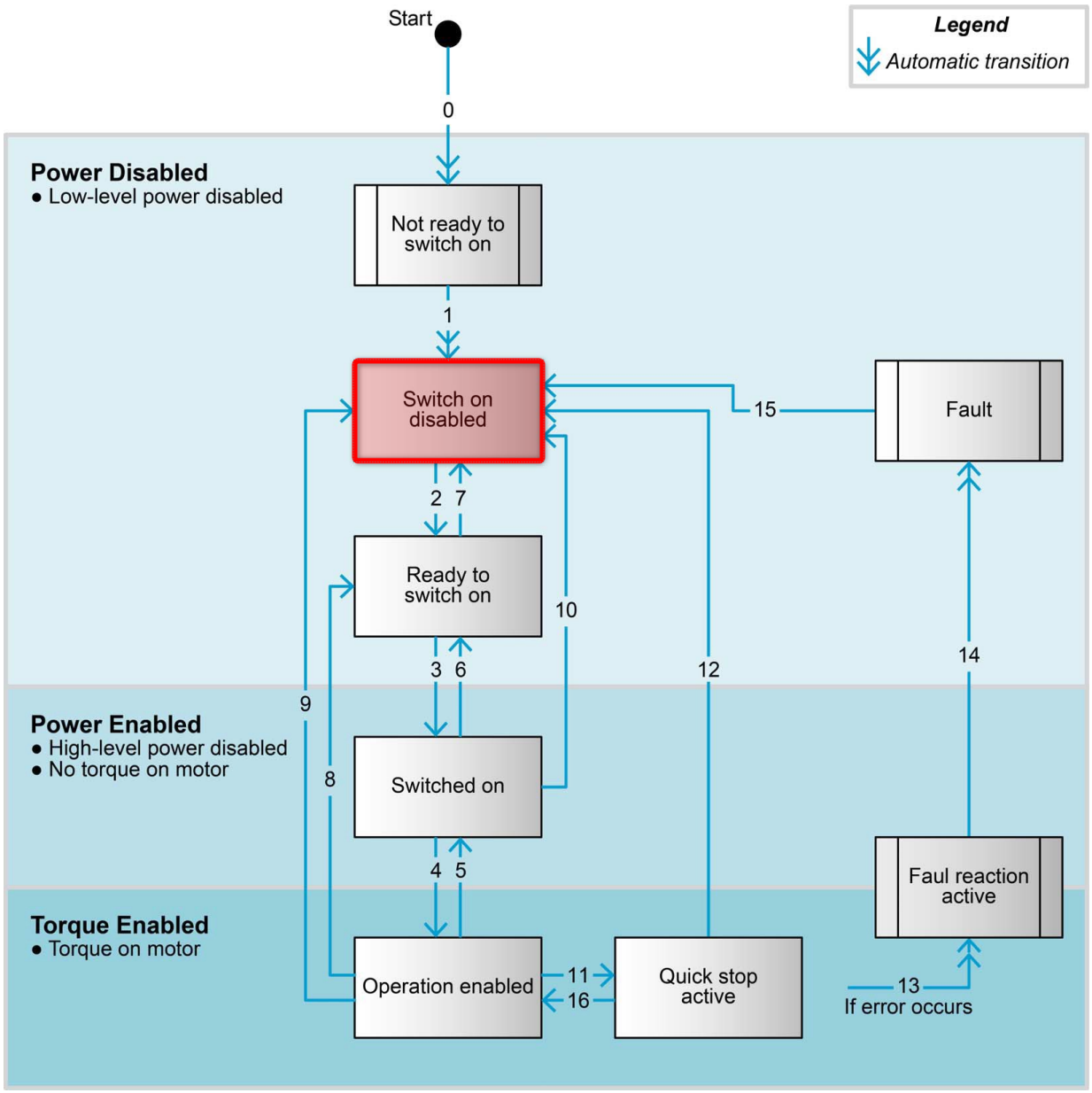
Command	Controlword LowByte [binary]	State transition (see state diagramm abov)
Shutdown	0xxx x110	2, 6, 8
Switch on	0xxx x111	3
Switch on & Enable operation	0xxx 1111	3, 4 (Automatic transition to state «Operation enabled» after execution of command «Switch on»)
Disable voltage	0xxx xx0x	7, 9, 10, 12
Quick stop	0xxx x01x	11
Disable operation	0xxx 0111	5
Enable operation	0xxx 1111	4, 16
Fault reset	0xxx xxxx → 1xxx xxxx	14, 15

Before you can move the pusher, you need to set the pump drive into *Operation Enabled* state. *Operation Enabled* means, the drive function is enabled and power is applied to the motor. Right after power on or after a reset, the drive is not in *Operation Enabled* state. To set the drive into Operation Enabled state, you need to control the internal drive state machine via the objects *0x6040 Controlword* and *0x6041 Statusword*.

