SO-1059

SRDO Add-on module

CANopen Source Code
Add-on for
CiA 304 Safety Framework

Software Manual

Edition May 2013

SYS TEC electronic GmbH
Am Windrad 2   08468 Heinsdorfergrund   Deutschland
Telefon: +49 3765 38600-0   Fax: +49 3765 38600-4100
Web: www.systec-electronic.com   Mail: info@systec-electronic.com

System House for Distributed Automation

contacts	Direct	Your local distributor
Address:	SYS TEC electronic GmbH Am Windrad 2 D-08468 Heinsdorfergrund GERMANY	Please find a list of our distributors under: www.systec-electronic.com/distributors
Ordering information:	+49 3765 / 38600-2110 info@systec-electronic.com	
Technical support:	+49 3765 / 38600-2140 support@systec-electronic.com	
Fax:	+49 3765 / 38600-4100	
Website:	http://www.systec-electronic.com	

5th Edition May 2013

Introduction

This manual is an extention of the „CANopen User Manual L-1020“ and describes the application layer of the SRDO module.

Section 1 provides some basic terms of the Safety Framework.

Section 2 explains the implementation and describes the user functions, user interfaces and data structures.

1 Basics “Framework for Safety-Relevant Communication”

The CiA Draft Standard Proposal 304 “CANopen Framework for Safety-Relevant Communication” defines the CANopen Protocol extensions for the integration of safety-relevant devices in CANopen networks. The protocol allows for using safety-targeted devices and non-safety-targeted devices in one CANopen network. Safety functions are realized via specific communication objects, the SRDOs (safety relevant data objects).

With the CANopen Safety Protocol it is possible to directly connect safety-targeted sensors and actuators. A safety-targeted control (e.g. PLC, safety monitor) is not needed. This enables the realization of logically comparable safety chains as in usual wired technology (e.g. the emergency power-off switch directly affects the safety relay).

The CiA-304 standard is integrated in the standard DIN EN 50325-5:2009

1.1 SRDO – Safety-Relevant Data Object

The SRDO communication follows the producer/consumer principle. This means that there is a SRDO producer and one or several SRDO consumers.
A SRDO consists of two CAN telegrams. The following rules apply to the generation of a SRDO:

1. The CAN identifier of the two CAN telegrams differ at least in two bit locations. The CAN identifier of the CAN telegram with normal data always is uneven. The CAN identifier of the CAN telegram with inverted data always is the subsequent even value.
2. The data oft two CAN telegrams is redundant. But the data of the second CAN telegram is inverted bit by bit.
3. A SRDO is transferred periodically whereas the distance between two SRDOs is determined by the SCT (safeguard cycle time).
4. The distance between the two CAN telegrams of a SRDO may not exceed the SRVT (safety relevant object validation time).
5. The order of the two CAN telegrams of a SRDO must be maintained. Firstly, the actual data is transferred and secondly, the inverted data is transferred.

The receiver checks the validity of a SRDO. The time and logical sequence of the CAN telegrams of a SRDO is compared to an expected value. Afterwards, the user data is verified. If errors are detected, it is switched to the secure state of the assigned actuators. The secure state is to be defined in dependence from the application of the device manufacturer and/or user.

Features of SRDOs (CAN identifier, SCT, SRVT, mapping) are stored in the object directory and checked for validity by a CRC (16 bit cyclic redundant check).

1.1.1 Communication parameters of a SRDO

The communication parameters of a SRDO define the transmission features and the COB-IDs of a SRDO.

The communication parameters of a SRDO are entries in the object directory (Index 0x1301 – 0x1340). They can be read and - if allowed - modified via the CAN bus by using service data objects (SDO).

Index	Subindex	Object data	Meaning
0x1301	0	Number of the following entries	
	1	Information Direction	Definition, if the SRDO is switched off (0), a TSRDO (1) or a RSRDO (2)
	2	Refresh Time / SCT	Distance between two transmissions of a SRDO
	3	SRVT	Distance between the two CAN messages of a SRDO
	4	Transmission Type	Type of transmission of the SRDO (fix 254)
	5	COB-ID 1	CAN identifier normal data
	6	COB-ID 2	CAN identifier inverted data

Table 1: Communication parameters for the first SRDO

Information Direction (Subindex 1)
The *Information Direction* is used to determine if the SRDO is switched off or if it is used as send or receive SRDO. The following values are possible:

Value	Meaning
0x00	the SRDO is switched off
0x01	the SRDO is switched on as send-SRDO
0x02	the SRDO is switched on as receiver-SRDO
0x03 – 0xFF	reserved

Table 2: Information Direction of a SRDO

Refresh Time / SCT (Subindex 2)
The *Refresh Time / SCT* sets the distance between two transmissions of a SRDO that is the distance between the first CAN messages of a SRDO. For send-SRDOs, the parameter is the distance between two transmissions of the SRDO. For receiver-SRDOs, this is the maximum time allowed between two transmissions of the SRDO for the SRDO to be valid.

The specification is given in milliseconds.

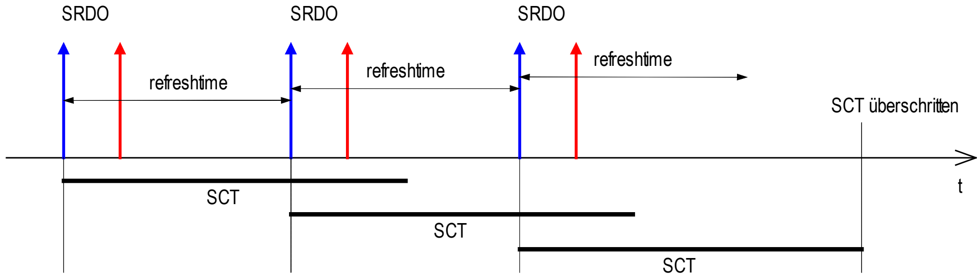


Figure 1: SCT principle

SRVT (Subindex 3)
The *SRVT* sets the maximum distance between the two CAN messages of a receiver-SRDO which is the time between the message with normal data and the message with inverted data. Send-SRDOs are directly sent one after another.

The specification is given in milliseconds.

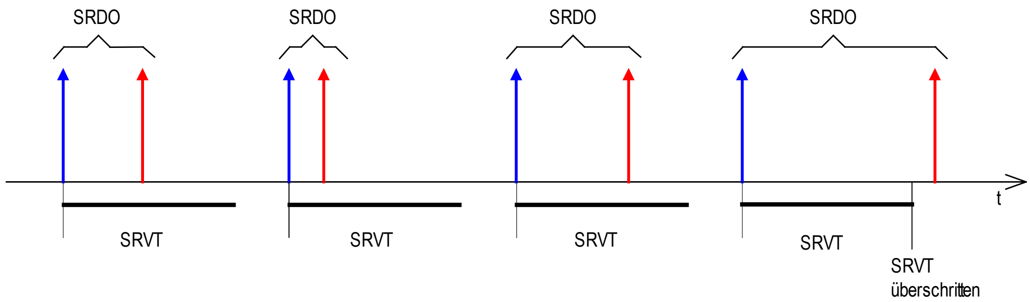


Figure 2: SRVT principle

Transmission Type (Subindex 4)
The *Transmission Type* sets the character of a SRDO transmission. Only value 254 is valid. This implies an asynchronous transfer (see CiA DS301).

COB-IDs (CAN identifier, Subindex 5 and 6)
COB-IDs 1 and 2 support the identification and definition of the priority of a SRDO for bus accesses. There may only be one sender (prodcer) for each CAN message, but several receivers (consumers). Values between 0x101 – 0x180 are acceptable. One SRDO always consists of two sequenced COB-Ids whereas COB-ID 1 is uneven and COB-ID 2 is the subsequent ID. Modifying the values is only possible if the SRDO is switched off which means subindex 1 *Information Direction* is set to 0.

Bit 31 – 11	***Bit 10 – 0***
reserved (0)	CAN Identifier

Table 3: Set-up of a COB-ID for a SRDO

1.1.2 Mapping parameter of a SRDO

Mapping parameters describe the data content of a SRDO. Mapping parameters are entries in the object directory (Index 0x1381 – 0x13C0). One mapping entry is structured such as the mapping of a PDO (see L-1020). But for the SRDO mapping, one entry is always generated for normal data and followed by a corresponding entry for inverted data.

Index	Sub-index	Object data	Meaning
0x1381	0	8	Number of mapped entries
	1	0x20000310	UNSIGEND16 to Index 0x2000, Subindex3 (normal data)
	2	0x21000310	UNSIGEND16 to Index 0x2100, Subindex3 (inverse data)
	3	0x20010108	UNSIGEND8 to Index 0x2001, Subindex1 (normal data)
	4	0x21010108	UNSIGEND8 to Index 0x2101, Subindex1 (inverse data)
	5	0x20010208	UNSIGEND8 to Index 0x2001, Subindex2 (normal data)
	6	0x21010208	UNSIGEND8 to Index 0x2101, Subindex2 (inverse data)
	7	0x20020620	REAL32 to Index 0x2002, Subindex6 (normal data)
	8	0x21020620	REAL32 to Index 0x2102, Subindex6 (inverse data)

Table 4: Exemplary Mapping Table for the first SRDO

1.1.3 CRC of a SRDO

To check the validity of the parameters of a SRDO, a CRC is calculated via the safety-relevant data of each SRDO. It is filed to Index 0x13FF in the object directory. The number of the subindex complies with the number of the SRDO. The following parameters go into the CRC:

Communication parameter:
a) 1 Byte Information Direction
b) 2 Byte Refresh Time / SCT
c) 1 Byte SRVT
d) 4 Byte COB-ID 1
e) 4 Byte COB-ID 2

Mapping parameter:
f) 1 Byte Subindex 0
g1) 1 Byte Subindex
h1) 4 Byte Mapping data
...
g128) 1 Byte Subindex
h128) 4 Byte Mapping data

The following polynom is used: $G(x) = X^{16} + X^{12} + X^{5} + 1$.
The start value for the CRC is 0x0000.

1.2 Configuration Valid

To make an entire SRDO configuration valid, a flag must be set to Index 0x13FE in the object dictionary. This flag is automatically set to an unvalid configuration for every write access that is done to a safety-relevant SRDO parameter. After completing the configuration, this flag must be set to a valid configuration.

Value	Meaning
0xA5	the configuration is valid
Other values	the configuration is valid

Table 5: Configuration Valid

General procedure of a configuration:
1.) Writing all safety-relevant parameters and the checksums
2.) Reading back all safety-relevant parameters and the checksums and comparison with the written parameters
3.) Setting the configuration to valid

This flag must be checked periodically by the application in the security cycle time. As long as this flag is not valid, the safe state must not be left.

1.3 Global Fail-Safe Command GFC

To increase the response time in safety-targeted systems, a GFC is defined that consists of a high-priority CAN telegram (CAN identifier 1). The GFC does not contain data and can be used by all participants. Afterwards, the initiating participant must send the corresponding SRDO.

The usage of GFC is optional. It is event-triggered and not safety-relevant, because there is no time monitoring.

For the GFC, the entry Global Fail-Safe Command parameter to Index 0x1300 is included in the object directory. The following values are possible:

Value	Description
0x00	GFC is not supported
0x01	GFC is supported
Other values	reserved

Table 6: Global Fail-Safe Command GFC

1.4 Predefined Connection Set

For the SRDO, the Predefined Connection Set of CiA DS301 is extended as follows:

Broadcast objects:

Object	**Function code**	**COB-ID**	**Index in the object directory**
GFC	0000	0x001	0x1300

Table 7: Extension Broadcast Predefined Connection Set

Peer-to-Peer Objects:

Object	**Function code**	**COB-ID normal data**	**COB-ID inverse data**	**Index in the object directory**
SRDO messages				
SRDO (Node-ID 1 – 32)	0010	0x101 – 0x13F	0x102 – 0x140	0x1301 – 0x1340 tx
SRDO (Node-ID 33 – 64)	0010	0x141 – 0x17F	0x142 – 0x180	0x1301 – 0x1340 rx

Table 8: Extension Peer-to-Peer Predefined Connection Set

1.5 Overview safety-targeted entries in the object directory

Index	Name	Object type	Data type	Attributs
0x1300	GFC parameter	var	u8	rw
0x1301	1. SRDO communication parameter	record	SRDO parameter	rw
...	...	...	...	...
0x1340	64. SRDO communication parameter	record	SRDO parameter	rw
0x1341	Reserved			
...	...			
0x1380	Reserved			
0x1381	1. SRDO mapping parameter	array	u32	rw
...	...	...	...	...
0x13C0	64. SRDO mapping parameter	array	u32	rw
0x13C1	reserved			
...	...			
0x13FD	reserved			
0x13FE	Configuration Valid	var	u8	rw
0x13FF	Safety Configuration Checksum	array	u16	rw

Table 9: SRDO entries in the object directory

1.6 Certification

The software package SO 1059 is an expansion pack for the CANopen Source Code SO 877. It cannot be certified as a single unit. The certification requires a self-contained unit with all the necessary software components. Therefore, the manufacturer of the device is always responsible for the certification.

