



AUTOMATIONSTECHNIK

Unternehmen der ZUNDEL Holding

CP1131

Function Blocks

Version 1.30
User Handbook

A u t o m a t i o n S y s t e m

CANtrol® //

Copyright © BERGHOF Automationstechnik GmbH

Reproduction and duplication of this document and utilisation and communication of its content is prohibited, unless with our express permission.
All rights reserved. Damages will be payable in case of infringement.

Disclaimer

The content of this publication was checked for compliance with the hardware and software described. However, discrepancies may arise, therefore no liability is assumed regarding complete compliance. The information in this document will be checked regularly and all necessary corrections will be included in subsequent editions. Suggestions for improvements are always welcome.

Subject to technical changes.

Trademark

CANtrol® // is a registered trademark of BERGHOF Automationstechnik GmbH

General Information on this Manual

Content:

This manual describes the CP1131 function blocks. The product-related information contained herein was up to date at the time of publication of this manual.

Completeness:

This manual is complete only in conjunction with the user manual entitled

'Introduction
to CANtrol Automation System'

and the product-related hardware or software user manuals required for the particular application.

Standards:

The CANtrol automation system, its components and its use are based on International Standard IEC 61131 Parts 1 to 4 (EN 61131 Parts 1 to 3 and Supplementary Sheet 1).
Supplementary Sheet 1 of EN 61131 (IEC 61131-4) entitled 'User Guidelines' is of particular importance for the user.

Order numbers:

Please see the relevant product overview in the 'Introduction to CANtrol Automation System' manual for a list of available products and their order numbers.

Ident. No.: 2801820

You can reach us at:

BERGHOF Automationstechnik GmbH

Harretstr. 1

72800 Eningen / Germany

Phone: +49 7121 / 894-0

Telefax: +49 7121 / 894-100

e-mail: info@berghof-automation.de

www.berghof-automation.de

BERGHOF Automationstechnik GmbH works in accordance with DIN EN ISO 9001

Update

Version	Date	Subject
1.00	25.09.97	
1.01	25.09.97	Additions to QAIO.LIB
1.02	29.09.97	Additions to COMASTER.LIB
1.03	11.02.98	Correction of errors in QAIO.LIB
1.04	20.03.98	Insertion of section String conversion
1.05	02.04.98	Additions to sections: Receiving data, Sending data, Receiving CAN messages, Transmitting CAN messages, CANopen Node Manager and SDO Manager
1.06	27.07.98	Errors corrected in section: Receiving data
1.07	12.08.98	Addition of Installing SVC to section: Serial interfaces (firmware version 1.11 and higher)
1.08	05.11.98	Errors corrected in sections: Receiving CAN messages and CANopen Port Init
1.09	22.04.99	Additions to section: Introduction (from V2.00)
1.10	17.05.99	Addition to section: TCP/UDP protocol driver via IP
1.11	18.05.99	Addition to section: Cycle time (SYSTIME.LIB)
1.12	08.06.99	Addition to sections: Initialisation, CANopen Port Init, new CANopen Port Re-start
1.20	17.11.00	Manual completely revised. New section: TPU blocks
1.30	21.04.06	Chapter, CP1131 Function Blocks: Update and separation into filial documents. Addition of new sections: CP1131 Multitasking, System Information, Application ID, CAN Interfaces and Analog Library for Dialog Controllers. New safety notes.