To clear the fault, we need to generate a rising edge for the *Fault reset* bit in the status word. That means we write the value *0x80* to the object *0x6040 Controlword*. Here is the example code from the RS233 C-Library:

```
long NemV4ClearFault(TNemesysV4* Nemesys)
{
  long Result;

  Result = NemV4IsInFaultState(Nemesys);
  CSI_RETURN_ON_ERROR(Result);

  // If we are not in fault state, then we can return now
  if (!Result)
  {
    return ERR_NOERR;
  }

  // First we clear the error history and then we clear the fault state
  Result = NemV4_ClearErrHist(Nemesys);
  CSI_RETURN_ON_ERROR(Result);

  return NemV4_SetControlWord(Nemesys, CW_FAULT_RST); // CW_FAULT_RST = 0x80
}
```

To check, if the fault has been cleared, simply read the Statusword and check if the fault bit is 0. Now we can set the drive into enabled state. To do this, we need to send the Controlwords to trigger state machine transitions until the drive is in state *Operation Enabled*. When the fault has been cleared, then the drive statemachine should be in state *Switch on disabled*.



To bring the drive into state *Operation Enabled*, we need to trigger transition 2, 3 and 4. To trigger the transition 2 we send the Controlword *0x06 -Shutdown*. After sending this Controlword we should be in state *Ready to switch on*. You can always read the Statusword to verify that you are in the expected state.

Now we send the Controlword *0x0F - Switch on and Enable operation*. This will first trigger transition 3 to state *Switched on* and afterwards automatically trigger transition 4 into state *Operation Enabled*. Now your device is ready for the first dosing command.

The following example code from the Nemesys RS232 C-Library shows, how a *SetEnabled* function can be implemented in C:

```
long NemV4SetEnabled(TNemesysV4* Nemesys)
{
  uint16_t state;
  long Result;
  uint8_t loopcnt = 0;

  Result = NemV4ReadStatusWord(Nemesys, &state);
  CSI_RETURN_ON_ERROR(Result);

  // if drive is in fault state or in fault reaction state then we cannot enable
  // drive because the user has to clear the fault state first
  if (STATE_IS_FAULT(state) || STATE_IS_FAULT_REAC(state)) {
    return -ERR_DS402_DRV_ENABLE_FAULT_STATE;
  }

  // loop until we reach the operation enable state or until we reach the loop limit 10
  do {
    if (STATE_IS_QUICK_STOP(state)) {
      Result = NemV4_ExecDrvStateCmd(Nemesys, CW_OP_EN, &state, DRV_PROC_TIME);
    }
    else if (STATE_IS_SWON_DIS(state)) {
      Result = NemV4_ExecDrvStateCmd(Nemesys, CW_SHUTDOWN, &state, DRV_PROC_TIME);
    }
    else if (STATE_IS_SWON_RDY(state)) {
      Result = NemV4_ExecDrvStateCmd(Nemesys, CW_SWON, &state, DRV_PROC_TIME);
    }
    else if (STATE_IS_SWON(state)) {
      // if operation is enabled we set the halt bit in order to avoid a running drive
      Result= NemV4_ExecDrvStateCmd(Nemesys, CW_OP_EN | CWBIT_HALT, &state, DRV_PROC_TIME);
    }
  } while (!STATE_IS_OP_EN(state) && (loopcnt++ < 10) && (ERR_NOERR == Result));


  if (loopcnt <= 10) {
    return Result;
  }
  else {
    return -ERR_DS402_TIMEOUT_STATUSWORD;
  }
}
```

7.7 Dosing

7.7.1 Introduction

Normally all dosing tasks are performed in *Profile Position Mode*. That means for each dosing task you need to set the volume (position), the flow rate (velocity) and you need to start/stop the pump using the device Controlword.

7.7.2 Starting dosage

To start a dosage you need to access the following objects:

Index	Subindex	Object	Description
0x6040	0	Controlword	Write to this object to start / stop dosage
0x6041	0	Statusword	Read current device state and status information from this object
0x607A	0	Target Position	Defines the volume (position) for the next dosage
0x6081	0	Profile Velocity	Defines the flow rate (velocity) for the next dosage

The pump supports absolute and relative dosing. A relative movement aspirates or delivers a certain volume. The position value that you write into object *0x607A Target Position* is relative to the current position. Write a negative position value to aspirate and a positive position value to displense a certain amount of fluid. After the movement, the syringe content is increased respectively decreased, by the volume, just as a bank account has its balance increased or decreased by a credit or a debit. There is a fixed relationship between the position of the piston and the content in the syringe.

IMPORTANT. Take care, that a relative movement does not move the pusher outside of the travel range.

An absolute movement moves the piston of the syringe so that the syringe content reaches the specified value. The position value that you write into object *0x607A Target Position* is an absolute value in the travel range from *Minimum Position – Maximum Position*. The actual movement is a delivering or an aspiration as required to fulfil this purpose, or even no movement at all if the content is already equal to the specified volume.

To start dosage, you need to perform the following steps:

(1) Enable the drive if it is not enabled yet according to the description in section Device Control via Controlword

(2) Ensure that the operation mode is *Profile Position*.

(3) Convert the volume and flow values into position and velocity values according to section Translation of volume / flow units.

(4) Write the position value into object *0x607A Target Position (increments)*

(5) Write the velocity value into object *0x6081 Profile Velocity (internal velocity units according to object 0x60A9 Si unit velocity)*

(6) Start the dosage by writing to the object *0x6040 Controlword*. Set the *Abs/rel* bit in the Controlword to select absolute or relative dosing.

