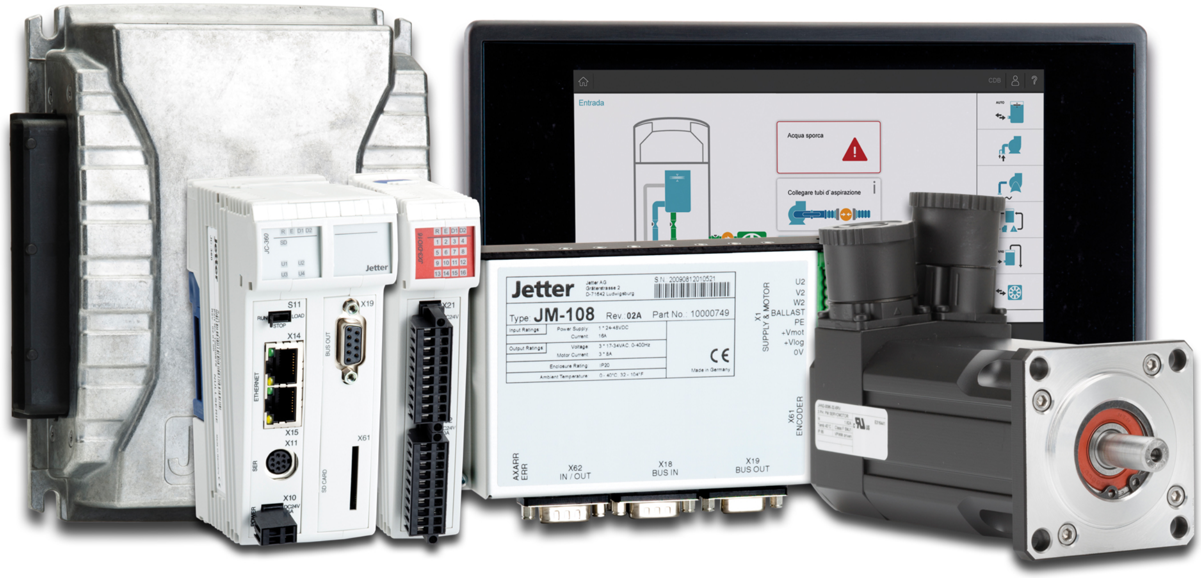


# Application-Oriented Manual

CANopen® STX API

60881083

We automate your success.

Item # 60881083

Revision 1.01

April 2017 / Printed in Germany

# Table of Contents

# 1 CANopen®

**The CANopen® standard**

CAN (Controller Area Network) was developed for data transmission in vehicles in the mid-eighties. Data transmission takes place on a serial data bus within real-time applications. In 1995, the specifications made so fare were handed over to the CiA e.V. (CAN in Automation association). From then on, the standard has been maintained and further developed within the framework of the CAN association. Since 2002, it has been available as a European standard (EN 50325-4 2002 Part 4: CANopen).

The hardware having got a CAN transceiver and a CAN controller to ISO 11898 is a prerequisite of applying CANopen®.

The CANopen® standard describes data interchange in a CAN-based network. According to this standard, both the basic communication mechanism (communication profile) and the functioning of the communicating devices (device profile) have been defined. This means that also the interpretation of process data that are being transmitted via bus is set under CANopen®.

**Documentation**

The CANopen® specifications can be obtained from the **CiA e.V. http://www.can-cia.org** homepage. The key specification documents are:

- CiA DS 301 - This document is also known as the communication profile and describes the fundamental services and protocols used under CANopen®.
- CiA DS 302 - Framework for programmable devices (CANopen® Manager, SDO Manager)
- CiA DR 303 - Information on cables and connectors
- CiA DS 4xx - These documents describe the behavior of a number of device classes in, what are known as, device profiles.

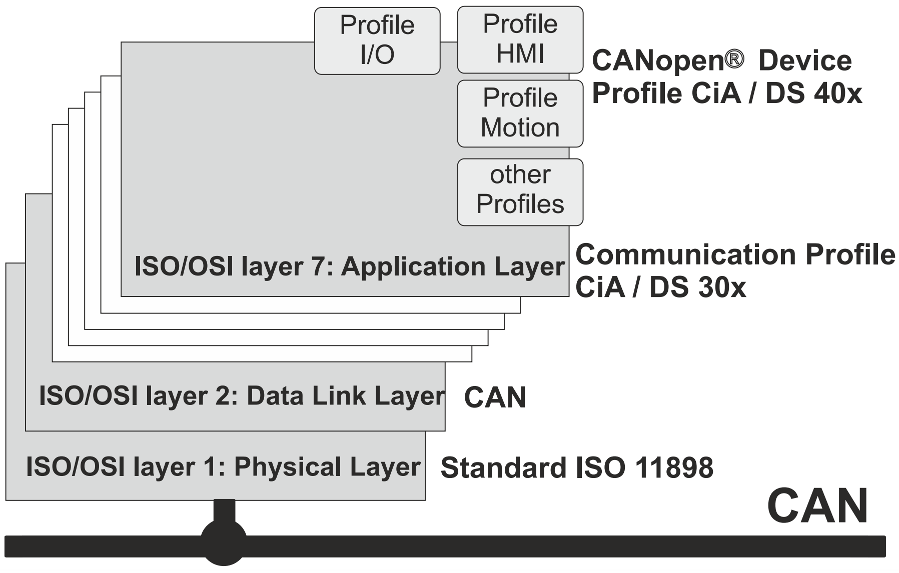
**Structural model of CANopen®**

The CANopen® protocol makes use of the CAN bus as transmission medium and sets the basic structures for network management, the usage of the CAN identifiers (message address), the timing behavior on the bus, the way of data transmission and user-specific profiles.

CANopen® defines the application layer as the common communication profile specified by the CiA within the standard DS 30x. It determines the individual communication pathway. As it is the case with some other field buses as well, a difference is made between real-time data and parameter data.

CAN has only been standardized for ISO-OSI layers 1 and 2 defined in ISO 11898.

# 1 CANopen®


Profile I/O
Profile HMI
Profile Motion
other Profiles
CANopen® Device Profile CiA / DS 40x
ISO/OSI layer 7: Application Layer
Communication Profile CiA / DS 30x
ISO/OSI layer 2: Data Link Layer
CAN
ISO/OSI layer 1: Physical Layer
Standard ISO 11898
CAN

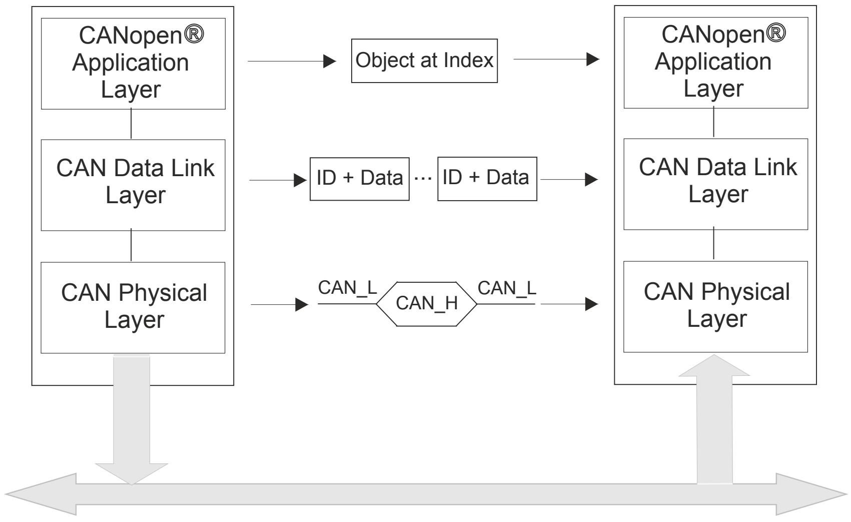
**Contents**

## Reference model

**CANopen® reference model**

The communication concept can be described in a similar way as the ISO-OSI reference model.

The following illustration shows the layers involved:



**Application layer**

The application layer provides a concept for configuration and communication of real-time data and of mechanisms for synchronizing various devices.

The functions being supplied by the application layer of an application have been distributed logically amont various service objects in the application layer. One service object offers one specific feature including all related services. These services are described in the service specification of the respective service object.

To make various applications interact, call the services of a service object in the application layer.  To physically establish these services, the respective object interchanges data with one or several other service objects via CAN network by means of a protocol. This protocol is described in the protocol specification of the related service object.

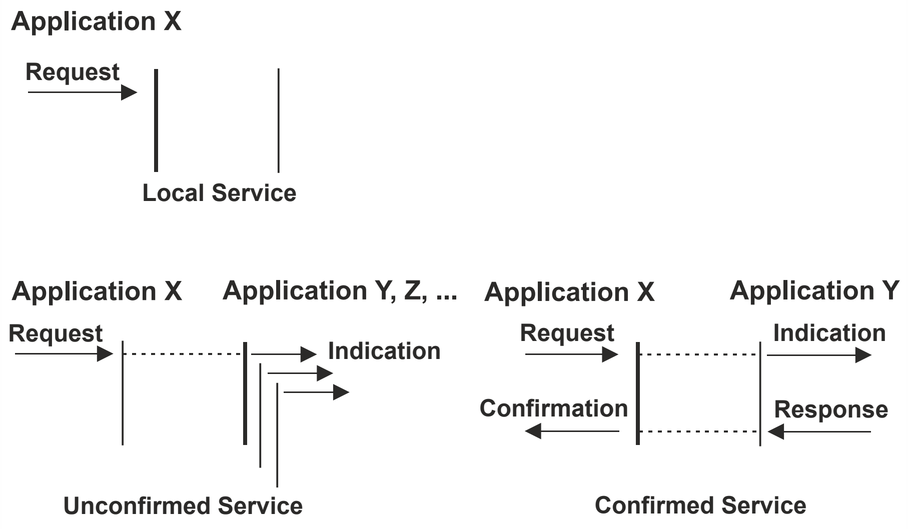
**Service elements**

By means of service elements, the application interacts with the application layer. There are four different types of service elements:

- The application issues a request for a service to the application layer.
- The application layer issues an indication to the application to signalize a service being requested.
- The application issues a response to the application layer, in order to reply to an indication received.
- The application layer issues a confirmation to the application, in order to signalize the event of a request being issued before.