Blank page

Contents

1.	GENERAL INSTRUCTIONS.....	11
1.1.	Hazard Categories and Indications	11
1.2.	Qualified users	11
1.3.	Use as Prescribed	12
2.	SERIAL INTERFACES (SIO.LIB).....	13
2.1.	Introduction	13
2.2.	IEC61131 User Interface	15
2.2.1.	Initialisation (SIO_INIT).....	15
2.2.2.	Receiving data (SIO_RECEIVE).....	16
2.2.3.	Transmitting data (SIO_TRANSMIT).....	17
2.2.4.	State request (SIO_STATE)	18
2.2.5.	Closing an interface (SIO_CLOSE)	19
2.2.6.	Initialising the SVC protocol on the cover (SIO_INITSVC).....	20
2.2.7.	Ending the SVC protocol (SIO_CLOSESVC).....	20
3.	CAN INTERFACES (ECAN.LIB)	21
3.1.	ECAN_V01.LIB.....	21
3.1.1.	General Information	21
3.1.2.	Brief Overview of the Most Important ECAN Function Blocks.....	22
3.2.	IEC61131 User Interface	24
3.2.1.	Initialization (ECAN_INIT).....	24
3.2.2.	Starting/Ending CAN Messages (ECAN_SET_SWFILTER)	27
3.2.3.	Hardware Acceptance Filter (ECAN_SET_HWFILTER)	28
3.2.4.	Receiving CAN Messages (ECAN_RX)	29
3.2.5.	Sending CAN Messages (ECAN_TX)	31
3.2.6.	Status Query (ECAN_STATE).....	33
3.2.7.	Starting/Ending a Gateway (ECAN_BRIDGE).....	34
3.2.8.	Querying the Own Node ID (ECAN_GET_NODEID).....	35
3.2.9.	Setting Error Limits (ECAN_SET_EWL)	36
3.2.10.	Reading the Error Register (ECAN_HWSTATE)	37
3.2.11.	Reading the Expanded Controller Status (ECAN_EXTSTATE)	38
3.2.12.	Setting the Listen Only Mode (ECAN_SET_LOM)	39
3.2.13.	Setting the Self Test Mode (ECAN_SET_STM)	40
4.	ANALOG INPUT/OUTPUT (QAIO.LIB).....	41
4.1.	Introduction	41
4.2.	IEC61131 User Interface	42
4.2.1.	Reading analog input values (QAIO_ANALOGIN)	42
4.2.2.	Writing analog output values (QAIO_ANALOGOUT)	43

5.	CANOPEN MASTER (COMASTER.LIB)	45
5.1.	COMASTER_V01.LIB	45
5.1.1.	Introduction to CANopen Master Library.....	45
5.1.2.	Initialising the CANopen interface.....	46
5.1.3.	Initialising the physical interface (COM_PORT_INIT).....	47
5.1.4.	Restarting the physical interface (COM_PORT_RESTART).....	48
5.1.5.	Initialising the CANopen master (COM_INIT).....	49
5.2.	Node Manager	50
5.2.1.	Logging on new CANopen slave nodes (COM_ADD_NODE).....	50
5.2.2.	Changing state of a CANopen slave (COM_CHANGE_STATE).....	51
5.2.3.	Getting state of a CANopen slave (COM_GET_State).....	52
5.2.4.	Activating node guarding (COM_GUARDING_ON).....	53
5.2.5.	Deactivating node guarding (COM_GUARDING_OFF).....	53
5.2.6.	Checking node guarding (COM_GUARDING_MSG).....	54
5.3.	SDO Manager	55
5.3.1.	Installing SDO (COM_INST_SDO).....	55
5.3.2.	Transmitting up to 4 data bytes by SDO (COM_SEND_SDO_EDATA).....	56
5.3.3.	Transmitting up to 256 data bytes by SDO (COM_SEND_SDO_DOMAIN).....	58
5.3.4.	Receiving up to 4 data bytes by SDO (COM_RECEIVE_SDO_EDATA).....	60
5.3.5.	Receiving up to 256 data bytes by SDO (COM_RECEIVE_SDO_DOMAIN).....	62
5.4.	PDO Manager	64
5.4.1.	Installing PDO (COM_INST_PDO).....	64
5.4.2.	Transmitting PDO telegram (COM_SEND_PDO).....	65
5.4.3.	Requesting PDO telegram (COM_REQUEST_PDO).....	66
5.4.4.	Receiving PDO telegram (COM_RECEIVE_PDO).....	67
5.4.5.	Updating the data of a synchronous transmit PDO (COM_UPDATE_PDO).....	68
5.4.6.	Configuring a synchronous transmit PDO (COM_CFG_SYNC_PDO).....	69
5.5.	SYNC Manager	70
5.5.1.	Installing sync telegram (COM_INST_SYNC).....	70
5.5.2.	Starting sync telegram (COM_START_SYNC).....	71
5.5.3.	Stopping sync telegram (COM_STOP_SYNC).....	71
5.6.	Error Codes	72
5.6.1.	CANopen error codes.....	72
5.6.2.	Internal CAN driver errors xxxx = internal error code.....	73
5.6.3.	Internal error xxxx = VRTXsa error code.....	73
5.6.4.	Error handling.....	74
6.	STRING CONVERSION (STRCNV.LIB)	75
6.1.1.	Converting DINT to STRING (CNV_DINT_TO_STR).....	75
6.1.2.	Converting REAL to STRING (CNV_REAL_TO_STR).....	77
6.1.3.	Converting STRING to DINT (CNV_STR_TO_DINT).....	78
6.1.4.	Converting STRING to REAL (CNV_STR_TO_REAL).....	78
6.1.5.	Converting STRING to array (CNV_STR_TO_ARRAY).....	79
6.1.6.	Converting array to STRING (CNV_ARRAY_TO_STR).....	79
6.1.7.	Converting real string to decimal number(CNV_REALSTR_TO_DINT).....	80