To start the dosage in step 6 we need to send the right Controlword. For profile position mode the following four Controlword bits are relevant:

Bit	Name	Desription
8	Halt	0 – Execute or continue positioning 1 – Stop drive
6	Abs / rel	0 – Target position is an absolute value 1 - Target position is a relative value
5	Change set immediately	• 0 - Finish actual positioning, then start next positioning. The actual positioning is considered as completed as soon as the position demand value reaches the target position • 1 - Abort actual positioning and start next positioning
4	New setpoint	A rising edge from 0 to 1 indicates a new setpoint

Because we do not want to trigger the drive state machine and we would like to keep the drive in Operation Enabled state, we always send a Controlword that is a combination of *0x0F - Switch on and Enable operation* and of the relevant profile position mode bits from the table above.

Lets do a small example: We would like to dose a certain volume, that means we need to execute a relative move – that means the *Abs / rel* bit (Bit 6) needs to be 1. We want to start the dosage immediately – that means the *Change set immediately* bit (Bit 5) needs to be 1. The Halt bit (Bit 8) needs to be cleared in order to enable positioning.

To signal a new setpoint, we need to generate a rising edge for the Bit 4 – *New setpoint*, To do this, we fist send the Controlword 0*x0F - Switch on and Enable operation* to clear the *New setpoint* bit. Now we can build our final Controlword and send it. To build the final Controlword we need to logically or the following values:

- 0x0F – Switch on and Enable operation
- 0x40 – Abs / rel (Bit 6)
- 0x20 – Change set immediately (Bit 5)
- 0x10 – New setpoint (Bit 4)

That means, sending the final Controlword 0x7F will start the motion.

Read the section *5.3 Profile Position Mode* in EPOS4-Firmware-Specification.pdf document for a detailed description of the Controlword bits and how to start / stop positioning.

HINT. You only need to write Target Position and Profile Velocity objects, if you want to change the values. If you would like to perform multiple dosing tasks with the same volume or the same flow rate, then you only need to trigger the Controlword.

The following C-Code from the Nemesys RS232 library shows, how to start a dosage. The code starts from point (4) of the steps above – that means, the drive is already enabled and in Profile Position Mode and the volume and flow values are already converted into position and velocity values:

```
long NemV4MoveToPos(TNemesysV4* Nemesys, int32_t dwPosAbs, uint32_t dwVelocity)
{
  long     Result;
  uint16_t  wControlWord;

  // first write the target position
  Result = CsiDevWriteObject(&Nemesys->Device, NemV4_OD_H607A_TARGET_POSITION,
    0, dwPosAbs);
  CSI_RETURN_ON_ERROR(Result);

  // Then write the profile velocity
  Result = CsiDevWriteObject(&Nemesys->Device, NemV4_OD_H6081_PROFILE_VELOCITY,
    0, dwVelocity);
  CSI_RETURN_ON_ERROR(Result);

  // Now start movement. The movement is started by a rising edge of the
  // control word bit CWBIT_NEW_SET_POINT. Therefor we first write a
  // controlword with the bit cleared and the the second control word with the
  // bit set
  wControlWord = CW_OP_EN;
  Result = NemV4_SetControlWord(Nemesys, wControlWord);
  CSI_RETURN_ON_ERROR(Result);
  wControlWord = CW_OP_EN | CWBIT_NEW_SET_POINT | CWBIT_IMMEDIATELY;
  return NemV4_SetControlWord(Nemesys, wControlWord);
}
```

7.7.3 Stopping dosage

To stop a running dosage set the *Halt* bit in the *Controlword*. The following C source code from the Nemesys RS232 library shows, how to stop a dosage:

```
long NemV4Stop(TNemesysV4* Nemesys)
{
  long     Result;
  uint16_t  ControlWord = CW_OP_EN | CWBIT_HALT;

  Result = NemV4_SetControlWord(Nemesys, ControlWord);
  return Result;
}
```

7.8 Valve Switching

If there is an external valve connected to the Nemesys I/O interface, you can switch the valve position by writing to the digital outputs object 0x60FE.

Index	Subindex	Object	Description	Type
0x60FE	1	Physical outputs	Read / write the state of all digital outputs	UNSIGNED32

The external valves are connected to the Nemesys digital outputs 1 and 2 which are mapped to the bits 16 and 17 of the digital outputs object.

Bit	Method
16	Digital output 1 - General Purpose Out A
17	Digital output 2 - General Purpose Out B

If you use a valve with only one switching position, such as the CETONI 3/2 way smartvalve, then you only need to switch the digital output 1. If you use a valve with more switching position, such as the CETONI 3-4 Way Contiflow Valve, then you need to switch both digital outputs. The following table lists some valves and their switching positions:

Valve Type	Valve Position	Dig. Out 1	Dig. Out 2	Bitmask (Bits 16 and 17)
3-2 Way Smartvalve	Port 1	0	-	0x00000000
	Port 2	1	-	0x00010000
3-4 Way Contiflow Valve	Closed	0	0	0x00000000
	Port 1	1	0	0x00010000
	Port 2	0	1	0x00020000
	Both ports open	1	1	0x00030000
3-3 Way Contiflow Ball Valve	Closed	0	0	0x00000000
	Port 1	1	0	0x00010000
	Port 2	0	1	0x00020000

IMPORTANT. If you write the digital outputs, ensure that you only modify the bits 16 and 17 that are relevant for valve switching. That means you either need to read the value before you change bits or you need to keep an internal shadow register.