**Service types**

A service type defines the service elements being exchanged between the application layer and the corresponding applications for a certain service of a service object.



- A *Local Service* only comprises the local service object. The application issues a request via its local service object. The requested service is executed without communicating with one or several partner service objects via CANopen® bus.

  This local service is performed, for example, in Jetter AG products if a CANopen® device changes its state in the state machine from, for example, pre-operational to operational.
- An *Unconfirmed Service* comprises one or more than one partner service objects. The application issues a request to its local service object. This request is forwarded to the partner service objects. These forward each request as an indication to its corresponding application. The result is not confirmed.

  This unconfirmed service is performed, for example, in Jetter AG products if PDOs (process data objects) are to be sent.
- A *Confirmed Service* can only comprise one partner service object. The application issues a request to its local service object. This request is forwarded to the partner service object. It forwards the request in the shape of an indication to the receiving application. The receiving application issues a response. The response is caused to the root service object, which, in turn, issues a confirmation to the requesting application.

  This confirmed service is performed, for example, in Jetter AG products if SDOs (service data objects) are to be sent.

## Data interchange via CAN bus

**Principle**

At CAN bus data transmission, no devices are addressed. Yet, the content of a message is flagged by an unambiguous identifier.

Besides flagging the content, the identifier also sets the priority of the message.

If any device is to send a message, it transmits both the message and the identifier to the CAN controller. The CAN controller takes on transmitting the message. If it is the only sending device at a certain moment, or if the message sent has got highest priority, all other controllers in the network will receive this message. The receiving CAN controller already decides whether this message is needed for its individual device. In order to perform the selection, the CAN controller is told during initializing, which messages must be assigned to the device. If the received message is not relevant for a device, it is ignored by its CAN controller.

Messages are transmitted bit by bit on a differential line pair (CAN high and CAN low line). In this case, there are two different states (dominant = 0 and recessive = 1) for bit information.

**Arbitration in the CAN network**

In principle, all devices at the CAN bus have got the same rights. In the bus, question/response behavior is not provided. Rather, each device is to trigger data transfer by itself. Arbitration is to be executed within a message and without destroying it.

On the bus, the level "active" being marked by "0" in the CAN frame is called dominant and the level "passive" marked by "1" in the CAN frame is called recessive. As a rule, a dominant level overrides a recessive bus state. This means that a device which is going to send a recessive level to the bus will be overridden by a dominant sending device.

As a rule, each device at the CAN bus overhears messages to be sent via the bus. Transmitting may only be started, if the bus is not occupied by a CAN frame at that instance. At transmitting, the present state of the bus is always compared with the own transmission frame.

If several controllers start transmitting simultaneously, the first dominant bit on the line determines prioritization of the message (dominance is given priority).

If a device which is going to write a recessive level to the bus recognizes that there is a dominant level on the bus, it interrupts its own transmission procedure to retry later. This way, the message of higher priority (the message of the lowest identifier) is kept on the bus in error-free state.

**Demands on the CAN network**

Arbitration results in the following demands:

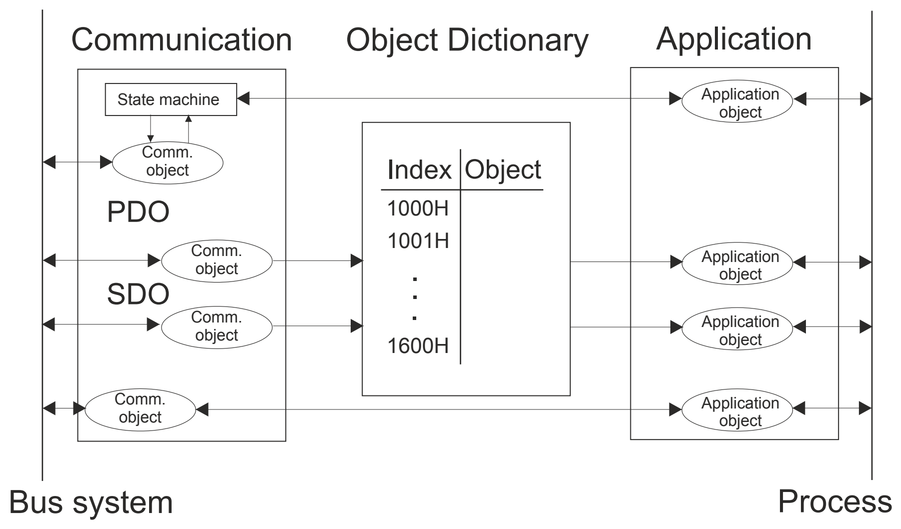
- 0 in a CAN frame represents the dominant level on the bus. This means that CAN identifiers of lower numbers have got higher priority.

  Thus, the more a message sent via the bus is prioritized, the lower must be the value of the identifier applied.
- Prioritizing is always carried out within a bit. This means that all network nodes must be within the line delay of 1 bit time (more precisely: 3/4). This relates to the forward and return travel of the signal.
- Because arbitration must take place within the identifier, there must be one sender per each identifier. This identifier, though, may be received by several devices at the CAN bus.

## Device model

**Device model**

The model of a CANopen® device is shown below.



It consists of the following elements:

- Communication:
  The functional unit lets communication objects and the related function transport data items via the basic network structure.
- Object dictionary:
  The object dictionary is a collection of all data items which influence the behavior of the application and communication objects as well as the state machine on this device.
- Application:
  In this context, application is the function range of the device regarding its interaction with the process environment.

## Object dictionary

**Introduction**

In the object dictionary, all variables and parameters (objects) of a CANopen® device are assembled. There, the process map is applied to the data. By means of the parameters, the functioning of a CANopen® device can be influenced.

**Configuration**

An object dictionary is structured in a way that several parameters for all devices of this category are obligatory, while others can be freely defined and used.

In CANopen®, the objects are assigned a number (the so-called index), by which they can be unambiguously identified and addressed. The objects can be simple data types, such as, for example, bytes, integers, longs or strings. In case of more complex structures, such as, for example, arrays and structures, a subindex is applied for addressing the individual elements.

The structure of the object dictionary, the assignment of the index numbers, and some obligatory entries are specified in the device profiles.

**The usage of index and subindex**

By means of a 16-bit index, any entry into the object library can be called. In case of a simple variable, the index directly relates to the value of this variable. In case of data records and arrays, though, the index addresses the entire data structure.

A subindex is defined, in order to be able to address individual data structure elements via the network. For individual entries in the object dictionary, such as, for example, UNSIGNED8, BOOLEAN, INTEGER32, etc., the subindex value is always zero. In case of complex entries into the object dictionary, such as, for example, arrays or data records of several data fields, the subindex relates to fields within a data structure, to which the index relates. The fields addressed by the subindex can consist of various data types.

**EDS file**

For the user, the object dictionary has been stored as EDS (Electronic Data Sheet) file. This EDS file contains all objects with their respective index, sub-index, name, data type, default value, minimum and maximum value and access options (read/write, transmission via SDO only, transmission via PDO, etc.).

This means that in an EDS file the entire function range of a CANopen® device is stored.

**Basic assignment of the object index numbers**

Below, an overview of the default object dictionary is given.

| Index (hex) | Function |
|---|---|
| 0000 | Unassigned |
| 0001 - 001F | Static Data Types |
| 0020 - 003F | Complex Data Types |
| 0040 - 005F | Manufacturer Specific Complex Data Types |
| 0060 - 007F | Device Profile Specific Static Data Types |
| 0080 - 009F | Device Profile Specific Complex Data Types |
| 00A0 - 0FFF | Reserved for later use |
| 1000 - 1FFF | Communication Profile Area |
| 2000 - 5FFF | Manufacturer Specific Profile Area |
| 6000 - 9FFF | Standardized Device Profile Area |
| A000 - BFFF | Standardized Interface Profile Area |
| C000 - FFFF | Reserved for later use |

## CANopen® communication

**Introduction**

Data interchange in CANopen® takes place via frames, by which the application data are transferred. For this, the service data objects (SDO), which serve data interchange with the object dictionary, and process data objects (PDO), which serve information interchange on the respective process states must be distinguished. Further, frames for network management and for error messages are defined.

**SDO and PDO - a comparison**

Generally, all entries of the object dictionary can be accessed via SDOs. In practice, SDOs are mostly used for initializing only during bootup. Within an SDO, only one object can be accessed. As a rule, SDOs are answered.