The necessities for certification depends on the level of security which shall be achieved. SIL[1]3 for example has a higher necessary requirements than SIL2.

For SIL3 certification the hardware needs to be designed with two channels (*see Figure 3* and *Figure 4*). Lower requirements can be build with a single-channel (see *Figure 5*). For this purpose the use of a Safety-CPU (e. g. TMS570LS by Texas Instruments) is recommended.

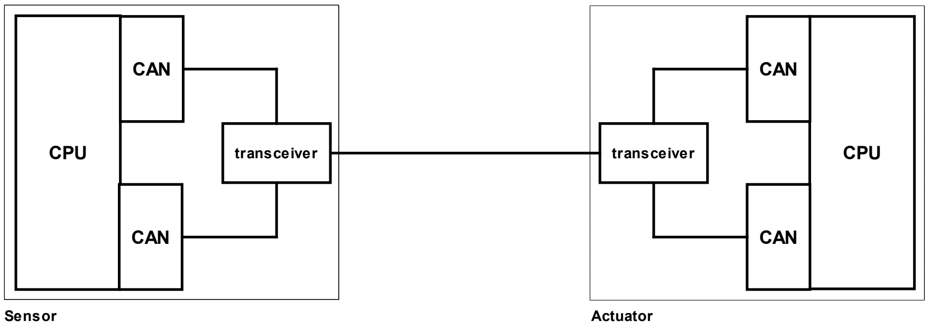


Figure 3: *two-channel hardware with CPU*

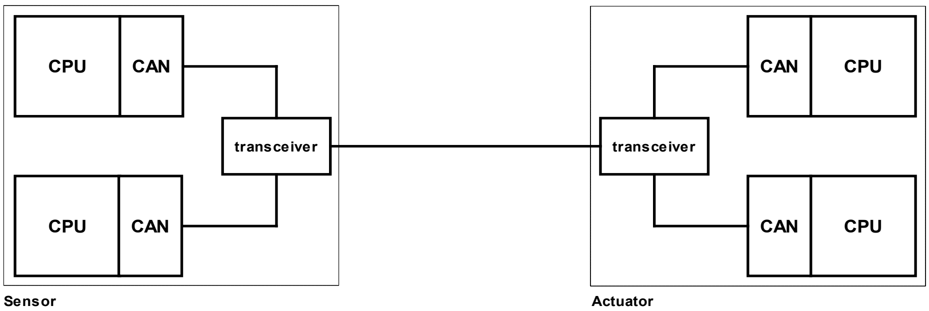


Figure 4: *two-channel hardware with two CPU's*

Figure 5: *single-channel hardware with Safety-CPU*

[1] SIL Safety Integrity Level

Furthermore, the implementation of additional security checks in the software is recommended. These are listed below:

- Repeated calculation of a CRC on the program memory
- Repeated testing of the RAM used
- Use of Watchdog
- Evaluation of exceptions that can occur due to programming errors (e. g. accesses to protected memory, accesses to unaligned addresses, etc.)

The extension package SO-1059 already provides the following options for security tests in the software:

- Calculating CRC over SRDO configuration
- Sending of SRDOs over two CAN messages with the normal and inverted data
- Monitoring the Safety Cycle Time SCT and Safety Related Validation Time SRVT as well as the inverted data for received SRDOs

If an error occurs, the software must always go into a safe mode for the switching outputs, so that no living beings can be injured or other machines destroyed.

It is recommended to coordinate the structure of the hardware with the admissions office before starting with the implementation.

2 Extension of the CANopen user layer

This section explains the extension of the *SYS TEC* CANopen Stack user layer described in L-1020. Moreover, it provides details about the data structures and API functions of the *SYS TEC electronic GmbH*-specific implementation of the CANopen standard CiA DS 304 - in the following called SRDO module.

The description contains the syntax of the functions, the parameter, the return value and explanations about the usage.
Section 2.13 explains the meaning of the return codes and the supported abort codes.

2.1 Limitations of the hardware

The usage of the SRDO module requires a CAN controller for which the chronological sequence of CAN messages on the CAN bus can be determined.

Currently, the SRDO module is adjusted to the SJA1000 CAN controller of the company Phillips. More CAN controllers will follow.

The number of high-prioritized buffer entries of the CAN controller in file obdcfg.h must be set to the minimum number of receive- and send-SRDO.

2.2 Limitations of the software

The SRDO module can only be operated with a particular configuration of the CAN driver. To do this, please put in the file copcfg.h the following defines to the following values:

```
CDRV_USE_HIGHBUFF     TRUE
CDRV_USE_BASIC_CAN    TRUE
CDRV_USE_IDVALID      TRUE
```

The number of high-priority buffer entries of the CAN controller in the file obdcfg.h is set at least to the number of receive and send SRDOs. To ensure safety you can increase this number.

The OD-Builder (up to version V1.19 of the date of this notice) can not be used to create object directory with SRDOs because the index objects between 0x1300 and 0x13FF use special macros not support by this version of the OD-Builder.

If the number of SRDOs needs to be increased copy the corresponding objects in the file obdict.h and adjust the object index resp. the subindex (see also chapter 2.10.1 and 2.10.2). If other objects needs to be extended or added you can create them in a temporary directory with the OD-Builder and copy&paste them to the actual file objdict.h.

2.3 Software structure

The SRDO module is integrated in the stack in parallel to the existing modules such as PDO or SDO.

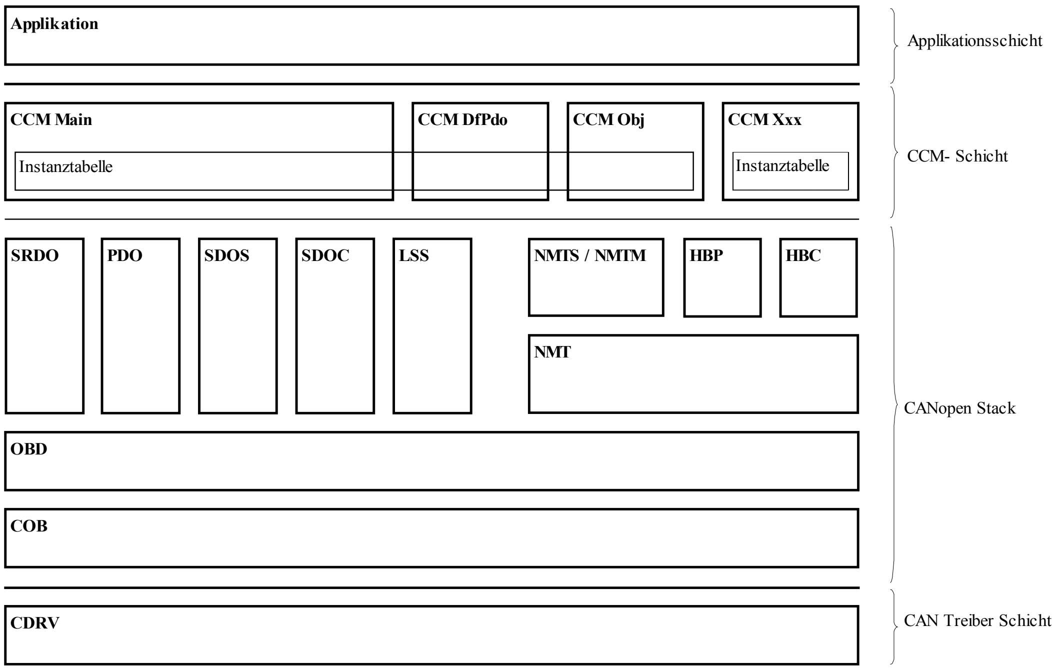


Figure 6: General software structure

The implementation contains two different SRDO modules:

SRDO.C	This module contains the services to define and transmit SRDOs.
SRDOSTC.C	This module provides the same services as SRDO.C, but it concerns the realization of static SRDO mapping.
CCMSRDO.C	User interface of the SRDO module

2.4 Configuration of the software

The software configuration is the same as in the standard CANopen stack also with the copcfg.h file. For SRDOs there are a few defines that are explained below. Lack of these defines in the file copcfg.h activate their default settings.

SRDO_USE_STATIC_MAPPING:

Value range:	FALSE, TRUE
Default:	FALSE
Meaning:	If TRUE static mapping is used instead of dynamic mapping of the SRDOs. The mapping then cannot be changed neither via SDO nor by the application during the runtime. Instead of SRDO.C SRDOSTC.C file must the be used.

SRDO_USE_DUMMY_MAPPING:

Value range:	FALSE, TRUE
Default:	FALSE
Meaning:	When using the dynamic SRDO mapping dummy objects can be mapped if this define is set to TRUE. This allows for Receive-SRDOs not having to implement any variable in the OD, if these variables are not important for a CANopen node.

SRDO_GRANULARITY:

Value range:	8, 16, 32, 64
Default:	8
Meaning:	This define determines the smallest resolution in bits of the variables of a SRDOs. Does this define ist set to 8, then up to 8 normal and 8 inverse data can be mapped into a SRDO. For value 16, the number halves to 4 normal and 4 inverse data, etc.

SRDO_ALLOW_GAPS_IN_OD:

Value range:	FALSE, TRUE
Default:	FALSE
Meaning::	This define is used to optimize the code requirements in SRDO module. If the SRDOs in the object directory sequentially implemented without gaps, then this define can be left to FALSE. In this case, the SRDOs for the checks are referenced more quickly. Are there some SRDOs missed in the object directory (e. g. only SRDO2 with communication index 0x1301 is going to be implemented but SRDO1 with index 0x1301 is missing – or SRDO1 and SRDO3 is going to be implemented, but SRDO2 is missing) then this define must be set to TRUE. In this case, the SRDOs are referenced by a search algorithm from which a higher runtime oft he softare results. See also chapter 2.10.2.

SRDO_USE_GFC:

Value range: FALSE, TRUE
Default: FALSE
Meaning:: If the GFC message is not needed in a project, then the API functions CcmSendGfc () and SrdoSendGfc () and the object 0x1300 can be omitted for reasons of optimization. In this case, the Define SRDO_USE_GFC must be set to FALSE.

SRDO_USE_PROGMONITOR:

Value range: FALSE, TRUE
Default: FALSE
Meaning: Is a project of the Program-Monitor not needed, then it can be removed for reasons of program code optimization by set this define to FALSE. The callback function AppProgMonEvent () is not called in this case.

SRDO_CHECK_SRVT_BEFORE_1STRX

Value range: FALSE, TRUE
Default: FALSE
Meaning: Should the SRVT also be monitored as the SCT cyclically by calling the SrdoProcess () function if only one of the two CAN messages of a SRDOs was received, then this constant must be set to TRUE. Is this constant set to FALSE, then an error is detected at the earliest, when the second CAN message was received after the SRVT or after the SCT has expired. With TRUE, an error is detected immediately after the SRVT.

2.5 Function of the SRDO module

The SRDO module takes over the SRDO processing for dynamic SRDO mapping (this means the mapping can be modified by the application or by the SDO during runtime).
Module SRDOSTC supports the static SRDO mapping.

For each SRDO, a structure with all relevant data is generated to accelerate the SRDO processing. Those structures are summarized in tables. The SRDO tables are part of the object directory.

Each SRDO uses variables that must be created by the application beforehand. During the mapping, addresses in the SRDO are directed to the corresponding variables. This means that there must be a variable for each mappable object. Therefore, when defining the object directory in file **objdict.h,** macro OBD_SUBINDEX_RAM_USERDEF or OBD_SUBINDEX_RAM_USERDEF_RG must be used for the respective object. The SRDO module checks the chosen parameters for each modification of the mapping. If the object does not exist or if it does not have a variable of the application, an errors is reported.

2.5.1 Sending SRDOs

SRDOs are directly sent from the application. Therefore, function **CcmSrdoSend()** is used. The Refresh Time is monitored in the application because only the application can assure that the normal and inverted datas are consistent before the CAN messages of a SRDO are sent.

It is important that the first sending must be held up by 0,5ms * NodeId after switching into the node state OPERATIONAL. The change of the node state is communicated to the application in function **AppCbNmtEvent()**.