7.	TCP/UDP PROTOCOL DRIVER VIA IP	81
7.1.	TCP_UDP_V01.LIB	81
7.1.1.	Introduction	81
7.1.2.	Transmission Control Protocol (TCP).....	82
7.1.3.	Internet Protocol (IP).....	83
7.1.4.	Ports used.....	83
7.1.5.	System restrictions.....	84
7.1.6.	General aspects.....	84
7.2.	UDP/IP Function Blocks	85
7.2.1.	Clearing the interface's buffer memory (IP_UDP_CLEAR_BUFFER).....	85
7.2.2.	Setting up a UDP interface (IP_UDP_CREATE_CONNECTOR).....	86
7.2.3.	Closing the interface (IP_UDP_DELETE_CONNECTOR)	87
7.2.4.	Reading a data block from an interface (IP_UDP_READ)	88
7.2.5.	State of an interface (IP_UDP_STATE)	89
7.2.6.	Writing data (IP_UDP_WRITE).....	90
7.3.	TCP/IP Function Blocks - Server functions	91
7.3.1.	Confirming a new client connection (IP_TCP_CONNECT_NEW_CLIENT).....	91
7.3.2.	Setting up a server (IP_TCP_REGISTER_SERVER)	92
7.3.3.	Determining the state of a TCP server (IP_TCP_SERVERSTATE).....	93
7.3.4.	Closing a TCP server (IP_TCP_UNREGISTER_SERVER).....	94
7.4.	TCP/IP Function Blocks - Client Functions	95
7.4.1.	Establishing a connection to a server (IP_TCP_CONNECT_TO_SERVER).....	95
7.5.	TCP/IP Function Blocks - Connection Functions	96
7.5.1.	Clearing down a connection (IP_TCP_DISCONNECT)	96
7.5.2.	Timeout Control During Line Disconnection (IP_TCP_DISCONNECT_TIMEOUT).....	97
7.5.3.	Reading data (IP_TCP_READ)	98
7.5.4.	Determining state of a connection (IP_TCP_STATE)	99
7.5.5.	Transmitting data (IP_TCP_WRITE)	100
7.6.	IP Function Blocks	101
7.6.1.	Reading own IP address (IP_OWN_NUMBER)	101
7.6.2.	Reading own Ethernet MAC address (IP_OWN_MAC_ADR).....	101
7.6.3.	Warning codes.....	102
7.6.4.	Error codes	102
8.	CYCLE TIME (SYSTIME.LIB).....	103
8.1.	SYSTIME_V01.LIB	103
8.1.1.	Introduction	103
8.2.	IEC61131 User Interface	104
8.2.1.	Setting up the cycle monitoring time (PLC_SetMaxCycleTime).....	104
8.2.2.	Retriggering cycle monitoring (PLC_RetriggerCycleTime).....	104
8.2.3.	Enabling calculation of cycle statistics (PLC_EnableCycleStatistics)	105
8.2.4.	Disabling calculation of cycle statistics (PLC_DisableCycleStatistics).....	105
8.2.5.	Resetting cycle statistics (PLC_ResetCycleStatistics)	105
8.2.6.	Reading cycle statistics (PLC_GetCycleStatistics).....	106