In principle, PDOs are a summary of objects (variables, respectively parameters) taken from the object dictionary. In a PDO, there can be 8 bytes max., which can consist of several objects, The technical expression is that the objects are mapped into a PDO.

| PDO (Process Data Object) | SDO (Service Data Object) |
|---|---|
| Real-time data | System parameters |
| No reply is transmitted to the frame (faster transmission) | A reply is transmitted to the frame (slower transmission) |
| High-priority identifiers | Low-priority identifiers |
| 8 bytes max. per frame | Data are distributed to several frames |
| Previously agreed on data format | Indexed data addressing |

**Further communications channels**

For network management and error messages, there are the following predefined logic communication channels:

- Communication objects for boot-up (i.e. network startup)
  Startup, stopping, reset of a node, etc.
- Communication objects for dynamic distribution of identifiers to DBT (Distributor)
- Communication objects for nodeguarding and lifeguarding - this way, the network can be monitored
- One communication object for synchronizing
- Communication objects for emergency messages (Emergency)

These have been set in CANopen®. They have got the characteristics of global broadcast (Broadcast).

**Default identifier distribution**

For CANopen® the following identifier distribution is predefined: In this case, the node number is embedded in the identifier.

| 11-bit identifier (binary) | Identifier (decimal) | Identifier (hexadecimal) | Function |
|---|---|---|---|
| 000000000000 | 0 | 0 | Network management |
| 000100000000 | 128 | 80h | Synchronization |
| 0001xxxxxxxx | 129 - 255 | 81h - FFh | Emergency |
| 0011xxxxxxxx | 385 - 511 | 181h - 1FFh | PDO1 (tx) |
| 0100xxxxxxxx | 513 - 639 | 201h - 27Fh | PDO1 (rx) |
| 0101xxxxxxxx | 641 - 767 | 281h - 2FFh | PDO2 (tx) |
| 0110xxxxxxxx | 769 - 895 | 301h - 37Fh | PDO2 (rx) |
| 0111xxxxxxxx | 897 - 1023 | 381h - 3FFh | PDO3 (tx) |
| 1000xxxxxxxx | 1025 - 1151 | 401h -47Fh | PDO3 (rx) |
| 1001xxxxxxxx | 1153 - 1279 | 481h - 4FFh | PDO4 (tx) |
| 1010xxxxxxxx | 1281 - 1407 | 501h - 57Fh | PDO4 (rx) |
| 1011xxxxxxxx | 1409 - 1535 | 581h - 5FFh | Send SDO |
| 1100xxxxxxxx | 1537 - 1663 | 601h - 67Fh | Receive SDO |
| 1110xxxxxxxx | 1793 - 1919 | 701h - 77Fh | NMT Error Control |
| xxxxxxxx = Node number 1 - 127 | | | |

**Note on default identifier distribution**

By means of the PDOx (tx) function, a device connected to the CANopen® bus can request another device connected to the bus to send a PDO frame with the same identifier and the desired data. This PDO frame will then be read by the requesting device.

By means of the PDOx (rx) function, a device connected to the CANopen® bus can request another device connected to the bus to read this PDO frame sent with the request and the data.

## The process data object PDO

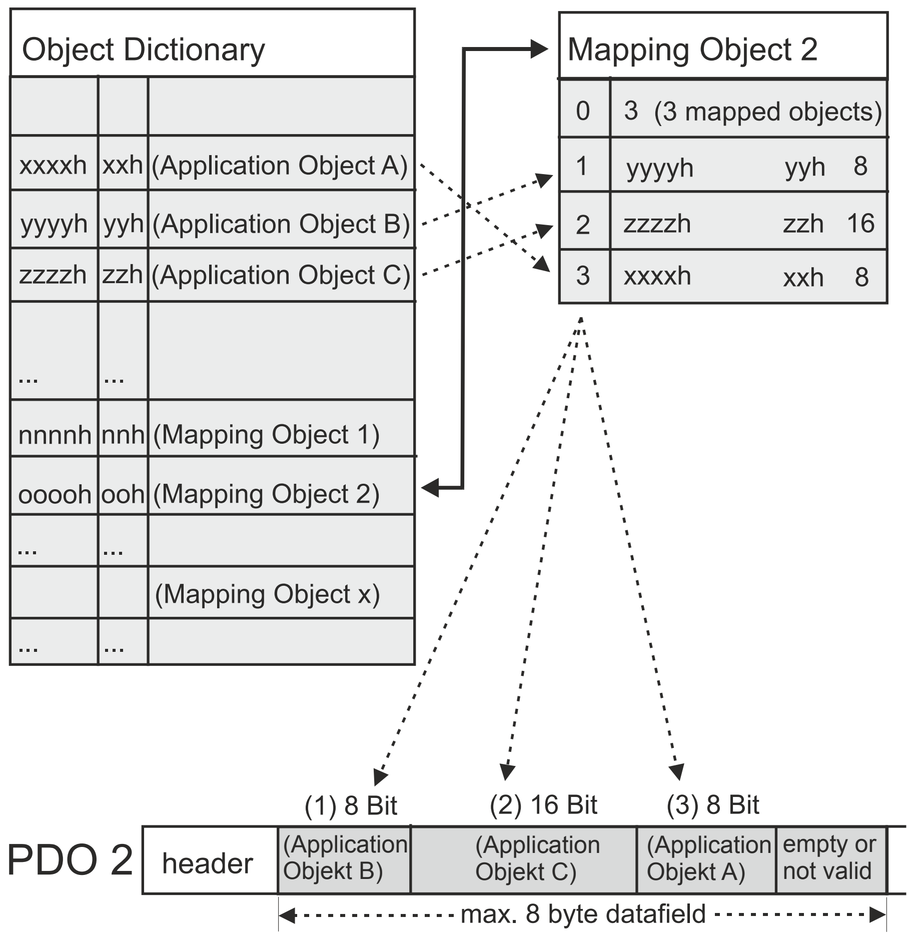
**Introduction**

The process data interchange with CANopen® is also a pure CAN bus, that is, without a protocol overhead. The broadcast property of the CAN bus completely remains as it is. This way, a message can be received and evaluated by all devices connected to the bus.

**PDO mapping**

As the protocol structure is missing in the frame, the node(s) at the bus, to which these data have been assigned, must be notified of how information is integrated into the data range of the PDO (which bit/byte is which value). For this declaration, so-called PDO mapping is applied, which allows for placing the desired information at a certain location in the data range of a PDO.

To allow for variable PDO data configuration, mapping itself is carried out in a specific mapping object. In principle, this is a table into which the objects to be mapped are entered.



**Various kinds of process data interchange**

Process data can generally be exchanged in various ways which can be applied within one network simultaneously (as a mixture, so to say).

- Event-driven data transfer
- Timer-driven data transfer
- Polling by remote frames
- Synchronized mode

**Event-driven data transfer**

In this case, the data of a node are transmitted as a message, as soon as the state has been changed.

If, for example, the level at a digital input of a CAN I/O device changes, transmitting the assigned message (PDO) is triggered.

If, for example, a device has got threshold values for an analog value, and if the threshold is reached, the assigned message (PDO) is sent as well.

**Timer-controlled data transfer**

In intervals of the so-called event time, messages are continually sent, even if the data haven't changed in between.

The inhibit time defines the minimum interval between two calls of a PDO service.

**Polling by remote frames**

In case of Remote Frame Polling, the CAN node which functions as master in the network, requires the desired information by query (by means of Remote Frame).

The node which owns this information, respectively the required data, then replies by transmitting the requested data.

As with CANopen® the message identifier also specifies the device address, the query is usually directed towards a specific device. All other CAN controllers in the network ignore this query.

**Synchronized mode**

CANopen® lets you query inputs and states of various nodes simultaneously and to make changes to inputs respectively states simultaneously.

This is supported by the synchronizing frame (SYNC). The sync frame is a broadcast of high importance, yet, without data content, to all bus nodes. The sync frame is dispatched by a bus node in cyclic mode and in set intervals (communication cycle).

Devices functioning in synchronized mode read their inputs (actual states) when the sync frame is received and send the data directly after this, as soon as the bus permits.

After the following sync frame, output data (state changes instructed via bus) are written to the outputs and executed.

## The service data object SDO

**Protocol structure**

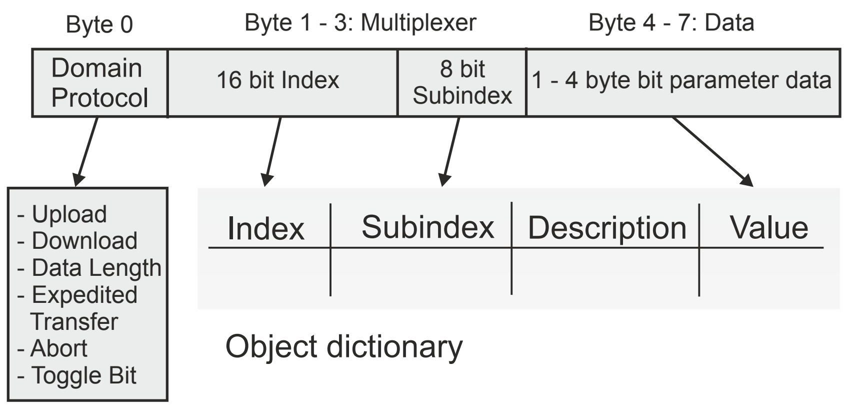
All CANopen® devices are equipped with a so-called object dictionary, which lets you access all parameters that are supported by the assembly.

As can be seen from the object dictionary, the object data have got an index of 16 bits. Parameters can be directly accessed via this index. Further, with each index there is a sub-index of 8 bits which enables further structuring within an index.

For this reason, a service data frame must have a protocol structure which exactly defines which parameter is to be addressed and how this parameter is to be dealt with.

A service data object consists of a domain protocol (8 bits), the index (16 bits), the sub-index (8 bits) and of up to 4 data bytes altogether.