7.9 Reading Analog Inputs

The Nemesys syringe pumps have two analog inputs. The analog input 1 is routed to the Nemesys I/O interface. You can use it, to read the pressure of an external pressure sensor or to read the pressure of the pressure sensor of a connected Contiflow valve.

The second analog input is used internally for measuring the internal force sensor.

Index	Subindex	Object	Description	Type
0x3160	1	Analog input 1 voltage	The voltage measured at analog input 1 [mV].	INTEGER16
0x3160	2	Analog input 2 voltage	The voltage measured at analog input 2 [mV].	INTEGER16

The returned value is the measured voltage in mV in the range from 0 – 10.000 mV. If there is a sensor connected to one of the inputs, you just need to translate the voltage value into the sensor value.

7.10 Force Monitoring

7.10.1 Overview

The Nemesys S and Nemesys M syringe pumps have an internal force sensor for force monitoring and safety stop in case of too high forces (pressures). The following table shows the device force limits for both pumps:

Pump Type	Device Force Limit (N)
Nemesys M	1300
Nemesys S	480

If the measured force raises above the force limit, the pump stops immediately with a quick stop.

7.10.2 Reading Internal Force Sensor

The Nemesys pumps have an internal force sensor that is connected to the internal analog input 2. To read the force sensor, you just need to read the analog input 2 value like written in section Reading Analog Inputs and then convert this value into a force value.

Pump Type	Force Sensor Scaling					
	x1 (mV)	y1 (N)	x2 (mV)	y2 (N)	Factor m	Offset n
Nemesys M	0	0	5650	1000	0,176991	0
Nemesys S	0	0	3000	400	0,133333	0

To convert between the analog input value and the force sensor value, you can use this formula:

$$\text{Force (N)} = \text{Analog Value (mV)} * y2\ (\text{N}) / x2\ (\text{mV})$$

That means, if you measure a voltage of 2800 mV in your Nemesys S syringe pump, then the internal force sensor measures the following force value:

$$\text{Force} = 2800\ \text{mV} * 400\ \text{N} / 3000\ \text{mV} \approx 267\ \text{N}$$

7.10.3 Setting a custom force limit

It is possible to reduce the force limit below the maximum device force by setting a custom force limit if this is required by your application. Setting a custom force limit can be done, by writing the analog output value 2. The analog output 2 is a threshold value for the internal force monitoring logic.

Index	Sub	Object	Description	Type
0x3182	2	Analog output general Purpose B	Output voltage in mV (-4000 mV - +4000 mV)	INTEGER32

To set a force limit, you need to translate the force into an output voltage. The following table lists the scaling factors depending on the device type:

Pump Type	Force Limit Scaling					
	x1 (mV)	y1 (N)	x2 (mV)	y2 (N)	Factor m	Offset n
Nemesys M	1650	1000	3920	0	-0,440529	1726,872
Nemesys S	740	400	1940	0	-0,333333	646,667

With the scaling parameters x1, y1, x2 and y2 you can calculate the scaling factor **m** and scaling offset **n**.

$$m = (y2 - y1) / (x2 - x1)$$
$$n = y1 - m * x1$$

Now you can convert a force value into a voltage value with the formula y = m * x + n. To convert from force into voltage, we simply solve the equation to x. That means, we can convert a force value into a voltage value with the following formula:

$$x = (y - n) / m$$

Let's do an example. We calculate the conversion factor and offset for a Nemesys M pump (or we simply take the values from the table above):

Scaling factor **m** = (y2 – y1) / (x2 – x1) = (0- 1000) / (3920 – 1650) = **-0,440529**
Scaling offset **n** = y1 – m * x1 = 1000 + 0,440529 * 1650 = **1726,872**

Now we can use **m** and **n** for conversion. We would like to limit the force to 500 N.

$$\text{Voltage (mV)} = (\text{Force} - n) / m = (500\ \text{N} - 1726{,}872) / -0{,}440529 = 2785$$

Now we can write the voltage value 2785 into the object 0x3182 Subindex 2.

IMPORTANT. Only force limits in the range from 0 to the maximum device force are allowed. That means, for Nemesys M the allowed force range is 0 – 1300 N and for Nemesys S the allowed force range is 0 – 480 N.

7.10.4 Reading Safety Stop Input

In case of a force that is higher than the configured force limit or higher than the maximum device force, a safety stop is executed. That means, the pump drive is stopped to prevent damage by too high forces. You can monitor the safety stop input to know, when a safety stop occurs.

Index	Sub	Object	Description	Type
0x60FD	0	Digital Inputs	Displays the state of the digital input functionalities	UNSIGNED32

The safety stop input is mapped into bit 28 of the digital input bitfield. So if you would like to check if the safety stop is active, read the digital inputs object *0x60FD Subindex 0* and test for bit 28.

7.10.5 Enable / Disable Force Monitoring

If the internal force monitoring detects a force higher than the force limit, the pump drive is stopped immediately with the safety stop input. It is not possible to move the pump in this state. To lower the force below the force limit, the force monitoring needs to get disabled. Enabling / Disabling the force limit is possible by writing to the object *0x3141 Subindex 2 Digital inputs polarity*.