8.2.7.	Reading the system time (SYS_GETTIMER)	107
9.	TPU BLOCKS FOR RAPID INPUTS/OUTPUTS	109
9.1.	Discrete I/O programming of TPU (TPUDIO.LIB).....	109
9.1.1.	Initialising a TPU channel as digital input or output (DIO_INIT)	109
9.1.2.	Ending discrete programming of inputs or outputs (DIO_CLOSE)	110
9.1.3.	Setting an output (DIO_SET)	111
9.1.4.	Reading an input (DIO_GET).....	111
9.2.	PWM Generation (TPUPWM.LIB)	112
9.2.1.	Setting up a PWM generator (PWM_INIT).....	112
9.2.2.	Changing a PWM parameter (PWM_CHANGE).....	113
9.2.3.	Ending PWM generator (PWM_CLOSE)	114
9.3.	Counters (COUNTER.LIB).....	115
9.3.1.	Setting up a counter (COUNTER_INIT).....	115
9.3.2.	Setting a counter (COUNTER_SET).....	116
9.3.3.	Reading a counter (COUNTER_GET)	116
9.4.	Quadrature Encoder (POSREL.LIB).....	117
9.4.1.	Selecting signal source for quadrature encoder (POSITION_SOURCE_SELECT).....	117
9.4.2.	Setting a quadrature encoder value (POSITION_REL SET)	118
9.4.3.	Reading a quadrature encoder value (POSITION_REL_GET)	118
10.	ANALOG LIBRARY FOR DIALOG CONTROLLERS (ANALOGIN.LIB)	119
10.1.	AnalogIn_V01.LIB	119
10.1.1.	Introduction.....	119
10.1.2.	Reading the Analog Value (AnalogIn).....	119
11.	SYSTEM INFORMATION (SYSINFO.LIB / SYSINFO-CP1131.LIB)	121
11.1.	SysInfo_V02.LIB and SysInfo-CP1131_V01.LIB	121
11.1.1.	Introduction.....	121
11.1.2.	Module Type Query (SYSINFO_GET_CPUTYPE).....	122
11.1.3.	Firmware Version Query (SYSINFO_GET_FWVERSION)	123
11.1.4.	Library Data Query (SYSINFO_GET_VERSION).....	124
11.1.5.	Querying Information from E-Bus Modules (SYSINFO_GET_QBUSMODTYPE).....	125
12.	APPLICATION ID (SVCAPPID.LIB)	127
12.1.1.	Reading the Application ID (SVC_GET_APPID)	127
12.1.2.	Writing the Application ID (SVC_SET_APPID).....	127
13.	CP1131 MULTITASKING (TASKMANAGER.LIB)	129
13.1.	Task Configuration.....	130
13.2.	The Library Functions.....	133
13.2.1.	Internal Function (CALLHOOK)	133
13.2.2.	Application Program Reaction Time (PLC_GetMaxTaskDelay)	133
13.2.3.	Recalling the Task Scheduler (PLC_RESCHEDULE)	134