The domain protocol specifies what is to be done to the parameters referred to by index and sub-index. New values which are to be assigned to certain parameters can be transferred within the data bytes.



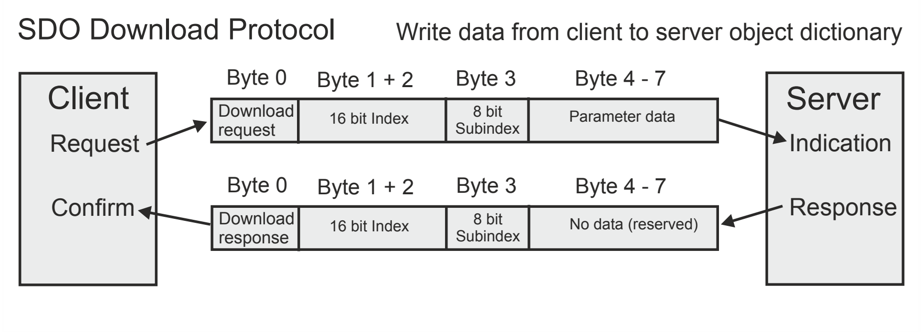
The 8 bytes of the SDO (as shown here) have been stored to the data range of the CAN message. The device is addressed by means of the SDO in the identifier.

An SDO transfer always comprises two frames as a minimum.

**Download protocol**

The following illustration shows data exchange in the case of an SDO download protocol being applied.

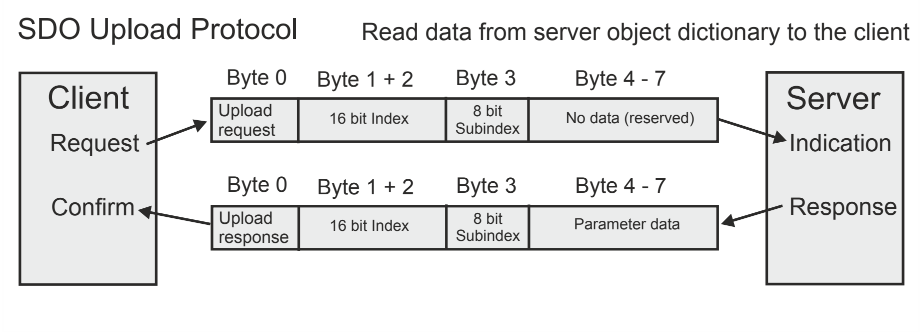
The data are written to the object dictionary of the server. The reply directed to the Client has got the same index and sub-index.



**Upload protocol**

The following illustration shows data exchange in the case of an SDO upload protocol being applied.

The data are read from the object dictionary of the server and transferred to the client. The reply directed to the Client has got the same index and sub-index.



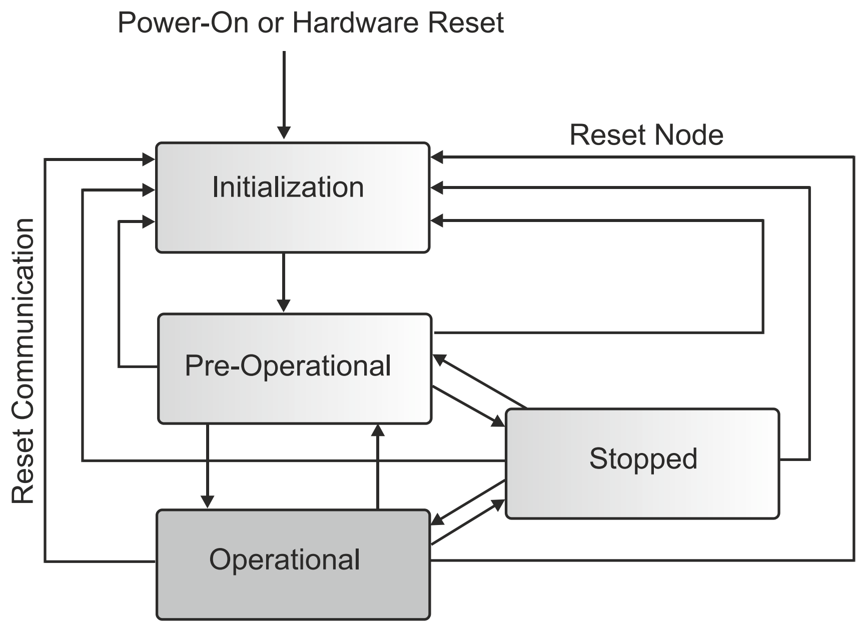
## Network management (NMT)

**State machine**

Each CANopen® device comprises a state machine which, after power-up, takes on the pre-operational state. In this state, the CANopen® device can be configured and parameterized via SDO. Communication via PDO is not permitted.

CANopen® devices by Jetter AG change their state my making a function call (CanOpenSetCommand) out of an STX program. This means that the device on which the program is running, sets all CANopen® devices (nodes) at the bus to the "Operational" state. In this state, PDO frames are sent and received. Access to the object dictionary via SDO is possible as well.

If a CANopen® device is set to Stop, communication via PDO or SDO is not possible any more. This state is made use of to evoke a certain behavior of the application. Defining this behavior belongs to the task area of the device profiles.



# 2 CANopen® STX API

**Introduction**

This chapter describes the STX functions of the CANopen® STX API.

**Application**

These STX functions are used in communication between this device and other CANopen® nodes.

**Terms and abbreviations**

In this chapter, the following terms and abbreviations are used:

| Term | Description |
|---|---|
| Node ID | Node identification number of the device:<br>This ID lets you address the device. |
| NMT | Network management |
| ro | Read only access |
| rw | Read/write access |

**Devices**

The following devices have got the CANopen®-STX-API feature:

| Category | Designation |
|---|---|
| Controller | JC-360(MC), JC-365(MC), JC-440(MC)<br>JCM-350-E01/E02, JCM-350-E03, JCM-620 |
| HMI | BTM07, BTM09(B), BTM010, BTM011(B), BTM012<br>JVM-104, JVM-407(B), JVM-507(B), JVM-604(B) |

**Contents**

## STX function: CanOpenInit()

**Introduction**

The function `CanOpenInit()` lets you initialize one of the CAN busses. The device then automatically sends the heartbeat message every second with the following communication object identifier (COB-ID): Node ID + 0x700.

**Function declaration**

```
Function CanOpenInit(
    CANNo:Int,
    NodeID:Int,
    const ref SWVersion:String,
) :Int;
```

**Function parameters**

The function `CanOpenInit()` has got the following parameters.

| Parameter | Description | Value |
|---|---|---|
| CANNo | CAN bus number | 0 ... CANMAX |
| NodeID | Node ID of the given device | 1 ... 127 |
| SWVersion | Reference to own software version<br>This software version is entered into the index 0x100A in the object directory. | String up to 255 characters |

**Return value**

This function transfers the following return values to the higher-level program.

| Return value | |
|---|---|
| 0 | OK |
| -1 | Error when checking parameters |
| -3 | Initialization has not worked |
| -4 | The JX2 system bus driver is activated |

**CANNo parameter**

This parameter specifies the number of the CAN interface. CANNo = 0 is assigned to the first interface. The number of CAN interfaces depends on the device. For information on the maximum number of CAN interfaces (CANMAX) refer to the chapters *Technical Specifications* and *Quick Reference* in the corresponding manual.

**How to use this function**

This function lets you initialize CAN bus 0. The device has node ID 20 (0x14).

```
Result := CanOpenInit(0, 20, 'Version: 01.00.0.00');
```

**Operating principle**

During initialization, the device processes the following process steps:

| Step | Description |
|---|---|
| **1** | First, the bootup message is sent as a heartbeat message. |
| **2** | As soon as the device goes into **pre-operational** status, it sends the heartbeat message **pre-operational**. |

**Access to the object directory**

If the device is in **pre-operational** state, it lets you access the object directory using SDO.

**NMT messages**

After initialization, NMT messages can be sent and received. The own heartbeat status can be changed with the function `CanOpenSetCommand`.

**Related topics**

- **STX function CanOpenSetCommand** (see page 24)

## STX function: CanOpenSetCommand()

**Introduction**

The function `CanOpenSetCommand()` lets you change the heartbeat status of the device itself and of all other devices (NMT slaves) on the CAN bus.

**Function declaration**

```
Function CanOpenSetCommand(
    CANNo:Int,
    iType:Int,
    Value:Int,
) :Int;
```

**Function parameters**

The function `CanOpenSetCommand()` has got the following parameters:

| Parameter | Description | Value |
|---|---|---|
| CANNo | CAN bus number | 0 ... CANMAX |
| iType | Command selection | See table below. |

| iType | Description: Value |
|---|---|
| CAN_CMD_HEARTBEAT | Only the own heartbeat status is changed. Selecting heartbeat states:<br>CAN_HEARTBEAT_STOPPED (0x04)<br>CAN_HEARTBEAT_OPERATIONAL (0x05)<br>CAN_HEARTBEAT_PREOPERATIONAL (0x7F) |
| CAN_CMD_NMT | The heartbeat status is changed for all other devices or for a specific device on the CAN bus. Selecting heartbeat states (NMT master):<br>CAN_NMT_OPERATIONAL (0x01) or CAN_NMT_START (0x01)<br>CAN_NMT_STOP (0x02)<br>CAN_NMT_PREOPERATIONAL (0x80)<br>CAN_NMT_RESET (0x81)<br>CAN_NMT_RESETCOMMUNICATION (0x82) |
| CAN_CMD_TIME_CONSUMER | This command lets you set the device to ready-to-receive state to allow time synchronization via CAN bus (CAN ID 0x100). Refer to document by CiA e.V. DS301 V402 *Selecting Synchronization*, page 59.<br>CAN_TIME_CONSUMER_DISABLE = 0<br>CAN_TIME_CONSUMER_ENABLE = 1 |
| CAN_CMD_TIME_PRODUCER | The time is published on the CAN bus. For more information on the structure refer to document DS301 by CiA e.V., CAN ID 0x100:<br>CAN_TIME_PRODUCER_SEND = 1 (for sending TIME_OF_DAY once) |

**Note**

The macro function `CAN_CMD_NMT_Value(NodeID, CAN_CMD_NMT)` is used to select the command CAN_CMD_NMT.