Index	Sub	Object	Description	Type
0x3141	2	Digital inputs polarity	Used to set the polarity of the digital input functionalities.	UNSIGNED16

To enable / disable force monitoring, you need to set bit 0 of the digital inputs polarity mask as follows:

- **Enable** force monitoring: set bit 0 to **0**
- **Disable** force monitoring: set bit 0 to **1**

To avoid changing any other bits in the polarity mask, you fist need to read the current polarity mask from the device, then change bit 0 in you software and then write the new polarity mask back to the device. The following example C source code from the Nemesys RS232 library shows this:

```
long NemV4EnableForceMonitoring(TNemesysV4* Nemesys, int Enable)
{
  long Result;
  uint32_t InputsPolarity;

  Result = CsiDevReadObject(&Nemesys->Device, NemV4_OD_H3141_DIG_INPUTS_PROPERTIES,
    2, &InputsPolarity);
  CSI_RETURN_ON_ERROR(Result);

  InputsPolarity &= ~0x01;
  if (!Enable)
  {
    InputsPolarity |= 0x01;
  }

  return CsiDevWriteObject(&Nemesys->Device, NemV4_OD_H3141_DIG_INPUTS_PROPERTIES,
    2, InputsPolarity);
}
```

7.10.6 How to resolve a force overload situation

If the internal force monitoring detects a force higher than the force limit, the pump drive is stopped immediately with the safety stop input. It is no longer possible to carry out any dosing with the pump in this state. To resolve this force overload situation, you need to reduce the force below the force limit threshold. There are different ways to reduce the pressure:

1. by using some kind of overpressure valve
2. by waiting until the pressure is released
3. by pulling back the syringe plunger (aspirating)

The force monitoring functionality has a hysteresis. In case of a force overload situation the current force needs to be ca. 0.1 kN lower then the current force limit to reset the safety stop input. If you have an overpressure valve or some switch to release the pressure, you can lower the force without moving the syringe pump pusher. If it is not possible for you, to lower the force until it is 0.1 kN below the force limit, then you can lower it less (for example 0.02 kN) and then elevate the force limit for a short time by a fixed offset (such as 0.09 kN) to overcome the hysteresis of 0.1 kN and to clear the safety stop input. The following example code from the Nemesys RS232 Library function `NemV4ClearForceSafetyStop` shows this:

```
long NemV4ClearForceSafetyStop(TNemesysV4* Nemesys)
{
  long Result;
  float ForceLimit_kN;
  float ElevatedForceLimit_kN;

  // First we read the current force limit
  Result = NemV4ReadForceLimit(Nemesys, &ForceLimit_kN);
  CSI_RETURN_ON_ERROR(Result);

  // Now we elevate the force limit to compensate for the hysteresis
  ElevatedForceLimit_kN = ForceLimit_kN + ElevatedForceOffset_kN;
  ElevatedForceLimit_kN = (ElevatedForceLimit_kN > Nemesys->MaxForce_kN)
    ? Nemesys->MaxForce_kN : ElevatedForceLimit_kN;
  Result = NemV4WriteForceLimit(Nemesys, ElevatedForceLimit_kN);
  CSI_RETURN_ON_ERROR(Result);
  CsiSleep(50); // give device some time to process the change

  // Now we restore the previous force limit
  return NemV4WriteForceLimit(Nemesys, ForceLimit_kN);
}
```

If you have no option to reduce the pressure via a switch or a valve, then you need to reduce the force by pulling the pusher. You can only move the pusher, if you disable force monitoring. That means, before you can reduce the force by pulling the pusher, you need to disable force monitoring.

ATTENTION. If force monitoring is disabled, you should only aspirate and never dispense to avoid damaging your syringe or the device!

To resolve a force overload situation by pulling the syringe pusher, you should follow the steps below:

(1) Disable force monitoring like written in section Enable / Disable Force Monitoring

(2) Command an aspiration with a low flow rate to lower the force

(3) As soon as the force drops below the force limit, the force monitoring stops the pump drive via the safety stop input. Just monitor the safety stop input, to know, when the force dropped below the force limit.

(4) If it is no possible for you, to lower the force 0.1 kN below the current force limit, then you can

elevate the force limit for a short time (some milliseconds) to clear the safety stop input

(5) Enable force monitoring like written in section Enable / Disable Force Monitoring

