User Documentation:
CANopen for 3S Runtime Systems

Document Version 1.1

tech_doc_e.doc / V1.2

CONTENT

1 Preliminary Remarks

1.1 General

CoDeSys is one of the leading systems for programming PLCs. To make the system “easy to use” for the customers, some important features were integrated into the programming system. Among them there is a CANopen configurator. With the CANopen configurator (from now on also called “configurator”) it’s possible to configure CANopen networks within CoDeSys. The PLC programmed by CoDeSys can be a master or a slave in such a network.

Up to the middle of year 2000 the configuration data were transferred to the target system after a program download.

On the target system a software module had to receive, to interpret, to maintain and process these data. The core target system provided by 3S – Smart Software Solutions was not able to do the real CANopen processing. So many customers integrated a CANopen protocol software in the runtime system.

In order to eliminate the disadvantages of this approach, 3S – Smart Software Solutions realized an own, restricted CANopen implementation. That is available as a CoDeSys IEC 61131-3 library in order to make it usable for a wide circle of users. The library uses very simple CAN basic functions, which are named CAN drivers.

Implementing the CANopen functions in the form of a CoDeSys library allows an easy scaling of the target system. So a CANopen function only will consume target system resources if it is actually used. To save further target system resources CoDeSys automatically generates a data base for the CANopen-Master function which exactly matches the configuration. Further on the update of the inputs and outputs on the target system not any longer must be interpreted but gets realized by automatically generated code.

As from CoDeSys-Version 2.3 it is possible to configure a CoDeSys-programmable PLC as a CANopen-Slave (CAN device). For this purpose a new submodule, which will be available via an entry in the configuration description file (.cfg-file), was integrated in the configurator.

The libraries described in this document can be used as from CoDeSys Version 2.3.5.0.

1.2 CANopen terms and 3S implementation

According to CANopen in a CAN network there are no masters and slaves.

According to CANopen there is a NMT-Master, a configuration master etc., always keeping in mind, that all participants in a CAN network have equal rights.

The 3S implementation proceeds on the assumption that a CAN network serves as periphery of a CoDeSys-programmable controller. As a consequence CoDeSys in the CAN configurator names a PLC “CAN-Master”. This master is NMT-master and configuration master. Normally the PLC will mind that the network can be taken into operation. The master starts the particular nodes which are known to it via the configuration. These nodes are named “slaves”.

In order also to approach the master to the status of a CANopen-node, an object dictionary for the master was introduced. The master also can act as a SDO server, not anymore only as a SDO-Client in the configuration phase of the slave. (Further details in a later chapter.)

1.3 Structure of the document

This document consists of three partitions: The first one describes the CANopen-Master library, the second describes the CANopen-Slave-library and there is a separate chapter for the CAN network variables.

The library descriptions each are structured as follows:

- Overview on the features of the library.
- Description of the default steps for working with the library.
- Description of the possibilities which the application has to influence the CAN stack and to get information from the stack.
- Overview on possible error states and their causes.

2 CANopen-Master library

2.1 Differentiation from other CANopen libraries

The library implemented by 3S – Smart Software Solutions differs in several respects from other systems available on the market. It has not been developed to make abundant other libraries which are provided by prestigious manufacturers, but pointedly has been optimized for the use with the CoDeSys programming and runtime systems.

The libraries are made according to the CiA DS301, V401 specification.

For users of the CoDeSys-CANopen-library the following advantages result:

- The implementation is independent from the target system and so usable on every CoDeSys-compatible PLC without changes.
- The customer gets a complete system from one distributor. This means, the configurator and the embedding of the CANopen-protocol into the runtime system are optimally coordinated.
- The CANopen-functionality is loadable.
  This means, the CANopen-functions can be reloaded and updated without changing the runtime system.
- The resources are used carefully. Memory is allocated depending on the used configuration, not for a maximum configuration.
- Automatic IO-update without additional measures.

The following functionalities, defined by CANopen, are supported by the CANopen-library of 3S – Smart Software Solutions (May 2005):

- Sending PDOs from the master => to the slaves
  Event-controlled (on change), time-controlled (repeat-timer) and as synchronous PDOs, i.e. each time a sync has been sent by the master as often as defined by the transmission type. Even an external sync-source can be used to initiate the sending of synchronous PDOs.
- Receiving PDOs (=> by the master) from the slave.
  Dependent on the slaves: event-controlled, request-controlled, acyclic and cyclic.
- PDO mapping
  if supported by the slaves
- SDO sending and receiving (unsegmented, i.e. 4 bytes per object dictionary entry)
  Automatic configuration of all slaves via SDOs at system start
  Application-controlled sending and receiving of SDOs to configured slaves

- Synchronisation
  Automatic sending of sync messages by the CANopen-Master
- Nodeguarding
  Automatic sending of guarding messages and guarding the lifetime for each correspondingly configured slave
- Heartbeat
  Automatic sending and guarding of heartbeat messages.
- Emergency
  Receiving and storing of emergency messages from the configured slaves
- The direct communication between slaves via PDOs is possible, but must be configured „manually“.

The following functions defined in CANopen are currently not supported by the 3S – Smart Software Solutions CANopen library (State: May 2005):

- Dynamic identifier assignment
- Dynamic SDO connections
- Blockwise SDO transfer
- The guarding of more than one heartbeat from the CAN Device viewpoint.
- Network variables according to CANopen specification DS302
- Modules according to DS405.
- All possibilities of the CANopen protocol not mentioned above

2.2 CANopen for the user

2.2.1 Used modules

To realize the CANopen-library easily and , the following modules are necessary:

For a simple and practical realization of the CANopen function under a CoDeSys programmable PLC the following modules are required:

- A CoDeSys programming system, as from version V2.3 SP5, with integrated CANopen-configurator.
- The STANDARD.LIB, providing the standard functions defined in IEC-61131 for the PLC.
- In CoDeSys programmable multitasking systems the libraries SysLibSem.lib (semaphores for the exclusive processing of certain code sections) and SysLibCallback.lib (IEC-function calls at certain events) are used to get a multitasking-save access on CAN.
- EDS-files for all slaves which are part of the network. The EDS-files are provided by the hardware manufacturer of the CANopen-Slaves.
- The library 3S_CANDRV.Lib, realizing the CAN base functions. The library is hardware-dependent and must be available as external (or internal) CoDeSys library.
  This library usually is provided by the PLC manufacturer and has a uniform interface to 3S_CANOPEN.lib.
- The library 3S_CanOpenManager.Lib, providing the CANopen base functionalities.
- One or several of the libraries 3S_CanOpenNetVar, 3S_CanOpenDevice and 3S_CanOpenMaster. Depends on the desired functionality.

2.2.2 How to create a project with CANopen

The creation of a new project with CANopen will be described step by step in the following. It is assumed that the CoDeSys programming system is installed and that also the target- and EDS-files have been installed resp. copied correctly.

2.2.2.1 Creating a new project and target selection

After having started CoDeSys choose => **File | New**

In the Target Settings dialog select the desired target system.

Then in tab **General** activate option **Support CANopen configuration** (if available; otherwise it should be made sure that the CANopen Configuration is supported by the target system, and just not switchable off by the user; if this is not the case the target system can not be used as CANopen-Master.)

2.2.2.2 Create the PLC application

After having closed the target settings dialog by OK, CoDeSys automatically opens a dialog for adding the main POU PLC_PRG. Here you can program an application.

2.2.3 Including the required libraries

The needed libraries 3S_CanOpenManager.LIB and 3S_CANDRV.LIB and one or several of the libraries 3S_CanOpenNetVar/Master/Device.lib must be included in the project. For this purpose use command **Window | Library Manager** and then for each of the libraries **Insert | Additional Library…** from the context menu of the library manager or from the menu bar.

Note:
When you insert one of the libraries 3S_CanOpenNetVar/Master/Device.lib, all libraries referenced by this library will be included automatically.

Including the libraries is an important step. By the presence of the libraries CoDeSys can recognize that a CANopen project is done, for which then special implicit program parts will be created automatically.

A description on the CoDeSys CAN configurator you find in the CoDeSys online help. See there how to configure a CANopen-Master and slaves.

2.3 Functionality of the libraries at runtime

2.3.1 Implicit services of the CANopen-Master library

The CANopenMaster library provides implicit services for the CoDeSys application that are sufficient for most of the implementations. These services are integrated transparently to the user and are available within the application without the need of explicit calls.

Among those services there are the following:

- Reset of all configured slaves at system's startup — Standard
  In order to reset the slaves, per default the NMT command "Reset Remote Node" is used, explicitly and singly for each slave. (As per CANopen NMT Network Management. The particular commands are described in document DSP301.) The experience of some users has shown that it is reasonable to reset the slaves with a ‚Reset All Remote Nodes' command, in order not to overstress the receive capacities of lower-performing slave controllers. This is possible by setting the appropriate flag in the application program.
- Polling the slave device type via SDO (Requesting object 0x1000) and comparing it to the configured Slave ID. — Standard
  Issuing an error state for those slaves that do not fit the configured type. If no device type has been received at all, the polling will be repeated after 0,5 seconds, except the slave is marked as "optional" in the configuration.

• Configuration of all error-free devices via SDO. Each SDO is watched for a response and will be repeated if the slave does not respond within the watch time.	Standard
• Automatic configuration of slaves via SDOs that send a boot-up message to the master while the bus is in operating mode.	Standard
• Start all slaves after having configured them faultlessly. For starting normally the NMT-command “Start remote node” is used. As is the case with Reset this command can be replaced by “Start all remote nodes”.	configurable
• Cycling sending of the sync message.	configurable
• Nodeguarding with lifetime-supervision for each slave and producing an error state for slaves at which the lifetime-supervision failed.	configurable
• Heartbeat of the master to the slaves and watching the heartbeats of the slaves.	configurable
• Receiving of emergency messages for each slave and storing the last received emergency messages separately for each slave.	Standard
• Receiving of PDO messages and task-consistent data transfer to the process image of the application.	Standard
• Sending PDO messages after end of the task, depending on the set „Transmission Type“ des PDOs.	Standard

2.3.2 Explicit services of the CANopen library

In addition to the implicit services described above, the CANopen library provides the following functions:

• Indication of the recently received PDOs via a flag which can be removed by the application.	Standard
• Application-triggered transmission of remote telegrams for receive-PDOs	Standard
• Application-triggered transmission of SDO objects to slaves	Standard
• Application-triggered reset of slaves with subsequent reconfiguration of slaves	Standard
• Specifically still possible: The application can send and directly receive own CAN messages to/from the bus. (Layer 2)	Standard

2.3.3 Starting the CANopen network

After a project download to the PLC or after a reset of the application the CAN network will be re-started by the master.

The restart always is done with the same sequence of actions:

- All slaves are reset, except they are marked in the configurator as not to be reset. Resetting is done singly by the NMT-command „Reset Node“ (0x81), always with the NodeID of the slave. Exception: the application has set flag „bUseResetAllNodes“; in this case the command for restarting the network is used once with NodeID 0, „Reset All Nodes“.
- All slaves get configured. For this purpose initially object 0x1000 of the slave gets polled. If the slave responds within the wait time of 0,5 seconds, the next configuration SDO will be sent. If a slave is marked “optional” and does not respond to the request within the wait time, it will be marked as “not existent” and no further SDOs will be sent to it. If a slave responds on the request for object 0x1000 with another type than configured in the lower 16 bits, it will be configured nonetheless, but marked as wrong type.
- All SDOs (including the polling for object 0x1000) are sent repeatedly until an answer of the slave is detected within the wait time. The application can watch the boot-up of the particular slaves and react if necessary. (see below)
- If the master has configured a heartbeat time unequal 0, the generation of the heartbeat will start immediately after the start of the master PLC.