2.5.2 Receiving SRDOs

Function **SrdoProcess()** is in charge of receiving SRDOs. This function must be called cyclically which is realized for function **CcmProcess()**.

2.5.3 Sending and receipt signaling of SRDOs

The sending and receipt is signaled to the application via two different ways.
One the one hand via the callback function of the application **AppSrdoEvent()** and **AppSrdoError()** and on the other hand via the state of a SRDO that is to be polled by the application. It is read with **CcmSrdoGetState()** and written with **CcmSrdoSetState()**.

The state of a SRDO is bit-coded in the following way:

```
TX-SRDO:
xx00 xxxxb   Sending was ok
xx01 xxxxb   Sending was incorrect
xx11 xxxxb   SRDO was edited

RX-SRDO:
xx00 xxxxb   Receipt was ok
xx01 xxxxb   Receipt was incorrect
xx11 xxxxb   SRDO was edited

SRDO-ERROR:
00xx xxxxb   Reset value
01xx xxxxb   Value prior to calling AppSrdoError()
10xx xxxxb   AppSrdoError() must set this value
```

The application must follow both ways.

Example for the receipt of a SRDO:
The SRDO module sets the state to “receipt OK”. Afterwards, the SRDO module calls function **AppSrdoEvent()**. This function checks if the state is set to „receipt OK“. If this is not the case, it is relevant to security. The application must react. If the state is correct, the status is set to “SRDO was edited”.

The state must also be checked for the application in the main loop. It must always be in state “SRDO was edited” because otherwise it would indicate that the SRDO in function **AppSrdoEvent()** was not edited. This would be satefy-critical.

With the implementation of the SRDO module we follow the philosophy that the safe state is taken only with the successful reception of the SRDOs. If an error appears during the runtime the safe state is left.

2.5.4 Logical monitoring of program run of the SRDO module

A logical monitoring of the program run is integrated in the SRDO module. Function **AppProgMonEvent()** is called with the respective Event for different program steps. The actual realization of the program run monitor takes place in the application function that is called.

2.6 Function of the SRDOSTC module

The SRDOSTC module replaces the SRDO module for static SRDO mapping. With the static SRDP mapping, the SRDOs are already mapped in the OD. The mapping cannot be modified by the application or the SDO. Thus, fewer CODE memory is needed.

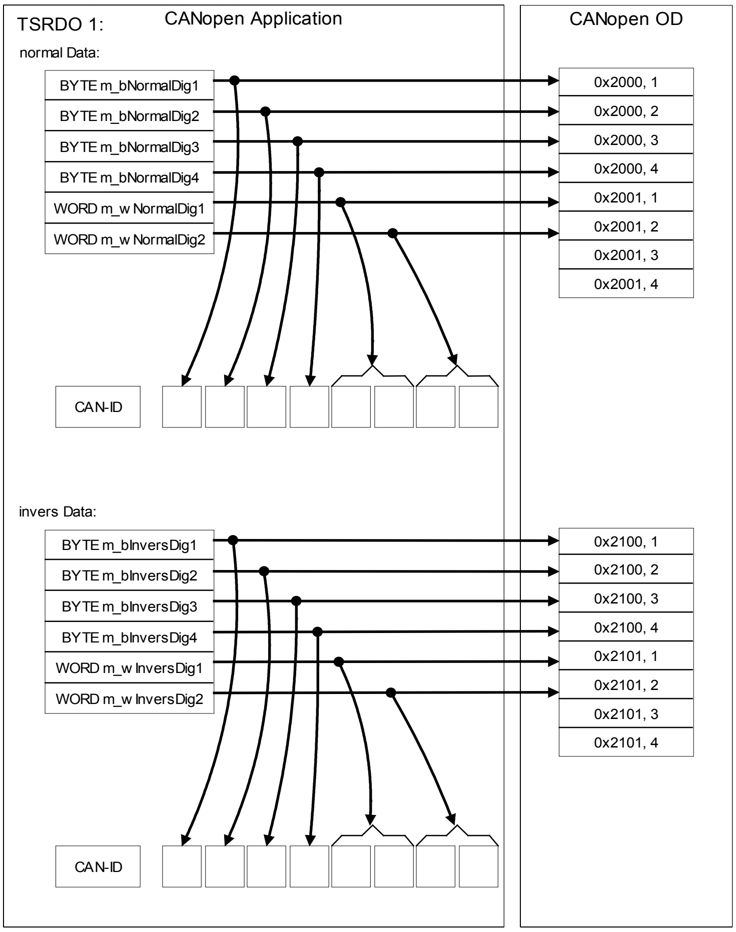


Figure 7: Figure of variable fields

The relation of SRDO variables in the application to data in the OD or to data in the CAN message is created via function **CcmStaticDefineSrdoVarField()**. The application must provide two times 8 connected data bytes maximum for each SRDO (which means without fill bytes → Struct Alignment 1). In this manual these data packages are called variable fields of a SRDO. Mapping the variable fields in the OD takes place in the application by calling function **CcmDefineVarTab()** or through macro OBD_SUBINDEX_RAM_EXTVAR (*see L-1024*) in the OD.

To use the static SRDO mapping, file SRDOSTC.C must be mounted instead of file SRDO.C. Moreover, define SRDO_USE_STATIC_MAPPING must be set to TRUE within file CopCfg.h.

Restriction:
For CPUs that do not support uneven accesses to data types larger BYTE, a mixed mapping of BYTE and WORD is not possible, e.g. for example:
BYTE – WORD – BYTE

But the following mapping is possible:
BYTE – BYTE – WORD

2.7 General program run

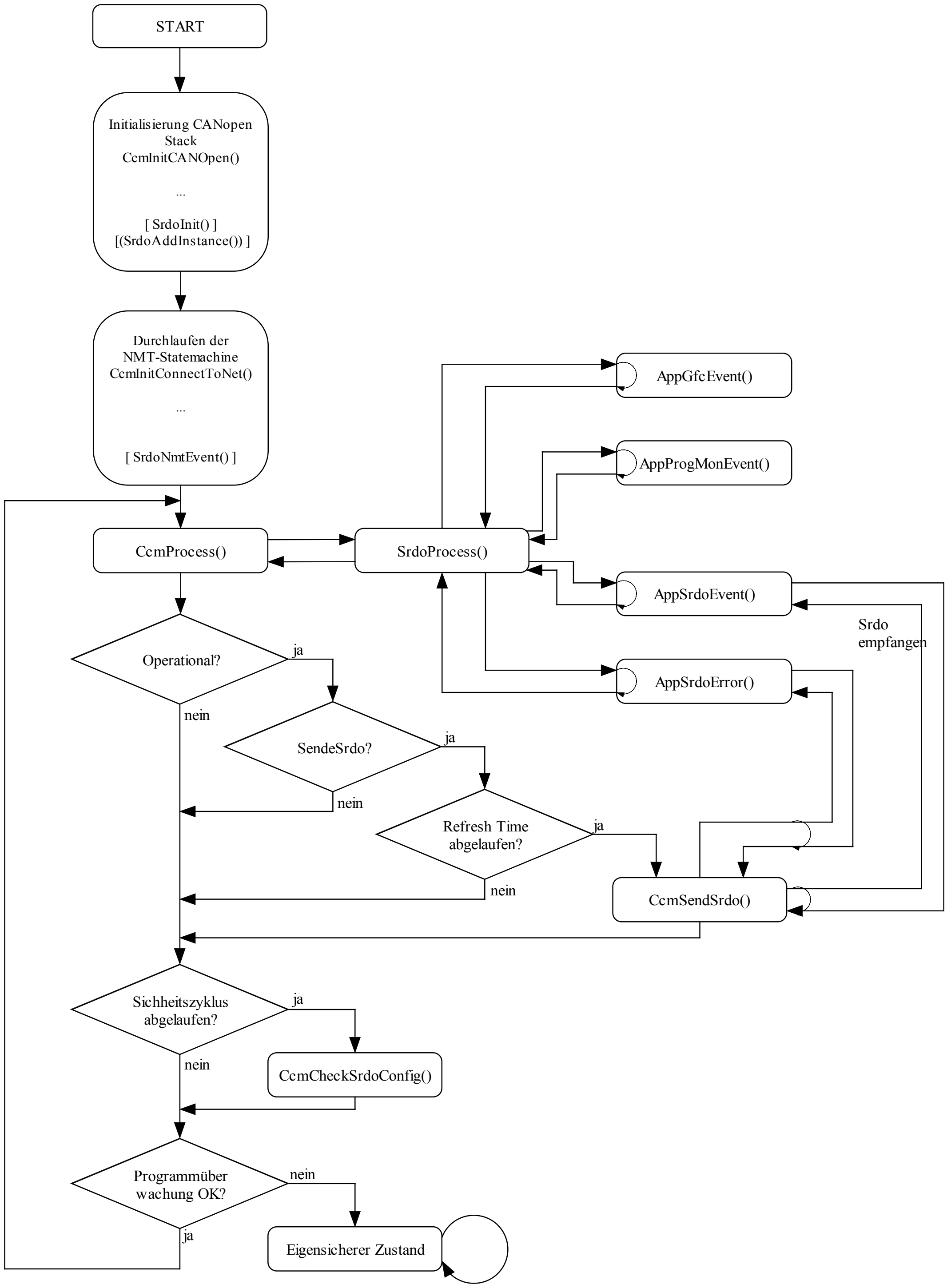


Figure 8: General program run

2.8 Extention of the CCM layer

File CCMMAIN.C is extended for the integration of the SRDO module.
The SRDO module must be activated in file COPCFG.H via define `CCM_MODULE_INTEGRATION`. Therefore, constant `CCM_MODULE_SRDO` must be added.

If the SRDO module is activated, function **CcmInitCANOpen()** executes the initialization of the SRDO module. The appropriate SRDO function is also called in function **CcmProcess()**.

In the following, user functions of the SRDO module are described.

2.8.1 Function CcmSendSrdo

Syntax:

```
#include <cop.h>
tCopKernel          PUBLIC  CCM_DECL_INSTANCE_HDL_
CcmSendSrdo (
                            WORD wSrdoCommuIndex_p);
```

Parameter:

CCM_DECL_INSTANCE_HDL_:	Instanz-Handle
wSrdoCommuIndex_p:	Object index of the communication parameter of the SRDO in the object directory

Return:

kCopSuccessful	The function was executed without error.

For more return codes, see 2.11.5 - Function SrdoSend().

Description:

The functions sends a SRDO specified via the communication index or it sends all SRDO if 0x0000 is specified as communication index. Before a SRDO sends CAN-messages all bits of the data are checked in terms of correct inverting. If at least one bit is not correct inverted all CAN messages of a SRDO are not sent and the callback function APPSrdoError() is called.