Values from 0 to 127 are permitted for the node ID parameter. 1 to 127 is the node ID for a specific device. If the command is to be sent to all devices on the CAN bus, use the parameter `CAN_CMD_NMT_ALLNODES(0)`.

**CANNo parameter**

This parameter specifies the number of the CAN interface. CANNo = 0 is assigned to the first interface. The number of CAN interfaces depends on the device. For information on the maximum number of CAN interfaces (CANMAX) refer to the chapters *Technical Specifications* and *Quick Reference* in the corresponding manual.

**Return value**

This function sends the following return values to the higher-level program.

| **Return value** | |
|---|---|
| 0 | OK |
| -1 | Error when checking parameters<br>Command not known |

**How to use this function (example 1)**

Task: Set the own heartbeat status to **operational**.

```
Result := CanOpenSetCommand(0, CAN_CMD_HEARTBEAT,
CAN_HEARTBEAT_OPERATIONAL);
```

**How to use this function (example 2)**

Task: Set the own heartbeat status and the status of all other devices on the CAN bus to **operational**.

```
Result := CanOpenSetCommand(0, CAN_CMD_NMT,
CAN_CMD_NMT_Value(CAN_CMD_NMT_ALLNODES, CAN_NMT_OPERATIONAL));
```

**How to use this function (example 3)**

Task: Set the heartbeat status of the device with the node ID 60 (0x3C) to **operational**.

```
Result := CanOpenSetCommand(0, CAN_CMD_NMT, CAN_CMD_NMT_Value(60,
CAN_NMT_OPERATIONAL));
```

**How to use this function (example 4)**

Task: Enable time synchronization via CAN bus (CAN ID 0x100).

```
Result := CanOpenSetCommand(0, CAN_CMD_TIME_CONSUMER,
CAN_TIME_CONSUMER_ENABLE);
```

**How to use this function (example 5)**

Task: Publish the time on the CAN bus.

```
Result := CanOpenSetCommand(0, CAN_CMD_TIME_PRODUCER,
CAN_TIME_PRODUCER_SEND);
```

## STX function: CanOpenUploadSDO()

**Introduction**

The function `CanOpenUploadSDO()` lets you access a particular object in the object directory of the message recipient and read the value of the object.

Data is exchanged in accordance with the SDO upload protocol. Supported transfer types are **segmented** (more than 4 data bytes) and **expedited** (up to 4 data bytes).

**Function declaration**

```
Function CanOpenUploadSDO(
    CANNo:Int,              // Number of the bus line
    NodeID:Int,             // Device ID
    wIndex:Word,
    SubIndex:Byte,
    DataType:Int,           // Type of the data to be received
    // Data length for the global variable DataAddr
    DataLength:Int,
    // Global variable into which the received value is entered
    const ref DataAddr,
    ref Busy: Int,          // Status of the SDO transmission
) :Int;
```

**Function parameters**

The `CanOpenUploadSDO()` function has got the following parameters:

| Parameter | Description | Value |
|---|---|---|
| CANNo | CAN bus number | 0 ... CANMAX |
| NodeID | Node ID of the message recipient | 1 ... 127 |
| wIndex | Index number of the object | 0 ... 0xFFFF |
| SubIndex | Subindex number of the object | 0 ... 255 |
| DataType | Type of data to be received | 2 ... 27 |
| DataLength | Data length of the global variable DataAddr | |
| DataAddr | Global variable into which the received value is to be entered | |
| Busy | Status of the SDO transmission | |

**Return value**

This function sends the following return values to the higher-level program.

| Return value | |
|---|---|
| 0 | OK |
| -1 | Error in checking parameters |
| -2 | Device in **Stop** status |
| -3 | DataType is greater than DataLength |
| -4 | Insufficient memory |

**CANNo parameter**

This parameter specifies the number of the CAN interface. CANNo = 0 is assigned to the first interface. The number of CAN interfaces depends on the device. For information on the maximum number of CAN interfaces (CANMAX) refer to the chapters *Technical Specifications* and *Quick Reference* in the corresponding manual.

**DataType parameter**

The following data types can be received.

| Byte types | CANopen® format | Jetter format |
|---|---|---|
| 1 | CANOPEN_INTEGER8<br>CANOPEN_UNSIGNED8 | Byte |
| 2 | CANOPEN_INTEGER16<br>CANOPEN_UNSIGNED16 | Word |
| 3 | CANOPEN_INTEGER24<br>CANOPEN_UNSIGNED24 | - |
| 4 | CANOPEN_INTEGER32<br>CANOPEN_UNSIGNED32<br>CANOPEN_REAL | Int |
| 5 | CANOPEN_INTEGER40<br>CANOPEN_UNSIGNED40 | - |
| 6 | CANOPEN_INTEGER48<br>CANOPEN_UNSIGNED48<br>CANOPEN_TIME_OF_DAY<br>CANOPEN_TIME_DIFFERENCE | - |
| 7 | CANOPEN_INTEGER56<br>CANOPEN_UNSIGNED46 | - |
| 8 | CANOPEN_INTEGER64<br>CANOPEN_UNSIGNED64<br>CANOPEN_REAL64 | - |
| n | CANOPEN_VISIBLE_STRING<br>CANOPEN_OCTET_STRING<br>CANOPEN_UNICODE_STRING<br>CANOPEN_DOMAIN | String |

**Busy parameter**

After successfully calling up the function, the **Busy** parameter is set to SDOACCESS_INUSE. With an error in transmission, **Busy** is set to SDOACCESS_ERROR. With a successful transmission, the function returns the number of bytes transmitted.

**Busy - Error codes**

With an error in transmission, **Busy** returns an error code. The following error codes are available:

**SDOACCESS_STILLUSED**
Another task is communicating with the same node ID.

**SDOACCESS_TIMEOUT**
The task has been timed out because the device with the specified node ID is not responding.
If the specified node ID does not respond within 1 second, the timeout bit is set.

**SDOACCESS_ILLCMD**
The response to the request is invalid.

**SDOACCESS_ABORT**
Access to the device with the specified node ID was aborted.

**SDOACCESS_SYSERROR**
General internal error

**Macro definitions**

The following macros have been defined in connection with this function:

**SDOACCESS_FINISHED (busy)**
This macro checks whether communication has finished.

**SDOACCESS_ERROR (busy)**
This macro checks whether an error has occurred.

## STX function: CanOpenDownloadSDO()

**Introduction**

The function `CanOpenDownloadSDO()` lets you access a particular object in the Object Directory of the message recipient and specify the value of the object. Data is exchanged in accordance with the SDO upload protocol. Supported transfer types are **segmented** or **block** (more than 4 data bytes) and **expedited** (up to 4 data bytes).

**Function declaration**

```
Function CanOpenDownloadSDO(
    CANNo:Int,              // Number of the bus line
    NodeID:Int,             // Device ID
    wIndex:Word,
    SubIndex:Byte,
    DataType:Int,           // Type of the data to be sent
    // Data length of the global variable DataAddr
    DataLength:Int,
    // Global variable holding the value to be sent
    const ref DataAddr,
    ref Busy: Int,          // Status of the SDO transmission
) :Int;
```

**Function parameters**

The `CanOpenDownloadSDO()` function has got the following parameters:

| Parameter | Description | Value |
|---|---|---|
| CANNo | CAN bus number | 0 ... CANMAX |
| NodeID | Node ID of the message recipient | 1 ... 127 |
| wIndex | Index number of the object | 0 ... 0xFFFF |
| SubIndex | Subindex number of the object | 0 ... 255 |
| DataType | Type of data to be sent | 2 ... 27 |
| DataLength | Data length of the global variable DataAddr | |
| DataAddr | Global variable into which the value to be sent is to be entered | |
| Busy | Status of the SDO transmission | |

**Return value**

This function sends the following return values to the higher-level program.

| Return value | |
|---|---|
| 0 | OK |
| -1 | Error when checking parameters |
| -2 | Device in **Stop** status (own heartbeat status) |
| -3 | DataType is greater than DataLength |
| -4 | Insufficient memory |

**CANNo parameter**

This parameter specifies the number of the CAN interface. CANNo = 0 is assigned to the first interface. The number of CAN interfaces depends on the device. For information on the maximum number of CAN interfaces (CANMAX) refer to the chapters *Technical Specifications* and *Quick Reference* in the corresponding manual.