- After all slaves have received their configuration SDOs, the nodeguarding will be started for those slaves that have been configured to be guarded.
- If the master has been configured for automatic startup, now each slave will be started individually by the master. For this purpose the NMT command “Start Remote Node“ (0x1) is used. If the application sets the flag „bUseStartAllNodes“, the command will be used with NodeId 0 and thus all slaves will be started with „Start all nodes“.
- At least once all configured Tx-PDOs (for the slaves these are the Rx-PDOs) will be sent. (It is always on the application to send RTRs for RTR-PDOs. For this purpose a method of the Rx-PDOs is available.)

2.4 Working with the CANopen-Master part in the application

There are various cases in which the IEC application must cooperate with the CANopen library. Among those is the detection of and reaction on an error.

Regard thereby the call sequence, which cannot be influenced by the application programmer. (The calls and in which tasks they are generated is also described in chapter 2.5, Implicit Calls’.)

Here an example for a CAN controller and a Rx/Tx-PDO each:

- CanRead(0); (* Implicitly created call *)
- pCanOpenMaster[0](); (* Implicitly created call *)
- pCanOpenPDO_Rx[0](); (* Implicitly created call *)
- <processing of the application code>
- pCanOpenPDO_Tx[0](); (* Implicitly created call *)
- MgrClearRxBuffer(wCurTask:= 1,wDrvNr := 0, dwFlags := 0, dwPara := 0); (*Implicitly created call for clearing the receive buffer.*)

2.4.1 Structure of the CANopen-Master

If the libraries 3S_CanOpenMaster/3S_CanOpenManager and 3S_CanDrv.lib are included in a project, CoDeSys implicitly (automatically) will create a Global Variables List and before/after the application code in certain tasks will add calls of library modules. The Global Variables List is named „CanOpen implicit Variables“ and will be filled by CoDeSys with the appropriate data from the PLC configuration.

Note: To view the initialization code in ST without going online, CoDeSys can be started by the command line option “/debug”. In this case (besides other useful information) the files CanOpenInitcode.exp and CanOpenBeforeTask/AfterTask_<Taskname>.exp will be in the compile directory.

The variables list „CanOpen implicit Variables“ in details:

The constants always indicate the bounds of the array which are declared in this variables list.

```
VAR_GLOBAL CONSTANT
    MAX_CTRLINDEX : INT := <Maximum Index of the CAN controller in the CANopen
                           configuration; 0, if there is one controller; 1 if there are two
                           controllers etc.>;
END_VAR

VAR_GLOBAL CONSTANT
    USE_CANOPEN_NODES : BOOL :=   <TRUE indicates whether there is at least 1 Slave below
                                  a master>;
    MAX_MASTERINDEX : INT :=      <Index in the controller table (in the Global Variables
                                  List of the CANopen-Manager library) the controllers are
                                  assigned to a master and not to a CanDevice. (for
                                  CanDevice see chapter 3) >;
    MAX_NODEINDEX : INT :=        <Size of the CanOpenNodes array>;
    MAX_SDOINDEX : INT :=         <Size of CanOpenSDOs array>;
```

```
    MAX_PDOINDEX_RX : INT :=        <Number of RX-PDOs which are available all over all
                                    CANopen-Masters>;
    MAX_PDOINDEX_TX : INT :=        <Number of TX-PDOs available all over all CANopen-
                                    Masters>;
    MAX_MASTER_ODENTRY_IDX : INT := <Number of object dictionary entries all over all
                                    CANopen-Masters>;
END_VAR
```

The configuration data are stored in the following arrays:

```
VAR_GLOBAL
    pCANopenMaster : ARRAY[0..MAX_MASTERINDEX] OF CanOpenMaster;
    pCanOpenNode : ARRAY[0..MAX_NODEINDEX] OF CanOpenNode;
    pCanOpenSDO : ARRAY[0..MAX_SDOINDEX] OF CanOpenSDO;
    pCanOpenPDO_Rx : ARRAY[0..MAX_PDOINDEX_RX] OF CanOpenPDO_Rx;
    pCanOpenPDO_Tx : ARRAY[0..MAX_PDOINDEX_TX] OF CanOpenPDO_Tx;
    ODMEntries: ARRAY[0..MAX_MASTER_ODENTRY_IDX] OF CanOpenODEntry;
END_VAR
```

Thereby the function block member variables are used as follows: (pure internal variables, declared in the VAR-sections, should not be written by the application; however some of them are of interest for diagnosis purposes.)

```
VAR_INPUT
    nStatus : INT;              (* Current status of the master. This status has nothing to do with the
                                   states defined according to CANopen. It is an own, internal definition.
                                   *)
    bMsgUsed: BOOL;             (* Can be used by the application in order to detect that a new SDO-
                                   Client request has been processed. Subsequently e.g. the object
                                   dictionary of the master can be inspected for changes or can be re-
                                   evaluated. (Only valid if the appropriate entries in the CoDeSys *.cfg-
                                   file effect that the master has an object dictionary. Otherwise without
                                   any meaning.)*)
    nRxIndex : INT;             (* Historically, in order to avoid compile errors in old projects; not
                                   used.*)
    wDrvNr : WORD;              (* Needed for the configuration: Number of the CAN controller, given
                                   from the library to the CAN driver in order to inform the driver on which
                                   controller the write and read requests refer to. The application may not
                                   write this parameter.*)
    bUseStartAllNodes : BOOL; (* Can be used by the application to determine whether the library
                                   should use command “Start Node”(FALSE) or “Start All Nodes”
                                   (TRUE). The application only once may write this flag, preferably
                                   during the first few cycles after start. Cyclic writing causes that single
                                   slaves will not be restarted after a failure.*)
    bUseResetAllNodes : BOOL; (* Analog to bUseStartAllNodes*)
END_VAR

VAR_OUTPUT
     bError : BOOL;             (* Historically, not used.*)
END_VAR

VAR (* Konfig *)
    sDrvName : STRING(40);      (* Historically, not used.*)
    wBaudrate : WORD;           (* Baudrate as used in the configuration, entered by CoDeSys.*)
    nFirstNodeNr : INT;         (* Index of the first slave belonging to the current master, entered by
                                   CoDeSys.*)
    nLastNodeNr : INT;          (* Index of the last slave belonging to the current master, entered by
                                   CoDeSys.*)
    SyncTimer : TON;            (* Timer used for the generation of the sync message.*)
    dwCOBID_Sync : DWORD;       (* COBID of the sync message as used in the configuration;
                                   entered by CoDeSys.*)
    dwHeartbeatTime : DWORD;(* Heartbeat generation time in ms, entered by CoDeSys.*)
```

```
    nNodeId : WORD;             (* NodeId of the master as used in the configuration, entered by
                                CoDeSys.*)
    HeartbeatTimer  :TON;       (* Timer which is used by the master for heartbeat generation.*)
    bSendHeartbeat : BOOL;      (* Internally set and reset by the master whenever a heartbeat
                                    message should be sent and as soon as it has been sent.*)
    byHeartbeatState : BYTE;    (* According to the state of the master here the CANopen-
                                conforming states 0, 4, 5 or 127 are entered, which are sent with the
                                heartbeat-message.*)
    wODMFirstIdx : WORD;        (* Index of the first MasterOD-entry in array ODMEntries.*)
    wODMCount : WORD;           (* Number of ODMEntries belonging to this master.*)
END_VAR
VAR
    bAutoStart : BOOL;          (* TRUE set by CoDeSys if the Autostart option of the master
                                has been set. *)
    bReentry: BOOL;             (* Historically, not used.*)
    bGefunden : BOOL;           (* Historically, not used.*)
    dwCOBID_NMT : DWORD := 16#0000; (* Exclusively used for starting the slaves. Never should
                                be changed by the application.*)
    nIndex : INT;               (* internal auxiliary variable of the master.*)
    dwMerker : DWORD;           (* historically, not used. *)
    dwSem : DWORD := 16#FFFFFFFF; (*    Intended for locking operations in multi-tasking
                                operation. Not used because the implicit calls always are executed
                                by one task only.*)
    a: INT;                     (* Lifecounter, only counted up in order to make the implicit calls
                                visible in monitoring mode. Herewith you can check whether the
                                master is called.*)
    bSynchSend : BOOL;          (* Always will get TRUE for the duration of one IEC-cycle, when
                                the master sends a sync message. The same also happens when
                                the master receives an externally generated sync message. If the
                                master itself generates the sync message, the flag will not be set
                                when receiving an external sync.*)
    bErrCodeNot0 : BOOL;        (* Used to delete the error code of the CAN driver for BusOFF
                                    (Errorcode = 1) with an offset of 1 cycle.*)
    MsgBuffer: CAN_Message      (* Receive buffer of the master. Only used for the SDO server
                                functionality and for receiving an external sync message.*);
    bSDOMsgUsed : BOOL;         (* Always set in case of an access on the object dictionary of the
                                master.*);
    bSDOReadrqActive : BOOL; (* The following variables are used in order to manage the
                            accesses          on the object dictionary of the master.*)
    bSDOWriterqActive : BOOL;
    bSub0NotFound, bSDOReadrspAbort, bInitiateRspSend, bSDOWriterspAbort,
                            bInitiateWrReqSend : BOOL;
    i, iActiveSegSDORead : INT;
    ucModus : BYTE;
    dwIdx : DWORD;
    wSegSDOReadSendOffs,wSegSDOReadSendLen : WORD;
    pActiveSegSDORead:POINTER TO BYTE;
    iActiveSegSDOWrite:INT := -1;
    pActiveSegSDOWrite:POINTER TO BYTE;
    wSegSDOWriteRecvOffs : WORD;
    wSegSDOWriteRecvLen : WORD;
    dwODEValue : DWORD;
    iSDOReadLen, iSDOWriteLen : INT;
    byAbortCode : ARRAY[0..3] OF BYTE;
    bSwap : BOOL;           (* Here the master indicates whether it is running on an INTEL or a
                            Motorola-byteorder machine.*)
END_VAR
```

2.4.2 Structure of the CANopen-node (Slave)

The access on the CANopen-nodes is performed via the implicit variable pCanOpenNode created by the programming system. This variable is an array of function blocks representing the configured nodes.

Thus methods of the nodes, like e.g. ResetNodes, are called as follows:

```
pCanOpenNode[0].ResetNode();
```

„0“ is the index of the node, not the NodeId. The NodeId is part of the function block.

All instance data of a node can be accessed reading and (partly) writing. (For this purpose please open the module interface in the library manager in CoDeSys).