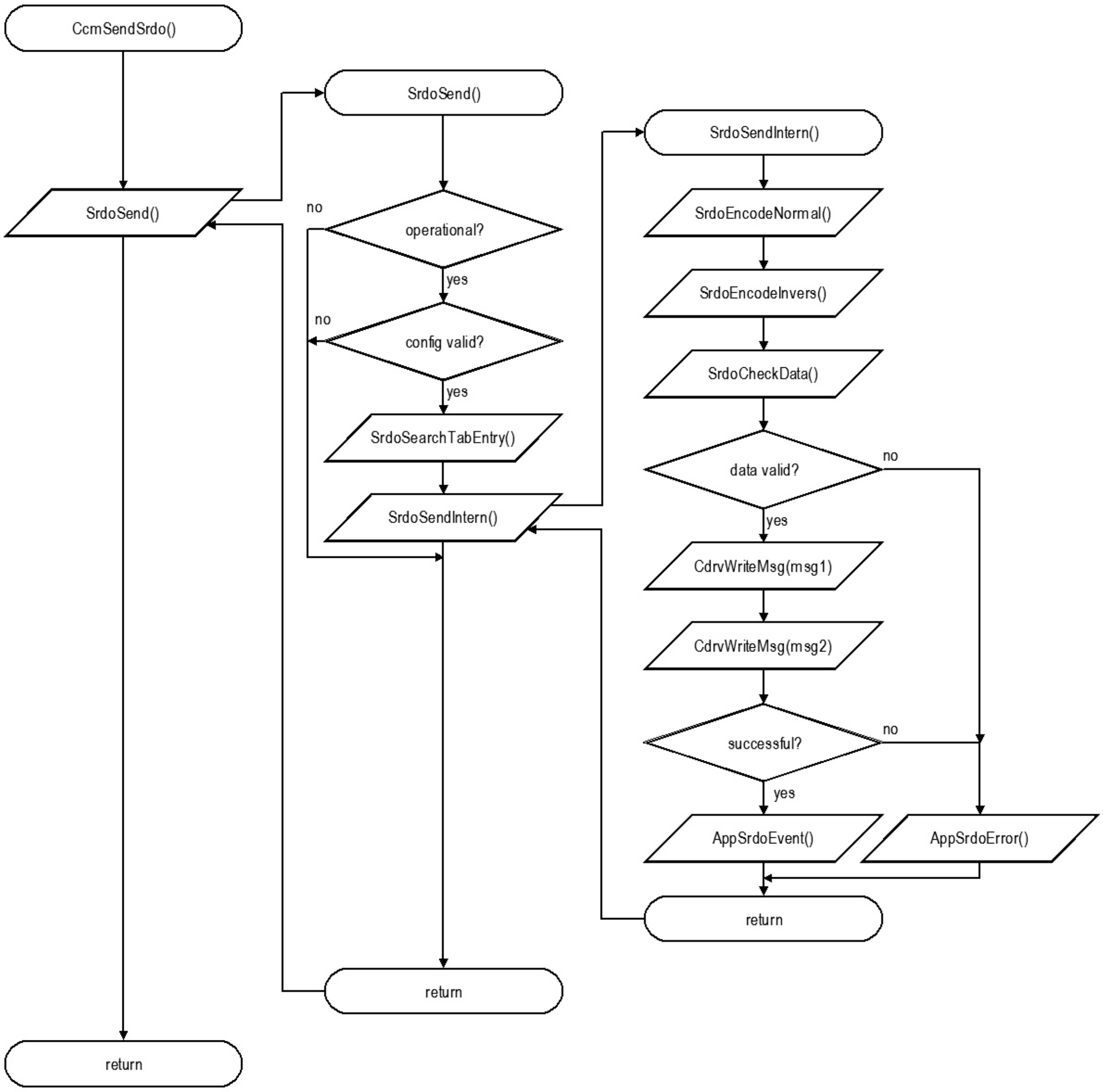


Figure 9: Principle for sending SRDOs

2.8.1.1 Function CcmCheckSrdoConfig

Syntax:

```
#include <cop.h>
tCopKernelPUBLIC CcmCheckSrdoConfig (
                        CCM_DECL_INSTANCE_HDL_
                        WORD * pwCommuIndex_p);
```

Parameter:

CCM_DECL_INSTANCE_HDL_: Instanz-Handle

pwCommuIndex_p: Pointer to a variable in which the function provides the communication index of the faulty SRDO in case of a faulty configuration

Return:

kCopSuccessful The function was executed without error.

For more return codes, see 2.11.7 - Function SrdoCheckConfig().

Description:

The function calculates the checksum (CRC) for all going SRDO (also deactivated SRDOs with direction = 0) and compares them to the one configured in the OD. If it identifies an error, it sends back the error and the communication index of the faulty SRDO. This function represents the API function for SrdoCheckConfig() and calls it. It is necessary to call this function as part of the diagnosis periodically in the diagnostic test interval. If an error is detected and the entry *Configuration Valid (Index 0x13FE)* is *valid (0xA5)* then has to be changed in the safe state.

Please note: The function SrdoCheckConfig() is called with the value 0xA5 by the SRDO module when the entry *Configuration Valid* is written (Index 0x13FE in the object directory).

2.8.2 Function CcmSendGfc

Syntax:

#include <cop.h>
tCopKernel PUBLIC CCM_DECL_INSTANCE_HDL)
CcmSendGfc (

Parameter:

CCM_DECL_INSTANCE_HDL: Instanz-Handle

Return:

kCopSuccessful The function was executed without error.

For more return codes, see 2.11.8 - Function SrdoSendGfc().

Description:

The function sends a GFC message.

It will be not available if the configuration of SRDO_USE_GFC is set to FALSE.
This function represents the API-function for Function SrdoSendGfc() and calls it.
The following SRDO must be transferred by the application via function **CcmSendSrdo()**.

2.8.3 Function CcmGetSrdoState

Syntax:

```
#include <cop.h>
tCopKernel                PUBLIC  CCM_DECL_INSTANCE_HDL_
CcmGetSrdoState (
                                  BYTE * pSrdoState_p,
                                  WORD wSrdoCommuIndex_p);
```

Parameter:

CCM_DECL_INSTANCE_HDL_:	Instanz-Handle
pSrdoState_p:	Pointer to which the functions copies the status
wSrdoCommuIndex_p:	Object index which contains communication parameters of the SRDO in the object directory

Return:

kCopSuccessful	The function was executed without error.

For more return codes, see 2.11.9 - Function SrdoGetState().

Description:

The function reads the status of a SRDO. For setup and usage of the status, please see 2.5.3.

2.8.4 Function CcmSetSrdoState

Syntax:

```
#include <cop.h>
tCopKernel                    PUBLIC  CCM_DECL_INSTANCE_HDL_
CcmSetSrdoState (
                                      BYTE SrdoState_p,
                                      WORD wSrdoCommuIndex_p);
```

Parameter:

CCM_DECL_INSTANCE_HDL_: Instanz-Handle

SrdoState_p: status to be set

wSrdoCommuIndex_p: Object index which contains communication parameters of the SRDO in the object directory

Return:

kCopSuccessful The function was executed without error.

For more return codes, see 2.11.10 - Function SrdoSetState().

Description:

The function writes the status of a SRDO. For setup and usage of the status, please see 2.5.3.

2.8.5 Function CcmGetSrdoParam

Syntax:

```
#include <cop.h>
tCopKernel                PUBLIC  CCM_DECL_INSTANCE_HDL_
CcmGetSrdoParam (
                            WORD wSrdoCommuIndex_p,
                            tSrdoCommuParam                 *
                            pSrdoCommuParam_p,
                            tSrdoMappParam * pSrdoMappParam_p);
```

Parameter:

CCM_DECL_INSTANCE_HDL_: Instanz-Handle

wSrdoCommuIndex_p: Object index which contains communication parameters of the SRDO in the object directory

pSrdoCommuParam_p: Pointer to the structure in which the function copies the values for *Information Direction* and *SCT*

pSrdoMappParam_p: Pointer to the structure in which the function copies the values for *Number of Mapped Objects* and the pointers to the mapped variables

Return:

kCopSuccessful The function was executed without error.

For more return codes, see 2.11.11 - Function SrdoGetCommuParam() and 2.11.12 - Function SrdoGetMappParam().

Description:

The function reads the parameters of a SRDO needed in the application. Those are the communication parameters *Information Direction* and *SCT* as well as the mapping parameters *Number of Mapped Objects* and the pointer to the mapped variables. The function only completes the structures if the transferred pointer is not the null-pointer. Structure tSrdoMappParam only exists for dynamic mapping.

This function represents the API-function for SrdoGetCommuParam() and the function SrdoGetMappParam() and calls them.

Structure tSrdoCommuParam is set up as follows:

```
typedef struct
{
    BYTE m_bDirection;  // direction of SRDO
                        // (0-invalid; 1-Tx; 2-Rx)
    WORD m_wSct;        // refresh time / SCT
} tSrdoCommuParam;
```

Structure tSrdoMappParam is set up as follows:

```
typedef struct
{
    BYTE       m_bNoOfMappedObjects;
                         // Number of mapped objects
    void MEM* m_apMappedVariable[SRDO_MAX_MAPPENTRIES];
                         // array of pointers to the
                            mapped variables
} tSrdoMappParam;
```

2.8.6 Function CcmStaticDefineSrdoVarFields

The function only is available for static SRDO mapping.

Syntax:

```
#include <cop.h>
tCopKernel PUBLIC
CcmStaticDefineSrdoVarFields(
                                  CCM_DECL_INSTANCE_HDL _
                                  WORD       wCommuIndex_p,
                                  void MEM* pNormalData_p,
                                  void MEM* pInversData_p);
```

Parameter:

CCM_DECL_INSTANCE_HDL_: Instanz-Handle

wCommuIndex_p: Communication index of the SRDO in the OD for which variables shall be defined.

pNormalData_p: Pointer to connected variable field which shall be linked (or mapped) with normal data of SRDO.

pInversData_p: Pointer to a connected variable field which shall be linked (or mapped) with inverse data of the SRDO.

Return:

kCopSuccessful The function was executed without error.

For more return codes, see 2.12.1 - Function SrdoStaticDefineVarField().

Description:

This function defines the variable fields for a SRDO. The application only modifies the variables via those variable fields. When sending a SRDO, those data bytes are copied from the variable field into the two CAN messages. When receiving a SRDO, the data bytes of the CAN messages are directly copied into the variable fields.

The function checks if the specified variable fields are conform with the variables to which the mapping in the OD points.

This function represents the API-function for SrdoStaticDefineVarFields() and calls it.

2.8.7 Function CcmCalcSrdoCrc

Syntax:
#include <srdo.h>
tCopKernel PUBLIC CcmCalcSrdoCrc (MCO_DECL_INSTANCE_PTR_
WORD wCommuIndex_p,
WORD* pwCrc_p);

Parameter:
MCO_DECL_INSTANCE_PTR_: Pointer to the instance

wCommuIndex_p: Index object contains the communication parameters of the SRDO in the object directory

pwCrc_p: Pointer to a WORD variable to return the 16-bit CRC in the calling function.

Return:
kCopSuccessful The function was executed without error.

kCopSrdoNotExist The SRDO chosen does not exist.

Description:
The function calculates the checksum (CRC) over a SRDO and returns it to the calling function. The calculation is done also when the SRDO is turned off. There is no comparison to the accuracy of the CRC. The application can use this function to update the CRC of a SRDOs if the configuration of SRDOs needs to be dynamically set new over the application (eg changing the COB-ID depending on the node ID). This function represents the API function SrdoCalcSrdoCrc () and calls it.

Note: The validity check of the CRC, that is the calculation of the CRC over the configuration data of a SRDO and comparison with the associated CRC in the index 0x13FF is made in the function **CcmCheckSrdoConfig ()**.

Example:

```
WORD  wTestCrc;
Ret = CcmCalcSrdoCrc (0x1301, &wTestCrc);
if (Ret != kCopSuccessful)
{
    goto Exit;
}
PRINTF1 ("Calculated CRC of SRDO1 = 0x%04X\n", wTestCrc);
```


2.9 Functions in the application

Function of the application that are called by the SRDO module as callback function are called directly and not via function pointer as for the rest in CANopen. Consequently, those functions must be available in the application and may not be renamed.

2.9.1 Function AppSrdoEvent

Syntax:

```
#include <cop.h>
tCopKernel              PUBLIC  CCM_DECL_INSTANCE_HDL_
AppSrdoEvent (
                                WORD wSrdoCommuIndex_p)
```

Parameter:

CCM_DECL_INSTANCE_HDL_: Instanz-Handle

wCommuIndex_p: Communication index of the SRDO in the OD

Return:

kCopSuccessful The function was executed without error.

All other return codes are reserved.

Description:

The function is called by the SRDO module if the transmission of a SRDO is accurate (receiving or sending). The status of the SRDO must be taken care of in the function according to chapter 2.5.3.

```
tCopKernel PUBLIC AppSrdoEvent (CCM_DECL_INSTANCE_HDL_
    WORD wSrdoCommuIndex_p)
{
BYTE bSrdoState;
tCopKernel Ret;

    Ret = CcmGetSrdoState (&bSrdoState,
                           wSrdoCommuIndex_p);
    if (Ret != kCopSuccessful)
    {
        ...
    }

    if ((bSrdoState & 0x30) != 0x00)
    {
        // Sicherheitskritischer Fehler !!!
        ...
    }

    // je nach Anwendung Information verarbeiten
    // beispielsweise Ausgänge des SRDO einschalten

    Ret = CcmSetSrdoState ((bSrdoState | 0x30),
                            wSrdoCommuIndex_p);
    if (Ret != kCopSuccessful)
    {
        ...
    }

    return kCopSuccessful;
}
```

2.9.2 Function AppSrdoError

Syntax:

```
#include <cop.h>
tCopKernel PUBLIC AppSrdoError  CCM_DECL_INSTANCE_HDL_
(
                                WORD wSrdoCommuIndex_p,
                                tCopKernel ErrorCode_p)
```

Parameter:

CCM_DECL_INSTANCE_HDL_: Instanz-Handle

wCommuIndex_p: Communication index of the SRDO in the OD

ErrorCode_p: Error code of the SRDO:

kCopSrdoSctTimeout	The SCT of a receive SRDO was exceeded.
kCopSrdoSrvtTimeout	The SRVT of a receive SRDO was exceeded.
kCopSrdoNotInOrder	The two CAN meassages of a SRDO have been received in the wrong order.
kCopSrdoDataError	The data of the CAN messages of a SRDO is not inverse.