(6) Now the pump is ready for the next dosing commands

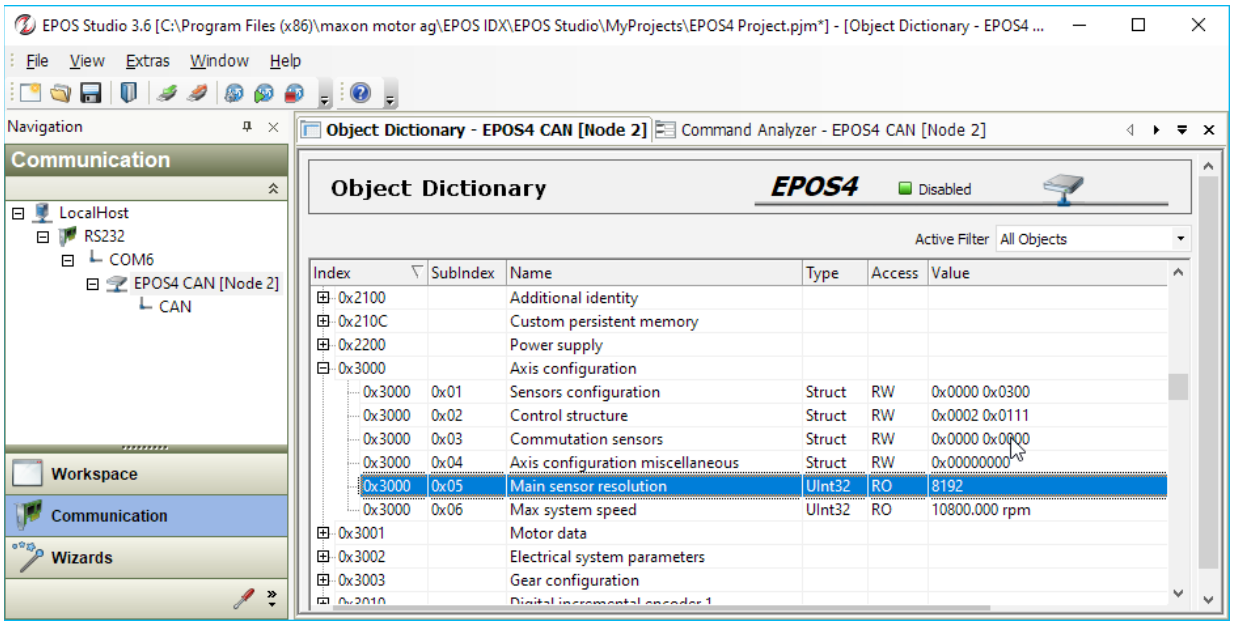
8 Development Tools

8.1 Tools for RS232 Protocol Implementation

Implementing the RS232 industrial protocol with CRC checksum is somewhat more difficult than implementing a simple ASCII protocol. To simplify and speed up the implementation, find errors in the protocol implementation or to monitor the serial frames, we recommend the following tools:

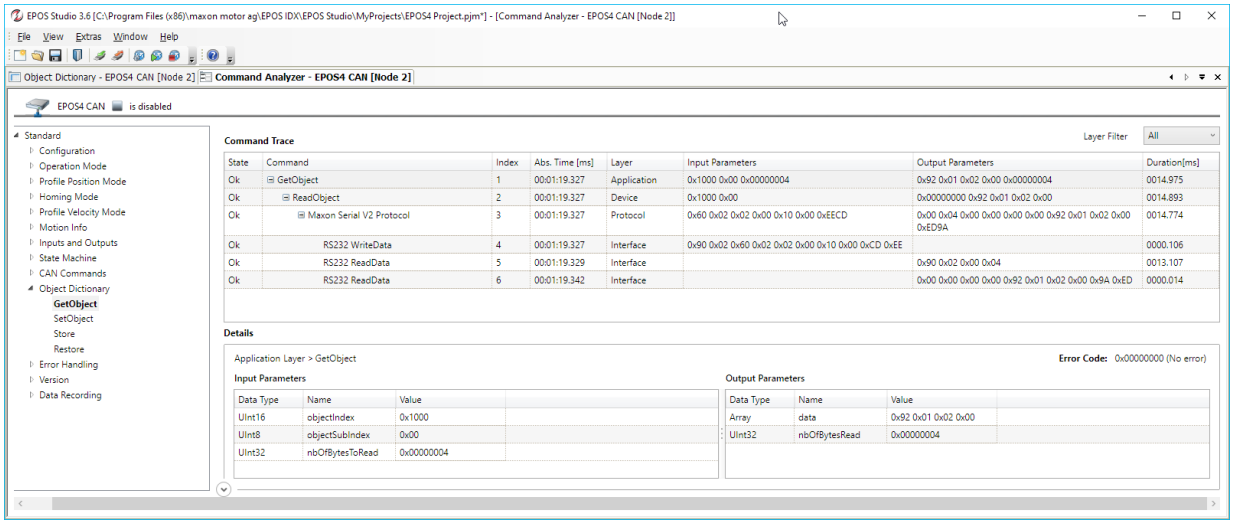
8.1.1 EPOS Studio

The EPOS Studio software is a powerful tool for access to all device parameters of the pump drive via RS232 or CAN interface. With the EPOS Studio Object Dictionary Tool it is possible to read and write entries of the CANopen object dictionary. With this tool you can modify parameters or verify, if your implementation has properly read or written certain parameters:



The EPOS Studio Command Analyzer will help you to analyze the low level RS232 protocol including