13.2.4. Task Identification (PLC_GetTaskID)	135
13.2.5. Task Information (PLC_GetTaskInfo)	136
13.2.6. Blocking a Task (PLC_SuspendTask)	137
13.2.7. Releasing a Task (PLC_ResumeTask)	137
13.2.8. Stopping the Cell Controller (PLC_STOP).....	138
13.2.9. Restarting the Cell Controller (PLC_REBOOT).....	138
14. SAVING APPLICATION DATA TO THE FLASH MEMORY (FOD_VXX.LIB)	139
14.1. FoD Application Area.....	139
14.1.1. When Can FoD Not be Used?	139
14.2. FoD Functional Overview	140
14.3. FoD Data Structure.....	140
14.4. FoD File Structure	141
14.5. Activating FoD.....	142
14.6. FoD Data Configuration (User Flash Configuration)	143
14.6.1. Creating an FoD Data Configuration	145
14.6.2. Editing an FoD Data Configuration.....	146
14.6.3. Importing FoD Data.....	147
14.6.4. Downloading/Uploading FoD Data with CP1131	148
14.6.5. Downloading/Uploading FoD Data with CNW	150
14.7. The FoD Library (FOD_Vxx.LIB)	151
14.7.1. FOD_SETUP	151
14.7.2. FOD_READ	152
14.7.3. FOD_READ_INFO.....	153
14.7.4. FOD_GET_MAXINDEX	154
14.7.5. FOD_WRITE.....	155
14.7.6. FOD_PREPARE_WRITE.....	156
14.7.7. FOD_RESET	157
14.7.8. FOD_DELETE	158
15. NOTES.....	159
15.1. Data interchange via addresses (DATAPTR) and markers.....	159
15.1.1. Transmitting data with an ARRAY at the CAN interface.....	159
15.1.2. Transmitting data with an address pointer at the CAN interface	160
15.1.3. Transmitting data with markers at the CAN interface	161
15.2. CAN interfaces (CAN.LIB)	162
15.2.1. Introduction	162
15.3. IEC61131 User Interface	164
15.3.1. Initialisation (CAN_INIT)	164
15.3.2. Logging CAN messages on and off (CAN_DEFINE_COBID)	166
15.3.3. Receiving CAN messages (CAN_RECEIVE)	167
15.3.4. Transmitting CAN messages (CAN_TRANSMIT)	169
15.3.5. State request (CAN_STATE).....	171
15.3.6. Logging a gateway on and off (CAN_DEFINE_GATEWAY)	172

15.3.7. Requesting own node ID.....	173
15.4. Quadrature Encoder (INCR.LIB).....	174
15.4.1. Introduction.....	174
15.5. IEC61131 User Interface.....	174
15.5.1. Sampling the position value (POSITION_REL_GET).....	174
15.5.2. Setting the position value (POSITION_REL_SET).....	175
16. ANNEX.....	177
16.1. Environmental Protection.....	177
16.1.1. Emission.....	177
16.1.2. Disposal.....	177
16.2. Maintenance/Upkeep.....	177
16.3. Repairs/Service.....	177
16.3.1. Warranty.....	177
16.4. Nameplate	178
16.5. Addresses and Bibliography.....	180
16.5.1. Addresses	180
16.5.2. Standards/Bibliography	180
17. INDEX	181

1. General Instructions

1.1. Hazard Categories and Indications

The indications described below are used in connection with safety instructions you will need to observe for your own personal safety and the avoidance of damage to property.

These instructions are emphasised by bordering and/or shading and a bold-printed indication, their meaning being as follows:



Immediate danger

Failure to observe the information indicated by this warning will result in death, serious injury or extensive property damage.



Potential danger

Failure to observe the information indicated by this warning may result in death, serious injury or extensive property damage.



Danger

Failure to observe the information indicated by this warning may result in injury or property damage.



No hazard

Information indicated in this manner provides additional notes concerning the product.

1.2. Qualified users

Qualified users within the meaning of the safety instructions in this documentation are trained specialists who are authorised to commission, earth and mark equipment, systems and circuits in accordance with safety engineering standards and who as project planners and designers are familiar with the safety concepts of automation engineering.