More error codes are possible from the CDRV module.

Return:

kCopSuccessful	The function was executed without error.

All other return codes are reserved.

Description:

The function is called by the SRDO module if the transmission of a SRDO is incorrect (receiving or sending). The status of the SRDO must be taken care of in the function according to chapter 2.5.3.

```
tCopKernel PUBLIC AppSrdoError (CCM_DECL_INSTANCE_HDL_
    WORD wSrdoCommuIndex_p,
    tCopKernel ErrorCode_p)
{
BYTE bSrdoState;
tCopKernel Ret;

    Ret = CcmGetSrdoState (&bSrdoState,
                           wSrdoCommuIndex_p);
    if (Ret != kCopSuccessful)
    {
        ...
    }

    if ((bSrdoState & 0x30) == 0x10)
    {

        // process information according to the application
        // for example switch outputs of SRDO off

        // Status "SRDO bearbeitet" on set
        bSrdoState |=  0x30;

        // toggle Bit 6 and 7
        bSrdoState ^= 0xC0;
    }
    else
    {
        // Safety Critical Error !!!
        ...
    }

    Ret = CcmSetSrdoState ((bSrdoState),
                            wSrdoCommuIndex_p);
    if (Ret != kCopSuccessful)
    {
        ...
    }

    return kCopSuccessful;
}
```

2.9.3 Function AppGfcEvent

Syntax:

#include <cop.h>
tCopKernel PUBLIC CCM_DECL_INSTANCE_HDL)
AppGfcEvent (

Parameter:

CCM_DECL_INSTANCE_HDL: Instanz-Handle

Return:

kCopSuccessful The function was executed without error.

All other return codes are reserved.

Description:

The function is called by the SRDO module when a GFG message is received.

```
tCopKernel PUBLIC AppGfcEvent (CCM_DECL_INSTANCE_HDL)
{

    // process information according to the application
    // for example change to intrinsically safe state

    return kCopSuccessful;
}
```

This function is not called if the configuration SRDO_USE_GFC is set to FALSE.

2.9.4 Function AppProgMonEvent

Syntax:

```
#include <cop.h>
tCopKernel                    PUBLIC  CCM_DECL_INSTANCE_HDL_
AppProgMonEvent (
                                      tProgMonEvent Event_p)
```

Parameter:

CCM_DECL_INSTANCE_HDL: Instanz-Handle

Event_p: Event of the executed program code:

```
kSrdoPMEvSctChecked
    SCT of a SRDO was tested

kSrdoPMEvSctNotCheckedItIsTx
     SCT of a SRDO was not tested, because it is a send SRDO

kSrdoPMEvSctNotCheckedItIsInvalid
SCT of a SRDO was not tested, because is is switched off

kSrdoPMEvSctNotCheckedNotOperational
     SCT of a SRDO was not tested, because the node is not in
     OPERATIONAL

kSrdoPMEvSrdoError
     found faulty SRDO (send- and receive SRDO)

kSrdoPMEvSrdoReceived
     a SRDO was received

kSrdoPMEvSrdoTransmitted
     a SRDO has been sent
```

Return:

kCopSuccessful The function was executed without error.

All other return codes are reserved.

Description:

The function is called by the SRDO module when certain program steps are processed. The application can setup a logical monitoring of the program run.

This function is not called if in the configuration file copcfg.h the Define SRDO_USE_PROGMONITOR is set to FALSE.

```
tCopKernel PUBLIC AppProgMonEvent (CCM_DECL_INSTANCE_HDL_
    tProgMonEvent Event_p)
{
    switch (Event_p)
    {
        case kSrdoPMEvSctChecked:
            // is called for each Rx SRDO
            wPMonValue_g += kPMonSctChecked;
            break;
        case kSrdoPMEvSctNotCheckedItIsTx:
            // is called for each Rx SRDO
            wPMonValue_g += kPMonSctNotCheckedItIsTx;
            break;
        case kSrdoPMEvSctNotCheckedItIsInvalid:
            // is called for each switched-off SRDO
            wPMonValue_g += kPMonSctNotCheckedItIsInvalid;
            break;
        case kSrdoPMEvSctNotCheckedNotOperational:
            // is called once for all SRDO
            wPMonValue_g += kPMonSctNotCheckedNotOperational;
            break;
        case kSrdoPMEvSrdoError:
            // is called for faulty SRDO
            wPMonValue_g += kPMonSrdoError;
            break;
        case kSrdoPMEvSrdoReceived:
            // is called for each received SRDO
            wPMonValue_g += kPMonSrdoReceived;
            break;
        case kSrdoPMEvSrdoTransmitted:
            // is called for each sent SRDO
            wPMonValue_g += kPMonSrdoTransmitted;
            break;
        default:
            break;
    }

    return kCopSuccessful;
}
```


2.9.5 Function AppCbNmtEvent

This function is called by the CANopen Stack when the NMT-Statemachine is running and must contain different event calls of the SRDO module:

kNmtEvResetCommunication: Notify variable fields by calling **CcmStaticDefineSrdoVarFields()** (for static mapping)
Initialisation of the SRDO communication parameter by calling **CcmWriteObject()** with the respective parameters

```
 // define all SRDOs in static SRDO modul
 Ret = CcmStaticDefineSrdoVarFields (0x1301,
     &SrdoNormalData.m_abSrdoData[0],
     &SrdoInversData.m_abSrdoData[0]);
if (Ret != kCopSuccessful)
{
     ...
}

 // write information direction into OD
 Ret = CcmWriteObject (0x1301, 1, &bDirection, 1);
 if (Ret != kCopSuccessful)
 {
     ...
 }

 // set configuration valid
 bTemp = 0xA5;
 Ret = CcmWriteObject (0x13FE, 0, &bTemp, 1);
 if (Ret != kCopSuccessful)
 {
     ...
 }
```

kNmtEvEnterPreOperational: SRDO may not be processed anymore (save NMT status to evaluate this in the main loop)

```
bSrdoState = kNotOperational;
```

kNmtEvEnterOperational: read the actual SRDO parameter by calling **CcmGetSrdoParameter()**
SRDO must be processed (save NMT status to evaluate this in the main loop)

```
CcmGetSrdoParam (0x1301, &SrdoCommuParam);

bSrdoState = kEnterOperational;
```

2.10 Object directory

Various safety-relevant entries of the object directory are described in chapter 1.

2.10.1 Macros for safety objects

There are special macros for the different SRDO entries for the realisation in the CANopen Software. Those are described in this chapter.

Please Note: The OD-builder (at the time of this note version V1.19) can not generate the specific macros for the SRDOs. Therefore, you should not use this tool for the creation of the object directory. Please read the chapter 2.2

Further information about the object directory is described in document L-1024 “CANopen Object Directory Software Manual”.

OBD_CREATE_SRDO_GFC_PARAM()

The macro OBD_CREATE_SRDO_GFC_PARAM is used to create entry “Global Fail-Safe Command Parameter” (Index 0x1300). The macro does not have parameters.

OBD_CREATE_SRDO_COMMU(ind,num,dir,sct,srvt,cob1,cob2)

and

OBD_BEGIN_SRDO_MAPP(ind,num,cnt)
    OBD_SUBINDEX_SRDO_MAPP(ind,sub,num,name,val)
OBD_END_SRDO_MAPP(ind)

The macro OBD_CREATE_SRDO_COMMU is used to define the communication parameter of the SRDO.
Macros OBD_xxx_SRDO_MAPP are used to define the mapping parameters of a SRDO. An entry of a SRDO always starts with the macro OBD_BEGIN_SRDO_MAPP. The different subindex entries are defined by macro Makro OBD_SUBINDEX_SRDO_MAPP. The entry ends with OBD_END_SRDO_MAPP.
Since there is always the communication parameter and the mapping parameter that correspond to one SRDO, it is important that for both the continuous numbers of the SRDO are set correctly.

- **ind**: Object index of the SRDO to be defined (0x1301 to 0x1340 and 0x1381 to 0x13C0 for the mapping)
- **num**: Continuous number from 0 to 63 for the corresponding entry in the table. The first always gets assigned the continuous number 0. The following entries always get the next larger number of the previous entry. For

example, if the SRDOs 0x1301, 0x1302 and 0x1305 are generated, the SRDO 0x1301 gets a 0, the 01302 a 1 and the 0x1305 a 2.

dir: Information direction of the SRDO. The value corresponds with the index 0x1301 to 0x1340 Subindex 1.

sct: Refresh-Time / SCT of the SRDO. The value corresponds with index 0x1301 to 0x1340 Subindex 2.

srvt: SRVT of the SRDO. The value corresponds with index 0x1301 to 0x1340 Subindex 3.

cob1: COB-ID 1 of the SRDO, this means CAN-Identifier of the message that contains normal data. The value corresponds with index 0x1301 to 0x1340 Subindex 5.

cob2: COB-ID 2 of the SRDO, this means CAN-Identifier of the message that contains inverse data. The value corresponds with index 0x1301 to 0x1340 Subindex 6.

cnt: Number of mapping entries of the SRDO. The value corresponds with the object entry 0x1381 to 0x13C0 Subindex 0.

sub: Subindex of the mapping entry that is to be defined

name: Object name

val: Default value for the mapping data that must be accepted after Reset

OBD_CREATE_SRDO_CFG_VALID()

The macro OBD_CREATE_SRDO_CFG_VALID is used to generate the entry “Configuration Valid” (Index 0x13FE). The macro does not have parameters.

OBD_BEGIN_SRDO_CRC(cnt)
OBD_SUBINDEX_SRDO_CRC(sub,name)
OBD_END_SRDO_CRC()

The macros are used to define the object entries “Safety Configuration Checksum” (Index 0x13FF).

cnt: Number of CRC table entries. If the indexes 0x1301 to 0x1340 contain gaps, CRC entries must be defined. The nth SRDO corresponds with the nth subindex of the CRC

sub: Subindex of the CRC entry that is to be defined

name: Object name

2.10.2 Advice for macros

Please note, the objects in the object dictionary have to be created in ascending order otherwise the CANopen Stack is not able to detect the objects in the OD. This means that the following macros must always be applied in the order listed below. Macros for ommunication and mapping parameters can occur multiple times, depending on how many SRDOs should be applied.

```
OBD_CREATE_SRDO_GFC_PARAM()
OBD_CREATE_SRDO_COMMU(…)
OBD_BEGIN_SRDO_MAPP(…)
OBD_CREATE_SRDO_CFG_VALID()
OBD_BEGIN_SRDO_CRC(…)
```

If several SRDOs are created in one OD it must be taken care that each SRDO has a consecutive number starting with 0. This number must be transferred to the macro OBD_CREATE_SRDO_COMMU() as the second parameter, to the macro OBD_BEGIN_SRDO_MAPP() also as second parameter and to the macro OBD_SUBINDEX_SRDO_MAPP() as the third parameter. The subsequent SRDO always gets the number inceased by one. The number for the communication parameters of a SRDOs is always the same number as the corresponding mapping parameters.