The components of the function block in detail:

```
VAR_INPUT
    nRxIndex : INT;                 (* historically, not used.*)
    nSDOSend : INT;                 (* Index of the current SDO, which is sent to the node during the
                                       configuration phase.*)
    bAutoStart : BOOL;              (* Always set TRUE as soon as the master sets the node to
                                       automatic start. *)
    nNewStatus: INT;                (* Status parameter used when action SetNodeStatus is called.*)
    bSynchSend: BOOL;               (* Transferred by the master at call of the module in order to
                                       indicate the receiving of an externally generated sync message or
                                       the generation of  a sync message by the master. Only used
                                       internally and immediately reset afterwards.*)
    dwHeartbeatTime : DWORD;        (* Interval in msec, according to which the master guards the
                                       receiving of a heartbeat message of the node. The configurator will
                                       enter the heartbeat time by using 1,5 times of the heartbeat time of
                                       the node.*)
    bSendReset: BOOL := TRUE;       (* Per default set by the master, except it wants to use
                                       “Reset All Nodes”. In this case the reset is switched off by the
                                       module.*)
END_VAR
VAR (* Konfig *)
    wDrvNr : WORD;                  (* Number of the CAN controller used by this module.*)
    wMasterIdx : WORD;              (* Index of the master, below which the current slave is running.*)
    ucNodeNr : BYTE := 1;           (* NodeID of the slave, as used in the configuration.*)
    dwNodeIdent : DWORD;            (* Type of the node, which is also saved in object 0x1000 of the
                                       node.*)
    dwGuardCOBID : DWORD;           (* COBID of the guard telegram; used when sending nodeguard
                                       telegrams to the slave.*)
    dwEmergCOBID : DWORD;           (*Emergency COBID; used when waiting for emergency telegrams
                                       of the slave.*)
    wDiagSegment : WORD;            (* In the future used for storing the diagnosis address.*)
    dwDiagOffset: DWORD;
    wFirstSDOIndex : WORD;          (* Range of configuration SDOs for this slave within the array of
                                       SDOs; entered by the configurator.*)
    wNummOfSDO : WORD;              (* Number of SDOs, generated by the configurator for this slave.*)
    wFirstRxPDOIndex : WORD;        (* In these and the following three components the PDOs for this
                                       slave will be identified.*)
    wNummOfRxPDO: WORD;
    wFirstTxPDOIndex: WORD;
    wNummOfTxPDO: WORD;
    GuardTime : TIME;               (* Nodeguard time as used in the configuration.*)
    GuardTimer : TON;               (* Timer for generation of the guard message.*)
    LifeTimer : TON;                (* Timer for guarding the slave. If within the guard time not at least
                                       one answer on a nodeguard telegram is received, a guard error
                                       will be issued. *)
    bDoInit : BOOL := TRUE;         (* Set to FALSE by the configurator, if option “Do not initialize” is
                                       set.*)
```

```
    bOptional :BOOL;              (* Set to TRUE by the configurator, if option "optional device" has
                                     been set. *)
    bHeartbeatConsumer: BOOL; (* Not used. Reserved for information purposes.*)
    tHeartBeatLifeTime : TON;     (* Watch time for the heartbeat messages of a slave. Always
                                     restarted as soon as a heartbeat message coming from the slave
                                     has been received. *)
END_VAR
VAR_OUTPUT
    byLastState : BYTE := 0;      (* Last status, as contained in the answer of the node on the last
                                     guard message.*)
    nStatus : INT;                (* Current status of the node
                                     0:  undefined.
                                     1:  Node will be reset by the master and re-configured.
                                         Master waits on a bootup-message of the node and sets (or
                                         after expiry of the given guard time ) on status 2.
                                     2:  Master waits ca. 300 msecs before requesting object 0x1000.
                                         Then the status is set to 3.
                                     3:  Master starts the configuration of the slave with a SDO-read-
                                         request (index 0x1000) to the node. All SDOs generated by the
                                         configurator will be sent to the node in this status. The
                                         generated SDOs are packed in an array of SDOs, where the
                                         node knows its first SDO and the number of its SDOs. (Instance
                                         components „wFirstSDOIndex" and „wNummOfSDO")
                                     4:  Node gets sent a „Start Node", depending on whether in the
                                         configuration of the master "Autostart" is activated or not.
                                     5:  Node receives/sends PDOs, Master also. Normal operation.
                                     97: Optional node has not been detected at start-up.
                                     98: Node has answered on a request for the device type
                                         (object 0x1000) with a different type.
                                     99: LifeTimer is exceeded, Nodeguarding error.*)
    nSDOActiv : INT;              (* Index of the active SDO, managed by the node; -1 means that no
                                     SDO is active *)
    EmcyMsg : CAN_Message;    (* Last emergency message received from the node.*)
    sdoConfig : CanOpenSendSDO; (*Instance of the CanManger-Lib function block, used for the
                                     configuration of the node. *)
END_VAR
VAR
    nSDOIndex : INT;              (* Current index of the configuration SDOs.*)
    bReady : BOOL;                (* Always when an emergency or guard message has been
                                     received,  bReady is set TRUE for the duration of one cycle.*)
    SDO_TimeOut: TON := ( PT := T#500ms );
    dwMerker : DWORD;             (* Lower 16 bits of object 0x1000, as read from the device.*)
    nStatusOld : INT;             (* Internal variable, used to detect a status change.*)
    bManualStart : BOOL := FALSE; (* Internally used by StartNode for advising the function block to
                                     start the slave.*)
    bDevTypeInvalid : BOOL;       (* TRUE indicates that the device type given back by the node on
                                     the request of object 0x1000, and the device type entered in the
                                     PLC Configuration are not identical. By setting the node status
                                     (which should be 98 if  bDevTypeInvalid is TRUE) manually (or by
                                     the IEC program) to 4, the node nevertheless can be activated. The
                                     application cannot write variable nStatus of a node. For this
                                     purpose the method „NodeStart" or SetNodeStatus(<new state>) of
                                     the nodes must be used.*)
    MsgBuffer : CAN_Message;  (* Receive buffer of the node. Internally, only used for special
                                     messages.*)
    a: INT;                       (* Counter, just used for indicating whether the module is called.*)
    iPDO : WORD;                  (* Internal variable, used for browsing all PDOs belonging to the
                                     current slave.*)
END_VAR
```

2.4.3 Interpretation and reaction on the states of the library

At start-up and during the operation of the CANopen network the library respectively the particular function blocks are running through different states.

During monitoring in CoDeSys these states can be seen in the global variables list „Can Open implicit variables“. Here the masters and nodes configured in the system are put into arrays of function blocks.

In the following the states of the variables are described in order to be able to recognize some standard situations. The respective reactions have to be handled by the application program.

2.4.3.1 Start-up of the CANopen-Master

During start-up of the CAN network the master runs through various states which primarily can be read from variable nStatus.

States **0**, **1** and **2** are run through by the master automatically within the first cycles after a PLC start. Status **3** of the master is kept for some time. In status 3 the master configures its slaves. For this purpose the slaves get sent one after the other all SDOs which have been created by the configurator. After having transmitted all configuration-SDOs to the slaves, the master changes to status **5** and remains in this status. Status 5 is the normal operation status for the master. After once having reached this status, the master remains in status 5. All other states are managed by function blocks representing the slaves.

If a slave does not respond on a SDO request (upload or download), the request will be repeated by the respective function block. The master leaves status 3, as mentioned above, but not until all SDOs have been transmitted successfully. Thus it can be detected whether a slave is missing or cannot correctly receive all SDOs. (Thereby it is irrelevant for the master whether a slave answers with a confirmation or with an abort, for the master it is only of concern whether any answer has come from the slave. An example for the detection of aborts during start-up see below.)

An exception is a slave which is marked ‘optional’. Optional slaves are only requested once for object 0x1000. If they do not respond within 0,5 seconds, the slave at this time will be ignored by the master and the master will change into status 5 without any reaction of the slave. If thenceforwards the slave is wanted to get re-configured by the master or to be checked for presence, the application must do that. (See add-on in 2.4.4.2, Boot-up of the network without automatic start.)

2.4.3.2 Boot-up of the CANopen-Slaves

A slave is represented by a function block instance in array pCanOpenNodes in the global variables list „Can open implicit Variables“. During boot-up of the CAN network the slave automatically runs through states -1, 1 and 2. These states mean the following: (Regard: they partially are internal states, which might be not detected by the application because they are only valid for the length of one cycle.)

- **-1** Node is reset by NMT-message „Reset Node“ and autonomously changes to status 1.
- **1** Node changes to status 2 after a maximum time of 2 sec. or immediately after having received its boot-up message.
- **2** Node automatically changes to status **3** after a delay time of 0,5 sec. This delay matches the experience that many CANopen devices not immediately after having sent their boot-up message are ready for receiving their configuration-SDOs. In status **3** the slave gets configured.

The slave remains in status 3 until having received all SDOs created by the configurator. Thereby it does not matter, whether the slave has responded to SDO transfers with an abort (error) or error-free. Decisive is just the fact of having got any response from the slave. The application can guard the responses on the SDO requests and itself can react on the responses of the slaves.

If option „Reset nodes“ is activated in the configurator, a reset of the node will be done after the transmission of object 0x1011 subindex 1, which after that gets value „load“. Thereby the library enters „0“ to this object in order to get the reset executed only once after the initialization of the application. Thereupon the slave will be requested by an upload of object 0x1000.

After having run through the configuration phase, the slave can change to the following states:

- It always changes to status **4**, with the following exceptions: It is an optional slave not detected as being available at the bus (request object 0x1000). It is available but on the request of object

0x1000 it has responded with a different type in the lower 16 bits than expected by the configurator.

- It changes to status **97** if it is optional (option „Optional device“ in the CAN configuration) and has not reacted on the SDO request for object 0x1000.
- It changes to status **98** if the device type (object 0x1000) does not match the configured one, after nevertheless having got all configuration-SDOs. (Concerning starting the nodes in this case please see add-on in 2.4.4.2, Boot-up of the network without automatic start.)

If the master is configured for automatic start, the slave will be started in status **4** (i.e. a „Start Node“-NMT-messages will be generated) and the slave will automatically change to status 5. Status **5** is the normal operation mode of the slave. If the master flag bUseStartAllNodes has been set by the application, then it will be waited until all slaves are in status 4 and after that all slaves will be started with NMT command „Start All Nodes“.

If the slave is in status **4** or higher, nodeguard messages will be sent to the slave, if nodeguarding is configured.

2.4.3.3 Nodeguarding / heartbeat errors

In case of a nodeguarding timeout the variable nStatus of the node will be set to 99. In order to restart the slave it is sufficient to reset the state of the node to 4. Therewith the node gets sent a „Start communication“ and the CANopen-Master restarts communicating. For this the method „NodeStart“ is used.

The master autonomously will do this as soon as the node restarts to react on nodeguard requests and if option „Autostart“ is activated. Thereby the node will be reconfigured or just restarted, depending on its status, which is contained in the response on the nodeguard requests.

If option Autostart is not activated, the application must call method NodeStart. Depending on the last saved status (which always is transmitted with the Heartbeat-/nodeguard message), the node will be restarted or reset.

For heartbeat errors the same proceeding is applicable.

2.4.3.4 Node does not respond in configuration phase

The application can time-guard the configuration phase of a slave and can set nodes, which remain in status 3, to „TimeOut“. The application can achieve this by calling method „NodeSetTimeoutState“ of the node (e.g. pCanOpenNode[0]. NodeSetTimeoutState();).

With NodeSetTimeoutState the status of the node will be set to 97. Thus subsequently the node can be treated like an optional node.

As soon as all configured nodes have reached a status > 4, the master changes to status 5, normal operation, where an exchange of process data via PDOs is done.

2.4.3.5 Evaluation of the responses of the slaves on the configuration SDOs

The application can log the complete configuration phase of a node, thus can „see“ each response. The responses on the configuration SDOs appear in sdoConfig of the node structure: for example in

pCanOpenNode[0].sdoConfig.ucAnswerBytes[0..7] (see also: description of function block CanOpenSendSDO.)

The master sends one SDO after the other as soon as a response from the slave has been received. Therefore the application must detect the change of the response in order to determine when a new SDO response has come in:

```
VAR
  ucOldAnwer : array[0..7] of BYTE;
  bChange : bool;
  i : INT;
END_VAR
FOR i := 0 to 7 DO
  IF pCanOpenNode[0].sdoConfig.ucAnswerBytes[i] <> ucOldAnswer[i] THEN
        ucOldAnswer := pCanOpenNode[0].sdoConfig.ucAnswerBytes;
        (* Here the new response can be evaluated, if desired by the application.*)
```

```
            (* Information on the abort codes see in the description of CanOpenSendSDO.*)
            EXIT; (* Leave loop at first change.*)

        END_IF
    END_FOR
```

2.4.4 Examples for application-controlled actions of the libraries

In the following some standard actions are described, which often have to be done by applications.

2.4.4.1 SDO Transfer via function block „CanOpenSendSDO“

In order to control the initiation of a SDO transfer by the application, use module CanOpenSendSDO of the CanOpenManager library.