**DataType parameter**

The following data types can be received.

| Byte types | CANopen® format | Jetter format |
|---|---|---|
| 1 | CANOPEN_INTEGER8<br>CANOPEN_UNSIGNED8 | Byte |
| 2 | CANOPEN_INTEGER16<br>CANOPEN_UNSIGNED16 | Word |
| 3 | CANOPEN_INTEGER24<br>CANOPEN_UNSIGNED24 | - |
| 4 | CANOPEN_INTEGER32<br>CANOPEN_UNSIGNED32<br>CANOPEN_REAL | Int |
| 5 | CANOPEN_INTEGER40<br>CANOPEN_UNSIGNED40 | - |
| 6 | CANOPEN_INTEGER48<br>CANOPEN_UNSIGNED48<br>CANOPEN_TIME_OF_DAY<br>CANOPEN_TIME_DIFFERENCE | - |
| 7 | CANOPEN_INTEGER56<br>CANOPEN_UNSIGNED46 | - |
| 8 | CANOPEN_INTEGER64<br>CANOPEN_UNSIGNED64<br>CANOPEN_REAL64 | - |
| n | CANOPEN_VISIBLE_STRING<br>CANOPEN_OCTET_STRING<br>CANOPEN_UNICODE_STRING<br>CANOPEN_DOMAIN | String |

| | |
|---|---|
| **Busy parameter** | After successfully calling up the function, the **Busy** parameter is set to SDOACCESS_INUSE. With an error in transmission, **Busy** is set to SDOACCESS_ERROR. With a successful transmission, the function returns the number of bytes transmitted. |
| **"Busy" error codes** | With an error in transmission, Busy returns an error code. The following error codes are available:<br><br>**SDOACCESS_STILLUSED**<br>Another task is communicating with the same node ID.<br><br>**SDOACCESS_TIMEOUT**<br>The task has been timed out because the device with the given node ID is not responding.<br>If the specified node ID does not respond within 1 second, the timeout bit is set.<br><br>**SDOACCESS_ILLCMD**<br>The response to the request is invalid.<br><br>**SDOACCESS_ABORT**<br>Access to the device with the specified node ID was aborted.<br><br>**SDOACCESS_BLKSIZEINV**<br>Communication error with Block Download<br><br>**SDOACCESS_SYSERROR**<br>General internal error |
| **Macro definitions** | The following macros have been defined in connection with this function:<br><br>**SDOACCESS_FINISHED (busy)**<br>This macro checks whether communication has finished.<br><br>**SDOACCESS_ERROR (busy)**<br>This macro checks whether an error has occurred. |

## STX function: CanOpenAddPDORx()

**Introduction**

The function `CanOpenAddPDORx()` lets you specify which process data, sent by other CANopen® devices, must be received.

Process data can be received only when a CANopen® device is sending them.

**Notes**

- Only if the CANopen® devices on the bus are in state **operational**, the PDO telegram is transmitted.
- The smallest time unit for the event time is 1 ms.
- The smallest time unit for the inhibit time is 1 ms.

**Function declaration**

```
Function CanOpenAddPDORx(
    CANNo:Int,              // Number of the bus line
    CANID:Int,              // CAN identifier
    // Starting position of data to be received
    BytePos:Int,
    DataType:Int,           // Data type of the data to be received
    // Data length of the global variable VarAddr
    DataLength:Int,
    // Global variable into which the received value is entered
    const ref VarAddr,
    // Cycle time for receiving a telegram
    // Event time
    EventTime: Int,
    // Minimum interval between two received messages
    // Inhibit time
    InhibitTime: Int,
    Paramset: Int,          // Bit-coded parameter
) :Int;
```

**Function parameters**

The `CanOpenAddPDORx()` function has got the following parameters:

| Parameter | Description | Value |
|---|---|---|
| CANNo | CAN bus number | 0 ... CANMAX |
| CANID | CAN identifier 11-bit<br>CAN identifier 29-bit | 0 ... 0x7FF<br>0 ... 0x1FFFFFFF |
| BytePos | Starting position of data to be received | 0 ... 7 |
| DataType | Data type of data to be received | 2 ... 13, 15 ... 27 |
| DataLength | Data length of the global variable VarAddr | |
| VarAddr | Global variable into which the received value is entered | |
| EventTime | Time lag between two telegrams (> InhibitTime) | |

| Parameter | Description | Value |
|---|---|---|
| InhibitTime | Minimum time lag between two telegrams received (< EventTime) | |
| Paramset | Bit-coded parameter | |

**Return value**

This function sends the following return values to the higher-level program.

| Return value | |
|---|---|
| 0 | OK |
| -1 | Error when checking parameters |
| -3 | DataType is greater than DataLength |
| -4 | Insufficient memory |

**CANNo parameter**

This parameter specifies the number of the CAN interface. CANNo = 0 is assigned to the first interface. The number of CAN interfaces depends on the device. For information on the maximum number of CAN interfaces (CANMAX) refer to the chapters *Technical Specifications* and *Quick Reference* in the corresponding manual.

## CANID parameter

The **CANID** parameter is used to transfer the CAN identifier. The CAN identifier is generated with a macro. The CAN identifier depends on the node ID of the other communicating user and on whether it is a PDO1, PDO2, PDO3 or PDO4 message.

**Macro definitions:**

```
#Define CANOPEN_PDO1_RX (NodeID)    ((NodeID) + 0x180)
#Define CANOPEN_PDO2_RX (NodeID)    ((NodeID) + 0x280)
#Define CANOPEN_PDO3_RX (NodeID)    ((NodeID) + 0x380)
#Define CANOPEN_PDO4_RX (NodeID)    ((NodeID) + 0x480)

#Define CANOPEN_PDO1_TX (NodeID)    ((NodeID) + 0x200)
#Define CANOPEN_PDO2_TX (NodeID)    ((NodeID) + 0x300)
#Define CANOPEN_PDO3_TX (NodeID)    ((NodeID) + 0x400)
#Define CANOPEN_PDO4_TX (NodeID)    ((NodeID) + 0x500)
```

**Example for calling up the macro:**

CANOPEN_PDO2_RX (64)

⇨ The resulting CAN identifier is: 2C0h = 40h + 280h

## Default CAN identifier distribution

For CANopen® the following CAN identifier distribution is predefined. In this case, the node number is embedded in the identifier.

| 11-bit identifier (binary) | Identifier (decimal) | Identifier (hexadecimal) | Description |
|---|---|---|---|
| 000000000000 | 0 | 0 | Network management |
| 000100000000 | 128 | 80h | Synchronization |
| 0001xxxxxxxx | 129 - 255 | 81h - FFh | Emergency |
| 0011xxxxxxxx | 385 - 511 | 181h - 1FFh | PDO1 (tx) |
| 0100xxxxxxxx | 513 - 639 | 201h - 27Fh | PDO1 (rx) |
| 0101xxxxxxxx | 641 - 767 | 281h - 2FFh | PDO2 (tx) |
| 0110xxxxxxxx | 769 - 895 | 301h - 37Fh | PDO2 (rx) |
| 0111xxxxxxxx | 897 - 1023 | 381h - 3FFh | PDO3 (tx) |
| 1000xxxxxxxx | 1025 - 1151 | 401h -47Fh | PDO3 (rx) |
| 1001xxxxxxxx | 1153 - 1279 | 481h - 4FFh | PDO4 (tx) |
| 1010xxxxxxxx | 1281 - 1407 | 501h - 57Fh | PDO4 (rx) |
| 1011xxxxxxxx | 1409 - 1535 | 581h - 5FFh | Send SDO |
| 1100xxxxxxxx | 1537 - 1663 | 601h - 67Fh | Receive SDO |
| 1110xxxxxxxx | 1793 - 1919 | 701h - 77Fh | NMT error control |
| xxxxxxxx = Node number 1 - 127 | | | |

**DataType parameter**

The following data types can be received.

| Byte types | CANopen® format | Jetter format |
|---|---|---|
| 1 | CANOPEN_INTEGER8<br>CANOPEN_UNSIGNED8 | Byte |
| 2 | CANOPEN_INTEGER16<br>CANOPEN_UNSIGNED16 | Word |
| 3 | CANOPEN_INTEGER24<br>CANOPEN_UNSIGNED24 | - |
| 4 | CANOPEN_INTEGER32<br>CANOPEN_UNSIGNED32<br>CANOPEN_REAL | Int |
| 5 | CANOPEN_INTEGER40<br>CANOPEN_UNSIGNED40 | - |
| 6 | CANOPEN_INTEGER48<br>CANOPEN_UNSIGNED48<br>CANOPEN_TIME_OF_DAY<br>CANOPEN_TIME_DIFFERENCE | - |
| 7 | CANOPEN_INTEGER56<br>CANOPEN_UNSIGNED46 | - |
| 8 | CANOPEN_INTEGER64<br>CANOPEN_UNSIGNED64<br>CANOPEN_REAL64 | - |
| n | CANOPEN_VISIBLE_STRING<br>CANOPEN_OCTET_STRING<br>CANOPEN_UNICODE_STRING<br>CANOPEN_DOMAIN | String |

**Paramset parameter**

The following parameters can be transferred to the function. Several parameters can be linked together using the Or function.

**CANOPEN_ASYNCPDORTRONLY**

Receive asynchronous PDOs by sending an RTR frame to the sender (after each expired EventTime). If there is no response to RTR frames, the request time increases to five times the EventTime.

**CANOPEN_ASYNCPDO**

Receive asynchronous PDOs.

**CANOPEN_PDOINVALID**

PDO not received. Disk space is reserved.

**CANOPEN_NORTR**

PDO cannot be requested by RTR (Remote Request).

Only if CANOPEN_ASYNCPDORTROnly has been set, an RTR is sent.

**CANOPEN_29BIT**

| | |
|---|---|
| | Use 29-bit identifier<br>Default: 11-bit identifier |

## STX function: CanOpenAddPDOTx()

**Introduction**

By calling up the `CanOpenAddPDOTx()` function, process data can be deposited on the bus.