Are the SRDOs in the object dictionary created with gaps, then the define SRDO_ALLOW_GAPS_IN_OD in the file copcfg.h must set to TRUE. With “gapes” is meant that for example SRDO1 and SRDO3 are created in the OD, but not SRDO2. In this case, the number of SRDO1 would be 0 and SRDO3 would get the serial number 1. A definite assignment of communication index and sequential number is then no longer possible. In order that the CANopen stack still can find the corresponding SRDO, the stack must implement a different search algorithm, which can lead to a higher running time. Therefore, please avoid such gaps in the object dictionary.


```
...
OBD_CREATE_SRDO_GFC_PARAM()

OBD_CREATE_SRDO_COMMU(0x1301, 0, 0, 0, 0, 0x101, 0x102)
OBD_CREATE_SRDO_COMMU(0x1302, 1, 0, 0, 0, 0x103, 0x104)

OBD_BEGIN_SRDO_MAPP(0x1381, 0, 6)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x01, 0, normal1, 0x20000108)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x02, 0, invert1, 0x21000108)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x03, 0, normal2, 0x20010110)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x04, 0, invert2, 0x21010110)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x05, 0, normal3, 0x20010210)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x06, 0, invert3, 0x21010210)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x07, 0, normal4, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x08, 0, invert4, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x09, 0, normal5, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x0A, 0, invert5, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x0B, 0, normal6, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x0C, 0, invert6, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x0D, 0, normal7, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x0E, 0, invert7, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x0F, 0, normal8, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1381, 0x10, 0, invert8, 0x00000000)
OBD_END_SRDO_MAPP(0x1381)

OBD_BEGIN_SRDO_MAPP(0x1382, 1, 0)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x01, 1, normal1, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x02, 1, invert1, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x03, 1, normal2, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x04, 1, invert2, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x05, 1, normal3, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x06, 1, invert3, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x07, 1, normal4, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x08, 1, invert4, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x09, 1, normal5, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x0A, 1, invert5, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x0B, 1, normal6, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x0C, 1, invert6, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x0D, 1, normal7, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x0E, 1, invert7, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x0F, 1, normal8, 0x00000000)
  OBD_SUBINDEX_SRDO_MAPP(0x1382, 0x10, 1, invert8, 0x00000000)
OBD_END_SRDO_MAPP(0x1382)

OBD_CREATE_SRDO_CFG_VALID ()

OBD_BEGIN_SRDO_CRC(SRDO_MAX_SRDO_IN_OBD)
  OBD_SUBINDEX_SRDO_CRC(1, crc_SRDO_1, 0)
  OBD_SUBINDEX_SRDO_CRC(2, crc_SRDO_2, 0)
OBD_END_SRDO_CRC()

...
```

communication index

mapping index

sequential number

Figure 10: Example of an OD with 2 SRDOs

2.11 Function descriptions of the SRDO module

2.11.1 Function SrdoInit

Syntax:

```
#include <srdo.h>
tCopKernel PUBLIC SrdoInit  MCO_DECL_PTR_INSTANCE_PTR_
(
                            tSrdoInitParam MEM* pInitParam_p);
```

Parameter:

MCO_DECL_PTR_INSTANCE_PTR_: Pointer to the instance pointer

pInitParam_p: Pointer to the parameter structure for initializing the SRDO module instance

Return:

kCopSuccessful The function was executed without error.

kCopSrdoGranularityMismatch The configured SRDO granularity is not supported.

Further return codes of the standard CANopen are not possible.

Description:

The function deletes the instance table and initializes the first instance by using function **SrdoAddInstance()**. The parameter structure **tSrdoInitParam** contains the parameters for initializing the instance and is setup as follows:

```
typedef struct
{

#if (COP_MAX_INSTANCES > 1)

    void MEM*           m_ObdInstance;
    void MEM*           m_CobInstance;
    void MEM*           m_CdrvInstance;

#endif

    tSrdoTabParam       m_SrdoTabParam;
    BYTE                m_bGranularity;
    BYTE MEM*           m_pbSrdoConfigValid;

} tSrdoInitParam;
```


2.11.2 Function SrdoAddInstance

Syntax:

```
#include <srdo.h>
tCopKernel PUBLIC  SrdoAddInstance(
                                        MCO_DECL_PTR_INSTANCE_PTR_
                                        tSrdoInitParam MEM* pInitParam_p);
```

Parameter:

MCO_DECL_PTR_INSTANCE_PTR_: Pointer to the instance pointer

pInitParam_p: Pointer to the parameter structure for initializing the SRDO module instance

Return:

kCopSuccessful The function was executed without error.

kCopSrdoGranularityMismatch The configured SRDO granularity is not supported.

Further return codes of the standard CANopen are possible.

Description:

This function adds a new instance to the SRDO module. Therefore, define COP_MAX_INSTANCES must be larger than 1. If there is no free entry available in the instance table, the functions sends back en error. The SRDO tables for this instance are initialized.
Chapter 2.11.1. contains the setup of the parameter structure **tSrdoInitParam**.

2.11.3 Function SrdoDeleteInstance

Syntax:

```
#include <srdo.h>
tCopKernel PUBLIC SrdoDeleteInstance (
                                    MCO_DECL_INSTANCE_PTR);
```

Parameter:

MCO_DECL_INSTANCE_PTR: Pointer to the instance

Return:

kCopSuccessful The function was executed without error.

Further return codes of the standard CANopen are possible.

Description:

This function deletes all generated communication objects of the stated instance and marks it as unused.

2.11.4 Function SrdoNmtEvent

Syntax:

```
#include <srdo.h>
tCopKernel                PUBLIC  MCO_DECL_INSTANCE_PTR_
SrdoNmtEvent (
                                  tNmtEvent NmtEvent_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_: Pointer to the instance

NmtEvent_p: a NMT event that occurred *(see L-1020)*

Return:

kCopSuccessful The function was executed without error.

Further return codes of the standard CANopen are possible.

Description:

The function processes a NMT event which was triggered via the NMT State Machine. An event induces a change of the NMT node status.For each node status, the execution of the SRDO module is controlled.

2.11.5 Function SrdoSend

Syntax:

```
#include <srdo.h>
tCopKernel          PUBLIC  MCO_DECL_INSTANCE_PTR_
SrdoSend (
                            WORD wSrdoCommuIndex_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_:	Pointer to the instance
wSrdoCommuIndex_p:	Object index of the communication parameters of the SRDO in the object directory

Return:

kCopSuccessful	The function was executed without error.
kCopSrdoNmtError	The action is not allowed in this NMT state.
kCopSrdoInvalidCfg	The action was tried with a faulty SRDO configuration.
kCopSrdoNotExist	The SRDO chosen does not exist.
kCopSrdoRxTxConflict	It was tried to send a receive SRDO.
kCopSrdoInvalid	The action was tried with a switched off SRDO.

More return codes of the standard CANopen are possible.

Description:

The function sends one SRDO that is stated via communication index or all SRDOs when 0x0000 is stated as communication index.

See also the related API Function CcmSendSrdo.

2.11.6 Function SrdoProcess

Syntax:

#include <srdo.h>
tCopKernel PUBLIC MCO_DECL_INSTANCE_PTR)
SrdoProcess (

Parameter:

MCO_DECL_INSTANCE_PTR:	Pointer to the instance

Return:

kCopSuccessful	The function was executed without error.
kCopSrdoNotHandledInApp	The SRDO error reported to the application was not processed correctly

Further return codes of the standard CANopen are possible.

Description:

The function is called instead of function **CobProcessReceiveQueue()**. It works on receiving CAN messages from the CANopen stack. Receive SRDOs is given a preferential treatment. In addition this function checks the SCT of all receiving SRDOs. If the SCT is expired, but received none of the two CAN messages of the SRDOs, then the function AppSrdoError () is called with the error code kCopSrdoSctTimeout. Is the constant SRDO_CHECK_SRVT_BEFORE_1STRX set to TRUE, this function checks the SRVT of all SRDOs. If only one of the two CAN messages was received and the SRVT has expired, then the function AppSrdoError () is called with the error code kCopSrdoSrvtTimeout.

This function is called cyclically. Variations in terms of the timing of the SRDOs depend on this function.

The function SrdoProcess () is called automatically by CcmProcess() from CcmMain.c once the SRDO is enabled in CCM_MODUL_INTEGRATION.

2.11.7 Function SrdoCheckConfig

Syntax:

```
#include <srdo.h>
tCopKernel                    PUBLIC  MCO_DECL_INSTANCE_PTR_
SrdoCheckConfig (
                                      WORD * pwCommuIndex_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_: Pointer to the instance

pwCommuIndex_p: Pointer to a variable in which the function stores the communication index of the faulty SRDO in case of faulty configuration

Return:

kCopSuccessful The function was executed without error.

kCopSrdoCfgCrcError The SRDO configuration is faulty (CRC).

Description:

The function calculates the check sum over all SRDO (also deactivated SRDOs with Direction = 0) and compares them to the check sum that is configured in the OD. If it detects an error, it sends back an error and the corresponding communication index of the faulty SRDO. The function is called by the SRDO module upon writing the entry *Configuration Valid* (Index 0x13FE in the object directory) with value 0xA5.

See also the related API function CcmCheckSrdoConfig().

2.11.8 Function SrdoSendGfc

Syntax:

```
#include <srdo.h>
tCopKernel              PUBLIC  MCO_DECL_INSTANCE_PTR)
SrdoSendGfc (
```

Parameter:

MCO_DECL_INSTANCE_PTR: Pointer to the instance

Return:

kCopSuccessful The function was executed without error.

Further return codes of the standard CANopen are possible.

Description:

The function sends a GFC message.

It will not be available if the configuration SRDO_USE_GFC is set to FALSE.

See also the related API function CcmSendGfc

2.11.9 Function SrdoGetState

Syntax:

```
#include <srdo.h>
tCopKernel              PUBLIC  MCO_DECL_INSTANCE_PTR_
SrdoGetState (
                                BYTE * pSrdoState_p,
                                WORD wSrdoCommuIndex_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_: Pointer to the instance

pSrdoState_p: Pointer to which the function copies the status

wSrdoCommuIndex_p: Object index that contains communication parameters of the SRDO in the object directory

Return:

kCopSuccessful The function was executed without error.

kCopSrdoNotExist The SRDO chosen does not exist.

Description:

The function reads the status of a SRDO. For setup and usage of the status see chapter 2.5.3. See associated API-function Function CcmGetSrdoParam.

2.11.10 Function SrdoSetState

Syntax:

```
#include <srdo.h>
tCopKernel                PUBLIC  MCO_DECL_INSTANCE_PTR_
SrdoSetState (
                                  BYTE SrdoState_p,
                                  WORD wSrdoCommuIndex_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_: Pointer to the instance

SrdoState_p: Status to be set

wSrdoCommuIndex_p: Object index that contains communication parameters of the SRDO in the object directory

Return:

kCopSuccessful The function was executed without error.

kCopSrdoNotExist The SRDO chosen does not exist.

Description:

The function writes the status of a SRDO. For setup and usage of the status see chapter 2.5.3.

2.11.11 Function SrdoGetCommuParam

Syntax:

```
#include <srdo.h>
tCopKernel                      PUBLIC
SrdoGetCommuParam (
                                        MCO_DECL_INSTANCE_PTR_

                                WORD wSrdoCommuIndex_p,
                                tSrdoCommuParam                     *
                                pSrdoCommuParam_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_: Pointer to the instance

wSrdoCommuIndex_p: Object index that contains communication parameters of the SRDO in the object directory

pSrdoCommuParam_p: Pointer to the structure in which the function copies the values for the Information Direction and SCT

Return:

kCopSuccessful The function was executed without error.

kCopSrdoNotExist The SRDO chosen does not exist.

Description:

The function reads the parameters of a SRDO that are necessary in the application. Those are *Information Direction* and *SCT*. See associated API-function
See associated API-Function CcmGetSrdoParam().

2.11.12 Function SrdoGetMappParam

Syntax:

```
#include <srdo.h>
tCopKernel PUBLIC SrdoGetMappParam  MCO_DECL_INSTANCE_PTR_
(
                                    WORD wSrdoCommuIndex_p,
                                    tSrdoMappParam * pSrdoMappParam_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_: Pointer to the instance

wSrdoCommuIndex_p: Object index that contains communication parameters of the SRDO in the object directory

pSrdoCommuParam_p: Pointer to the structure in which the function copies the values for the Number Of Mapped Objects and the Variable pointer

Return:

kCopSuccessful	The function was executed without error.
kCopSrdoNotExist	The SRDO chosen does not exist.

Description:

The function reads the mapping parameters of a SRDO that are necessary in the application. Thos are *Number of Mapped Objects* and the Pointers to the mapped variables. See associated API-Function CcmGetSrdoParam().

2.11.13 Funktion SrdoCalcSrdoCrc

Syntax:

```
#include <srdo.h>
tCopKernel PUBLIC SrdoCalcSrdoCrc (      MCO_DECL_INSTANCE_PTR_
                                         WORD wCommuIndex_p,
                                         tSrdoTabEntry MEM* pSrdoEntry_p,
                                         WORD* pwCrc_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_:	Pointer to instance
wCommuIndex_p:	Object index which contains the communication parameters of the SRDO in the object directory
pSrdoEntry_p:	Must always be passed with 0
pwCrc_p:	Pointer to a WORD variable for receiving 16 Bit CRC within the calling function.

Return:

kCopSuccessful	The function was executed without error.
kCopSrdoNotExist	The SRDO chosen does not exist.

Description:
The function calculates the checksum (CRC) (CRC) over a SRDO and returns it to the calling function. The calculation is done also when the SRDO is turned off. There is no comparison to the accuracy of the CRC.

The application can use this functionality via the API function CcmCalcSrdoCrc () to update the CRC of a SRDOs when the configuring of a SRDOs on the application must be set new dynamically (e.g. changing the COB-ID depending on the node ID).