Interface of the module:

```
VAR_INPUT
    Enable  : BOOL;       (* Rising edge at this flag.*)
    wDrvNr : WORD;        (* Index of the CAN controller to be used. 0 for the first one.*)
    ucNodeId : BYTE;      (* NodeId of the SDO server.*)
    wIndex  : WORD;       (* Index of the object dictionary entry.*)
    bySubIndex : BYTE;    (* SubIndex*)
    ucModus: BYTE;        (* SDO-mode, 16#40 for read-request,
                             use 16#23 for 4-byte-write-request.
                             use 16#27 for 3-byte...
                             use 16#2B for 2-byte...
                             use 16#2F for 1-byte...
                             use 16#21 for a download with more than 4 Bytes.*)
    ucByte0  : BYTE;      (* The 4 possible data bytes in the expedited transfer. *)
    ucByte1  : BYTE;
    ucByte2  : BYTE;
    ucByte3  : BYTE;
    aAbortCode : ARRAY[0..3] OF BYTE;

    (* Additional input parameters in case of a segmented transfer. *)
    (* For this mode 16#41, 16#40 for segmented reading and 16#21 for segmented writing. In
    these cases a buffer must be defined for the data.*)
    dwDataBufferLength : DWORD;
    pDataBuffer : POINTER TO BYTE;
END_VAR

VAR_OUTPUT
    bWaitForAnswer : BOOL;  (* During transmission this flag is set TRUE at least for the
                               duration of one cycle and after the transmission it is set FALSE.*)
    bAnswerRec : BOOL;      (* Only in case of an error-free transmission this flag is set TRUE.*)
    ucAnswerBytes : ARRAY[0..7] OF BYTE;
    iAnswerLength : INT;
    bToggleUnequal: BOOL;
    bAbortRec:BOOL;         (* If an there was an abort, this variable will be set TRUE; the abort
                               code appears in aAbortRec.*)
    aAbortRec : ARRAY[0..3] OF BYTE;
END_VAR

VAR
    EnableOld : BOOL;
    bAnswer : BOOL;
    n: INT;
    dwActiveCOBId : DWORD;
    dwAnswerId : DWORD;
    Buffer : CAN_Message;
    bExpedited : BOOL;          (* Here the module shows whether the reading has been
                                   done via an expedited or a segmented transfer.*)
    bWriteActive : BOOL;
```

```
        bReadActive : BOOL;
        dwDataOffset : DWORD;          (* After the data transfer the number of actually read
                                          resp. written bytes appears in this variable.*)
        dwDataReadLength : DWORD; (* The estimated length of a SDO upload (from the
                                          1. response message).*)
        ucUploadRequest: BYTE;
        bLastToggle: BOOL;
        iCurSDODataLen : INT;
        ucDownloadRequest: BYTE;
END_VAR
```

In order to be able to use this module, for each simultaneous transfer a static instance of the module must be created (static means: not as a local variable of a function). It is not possible (per CANopen definition) to open multiple SDO channels to the same slave at the same time.

Send SDO (expedited Mode):

The input parameters to be passed to the module: address (NodeID) of the target node, (0-based) number of the CAN network on which the module should work, the 4 data bytes, index and subindex of the target object. Further on the mode in which the module should work must be passed: For sending of

- 4 Bytes: mode 16#23,
- 3 Bytes: mode 16#27,
- 2 Bytes: mode 16#2B,
- 1 Byte: mode 16#2F.

Example: If a 4-byte value should be written to a SDO server, the call of the module must look like shown in the following:

```
sdo(
Enable:= TRUE,
wDrvNr:= 0,
ucNodeId:= 2,
wIndex:= 16#4002,
bySubIndex:= 0,
ucModus:= 16#23,
aAbortCode:=aAbort,
ucByte0 := DWORD_TO_BYTE(dwWrite),
ucByte1 := DWORD_TO_BYTE(SHR(dwWrite,8)),
ucByte2 := DWORD_TO_BYTE(SHR(dwWrite,16)),
ucByte3 := DWORD_TO_BYTE(SHR(dwWrite,24)));
```

Thereby sdo is an instance of CanOpenSendSDO.

To check whether the transfer has been terminated for example could be programmed like follows::

```
IF (sdo.bAnswerRec OR NOT sdo.bWaitForAnswer) then
        ….. (* Transfer is terminated. If bAnswerRec is not TRUE, an error has occurred.*)
END_IF
```

Receive SDO:

Like Sending SDO, except that 4 x 0 data Bytes are passed and mode 16#40.

Each other input value for „Mode“ will lead to an undefined behaviour of the module.

Thus the call of the module e.g. could look as follows:

```
sdo(
Enable:= TRUE,
wDrvNr:= 0,
ucNodeId:= 2,
```

```
wIndex:= 16#4001,
bySubIndex:= 0,
ucModus:= 16#40,
aAbortCode:=aAbort);
IF sdo.bAnswerRec AND NOT sdo.bAbortRec THEN
        dwRead := SHL(BYTE_TO_DWORD(sdo.ucAnswerBytes[7]),24);
        dwRead := dwRead + SHL(BYTE_TO_DWORD(sdo.ucAnswerBytes[6]),16);
        dwRead := dwRead + SHL(BYTE_TO_DWORD(sdo.ucAnswerBytes[5]),8);
        dwRead := dwRead + BYTE_TO_DWORD(sdo.ucAnswerBytes[4]);
END_IF
```

It is proceeded on the assumption that in index 4001, Sub0 there is a 4-byte value which always can be read with an “expedited” transfer.

If it is unknown whether the value is transmitted by the SDO server via expedited transfer or not, always a data buffer must be defined. In this case, after the transfer has been done, it can be determined via the flag bExpedited whether the data will appear in ucAnswerBytes[4..7] or in the buffer.

The input parameter aAbortCode serves to inform the module about the abort code which should be sent in case the communication is aborted by the application per deleting the enable input. (Always when a rising edge is detected at the Enable input of the module during a running transfer, an abort will be generated.)

The module itself does not watch timeouts. Thus the application must watch the SDO transfer and e.g. via call

```
Sdo(Enable := FALSE, aAbortCode[0]:=0,aAbortCode[1] := 0,aAbortCode[2] := 4,aAbortCode[3] := 5);
```

must pass the abort code „SDO Protocol timed out“. This will be sent at a falling edge at Enable if the protocol has not yet finished the transfer.

If more than 4 bytes should be read, the address of a buffer and a maximum size of this buffer can be passed to the module:

```
sdo(
Enable:= bReadString,
wDrvNr:= 0,
ucNodeId:= 2,
wIndex:= 16#4000,
bySubIndex:= 2,
ucModus:= 16#40,
aAbortCode:=aAbort,
dwDataBufferLength:= SIZEOF(str) ,
pDataBuffer:= ADR(str));
```

In this example str is a string. The size of this string within the memory can be determined by means of the SIZEOF operator. The corresponding number of characters will be maximum transferred by the module. The actually read number of bytes appears after the transfer in dwDataOffset.

General:

The module always sends the request on COBID: 16#600 + NodeId, the response of the SDO server always is expected on COBID 16#580 + NodeId.

The response data are transparent, i.e. the response message on the SDO request simply appears in ucAnswerBytes. This however is only valid for the case of an expedited Read Request (Expedited Upload). The meaning of the particular bytes is described in the CANopen protocol specification :

- Byte ucAnswerBytes[0] contains the server command specifier (ssc): 0x60 as a response on a download request. 0x4y as a response on an upload request (y: Bits 0..3 are allocated as follows: Bit 1 must be 1, otherwise the SDO server responds with a segmented upload not supported by the module). 0x80 means that the transfer has been aborted (Abort).
- Bytes ucAnswerBytes[1..2] contain the index. (Intel-byte-order)

- Bytes ucAnswerBytes[3] contain the subindex.
- Bytes ucAnswerBytes[4..7] contain the data of the response, in case of an expedited SDO upload.

In case of an abortion the abort code appears in aAbortRec, see the following table (from DSP3.01V401):

0503 0000h	Toggle bit not alternated.
0504 0000h	SDO protocol timed out.
0504 0001h	Client/server command specifier not valid or unknown.
0504 0002h	Invalid block size (block mode only).
0504 0003h	Invalid sequence number (block mode only).
0504 0004h	CRC error (block mode only).
0504 0005h	Out of memory.
0601 0000h	Unsupported access to an object.
0601 0001h	Attempt to read a write only object.
0601 0002h	Attempt to write a read only object.
0602 0000h	Object does not exist in the object dictionary.
0604 0041h	Object cannot be mapped to the PDO.
0604 0042h	The number and length of the objects to be mapped would exceed PDO length.
0604 0043h	General parameter incompatibility reason.
0604 0047h	General internal incompatibility in the device.
0606 0000h	Access failed due to an hardware error.
0607 0010h	Data type does not match, length of service parameter does not match
0607 0012h	Data type does not match, length of service parameter too high
0607 0013h	Data type does not match, length of service parameter too low
0609 0011h	Sub-index does not exist.
0609 0030h	Value range of parameter exceeded (only for write access).
0609 0031h	Value of parameter written too high.
0609 0032h	Value of parameter written too low.
0609 0036h	Maximum value is less than minimum value.
0800 0000h	general error
0800 0020h	Data cannot be transferred or stored to the application.
0800 0021h	Data cannot be transferred or stored to the application because of local control.
0800 0022h	Data cannot be transferred or stored to the application because of the present device state.
0800 0023h	Object dictionary dynamic generation fails or no object dictionary is present (e.g. object dictionary is generated from file and generation fails because of an file error).

2.4.4.2 Boot-up of the network without automatic start

Sometimes it is required that the application determines the time at which the CANopen-Slaves should be started. For this purpose option „Automatic startup“ of the CAN master must be activated in the configuration. Then the application is responsible for starting the slaves.

The normal way to start a slave via a library is to call the method NodeStart of the respective slave. This method call after the configuration effects that slaves, for which nodeguarding resp. heartbeat is activated, will be started depending on their current status, which is contained in their nodeguard/heartbeat message. If the current status of a slave is not yet „OPERATIONAL“, a „Start Node“-NMT-command will be sent to the slave.

If the method is used to start a slave which does not use nodeguarding, the slave always will get sent the complete SDO-set and afterwards a „StartNode“-command will be sent, if the startup was o.k.(means the device type matched the one in the configuration). In order to start such a slave after the automatic configuration phase (after which it needs no more configuration SDOs because it just has received those), it is easier to self-generate a „Start Node“-NMT-command. For this purpose a call of the CanOpenWriteMSG function block instance can be used (This instance always is available globally in the manager library.)

```
CanOpenWriteMSG(  wDrvNr := pCanOpenNode[xx].wDrvNr, dwCanID := 0,
                  ucLen := 2, bRtrFrame := FALSE,
                  ucByte1 := 16#01, ucByte2 := pCanOpenNode[xx].ucNodeNr);
```

Thus always one StartNode-NMT-command is put on the bus. The parameters of the module thereby are read from the array containing the descriptions of the slaves.

IN order to start the complete network, the NMT message also – as described above – can be self-generated: (This procedure always must be utilized for nodes which are working without nodeguarding/heartbeat.)

```
CanOpenWriteMSG(  wDrvNr := pCanOpenMaster[xx].wDrvNr, dwCanID := 0,
                  ucLen := 2, bRtrFrame := FALSE,
                  ucByte1 := 16#01, ucByte2 := 0);
```

This NMT command is named „Start All Nodes“.

In order to start a slave, which was brought to status 98 during boot-up (device type does not match with the configured type), it is sufficient to use method SetNodeStatus(nNewStatus := 4). If the master has been configured to start automatically, the node now will be started. Otherwise the application must additionally call method NodeStart of the node:

```
pCanOpenNode[iNodeXX].NodeStart();
```

If the device type of the node did not fit the one in the configuration, the node has to be started (if desired) by using the explicit generation of a NMT message, see above. The use of NodeStart will only lead to a sending of the whole set of configuration SDOs with no Start Node command at the end.

For an optional slave which was not present at the start-up of the master and which therefore currently is in status 97, the same procedure can be utilized or alternatively the method NodeReset can be called. This method always causes that the slave is re-configured and then started. (Indeed it will only be started if the autostart-option is activated. Otherwise it must be started manually.)

2.4.4.3 Stopping a node

By calling method NodeStop, analogue to NodeStart, a NMT-command for stopping the slave will be created. Thereupon the slave normally will change to status „Stopped“.

2.4.4.4 Resetting a node

By calling method ResetNode the respective slave will be reset and re-configured.