1.3. Use as Prescribed

This is a modular automation system based on the CANbus, intended for industrial control applications within the medium to high performance range.

The automation system is designed for use within Overvoltage Category I (IEC 364-4-443) for the controlling and regulating of machinery and industrial processes in low-voltage installations in which the rated supply voltage does not exceed 1,000 VAC (50/60 Hz) or 1,500 VDC.

Qualified project planning and design, proper transport, storage, installation, use and careful maintenance are essential to the flawless and safe operation of the automation system.

The automation system may only be used within the scope of the data and applications specified in the present documentation and associated user manuals.

The automation system is to be used only as follows:

- as prescribed,
- in technically flawless condition,
- without arbitrary or unauthorised changes and
- exclusively by qualified users

The regulations of the German professional and trade associations, the German technical supervisory board (TÜV), the VDE (Association of German electricians) or other corresponding national bodies are to be observed.

Safety-oriented (fail-safe) systems

Particular measures are required in connection with the use of SPC in safety-oriented systems. If an SPC is to be used in a safety-oriented system, the user ought to seek the full advice of the SPC manufacturer in addition to observing any standards or guidelines on safety installations which may be available.



As with any electronic control system, the failure of particular components may result in uncontrolled and/or unpredictable operation.

All types of failure and the associated fuse systems are to be taken into account at system level. The advice of the SPC manufacturer should be sought if necessary.

2. Serial Interfaces (SIO.LIB)

2.1. Introduction

The serial interfaces of the Dialog/Cell Controller can be operated with a standard user interface using IEC61131 function blocks (initialize, receive, transmit, etc.). Internally, each function block is based on an interrupt-controlled receive queue and send queue, each of which is fixed in size (256 bytes).

The 3964/3964R protocol driver is not supplied with the CP1131, but can be ordered separately.

In the SIO function blocks, the interfaces are addressed by means of a number. This number is assigned to the variable 'CHANNEL'. Interface selection is allocated as follows:

Interface	CHANNEL
Serial programming interface	255
SIO 1	0
SIO 2	1
SIO 3	2
SIO 4	3

The following serial interface assignments apply to those dialog controllers with Ethernet interface (e.g., CEDISP29):

Interface	CHANNEL
Serial programming interface	255
SIO 1	10
SIO 2	11

The serial program interface (CHANNEL 255) normally is intended to be a service and programming interface for the corresponding configuration tools and programming tools, and is not initially available for application-specific data interchange.

When delivered, this interface is configured to contain the Service Channel Protocol (SVC), which gives configuration tools and programming tools access to the cell and dialog controllers. However, the application can use function blocks to disable this protocol, and then, after restarting, the interface can be used to interchange user data.



If the serial interface on the cover is configured to a specific application, the controller can only be programmed through the CAN / Ethernet interface. Applications will often use a hardware input to facilitate interchange between SVC and application-specific protocol.

If the user program is stored in the flash memory, the currently installed status (SVC protocol active or inactive) of the serial interface will be saved when the power supply is disconnected, and then automatically installed again on re-starting. The SVC protocol stays set to active all the time if the program is not stored permanently.



After switching over to the SVC protocol, the user program will no longer be able to communicate via the serial interface on the cover of the cell controller.

This can give rise to undefined plant states.