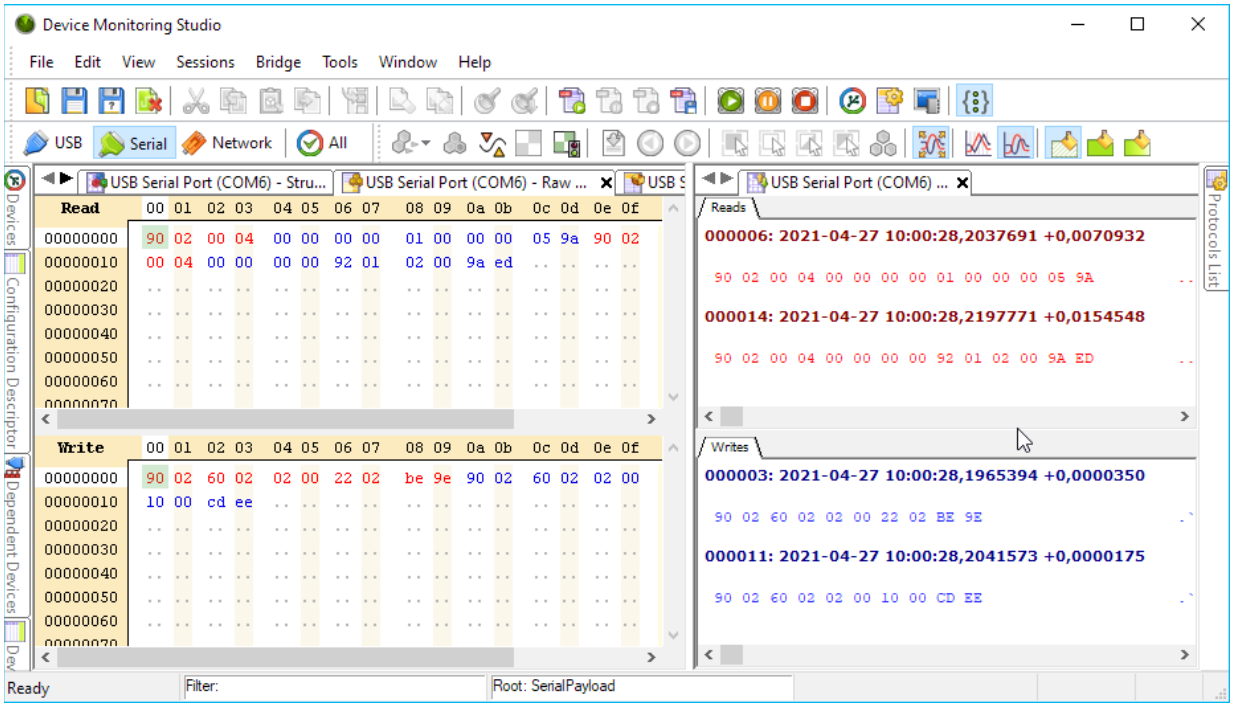
checksum calculation. With this tool you can execute certain commands and access the object dictionary and you will see the corresponding serial protocol frames including CRC and stuff bytes.



You can download the EPOS Studio software here: Download.

8.1.2 Serial Port Monitor

With a serial port monitor, you can monitor the low level data frames on the serial line.



This helps you to see and understand the RS232 frame structure and RS232 checksum calculation. It will also help you, to find and trace errors in your serial protocol implementation. At CETONI we use this serial port monitor: https://www.hhdsoftware.com/serial-port-monitor.

8.1.3 Nemesys V4 RS232 Library Documentation

The Nemesys RS232 Library is an open source implementation of the industrial RS232 protocol in plain C language. The library is well structured and well documented. If you understand C language a little bit, then this library will be a valuable helper for you and you can use it as a template for your implementation. You can browse the online documentation here.

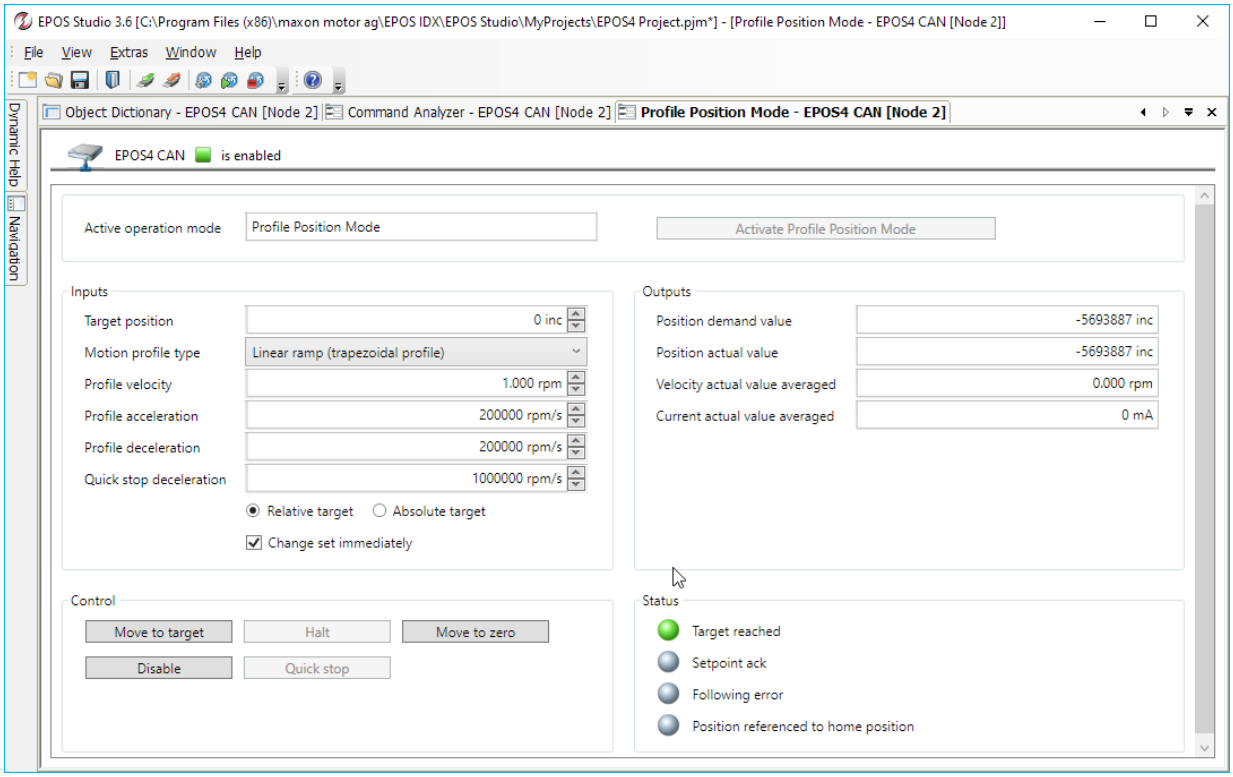
If you would like to learn about the low level serial serial protocol implementation, then you should look into the CSI library. If you would like to learn about the implementation of the Nemesys V4 pump functionality, then you should look into the Nemesys V4 API of the Nemesys RS232 Library.