2.4.4.5 Starting the network by bUseStartAllNodes

In a CAN network with many participants (mostly more than 8) it often happens, that NMT-messages which quickly follow each other are not detected by all (mostly slow) IO-nodes (slaves). This happens due to the reason that all those nodes must listen to all messages with ID 0. NMT-messages which are sent in a too quick succession overstress the receive logic of such nodes. A symptom for this fact is that these nodes sometimes are not started. Relief can only be produced by reducing the number of quickly succeeding NMT-messages. Additionally the application can cause the CANopenMaster library to use command “Start All Nodes” instead of starting all nodes singly per „Start Node“.

For this purpose the application must set the flag bUseStartAllNodes of the master uniquely, always at start of the controller, e.g. like shown in the following (bInit is defined by the application):

```
IF NOT bInit THEN
        pCanOpenMaster[0].bUseStartAllNodes := TRUE;
        bInit := TRUE;
END_IF
```

Attention: This flag may not be set cyclically, because then single nodes after a failure would not be re-started correctly.

Always when the application uses this command for starting the network, also nodes which – e.g. because they had responded with a wrong device type – are in status 98, will be started. However, then PDOs for these nodes remain deactivated. To unlock them the application must act like described in 2.4.4.2, Boot-up of the network without automatic start. If it is not sure whether all slaves have been parameterized with the correct device type, member „dwMerker“ of the particular nodes can be regarded. Here the device type appears, which has been read by the slave via request of object 0x1000. (In member dwNodeIdent the configured type is stored.) dwMerker is always valid, if it is unequal 0.

2.4.4.6 Initialization of the network via bUseResetAllNodes

For the same reasons as described for bUseStartAllNodes there are cases in which it is better to use NMT-command „Reset All Nodes" instead of "Reset Node" for each single node.

For this the application uniquely must set flag bUseResetAllNodes of the master:

```
IF NOT bInit THEN
        pCanOpenMaster[0]. bUseResetAllNodes := TRUE;
        bInit := TRUE;
END_IF
```

Attention: This flag may not be set cyclically because then single nodes would not be re-started correctly after a failure.

2.4.4.7 Guarding the configuration phase

The configuration phase of the slaves can be guarded. First of all the master must be guarded in its boot-up phase, because it will get caught at nStatus = 3, if a non-optional slave is not found during boot-up.

Slaves, at which a problem occurs during the configuration phase, will get caught at nStatus = 3 or will directly change to error state > 5 after the configuration phase.

The following examples show application code which can be used to keep working with the CAN network even in certain situations resp. to get diagnosis information.

During the guarding of the boot-up phase it must be regarded that per IEC-cycle of the task, which calls the CANopen-Master (also see implicit calls), only one SDO can be transmitted to each slave. That is, the boot-up phase might take long time, depending on the maximum number of SDOs which must be transferred to a slave.

2.4.4.7.1 Access on the state of the CANopen-Master

In order to avoid that application code is processed although the IO-network is not yet ready, the status of the master must be polled:

```
IF pCanOpenMaster[0].nStatus = 5 THEN
    <application code>
END_IF
```

2.4.4.7.2 Time guarding in the boot-up phase

```
VAR
    Guarding : TON;
    iSlave : INT;
    bSlaveFound: BOOL;

END_VAR

Guarding(IN:= pCanOpenMaster[0].nStatus < 5, PT := T#10 );

IF Guarding.Q THEN
    (* figuring out whether there is a non-configurable slave *)
    FOR iSlave := pCanOpenMaster[0]. nFirstNodeNr  TO pCanOpenMaster[0]. nLastNodeNr
    DO
        IF pCanOpenNode[iSlave]. nStatus < 4 THEN
            bSlaveFound := TRUE;
            EXIT;
        END_IF
    END_FOR
END_IF
```

After this sequence in variable iSlave the number (index) of the fist slaves appears, which is still in status 3 after expiration of the guarding time.

2.5 Implicit calls

CoDeSys creates implicit calls to the CANopen libraries.

If CoDeSys is started with command line option /debug, these calls are stored to the compile directory as files (name starts with CANopen…., extension is ".exp").

The calls are generated after the following pattern:

If there is no module parameter 'UpdateTask' of the CAN master, then the call of the CAN master will be generated in the alphabetically first task. Otherwise this module parameter specifies the name of the task in which the CAN master is called.

The Rx- and Tx-PDOs each are called in that task, which has the highest priority among all tasks referencing the PDO.

If PDOs or the master are called according to the above described pattern in an event task, a warning will be issued.

In order to shift PDO calls in another task, IO references must be shifted to this task. In order to shift the master call the module parameter 'UpdateTask' must be modified, or another task must be made to the first one in the list, by changing its name.

2.6 Object dictionary of the CANopen-Master

In some cases it is helpful if the CAN master has an own object dictionary, e.g. for the data exchange of the application with other CAN nodes.

The object dictionary of the master is created via an EDS-file during compilation and is pre-allocated with values.

The EDS-file to be used is defined by two entries in the cfg-file:

```
MasterEDS=TRUE
MasterEDSFile=MyMaster.eds
```

In this example the first file named „MyMaster.eds" which is found all over all configuration directories, will be used.

The object dictionary is an array of the following structure:

```
TYPE CanOpenODEntry :
STRUCT
    dwIdxSubIdxF : DWORD;   (* The structure of this component is 16#iiiissff, whereby iiii
                               signifies 2 bytes index, ss signifies 1 byte subindex and ff
                               signifies 1 byte flags. *)
    dwContent : DWORD;      (* Content of the entry.*)
    wLen : WORD;            (* Data length.*)
    byAttrib : BYTE;        (* Originally intended for defining the access right, but can
                               be arbitrarily used by the master application. *)
    byAccess : BYTE;        (* Formerly access right, but can be arbitrarily used by the
                               master application.*)
END_STRUCT
END_TYPE
```

The CoDeSys user interface provides no editor for the object dictionary.

The EDS-file just defines with which objects the object dictionary has to be created. Thereby the entries always are created with length 4 and the flags (lowest prior byte of the component of an object dictionary entry dwIdxSubIdxF) always with 16#41 resp. 16#01 for the last entry.

If an object dictionary is available in the master, the master can act as a SDO server in the network.

Always when a client performs a writing access an object dictionary entry, the application will be notified through the master flag bMsgUsed. This flag is implemented as an input variable of the master so that the application can reset the flag. The master only sets the flag.

The application can use the object dictionary by directly writing or reading entries or by the assignment of entries to IEC variables. In the latter case these IEC variables will be directly accessed when reading or writing from/to another node.

An entry can be accessed as follows if index/subindex are known:

```
I := GetODMEntryValue(16#iiiiss00, pCanOpenMaster[0].wODMFirstIdx,
                      pCanOpenMaster[0].wODMFirstIdx + pCanOpenMaster[0]. wODMCount;
```

Therewith the index of the entry is available in I.

Now the components of the entry can be accessed directly:
For example: In order to get an entry directly pointing on an IEC variable, it is sufficient to enter address, length and flags:

```
ODMEntries[I].dwContent := ADR(<variable name>);
ODMEntries[I].wLen := sizeof(<variable name>);
ODMEntries[I]. dwIdxSubIdxF := ODMEntries[I]. dwIdxSubIdxF OR OD_ENTRYFLG_WRITE
                               OR OD_ENTRYFLG_ISPOINTER;
```

In order just to modify the content of the entry it is sufficient to modify dwContent.

3 CanDevice

A CoDeSys programmable PLC can appear in a CAN network also as a CANopen-slave (also named CANopen-node, in the following named „CanDevice").

3.1 Functionality

The CanDevice library together with the CANopen-configurator provide the following functionality for the user:

- Configuration of the following properties in CoDeSys: NodeGuarding/Heartbeat, Emergency, NodeID and baudrate, on which the device should work.
- Together with the Parameter Manager in CoDeSys a default PDO mapping can be created, which can be modified by the master during run time. The change of the PDO mapping is done during the configuration phase by the master. By the mapping IEC variables of the application can be mapped to PDOs.
- The CanDevice library provides an object dictionary, the size of which is determined by CoDeSys during compilation. This dictionary contains all objects describing the CAN device and additionally those which are defined by the Parameter Manager. In the Parameter Manager together with the CanDevice only the list types „Parameters" and "Variables" can be used.
- The library manages accesses on the object dictionary, thus acting as SDO server at the bus.
- The library watches the nodeguarding resp. the heartbeat consumer time (only of one producer each) and sets the corresponding error flags for the application.
- Creation of an EDS-file, describing the configured properties of the CanDevice in a way that the device can be configured as slave below a CAN master.

The CanDevice library explicitly not provides the following functionalities which are described in CANopen. (All possibilities of the CANopen protocol, mentioned here and in the upper paragraph, also are not implemented) :

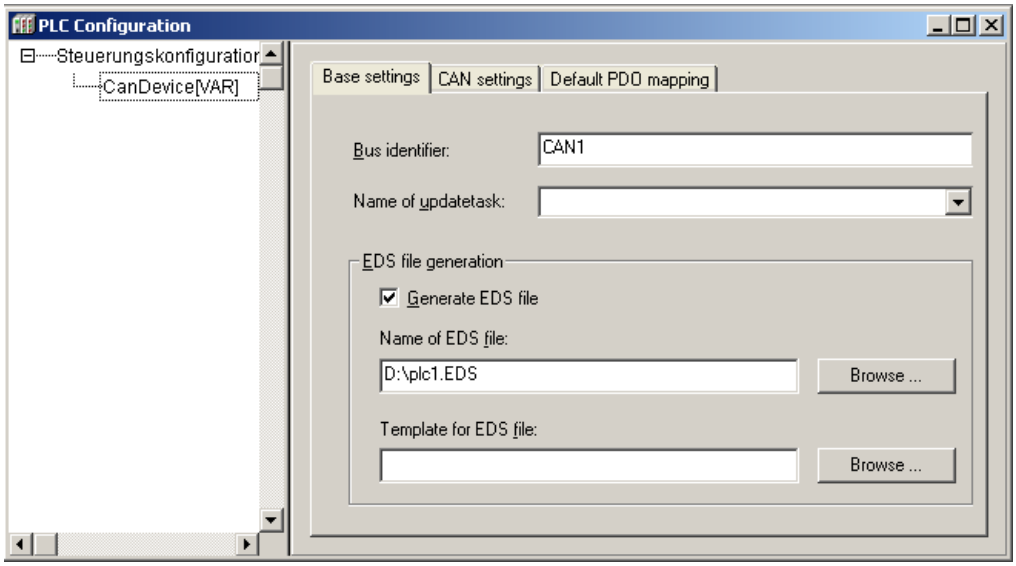
- Dynamic SDO, PDO identifier.
- Blocking SDO transfer.
- Implicit generation of emergency messages. Emergency messages always must be created by the application. For this purpose the library provides a function block, which can be used by the application.
- Dynamic changes of the PDO properties resp. runtime properties always are updated at the arrival of a StartNode NMT-message and at each access on an object in the communication properties of the CanDevice.

3.2 Configure CanDevice

In order to configure a CanDevice, in the context menu of the PLC Configuration choose command Insert subelement -> CanDevice. This command is only available, if at least one .cfg-file (manufacturer-specific description of the PLC configuration) contains an entry for a CanDevice:

```
[Module.CanDevice]
Name=CanDevice
Id=78379
DeviceType=CANDEVICE
BasisPrmDlg=FALSE
Class=<class, which can be inserted at the root module of the PLC configuration >
Icon=codsmall.ico (optional)
FixedNumOfPDOs=1 (optional, default mapping is set to 4 PDOs)
```

Now you get the following configuration dialog:



3.3 CanDevice Settings

Bus identifier: currently not used.

Name of updatetask: Name of a task, in which the CanDevice is called.

If from the current settings an EDS-file should be generated, in order to be able to include the CanDevice in any desired master configuration, activate option **Generate EDS-file** and enter a file name. Optionally in addition a **template** file can be specified, the entries of which will be added to the EDS-file of the CanDevice. (In case of overlaps the defaults of the template will not be overwritten.)