However, that should not mean that other CANopen® devices on the bus can also read this process data.

**Notes**

- Only if the CANopen® devices on the bus are in state **operational**, the PDO telegram is transmitted.
- As soon as there are any changes to the process data, another PDO telegram is transmitted immediately.
- The smallest time unit for the event time is 1 ms.
- The smallest time unit for the inhibit time is 1 ms.
- Any unused bytes of a telegram are sent as null.

**Function declaration**

```
Function CanOpenAddPDOTx(
    CANNo:Int,            // Number of the bus line
    CANID:Int,            // CAN identifier
    BytePos:Int,          // Starting position of the data to be sent
    DataType:Int,         // Data type of the data to be sent
    // Data length of the global variable VarAddr
    DataLength:Int,
    // Global variable holding the value to be sent
    const ref VarAddr,
    // Cycle time for sending a telegram
    // Event time
    EventTime: Int,
    // Minimum interval between two transmitted messages
    // Inhibit time
    InhibitTime: Int,
    Paramset: Int,        // Bit-coded parameter
) :Int;
```

**Function parameters**

The `CanOpenAddPDOTx()` function has got the following parameters:

| Parameter | Description | Value |
|---|---|---|
| CANNo | CAN bus number | 0 ... CANMAX |
| CANID | CAN identifier 11-bit<br>CAN identifier 29-bit | 0 ... 0x7FF<br>0 ... 0x1FFFFFFF |
| BytePos | Starting position of data to be sent | 0 ... 7 |
| DataType | Data type of data to be sent | 2 ... 13, 15 ... 27 |
| DataLength | Data length of the global variable VarAddr | |
| VarAddr | Global variable into which the value to be sent is entered | |

| Parameter | Description | Value |
|---|---|---|
| EventTime | Time lag between two telegrams (> InhibitTime) | |
| InhibitTime | Minimum time lag between two telegrams to be sent (< EventTime) | |
| Paramset | Bit-coded parameter | |

**Return value**

This function sends the following return values to the higher-level program.

| Return value | |
|---|---|
| 0 | OK |
| -1 | Error when checking parameters |
| -3 | DataType is greater than DataLength |
| -4 | Insufficient memory |

**CANNo parameter**

This parameter specifies the number of the CAN interface. CANNo = 0 is assigned to the first interface. The number of CAN interfaces depends on the device. For information on the maximum number of CAN interfaces (CANMAX) refer to the chapters *Technical Specifications* and *Quick Reference* in the corresponding manual.

**CANID parameter**

The **CANID** parameter is used to transfer the CAN identifier. The CAN identifier is generated with a macro. The CAN identifier depends on the node ID of the other communicating user and on whether it is a PDO1, PDO2, PDO3 or PDO4 message.

**Macro definitions:**

```
#Define CANOPEN_PDO1_RX (NodeID)    ((NodeID) + 0x180)
#Define CANOPEN_PDO2_RX (NodeID)    ((NodeID) + 0x280)
#Define CANOPEN_PDO3_RX (NodeID)    ((NodeID) + 0x380)
#Define CANOPEN_PDO4_RX (NodeID)    ((NodeID) + 0x480)

#Define CANOPEN_PDO1_TX (NodeID)    ((NodeID) + 0x200)
#Define CANOPEN_PDO2_TX (NodeID)    ((NodeID) + 0x300)
#Define CANOPEN_PDO3_TX (NodeID)    ((NodeID) + 0x400)
#Define CANOPEN_PDO4_TX (NodeID)    ((NodeID) + 0x500)
```

**Example for calling up the macro:**

CANOPEN_PDO2_RX (64)

⇨ The resulting CAN identifier is: 2C0h = 40h + 280h

**Default CAN identifier distribution**

For CANopen® the following CAN identifier distribution is predefined. In this case, the node number is embedded in the identifier.

| 11-bit identifier (binary) | Identifier (decimal) | Identifier (hexadecimal) | Description |
|---|---|---|---|
| 000000000000 | 0 | 0 | Network management |
| 000100000000 | 128 | 80h | Synchronization |
| 0001xxxxxxxx | 129 - 255 | 81h - FFh | Emergency |
| 0011xxxxxxxx | 385 - 511 | 181h - 1FFh | PDO1 (tx) |
| 0100xxxxxxxx | 513 - 639 | 201h - 27Fh | PDO1 (rx) |
| 0101xxxxxxxx | 641 - 767 | 281h - 2FFh | PDO2 (tx) |
| 0110xxxxxxxx | 769 - 895 | 301h - 37Fh | PDO2 (rx) |
| 0111xxxxxxxx | 897 - 1023 | 381h - 3FFh | PDO3 (tx) |
| 1000xxxxxxxx | 1025 - 1151 | 401h -47Fh | PDO3 (rx) |
| 1001xxxxxxxx | 1153 - 1279 | 481h - 4FFh | PDO4 (tx) |
| 1010xxxxxxxx | 1281 - 1407 | 501h - 57Fh | PDO4 (rx) |
| 1011xxxxxxxx | 1409 - 1535 | 581h - 5FFh | Send SDO |
| 1100xxxxxxxx | 1537 - 1663 | 601h - 67Fh | Receive SDO |
| 1110xxxxxxxx | 1793 - 1919 | 701h - 77Fh | NMT error control |
| xxxxxxxx = Node number 1 - 127 | | | |

**DataType parameter**

The following data types can be received.

| Byte types | CANopen® format | Jetter format |
|---|---|---|
| 1 | CANOPEN_INTEGER8<br>CANOPEN_UNSIGNED8 | Byte |
| 2 | CANOPEN_INTEGER16<br>CANOPEN_UNSIGNED16 | Word |
| 3 | CANOPEN_INTEGER24<br>CANOPEN_UNSIGNED24 | - |
| 4 | CANOPEN_INTEGER32<br>CANOPEN_UNSIGNED32<br>CANOPEN_REAL | Int |
| 5 | CANOPEN_INTEGER40<br>CANOPEN_UNSIGNED40 | - |
| 6 | CANOPEN_INTEGER48<br>CANOPEN_UNSIGNED48<br>CANOPEN_TIME_OF_DAY<br>CANOPEN_TIME_DIFFERENCE | - |
| 7 | CANOPEN_INTEGER56<br>CANOPEN_UNSIGNED46 | - |
| 8 | CANOPEN_INTEGER64<br>CANOPEN_UNSIGNED64<br>CANOPEN_REAL64 | - |

| Byte types | CANopen® format | Jetter format |
|---|---|---|
| n | CANOPEN_VISIBLE_STRING<br>CANOPEN_OCTET_STRING<br>CANOPEN_UNICODE_STRING<br>CANOPEN_DOMAIN | String |

**Paramset parameter**

The following parameters can be transferred to the function. Several parameters can be linked together using the Or function.

**CANOPEN_ASYNCPDORTRONLY**

Send asynchronous PDOs by receiving an RTR frame.

This feature is not yet supported at the moment.

**CANOPEN_ASYNCPDO**

Send asynchronous PDO.

**CANOPEN_PDOINVALID**

PDO not sent. The required disk space is reserved.

**CANOPEN_NORTR**

PDO cannot be requested by RTR (Remote Request).

**CANOPEN_29BIT**

Use 29-bit identifier

Default: 11-bit identifier

## Heartbeat monitoring

**Introduction**

The heartbeat protocol is for monitoring the activity of communication partners. If the inactivity exceeds the set interval (Heartbeat consumer time), the status is set to **offline**.

The application program lets you define heartbeat functions, such as

- Displaying information to the user
- Rebooting the device
- Ignoring process data

**Prerequisites**

**Heartbeat monitoring** is available only for specific devices and its availability depends on the OS version, for further details refer to the quick reference on the respective device.

**Registers for heartbeat monitoring**

Heartbeat monitoring uses the following registers:

| Register | Description | Data type | Attributes |
|---|---|---|---|
| 40x001 | Own heartbeat status of the device;<br>Value range:<br>0 = Bootup<br>4 = Stopped<br>5 = Operational<br>127 = Preoperational<br>255 = Offline (default value) | Int | ro (read only) |
| 40x100 | The heartbeat status of all monitored node IDs has changed. Value range:<br>0 = False<br>1 = True | Bool | rw (read and write) |
| 40x101 ... 40x227 | Heartbeat status of nodes with ID 1 ... 127; value range:<br>0 = Bootup<br>4 = Stopped<br>5 = Operational<br>127 = Preoperational<br>255 = Offline (default value) | Byte | ro |
| 40x229 ... 40x355 | Heartbeat timeout of nodes with ID 1 ... 127; value range:<br>0 ... 65535 [ms] | Word | rw |

In the register number, the letter **x** represents the number of the CAN bus line used: x = 0 ... CANMAX.

**Launching heartbeat monitoring**

To launch heartbeat monitoring, proceed as follows:

| Step | Action |
|---|---|
| **1** | Enable heartbeat monitoring:<br>Enter the timeout value into the corresponding register. This value must range between 1 and 65535 [ms]. Example:<br>For CAN 0 and node ID 1: Enter a timeout value of 3000 [ms] into register 400229. |
| **2** | Define in your application program how the device is to respond to individual values in the heartbeat status register.<br>When the state in register 40x101 ... 40x227 changes, the value in register 40x100 changes to 1 (true). |
| **3** | Reset the value in register 40x100 to zero (false).<br>This step ensures that subsequent changes in register 40x101 ... 40x227 can be displayed. |

Heartbeat monitoring starts on receipt of the first heartbeat (including bootup message). The DLC (Data Length Code) of the heartbeat message must be 1.