8.2 Tools for CANopen implementation

8.2.1 EPOS Studio

The EPOS Studio software is a powerful tool for access to all device parameters of the pump drive via RS232 or CAN interface. With the EPOS Studio Object Dictionary Tool it is possible to read and write entries of the CANopen object dictionary. With this tool you can modify parameters or verify, if your implementation has properly read or written certain parameters.

The software also allows you to execute positioning commands via its graphical interface.



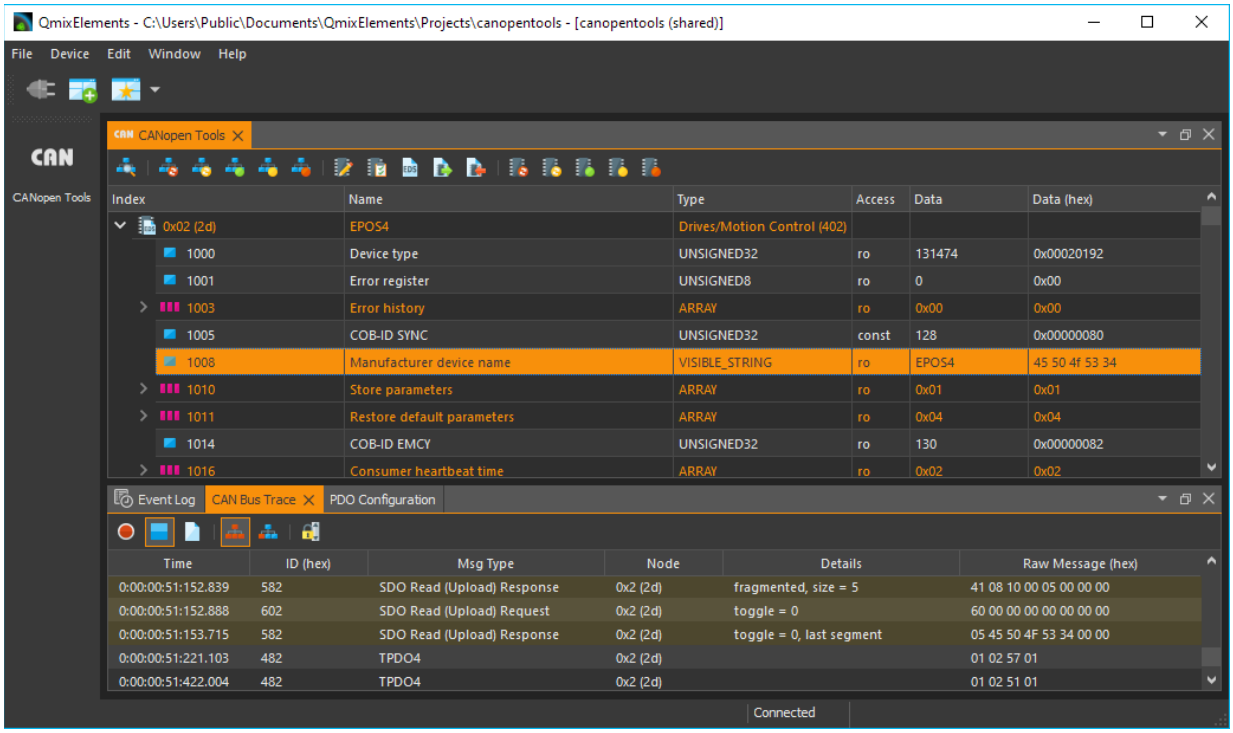
You can download the EPOS Studio software here: Download.

8.2.2 CETONI Elements CANopen Tools Plugin

The CETONI Elements software from CETONI has an CANopen-Tools Plugin which transforms the software into a powerful tool to configure CANopen devices, access the CANopen object dictionary of the Nemesys pumps and to monitor, log and analyze the CAN-Bus traffic and the CANopen protocol of the pumps.

This tool will help you to read and write object dictionary entries and to monitor the CAN-bus traffic of

your PLC, PC or embedded control device connected to the Nemesys pumps.



Read the section *CANopen Tools Workbench* in the CETONI Elements manual to learn how to open and use this tool.