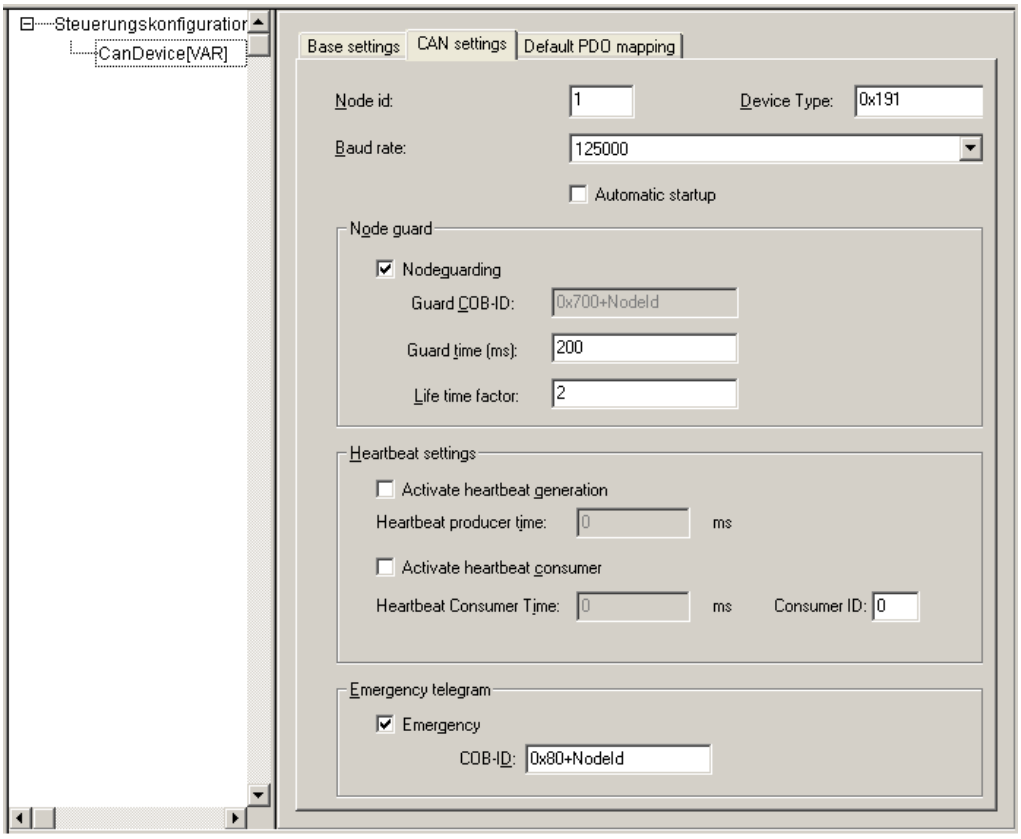
In tab CAN settings:

Here the **Node id** and the **Baud rate** can be set.

The **Device Type** (default value of object 0x1000, entered in the EDS-file) is predefined with „0x191" (Standard IO Device) and can be arbitrarily modified by the user.

The index of the CAN controller results (0-basing) from the position of the CanDevice in the PLC configuration.

The **Nodeguard** parameters, die **Heartbeat** settings and the **Emergency** COBID are defined here. The device here only can be configured for guarding one heartbeat.



In tab „Default PDO Mapping“:

Here the assignment between local object dictionary (Parameter Manager) and the PDOs sent/received by the CanDevice is determined.

In the used object dictionary entries (parameter lists of type ‘Variables’) an assignment of object index/subindex to variables of the application is done. Thereby it must be regarded, that subindex 0 of an index containing more than one subindices, implicitly will be used to store the number of the subindices.

Example:

> Purpose: On the first receive-PDO (COB-Id = 512 + NodeId) of the CanDevice variable PLC_PRG.a should be received.
>
> Thus in the Parameter Manager an index/subindex must be connected with variable PLC_PRG.a. For this purpose add a line in a variables list and enter index and subindex. In the access field only „Write Only“ is applicable, because in a receive-PDO only a writeonly-value can be mapped. In column “Variable” enter PLC_PRG.a, or press F2 and select this variable.
>
> **Note**: The Parameter Manager editor is only available, if in the Target Settings in „Network functionality“ option „Support parameter manager“ is activated and parameterized with useful index/subindex ranges.
>
> After that in the ‘Default PDO Mapping’ of the CanDevice the index/subindex entry is assigned as a mapping entry to a receive-PDO. The properties of the PDO can be defined in the Properties dialog which is known from the CAN slave configuration below a master.

Only objects of the Parameter Manager, which have got attribute „read-only“ resp. „write-only“, will be marked as mappable in a possibly generated EDS-file and will appear in the list of mappable objects.An object which has been created in the Parameter Manager gets assigned a default value in the EDS-file. This default is either the initial value of the variable (variables lists) or the value defined by the user (parameter lists).

All other objects will be marked as not mapable in the EDS-file.

3.4 Generate EDS-file

If an EDS-file should be generated from the configuration of the CanDevice, in order to be able to use the device in the configuration of a CANopen-Master, it is sufficient to activate the functionality in tab “Base settings”. Additionally the file name of the file to be generated must be defined.

Optionally a template for the EDS-file can be specified.

Entries available in the template will not be overwritten by CoDeSys. So it is possible to define parts of the EDS-file, which should not be changed, by a template file.

All the user defined entries in the Parameter Manager get the initial value of a variable (variable list) or the entered value (parameter list) as the DefaultValue in the EDS file.

In the following see an example showing the content of an EDS-file in detail:

```
[FileInfo]
FileName=D:\CoDeSys\lib2\plcconf\MyTest.eds
FileVersion=1
FileRevision=1
Description=EDS for CoDeSys-Project:
D:\CoDeSys\CanOpenTestprojects\TestHeartbeatODsettings_Device.pro
CreationTime=13:59
CreationDate=09-07-2005
CreatedBy=CoDeSys
ModificationTime=13:59
ModificationDate=09-07-2005
ModifiedBy=CoDeSys

[DeviceInfo]
VendorName=3S Smart Software Solutions GmbH
ProductName=TestHeartbeatODsettings_Device
ProductNumber=0x33535F44
ProductVersion=1
ProductRevision=1
OrderCode=xxxx.yyyy.zzzz
LMT_ManufacturerName=3S GmbH
LMT_ProductName=3S_Dev
BaudRate_10=1
BaudRate_20=1
BaudRate_50=1
BaudRate_100=1
BaudRate_125=1
BaudRate_250=1
BaudRate_500=1
BaudRate_800=1
BaudRate_1000=1
SimpleBootUpMaster=1
SimpleBootUpSlave=0
ExtendedBootUpMaster=1
ExtendedBootUpSlave=0

[DummyUsage]
Dummy0000=0
Dummy0001=0
Dummy0002=0
Dummy0003=0
Dummy0004=0
Dummy0005=0
Dummy0006=0
Dummy0007=0

[MandatoryObjects]
```

```
SupportedObjects=2
1=1000
2=1001

[1000]
SubNumber=0
ParameterName=Device Type
ObjectType=0x7
DataType=0x7
AccessType=ro
DefaultValue=0x191
PDOMapping=0

[1001]
SubNumber=0
ParameterName=Error Register
ObjectType=0x7
DataType=0x5
AccessType=ro
DefaultValue=
PDOMapping=0

[OptionalObjects]
SupportedObjects=8
1=0x1003
2=0x1005
3=0x100C
4=0x100D
5=0x1014
6=0x1016
7=0x1017
8=0x1018

[ManufacturerObjects]
SupportedObjects=5
1=0x3000
2=0x3001
3=0x3002
4=0x4000
5=0x5000

[1003]
SubNumber=2
ParameterName=Predefined error field

[1003sub0]
ParameterName=Number of entries
ObjectType=0x7
DataType=0x5
AccessType=rw
DefaultValue=1
PDOMapping=0

[1003sub1]
ParameterName=Predefined Error field
ObjectType=0x7
DataType=0x7
AccessType=rw
DefaultValue=0x0
PDOMapping=0

[1005]
SubNumber=0
ParameterName=COBID Synch Message
ObjectType=0x7
DataType=0x7
AccessType=rw
```

```
DefaultValue=0x80
PDOMapping=0

[100C]
SubNumber=0
ParameterName=Guard Time
ObjectType=0x7
DataType=0x6
AccessType=rw
DefaultValue=0
PDOMapping=0

[100D]
SubNumber=0
ParameterName=Lifetime Factor
ObjectType=0x7
DataType=0x5
AccessType=rw
DefaultValue=0
PDOMapping=0

[1014]
SubNumber=0
ParameterName=COBId Emergency
ObjectType=0x7
DataType=0x7
AccessType=rw
DefaultValue=0x80+$NodeId
PDOMapping=0

[1016sub0]
ParameterName=Nums consumer heartbeat time
ObjectType=0x7
DataType=0x5
AccessType=rw
DefaultValue=1
PDOMapping=0

[1016sub1]
ParameterName=Consumer heartbeat time
ObjectType=0x7
DataType=0x7
AccessType=rw
DefaultValue=0x0
PDOMapping=0

[1017]
SubNumber=0
ParameterName=Producer heartbeat time
ObjectType=0x7
DataType=0x6
AccessType=rw
DefaultValue=0
PDOMapping=0

[1018]
SubNumber=4
ParameterName=Vendor identification

[1018sub0]
ParameterName=Number of entries
ObjectType=0x7
DataType=0x5
AccessType=ro
DefaultValue=2
PDOMapping=0
```

```
[1018sub1]
ParameterName=VendorID
ObjectType=0x7
DataType=0x7
AccessType=ro
DefaultValue=0x0
PDOMapping=0
[1018sub2]
ParameterName=Product Code
ObjectType=0x7
DataType=0x7
AccessType=ro
DefaultValue=0x0
PDOMapping=0
```

For the meaning of the particular objects please see the CANopen specification DS301.

The EDS-file contains, besides the mandatory entries, the definitions for SYNC, Guarding, Emergency and Heartbeat. If this objects are not used, as Guarding and Heartbeat in the shown example, the default values are set to 0. But these objects are available at runtime in the object dictionary of the CanDevice, thus they are described nevertheless in the EDS-file.

Analogue this is valid for the (not listed here) entries for the communication and mapping parameters. Always all 8 possible (bit mapping is not supported by the library!) subindices of the mapping objects 0x16xx resp. 0x1Axx are available, but potentially not regarded in subindex 0.

Note: With entry FixedNumOfPDOs=1 in the cfg-file, as described above, the mapping objects, if e.g. no default mapping is predefined, always are set up for 4 PDOs, so that at runtime 4 PDOs are available for the mapping of variables. Subindex 0 of the mapping objects however is pre-set with 0, because none of the objects is allocated.

3.5 Modifying the default mapping by the master configuration

The given PDO mapping (in the CanDevice configuration) can be modified within certain bounds by the master.

Thereby the rule must be observed, that CanDevice is not able to create new object dictionary entries, which are not yet part of the default mapping (Default PDO Mapping in the CanDevice configuration). So e.g. for a PDO, which contains one object mapped in the Default PDO Mapping, it is not possible to map a second object in the master configuration.

Thus the mapping modified by the master-configuration maximum can contain the PDOs defined in the default-mapping. Within these PDOs 8 mapping entries (subindices) are available.

Errors which possibly occur in this context are not indicated bit the redundant PDO definitions / redundant mapping entries will be treated as non-existent.

The PDOs always must be set up in the master starting with 16#1400 (receive PDO communication parameters) resp. 16#1800 (send-PDO communication parameters) and must follow each other without a break.

The default mapping of the CanDevice always is available after a program download. It depends on the master configuration, whether none-available mappings (not available in the master configuration but available in the default configuration) will be explicitly overwritten by the master.