**Terminating heartbeat monitoring**

To terminate heartbeat monitoring, proceed as follows:

| Step | Action |
|---|---|
| **1** | Disable heartbeat monitoring:<br>Enter a timeout value of 0 [ms] into the timeout register. |

**Emergency message**

When a heartbeat timeout is detected, an emergency message is sent automatically.
On receipt of the next heartbeat message, the emergency message is reset.

**Example:**
The following emergency message is tripped:

| Reference | Value |
|---|---|
| Error code | 0x8130 |
| Error Register | 0x81 |
| Manufacturer error | 0x00,NodeID,0x00,0x00,0x00 |

The message on the CAN bus looks as shown below:

- Own NodeID 5
- Monitored NodeID 1
- ID: 0x85 DLC = 8 Data: 0x30 0x81 0x81 0x00 0x01 0x00 0x00 0x00

**Emergency message Rx**

The declaration of the emergency message Rx consists of the following elements:

```
CanOpenAddEmergencyRx(
    CANNo:Int,        // Number of the bus line
    NodeID:Int,       // Node ID
    // Status, number of valid messages
    ref stCanOpenEmergencyStat:CanOpenEmergencyStat,
    // Array holding the emergency messages
    ref CanOpenEmergencyMSG:CanOpenEmergencyArray,
    ):int
```

**Example:**

The above program lines must be included into the corresponding tasks of your application program. The example below shows an emergency message from a device with node ID 21.

```
...
// Initializing the CAN bus once.

...

// Defining global variables
Var
    stCanOpenEmergencyMsg : ARRAY[5] of CanOpenEmergencyMsg;
    stCanOpenEmergencyStat : CanOpenEmergencyStat;
End_Var;

stCanOpenEmergencyStat.lBuffer := sizeof(stCanOpenEmergencyMsg);
iRet:= CanOpenAddEmergencyRx(0,                          // CANNo.
                             21,                         // NodeID
                             stCanOpenEmergencyStat,     // Status
                             stCanOpenEmergencyMsg);     // Array

...
```

**The above program lines produce the following result:**

When the device with node ID 21 receives an emergency message, the value in register 400100 switches from 0 to 1 (true).

Reset this value always to 0 (false). In doing so, you make sure that new emergency messages are displayed.

**Emergency message Tx**

The declaration of the emergency message Tx consists of the following elements:

```
CanOpenAddEmergencyTx(
    // Number of the bus line
    CANNo:int,
    // For error code see CiA DS 301 V4.02 page 60
    // or CiA DS 4xx (device profile)
    ErrorCode:word,
    // Error register (object 0x1001)
    ErrorRegister:byte,
    // 5 bytes can be used at the user's discretion
    ManufacturerArray:ByteArray5,
    // True = An error has occurred
    // False = Error has been cleared (acknowledged)
    bSet:bool
    ):Int;
```

## CANopen® object dictionary

**Supported objects**

The operating system of the CANopen® devices supports the following objects:

| Index (hex) | Object (code) | Object name | Type | Attribute s |
|---|---|---|---|---|
| 1000 | VAR | Device type | Unsigned32 | ro (read only) |
| 1001 | VAR | Error register | Unsigned8 | ro |
| 1002 | VAR | Manufacturer status | Unsigned32 | ro |
| 1003 | ARRAY | Pre-defined error field | Unsigned32 | ro |
| 1008 | VAR | Manufacturer device name | String | const |
| 1009 | VAR | Manufacturer hardware version | String | const |
| 100A | VAR | Manufacturer software version | String | const |
| 100B | VAR | Node ID | Unsigned32 | ro |
| 1017 | VAR | Producer heartbeat time | Unsigned16 | rw (read & write) |
| 1018 | RECORD | Identity | Identity | ro |
| 1200 | RECORD | Server 1 - SDO parameter | SDO parameter | ro |
| 1201 | RECORD | Server 2 - SDO parameter | SDO parameter | rw |
| 1203 | RECORD | Server 3 - SDO parameter | SDO parameter | rw |
| 1203 | RECORD | Server 4 - SDO parameter | SDO parameter | rw |

**Device Type object (index 0x1000)**

The structure of the **Device Type object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x1000 | 0 | 0x0000012D | Device type (read-only) |

**Error Register object (index 0x1001)**

The function `CanOpenAddEmergencyTx()` lets you set the bits in this register.

The structure of the **Error Register object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x1001 | 0 | 0 | Error register (read-only) |

This object implements the CANopen® error register functionality.

The following error messages may appear:

- Bit 0 = Generic error
- Bit 1 = Current error
- Bit 2 = Voltage error
- Bit 3 = Temperature error
- Bit 4 = Communication error (overrun, error state)
- Bit 5 = Specific device profile error
- Bit 6 = Reserved (always 0)
- Bit 7 = Manufacturer-specific error

**Pre-defined Error Field object (index 0x1003)**

The structure of the **Pre-defined Error Field object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x1003 | 0 | 0 | Number of errors entered in the array's standard error field |
| | 1 | 0 | Most recent error<br>0 indicates no error |
| | 2 ... 254 | - | Earlier errors |

This object shows a history list of errors that have been detected by the device. The maximum length of the list is 254 errors. The list content is deleted on restart.

**Composition of standard error field**

2-byte LSB: Error code

2-byte MSB: Additional information

**Manufacturer Device Name object (index 0x1008)**

The structure of the **Manufacturer Device Name object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x1008 | 0 | device name | Hardware name |

**Manufacturer Hardware Version object (index 0x1009)**

The structure of the **Manufacturer Hardware Version object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x1009 | 0 | | OS version of the device |

**Manufacturer Software Version object (index 0x100A)**

The structure of the **Manufacturer Software Version object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x100A | 0 | | Software version of the application program that runs on the device |

The entry in this index is made via the parameter **SWVersion** of the STX function `CanOpenInit()`.

**Node ID object (index 0x100B)**

The structure of the **Node ID object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x100B | 0 | | Node ID of the given device |

**Producer Heartbeat Time object (index 0x1017)**

The structure of the **Producer Heartbeat Time object** is shown in the following table.

| Index | Subindex | Default | Description |
|---|---|---|---|
| 0x1017 | 0 | 1,000 [ms] | Heartbeat time |

**CANopen® registers**

The table below lists the device registers associated with the CANopen® Object Dictionary.

The letter x in the register number represents the CAN bus number ranging from 0 ... CANMAX.

| Register number | Description | Value range | Attributes | Data type |
|---|---|---|---|---|
| 40x000 | Own node ID | 1 ... 127 | rw (read & write) | Int |
| 40x001 | Own heartbeat status | 0 = Bootup<br>4 = Stopped<br>5 = Operational<br>127 = Preoperational<br>255 = Offline | ro (read only) | Int |
| 40x002 | | Refer to object 0x1001 | ro | Int |
| 40x019 | | | ro | Int (IP format) |

| Register number | Description | Value range | Attributes | Data type |
|---|---|---|---|---|
| 40x020 | | | rw | Int |
| 40x021 | | | rw | Int |
| 40x022 | | | rw | Int |
| 40x023 | | | rw | Int |
| 40x030 | | | rw | Int |
| 40x100 | | | rw | bool |
| 40x400 | | | rw | bool |
| 40x101 ... 40x227 | Node ID 1 ... 127 Status | 0 = Bootup<br>4 = Stopped<br>5 = Operational<br>127 = Preoperational<br>255 = Offline (default) | ro | byte |
| 40x229 ... 40x355 | Node ID 1 ... 127 timeout | 0 ... 65535 ms | rw | word |

# 3 Jetter-specific use of CANopen® object dictionaries

**Purpose of this chapter**

This chapter contain a table giving an overview of the generally known CANopen® objects implemented by Jetter AG.

| Index (hex) | Object name | Object (code) | Type |
|---|---|---|---|
| 1000 | Device Type | VAR | Unsigned32 |
| 1001 | Error Register | VAR | Unsigned8 |
| 1002 | Manufacturer Status | VAR | Unsigned32 |
| 1003 | Pre-defined Error Field | ARRAY | Unsigned32 |
| 1008 | Manufacturer Device Name | VAR | String |
| 1009 | Manufacturer Hardware Version | VAR | String |
| 100A | Manufacturer Software Version | VAR | String |
| 100B | Node-ID | VAR | Unsigned32 |
| 1017 | Producer Heartbeat Time | VAR | Unsigned16 |
| 1018 | Identity | RECORD | Identity (23h) |
| 1200 | Server 1 - SDO parameter | RECORD | SDO parameter (22h) |
| 1201 | Server 2 - SDO parameter | RECORD | SDO parameter (22h) |
| 1203 | Server 3 - SDO parameter | RECORD | SDO parameter (22h) |
| 1203 | Server 4 - SDO parameter | RECORD | SDO parameter (22h) |
| 1600 | Receive PDO mapping Parameter | ARRAY | Unsigned32 (21h) |
| 1A00 | Transmit PDO mapping Parameter | ARRAY | Unsigned32 (21h) |
| 2000 | Features | ARRAY | Unsigned32 |
| 4554 | OS Update | ARRAY | Unsigned32 |
| 4555 | Electronic Datasheet | ARRAY | Unsigned32 |
| 4556 | System Parameters | ARRAY | Unsigned32 |
| 4557 | OS Status | ARRAY | Unsigned32 |
| 4559 | Detailed Software Version | ARRAY | Unsigned32 |
| 4565 | ENP SDO | ARRAY | Unsigned32 |