2.12 Function descriptions of the SRDOSTC module

The following functions of the SRDO module are also implemented in the SRDOSTC module. Their meanings and syntax can be taken from 2.11:

> SrdoInit(), SrdoAddInstance(), SrdoDeleteInstance(), SrdoNmtEvent(), SrdoSend(), SrdoProcess(), SrdoCheckConfig(), SrdoSendGfc(), SrdoGetState(), SrdoSetState(), SrdoGetCommuParam().

2.12.1 Function SrdoStaticDefineVarFields

Syntax:

```
#include <srdo.h>
tCopKernel PUBLIC
SrdoStaticDefineVarFields(
                                        MCO_DECL_INSTANCE_PTR
                                        _
                                        WORD        wCommuIndex_p,
                                        void MEM* pNormalData_p,
                                        void MEM* pInversData_p);
```

Parameter:

MCO_DECL_INSTANCE_PTR_:	Pointer to the instance
wCommuIndex_p:	Communication index of the SRDO in the OD whose variables shall not be defined.
pNormalData_p:	Pointer to a coherent variable field that shall be linked (or mapped) to the normal data of the SRDO.
pInversData_p:	Pointer to a coherent variable field that shall be linked (or mapped) to the inverse data of the SRDO.

Return:

kCopSuccessful	The function was executed without error.
kCopSrdoNotExist	The SRDO chosen does not exist.
kCopSrdoErrorMapp	The mapping of a SRDO is faulty.
kCopSrdoLengtExceeded	The length of the SRDO chosen Mapping exceeds 64 Bit.

Description:

This function defines variable fields for a SRDO. The application only changes the variables via those variable fields. When sending a SRDO, those data byetes are copied from the variable field into the two CAN messages. When receiving a SRDO, the data bytes of the CAN messages are directly copied into the variable fields.
The function verifies, if the stated variable fields correspond with the variables to which the mapping points in the OD.

2.13 Extended CANopen Return codes

The CANopen Return codes are defined in file **errordef.h**.

Error code	Description
kCopSuccessful	The function was executed without error.
kCopSrdoNotExist	The SRDO chosen does not exist.
kCopSrdoGranularityMismatch	The configured SRDO granularity is not supported.
kCopSrdoCfgTimingError	The SRDO configuration is faulty (time configuration SCT – SRVT).
kCopSrdoCfgIdError	The SRDO configuration is faulty (COB-Ids).
kCopSrdoCfgCrcError	The SRDO configuration is faulty (CRC).
kCopSrdoNmtError	The action is not allowed in this NMT state.
kCopSrdoInvalidCfg	The action was tried with a faulty SRDO configuration.
kCopSrdoInvalid	The action was tried with a switched off SRDO.
kCopSrdoRxTxConflict	It was tried to send a receive SRDO.
kCopSrdoIllegalCanId	The CAN Identifier is not valid.
kCopSrdoCanIdAlreadyInUse	The CAN Identifier is already being used.
kCopSrdoNotInOrder	The two CAN meassages of a SRDO have been received in the wrong order.
kCopSrdoSctTimeout	The SCT of a receive SRDO was exceeded.
kCopSrdoSrvtTimeout	The SRVT of a receive SRDO was exceeded.
kCopSrdoCanIdNotValid	At least on of the two received CAN Identifier of a SRDO is faulty.
kCopSrdoDlcNotValid	At least on of the two received CAN message lenghts of the SRDO is faulty.
kCopSrdoErrorMapp	The mapping of a SRDO is faulty.
kCopSrdoDataError	The data of the CAN messages of a SRDO is not inverse.
kCopSrdoLengtExceeded	The length of the SRDO chosen Mapping exceeds 64 Bit.
kCopSrdoNotHandledInApp	The SRDO error reported to the application was not processed correctly

3 Reference environment TMDX570LS20SMDK

Texas Instruments provide the development board TMDX570LS20SMDK. It serves as a reference environment for our safety extension.

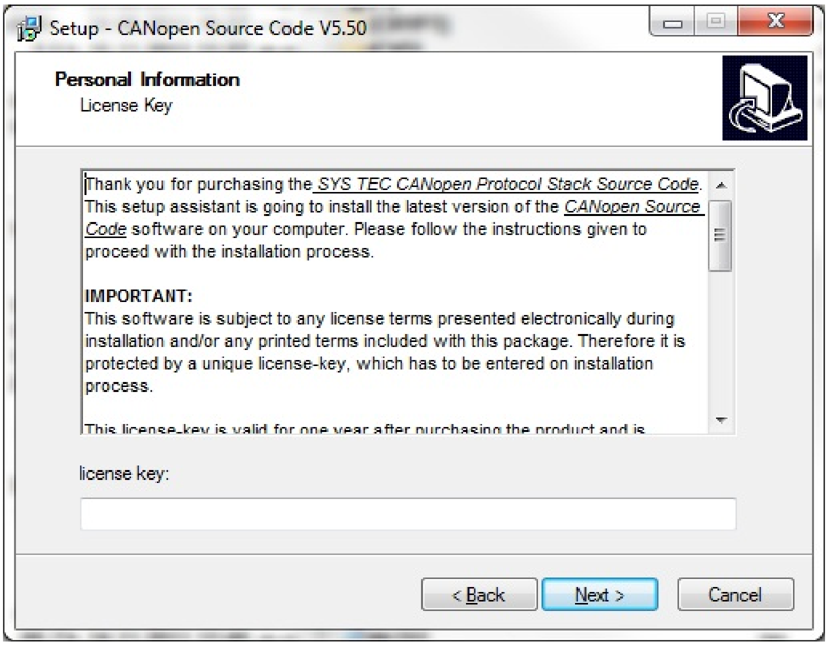
For the handling of projects in our extension, there are several things to consider. This chapter describes all these things to help you get started with the project and the hardware.

3.1 Installation of the development environment

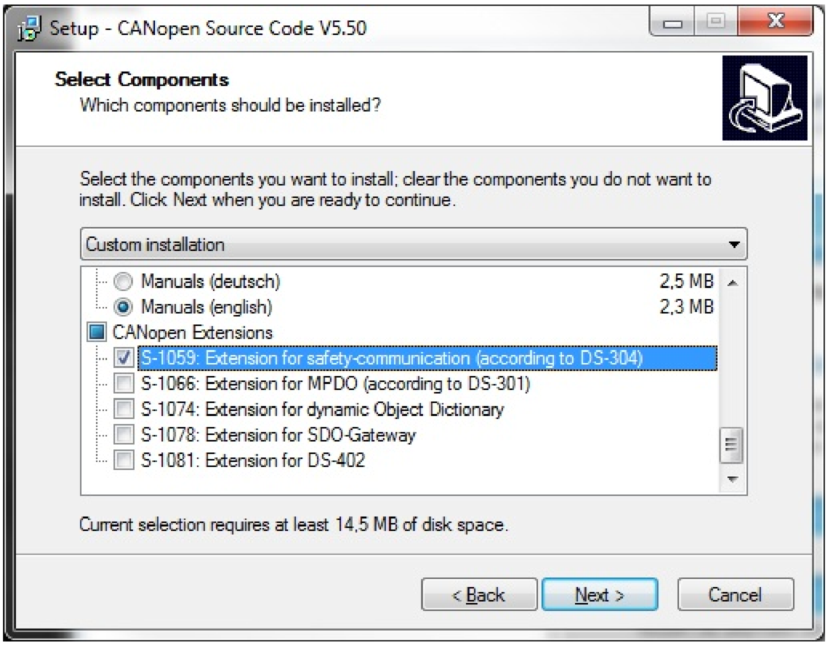
With the development kit TMDX570LS20SMDK you have received a CD with the Code Composer Studio development environment. The safety demo was created and tested with version V4.2.3. Install the development software of this CD and continue with the installation of the CANopen software.

3.2 Installation of the CANopen software

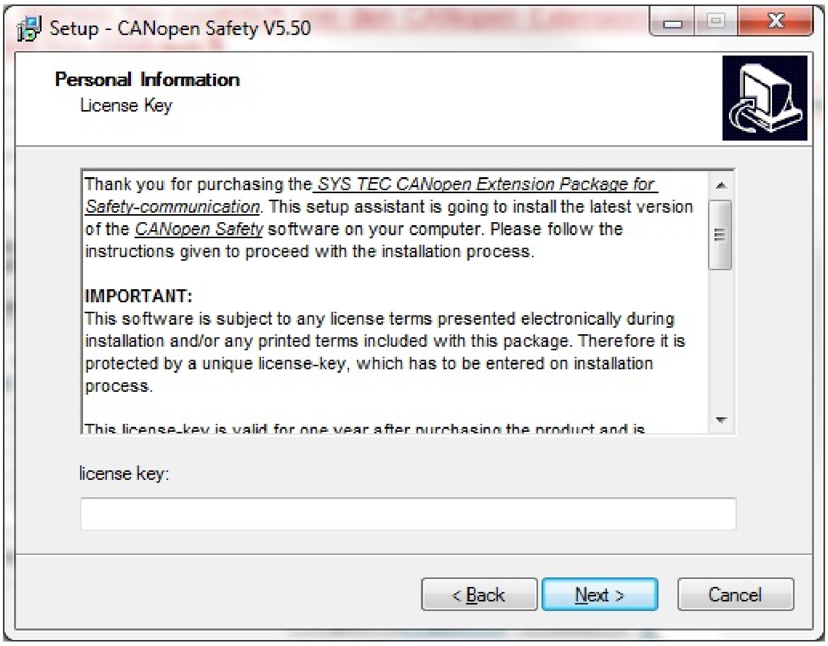
The CANopen stack SO 877 must be installed first. Start the installation from the SYS TEC electronic product CD autorun menu. The version of the CANopen stack must necessarily be greater than or equal to V5.51. In an earlier version the project for the TMS570LS does not exist. After the welcome screen, accepting the license agreement and enter the user information you will see the following dialog box for entering the license key of our CANopen stack:



Enter the purchased license key and press "Next". In the following dialog, select the demo projects. Also select the software package SO 1059 from the CANopen extensions.



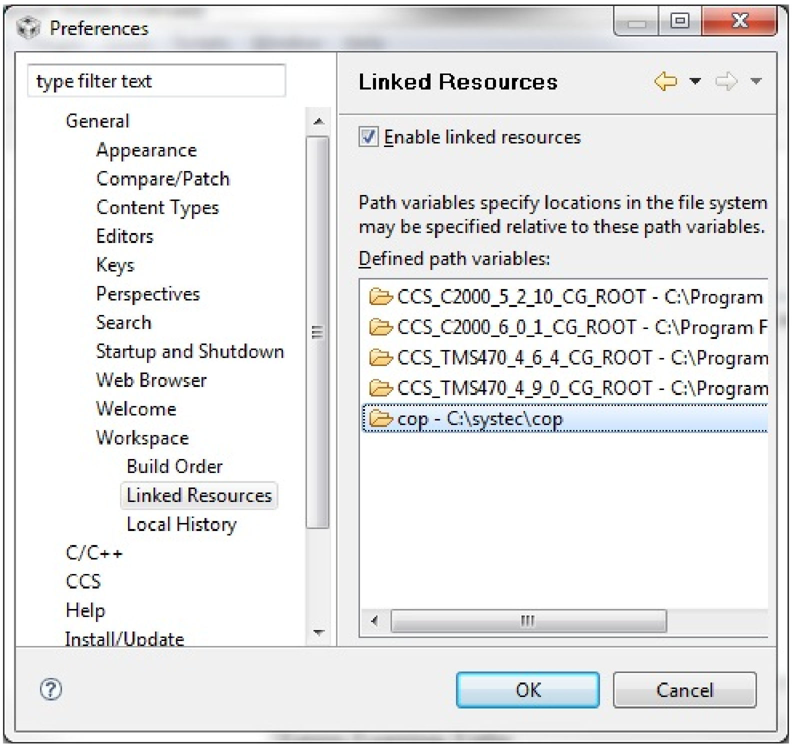
Follow any prompts in the setup. After installing the extension of SO 877 SO 1059 will automatically be installed. You need to enter another license key for SO 1059.