3.6 Working with the CanDevice in the application program

3.6.1 Module CanopenDevice

The module is instanced once for each configured CanDevice. The components of the function block are used as follows:

bAutoStart : BOOL;	Setting of option “Automatic startup”
ucNodeId : BYTE;	Node-ID of the CanDevice.
wDrvNr : WORD;	Index of the CAN controller in the configuration.
wODStart : WORD;	Index of the first object dictionary entry of the CanDevice.
wODEnd : WORD;	Index of the last object dictionary entry of the CanDevice.
wPDOStartRx : WORD;	Index of the first Rx-PDOs of the CanDevice within the array of RxPDOs (pCanDevPDO_Rx).
wPDOCountRx : WORD;	Number of RxPDOs of the CanDevice.
wPDOCountRxCurMap : WORD;	Running variable for the configuration of the PDO Mappings of the object dictionary.
wPDOStartTx : WORD;	Index of the first Tx-PDOs of the CanDevice within the array of TxPDOs (pCanDevPDO_Tx).
wPDOCountTx : WORD;	Number of TxPDOs of the CanDevice.
wPDOCountTxCurMap : WORD;	Running variable for the configuration of the PDO Mappings of the object dictionary.
ucNodeIdNew : BYTE;	This variable is set by init-code as follows: After calling the CAN driver function CanInit, in which the driver can set a new NodeID, e.g. from a rotary switch, this variable gets assigned the value: WORD_TO_BYTE(gCanInterface[1].pArray[1]) . Thus CAN drivers must write a new NodeID in pArray[1].
ErrorCode : WORD;	If the application wants to send an emergency message, here the input parameter ErrorCode (according to CANopen) of function block SEND_EMERGENCY is entered.
ErrorRegister : BYTE;	Like ErrorCode.
ManufacturerErrorField : ARRAY[0..4] OF BYTE;	Analogue to ErrorCode.
bEmergencySent : BOOL;	Always when an emergency message has been sent, this variable is set TRUE.bClearResetFlags : BOOL; The application can set this flag to reset the flags that show the reception of reset-NMT-messages.
bClearODEntryWritten : BOOL;	The application can set this flag to reset the flags that show write access to the object dictionary.
bInit: BOOL;	Used to detect the first cycle.
bMsgUsed : BOOL;	Internal variable, always set when a device message was used by the device.
bUseOldBootupToo : BOOL;	Flag, which can be set by the application in the first cycle in order to cause the device to send also the old bootup message (emergency with 0 bytes data length).
bGuardError: BOOL;	Set by the library if no guard message has arrived within a time span of GuardTime * LifeTimeFactor.
ucState: BYTE := 127;	Current status of the device, according to CANopen.
bHeartbeatError: BOOL;	If the device is configured to be heartbeat consumer, this flag will be set if no new heartbeat message has been notified within the configured heartbeat consumer time.

bResetCommReceived : BOOL;	Reset Communication NMT Message was received. A flag for the application.
bResetNodeReceived : BOOL;	Reset Node NMT Message was received. A flag for the application.
bEnterPreOpReceived : BOOL;	Enter Preoperational NMT Message was received. A flag for the application.
bODEntryWritten: BOOL;	An object dictionary entry was written by SDO-access.
iLastODEntryWritten: INT;	Index of last written entry.
i, iIdx: INT;	Internal running variables.
dwIdx: DWORD;	internal variable for the object dictionary access.
dwIdxHelp : DWORD;	Internal.
bSDOReadrqActive : BOOL := FALSE;	Internal variable for SDO server functionality.
bSDOWriterqActive : BOOL := FALSE;	Internal variable for SDO server functionality.
bSDOReadrspAbort : BOOL := FALSE;	Internal variable for SDO server functionality.
bSDOWriterspAbort : BOOL := FALSE;	Internal variable for SDO server functionality.
ucModus: BYTE;	Internal variable for SDO server functionality.
dwODEValue: DWORD;	Internal variable for SDO server functionality.
bSwap: BOOL;	Internal variable, used to determine whether the CanDevice is running on a processor using Intel or Motorola Byteorder. According to CANopen the order of the bytes in a SDO transfer always is Intel Byteorder.
byAbortCode: ARRAY[0..3] OF BYTE;	Internal variable for SDO server functionality.
iIdxCom: INT;	Internal variable
dwIdCom: DWORD;	Internal variable
dwIdMap: DWORD;	Internal variable
iIdxMap: INT;	Internal variable
iPDOcur: INT;	Internal variable
ucCurPDOLen : BYTE;	Internal variable
iNumOfMappedObjects: INT;	Internal variable
iMapObj: INT;	Internal variable
iCurRxPDO: INT;	Internal variable
iCurTxPDO: INT;	Internal variable
bSendBootUp: BYTE;	Internal variable, always set TRUE when a bootup message should be sent.
bSendNodeGuard: BOOL;	Internal variable for sending the nodeguard-response.
ucToggle: BYTE;	Toggle-bit of the nodeguard response.
bReentry : BOOL;	Historically
tGuardLifeTime : TON;	Guarding timer for nodeguarding.
bGuardingEnabled : BOOL;	Flag, always set when GuardLifeTime <> 0.
tHeartbeatProducer : TON;	Heartbeatproducer time. According to this time interval heartbeat messages are created.
tHeartbeatConsumer : TON;	Timer for guarding the arrival of heartbeat messages.
byHeartbeatConsumerID : BYTE;	ID on which heartbeat messages are expected. Always only one ID is guarded at a time.

bCurPDOChanged: BOOL;	Internal.
bTransmitCyclic: BOOL;	Internal.
dwSynchCobId : DWORD;	COBId, on which the sync message is expected.
dwSynchWinLen : DWORD;	Configured syncWindowLength.
dwSynComCycle: DWORD;	Configured sync interval.
diHelp : DINT;	Internal.
dwHelp : DWORD;	Internal.
MsgBuffer : CAN_Message;	Buffer for received messages.
a: INT;	Counter, incremented with each cycle in order to get monitored the number of calls.
dwCobIdPDOMin,dwCobIdPDOMax,dwAcceptance:DWORD;	These values are passed to the driver to enable it to set an acceptance mask.
bInitiateRspSend : BOOL;	Internal.
bInitiateWrReqSend : BOOL;	Internal.
iActiveSegSDORead:INT := -1;	Internal, for the segmented SDO transfer.
pActiveSegSDORead:POINTER TO BYTE;	Internal, for the segmented SDO transfer.
wSegSDOReadSendOffs : WORD;	Internal, for the segmented SDO transfer.
wSegSDOReadSendLen : WORD;	Internal, for the segmented SDO transfer.
iActiveSegSDOWrite:INT := -1;	Internal, for the segmented SDO transfer.
pActiveSegSDOWrite:POINTER TO BYTE;	Internal, for the segmented SDO transfer.
wSegSDOWriteRecvOffs : WORD;	Internal, for the segmented SDO transfer.
wSegSDOWriteRecvLen : WORD;	Internal, for the segmented SDO transfer.
wSwapCheck : WORD := 16#55AA;	Internal, must remain unmodified.
bErrCodeNot0: BOOL;	Check, whether the driver has set a fatal error (BusOff).
dwEmergCobId : DWORD;	COBId, used for sending emergency messages.
iSDOWriteLen : INT;	Internal.
iSDOReadLen : INT;	Internal.
bGuardingStarted: BOOL;	Set as soon as the first guard message has arrived.
bFirstHeartbeatRecv: BOOL;	Set as soon as the first heartbeat message has arrived.
bSub0NotFound: BOOL;	Internally used in order to distinguish the abort codes „Index not found“ and „SubIndex not found“.

3.6.2 Access on the object dictionary entries by the application program

Naturally there are object dictionary entries which are mapped on variables (Parameter Manager).

But there are also the implicitly generated entries of the CanDevice, which cannot be mapped via the Parameter Manager in a variable. These entries only appear in array ODentries at runtime. In the monitoring mode (online mode) of CoDeSys all object dictionary entries can be viewed in the global variables list of 3S_CanOpenDevice.lib.

The access on the content of these objects must, exactly as the accesses on the parameter lists of the Parameter Manager, be done application-controlled:

Uniquely the index of the entry must be evaluated. For this purpose write (in the following example for the first CanDevice in the configuration):

Index := FindBinary(16#iiiiss00, CanOpenDev[0].wODStart, CanOpenDev[0].wODEnd);

After that the content of the object can be read via GetODEntryValue(Index).

With SetODEntryValue the content of an object can be manipulated. Alternatively a direct access on the content is possible, if it is known – like it is always the case for implicit objects – that the basis data type is a numeric type:

ODEntries[Index].dwContent := xxxxx;

3.6.3 Changing the PDO properties at runtime

If the properties of a PDO should be changed during runtime, this can be done by another node via SDO write accesses, like described by CANopen. Alternatively a new property can be written directly, like e.g. the event time of a send-SDO, and subsequently a StartNode-NMT-command can be sent to a node although this node is started already. This effects that the device will re-interpret the object dictionary values.

But of course also a local application can change the properties of a PDO.

For this purpose the application just must modify, as described above, the contents of the respective object(s) and subsequently must call method SetupPDOTable of the CanDevice:

```
CanOpenDev[0].SetupPDOTable();
```

3.6.4 Sending emergency messages by the application program

In order to transmit an emergency message via the application program, function block SEND_EMERGENCY can be used.

The module has the following interface:

```
VAR_INPUT
    ENABLE : BOOL;          * As long as Enable is TRUE, with each call an emergency message
                              is transmitted.*)
    ErrorCode : WORD;       (* Error code; according to CANopen some values are already pre-
                              defined. (See table below.)*)
    ErrorRegister : BYTE;   (* The error register is content of object 1001h.*)
    ManufacturerErrorField : ARRAY[0..4] OF BYTE; (* Here the application can define its own
                                                     error codes.*)
    DeviceIndex : INT;      (* The 0-based index of the device, which should send the
                              emergency messages.*)
END_VAR

VAR_OUTPUT
    bReady : BOOL;          (* As soon as the message has been sent, bReady gets FALSE.*)
END_VAR
```

Table with error codes, from DSP301:
00xx Error Reset or No Error
10xx Generic Error
20xx Current
21xx Current, device input side
22xx Current inside the device
23xx Current, device output side
30xx Voltage
31xx Mains Voltage
32xx Voltage inside the device
33xx Output Voltage
40xx Temperature
41xx Ambient Temperature
42xx Device Temperature
50xx Device Hardware
60xx Device Software
61xx Internal Software
62xx User Software

63xx Data Set
70xx Additional Modules
80xx Monitoring
81xx Communication
8110 CAN Overrun (Objects lost)
8120 CAN in Error Passive Mode
8130 Life Guard Error or Heartbeat Error
8140 recovered from bus off
8150 Transmit COB-ID
82xx Protocol Error
8210 PDO not processed due to length error
8220 PDO length exceeded
90xx External Error
F0xx Additional Functions
FFxx Device specific

4 CAN network variables

Network variables can be used to exchange data between one or multiple PLCs. The mechanism should be easy to handle fort he user. Currently network variables are implemented for CAN and UDP. The variables' values are exchanged automatically on the basis of broadcast messages. In UDP these are realized as broadcast telegrams, in CAN as PDOs. Concerning the used protocol these services are not-confirmed services, i.e. there is no control whether the message reaches the receiver. Network variables exchange is a 1 (sender) to n (receiver) - connection.

The **Parameter-Manager** is another possibility to exchange variables. This is a 1 to 1 – connection, using a confirmed protocol. This means that the user can guard, whether the message has reached the receiver. The exchange is not done automatically but via calling function blocks in the application program.

4.1 CAN network variables for the user

To use the network variables with CoDeSys the user must perform the following steps:

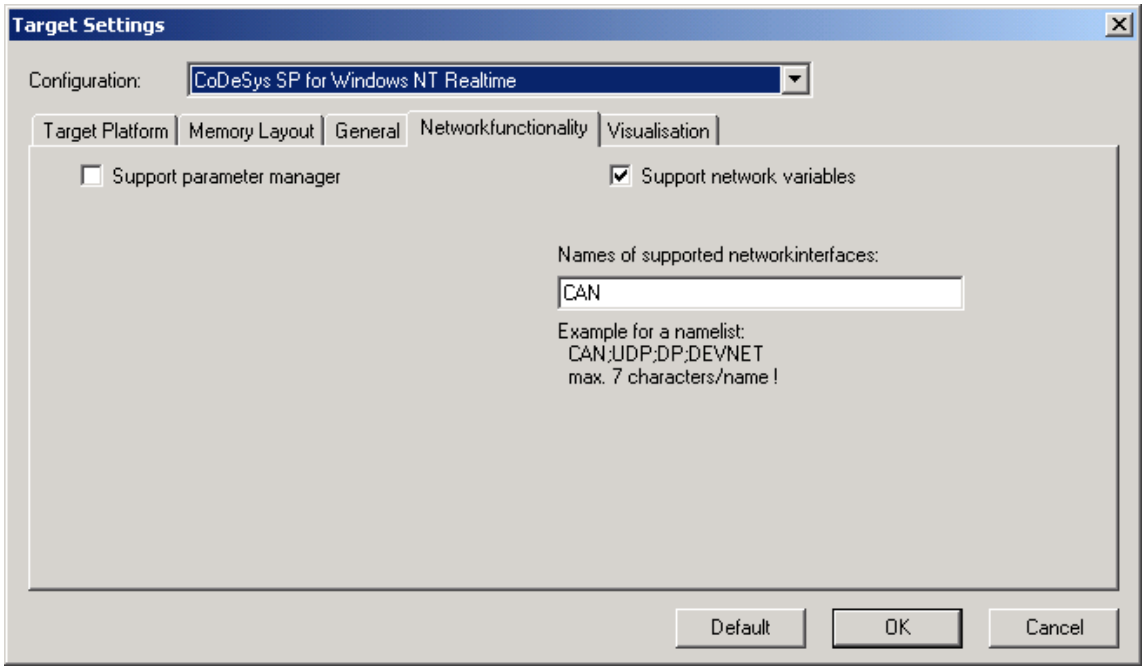
4.1.1 Preconditions

A CoDeSys programming system Version 2.3 SP5 (or higher) must be installed and the following libraries must be available:

3s_CanDrv.lib, 3S_CanOpenManager.lib and 3S_CanOpenNetVar.lib

CoDeSys automatically generates the required initialization code and the calls of the network POUs at the start and the end of each cycle.

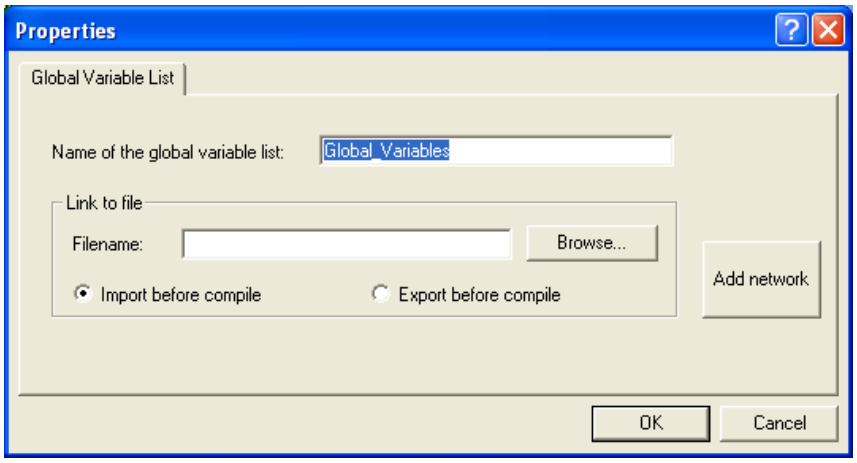
4.1.2 Target Settings



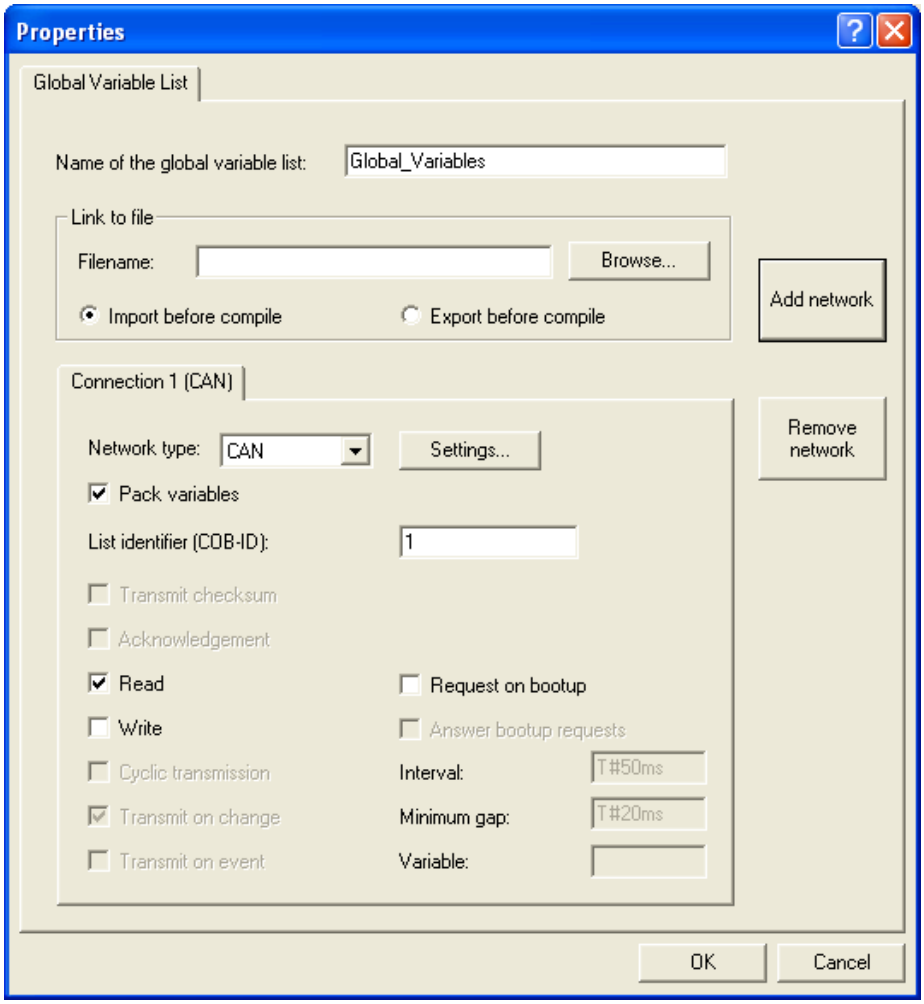
Activate option **Support network variables** in the dialog box 'Target Settings' (tab 'Networkfunctionality'). At **Names of supported networkinterfaces** enter the name of the desired network, e.g. CAN. The required libraries will then automatically be included in the project with the next compile.

4.1.3 Settings in the Global Variables Lists

Add a new global variables list. Here you define the variables, which you want to exchange with the other PLCs. Use the command 'Object Properties' to open the following dialog:



Click on the button **Add Network** to get the options for defining the network properties of the current variables list. If you have set up several network connections, then you can configure several connections for each Global variable list.



The options and their meaning:

- **Network type:** one of the network types which are defined in the Target Settings dialog (tab 'Networkfunctionality').
- **Read**: The values of the variables in this list should be read by one or several of the other PLCs in the network.
- **Write**: The values of the variables in this list should be sent to other PLCs. It is recommended to set only one of the options "Read" or "Write" for a variables list. If some variables of a project should be read and written, then use two variables lists: one for reading, one for writing. Besides that it is recommended to have only one PLC in a network which is used to send a certain variables list.
- **Cyclic Transmission**: Only valid, if „Write“ is activated. The values will be sent in the defined **Interval**, no matter whether they have changed or not.
- **Transmission at Change**: The values will only be sent in case they have changed or when the time, set at **Min. interval**, is over.
- **Pack Variables**: If this option is activated, as many variables as possible will be packed in one transmission unit. For UDP a transmission unit has a maximum size of 256 Bytes, for CAN 8 Bytes. If one transmission unit is to small to catch all variables, then further units will be generated. If the option is deactivated, each variable will be packed in a separate transmission unit. If ‘Transmission at Change’ is activated, for each particular transmission unit it will be checked whether it has been changed and is to be sent.
- **List identifier (COBID)**: This is an unique identifier to mark the variables lists which should be exchanged between several projects. Only lists which have the same identifier can exchange their data. But assure that the lists, which have the same identifier, also contain identical definitions! For this it is recommended to use the feature **Link to file** to export the desired variable list from one project and to import it into the other projects.
  WARNING: In a CAN network the list identifier will be used directly as COBID of the CAN message(s). There’s no check for collisions of identifiers used here and the ones which are used in the CANopen-configuration for IO-PDOs.

To exchange data properly between two controllers, the definition of the variable lists in the two projects must be same. You can use the “Link to file” feature to assure that both projects use the same variable list. One project should export the file before compile, the other projects import it before compile.

Besides simple data types, a variable list may also contain structures and arrays. Elements of these compound data types are sent individually

If a variable list is bigger than the PDO size of the respective network, it is sent using several PDOs. This means that it can not be assured that all data of a variable list is received in the same cycle. Portions of the variable list may arrive in different PLC cycles. This can happen also for variables of array or structure data types.

Options not supported for CAN:

- Transfer checksum.
- Answer bootup requests.
- Confirmed transfer.
- Request on Bootup.

4.1.4 Generated calls

An implicit variable list will be created. For each PDO (sending or receiving) an array entry of type NetVarPDO_Rx_CAN resp. NetVarPDO_Tx_CAN will be inserted.

For debug purposes a file containing the generated declarations will be created. The file is named NetworkGlobalVars_CAN.exp and will be found in the compile files directory. (Only if CoDeSys has been started with command line option “/debug”.)

Initialization code is generated, which will initialize the generated data. The initialization code will be called by the GlobalInit POU, which in turn is called immediately after a download in order to initialize the project data.

At the begin and at the end of a task implicit calls of library functions will be generated:

- For each task, which uses network variables, initially a receive call will be generated for each PDO.
- For each task, which uses network variables, initially a (sole) instance of the POU NetVarManager_Udp_FB bzw. NetVarManager_Can_FB will be called:

```
NetVarManager_CAN();
```

- For each task, which uses network variables, at the end a send call will be generated for each PDO:

The calls all in all:

CAN_Read(0);

NetVarManager_CAN();

pNetVarPDO_Rx_CAN[0]();

<application code>

MgrClearRxBuffer(wCurTask:= 1,wDrvNr := 0, dwFlags := 0, dwPara := 0);

pNetVarPDO_Tx_CAN[0]();

5 Appendix

5.1 List of SDOs usually created for a slave

The CoDeSys CANopen configurator per default creates the following SDOs:

- Read access on object 0x1000 in order to check the type of the node.
- Write access on all objects of the EDS-file, which in the current configuration are not matching the default value defined in the EDS-file and which have got assigned read access in the EDS-file.
- If in the current configuration „Create all SDOs“ is activated, all communication- and mapping parameters (1400er, 1600er, 1800er and 1A00er objects) are created each time, the ServiceData objects (own tab in the configurator) however still only in case of a modification towards the EDS-file.

5.2 Monitoring transmit buffer overruns.

In the global varlist of the 3S_CanOpenManager.lib now an array of flags is available for monitoring transmit buffer overruns to the application:

g_CanMgrTxBufferOverrun[<Controllerindex>].

The application can use these flags for diagnostic and messaging purposes. The flags have to be reset by the application. The library sets the flags, only.

Change History

Version	Description	Editor	Date
1.0	Translation and Release (according to German version 1.0)	AF, MN	01.12.05
1.1	Update (with CoDeSys v2.3.6.0): Chap. 3.3 (#5445), Chap. 5.2 new (#5465), Chap. 2.4.4.2 (#5449, #5450), Chap. 3.6.1(#5446)	AF	11.01.-24.06.2005
1.1	Formal Review, Release	MN	24.06.2005