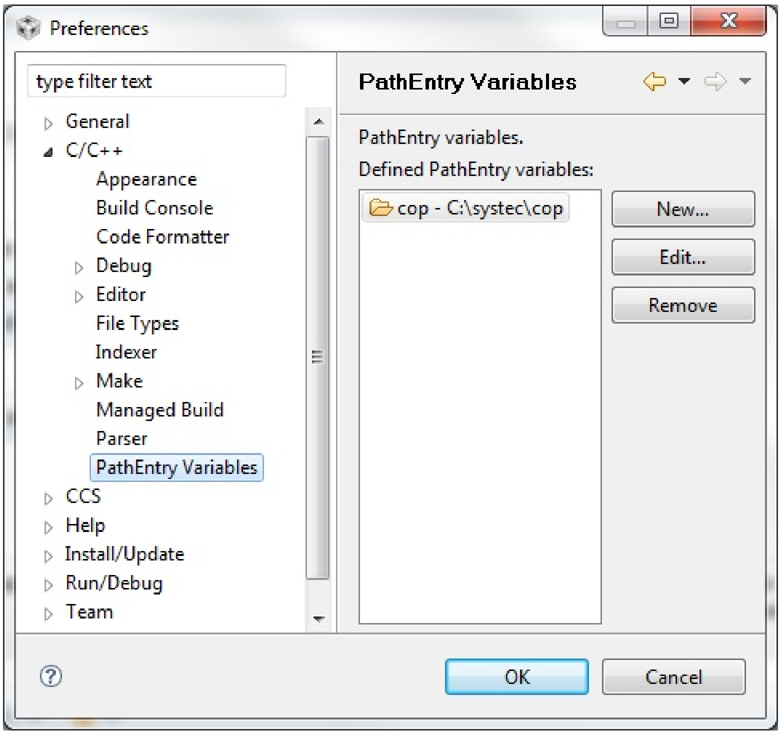
3.3 Import of the safety demo in Code Composer Studio

When the installation of the CANopen stack and the safety expansion is completed, you will find in *C:\systec\cop\target\TMDX570LS20SMDK\no_os\Code Composer Studio\demo_srdo_actor* the demo for the actuator on the TMS570LS Development Kit. Please make sure that the files .ccsproject, .cdtbuild, .cdtproject and. project are not set to "hidden" in the directory. Otherwise, the project can not be imported to the Code Composer Studio. Please remove the attribute "hidden" when it should be set. Now start the Code Composer Studio. You will be prompted to create a workspace. Close this dialog by entering a directory of your choice.

In Code Composer Studio call up the menu **Window** -> **Preferences**. Expand the menu on the left part of the window, click **General** -> **Workspace** -> **Linked Resources**. In the right window use the **New** button to create a new entry: name "cop" and location "C:\systec\cop". Please pay attention to the case-sensitive. At the end the dialog should look at as follows:

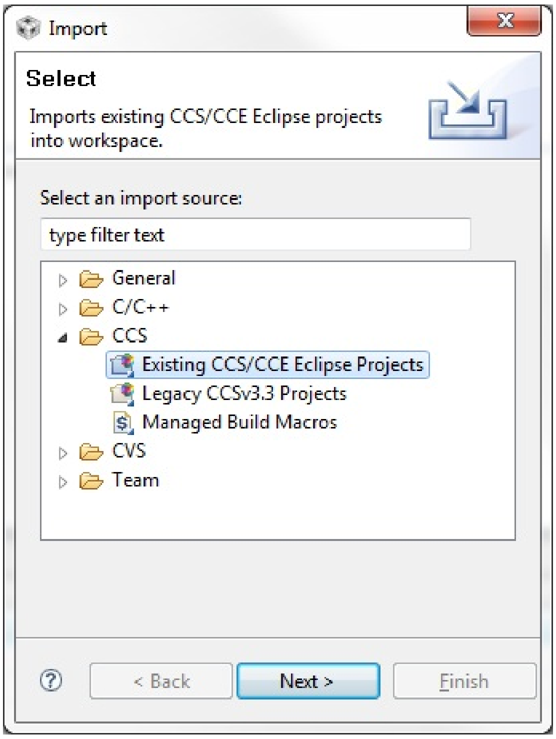


In the left part of the window change to **C/C++ → PathEntry Variables**. Add tehere also a new entry with the button **New…** and named it **cop** and add it to ***C:\systec\cop***.

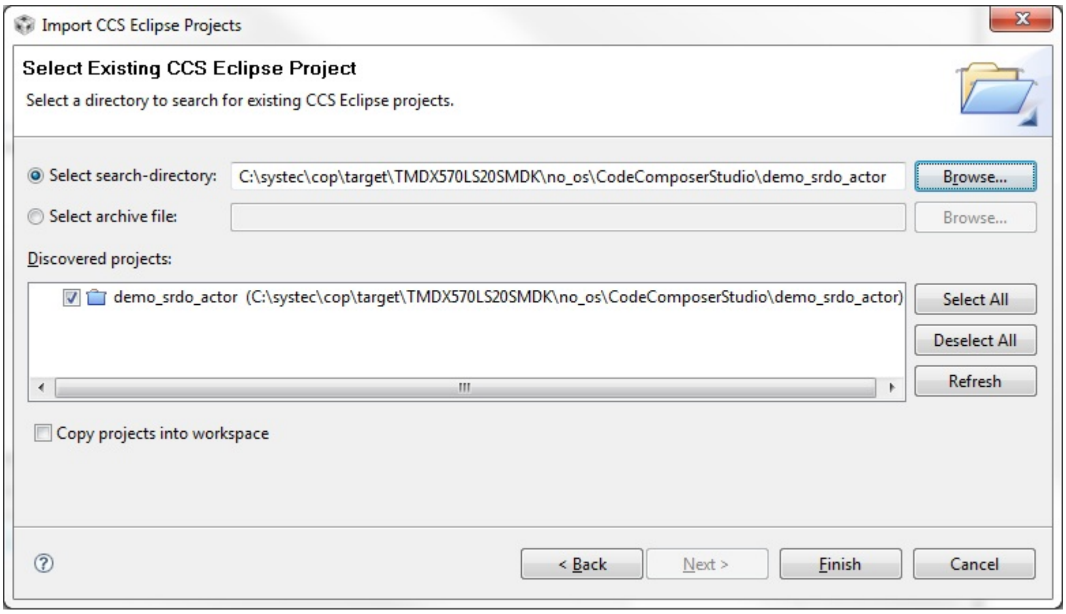


Confirm your entry with OK.

Now import the project from the menu **File → Import…** . Select in the following dialog under ***CCS*** the line **Existing CCS/CCE Eclipse Projects** and confirm with **Next**.



In the following dialog select over **Browse** the path to the demo and then click **Finish.**



If all these steps have been carried out without problems, the project can be (re-) created.

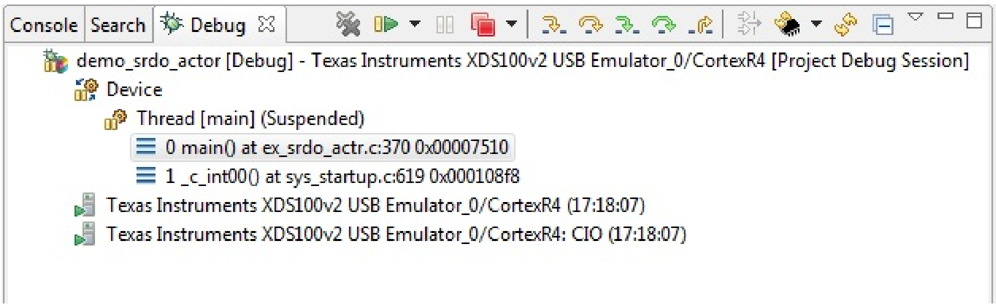
3.4 Debugging the Demo on the hardware

Now connect the TMS570 Development Kit to the PC. Just use the included USB cable and plug it on the top board into the USB mini jack labeled **XDS100V2**. Now Windows search for the device drivers of the Development Kit. These device drivers were installed with the installation of the Texas Instruments CD.

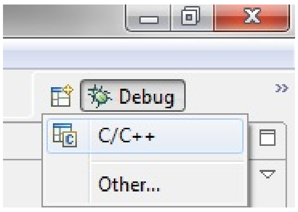
After installing the device driver, you can connect the power supply to the Development Kit. In the scope of delivery of the Development Kit is a 12V power supply included. Connect it to the jack on the top board next to the USB mini-jack.

With right-click on the project on the left side in the window (in Code Composer Studio) you can now choose from the context menu **Debug As → Debug Session**. On the very first time you must select the type of CPU, choose **TMS570LS20216SZWT.** After confirming the Code Composer programs the demo into the flash of the microcontroller and stops in the main() function.

In the Debug window, you can now control the program execution with the symbols.



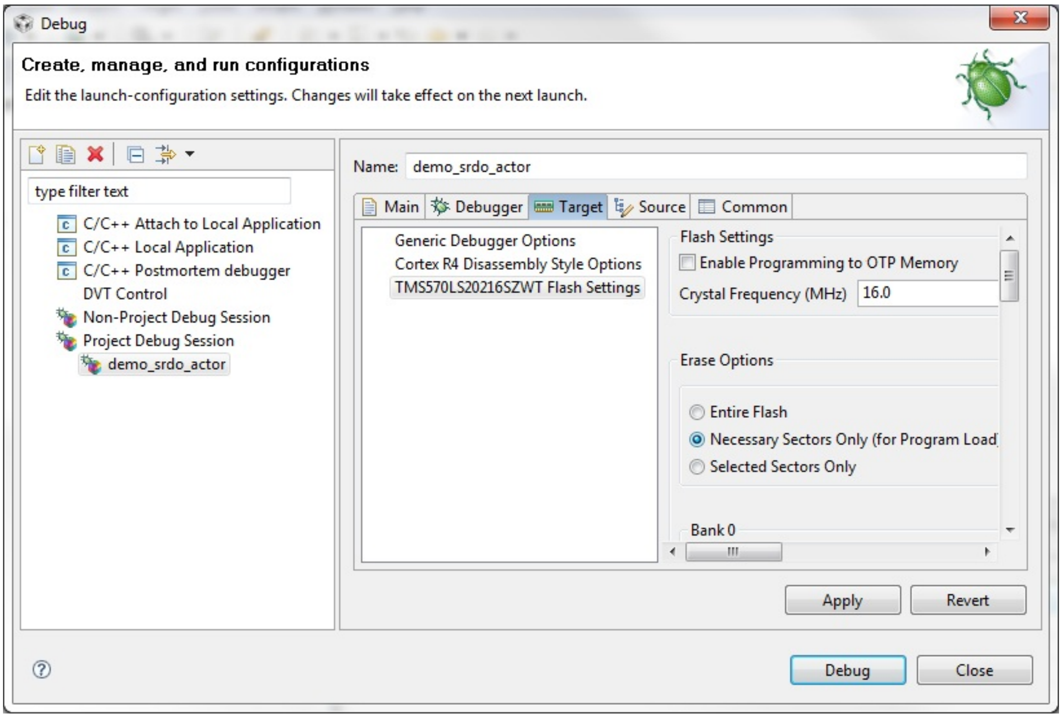
Do you want to stop debugging, then simply change the perspective back to **C/C++**. Click in the upper right part of the Code Composer Studio on the icon next to **debug**. The following Contex Menu **C/C++** will be offered.



Now you are back to the Project Explorer of the Code Composer Studio.

The programming of the firmware in the flash takes a long time, first the entire flash is erased. Therefore, you should change the debug options so that only the flash sectors will be erased, which are used by the application.

Click with the right mouse button on the project and select the context menu **Debug As -> Debug.** In the next dialog switch on the right side of the window, click the tab sheet **Target**. Select the line **TMS570LS20216SZWT Flash Settings**. Now you can find the **Erase Options** on the right side. Selct ***Necessary Sectors Only*** and press ***Apply***.


Debug
Create, manage, and run configurations
Edit the launch-configuration settings. Changes will take effect on the next launch.
type filter text
C/C++ Attach to Local Application
C/C++ Local Application
C/C++ Postmortem debugger
DVT Control
Non-Project Debug Session
Project Debug Session
demo_srdo_actor
Name: demo_srdo_actor
Main
Debugger
Target
Source
Common
Generic Debugger Options
Cortex R4 Disassembly Style Options
TMS570LS20216SZWT Flash Settings
Flash Settings
Enable Programming to OTP Memory
Crystal Frequency (MHz) 16.0
Erase Options
Entire Flash
Necessary Sectors Only (for Program Load)
Selected Sectors Only
Bank 0
Apply
Revert
Debug
Close

Index

Document:	CiA 304 Safety Framework
Document number:	L-1077e_4, Edition October 2012

How would you improve this manual?

Have you found any mistake in this manual? Page

Sent in by:

Customer number:

Name:

Company:

Address:

Please submit to: SYS TEC electronic GmbH
Am Windrad 2
D-08468 Heinsdorfergrund
GERMANY
Fax : +49 (0) 3765 / 38600-4100

Published by

Best.-Nr. L-1077e_5
Printed in Germany