2.2. IEC61131 User Interface

2.2.1. Initialisation (SIO_INIT)

SIO_INIT

Declaration

```

FUNCTION_BLOCK SIO_INIT
VAR_INPUT
    CHANNEL      :      INT;
    BAUDRATE     :      DWORD;
    DATABITS     :      INT;
    STOPBITS     :      INT;
    PARITY       :      INT;
END_VAR
VAR_OUTPUT
    STATE        :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 . . 3, 255	Serial interface
	BAUDRATE	1200, 2400, 4800, 9600, 19200, 38400	Serial transmission rate
	DATABITS	7, 8	Number of databits
	STOPBITS	1, 2	Number of stopbits
	PARITY	0	No parity
		1	Even parity
2		Odd parity	
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	BAUDRATE invalid
		4	DATABITS invalid
		5	STOPBITS invalid
		6	PARITY invalid

Description

This function block sets up the desired parameters in the serial interface identified by 'CHANNEL': 'BAUDRATE', 'DATABITS', 'STOPBITS' and 'PARITY'. The block also empties the receive queue and send queue, and ends any other protocol that may be installed on the specified interface. The interface on the cover is subject to the following restrictions: only 8 databits, only 1 stopbit.

2.2.2. Receiving data (SIO_RECEIVE)

SIO_RECEIVE

Declaration

```

FUNCTION_BLOCK SIO_RECEIVE
VAR_INPUT
    CHANNEL      :      INT;
    LENGTH       :      INT;
    DATAPTR      :      DWORD;
END_VAR
VAR_OUTPUT
    COUNT        :      INT;
    DATA        :      ARRAY[0..255] OF BYTE;
    STATE        :      INT;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>	
Input parameters	CHANNEL	0..3, 255	Serial interface	
	LENGTH	0..255	Indicates the maximum number of data bytes that can be processed	
	DATAPTR		Address of a data object	
Output parameters	COUNT	0..255	Indicates the number of data bytes that have actually been received (COUNT <= LENGTH).	
	DATA		Valid only when DATAPTR = 0 The number of received data bytes indicated in COUNT is present at and from index position 0.	
	STATE	0		Function successfully executed
		1		Internal error
2			CHANNEL invalid	
3			LENGTH invalid	
	4		Receive queue overrun	
	9		SIO channel has not been initialised	
	10		Invalid area/range for DATAPTR	

Description

This function block fetches (at the most) the number of bytes indicated in 'LENGTH' from the receive queue of the serial interface identified by 'CHANNEL':
 In principle, the function block returns immediately to the invoker, even when there are fewer than 'LENGTH' bytes in the receive queue.
 The number of bytes actually fetched out and copied to 'DATA' or 'DATAPTR' can be read off from the output variable 'COUNT'.

2.2.3. Transmitting data (SIO_TRANSMIT)

SIO_TRANSMIT

Declaration

```

FUNCTION_BLOCK SIO_TRANSMIT
VAR_INPUT
    CHANNEL      :   INT;
    LENGTH       :   INT;
    DATAPTR      :   DWORD;
    DATA        :   ARRAY[0..255] OF BYTE;
END_VAR
VAR_OUTPUT
    STATE       :   INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 .. 3, 255	Serial interface
	LENGTH	0 .. 255	Indicates the number of data bytes that are to be transmitted
	DATAPTR		Address of a data object
	DATA		Valid only when DATAPTR = 0 The number of data bytes to be sent, as indicated in COUNT, is present at and from index position 0
Input parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	LENGTH invalid
		4	Send queue does not have enough free entries
		9	SIO channel has not been initialized
	10	Invalid range for DATAPTR	

Description

This function block enters the number of bytes indicated in 'LENGTH' into the send queue of the serial interface identified by 'CHANNEL'.

If there are not enough free entries in the send queue, the function block returns immediately with an error code indicated in 'STATE', showing either that all stipulated bytes have been transmitted (STATE = 0), or that none at all have been sent (STATE <> 0).

2.2.4. State request (SIO_STATE)

SIO_STATE

```

Declaration      FUNCTION_BLOCK SIO_STATE
                    VAR_INPUT
                        CHANNEL      :      INT;
                    END_VAR
                    VAR_OUTPUT
                        TXCOUNT      :      INT;
                        RXCOUNT      :      INT;
                        STATE          :      INT;
                    END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..3, 255	Serial interface
Output parameters	TXCOUNT	0..255	Number of bytes in the send queue of "CHANNEL"
	RXCOUNT		Number of bytes in the receive queue of "CHANNEL"
	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	LENGTH invalid
		4	Receive queue overrun
		5	BREAK has been received
		6	FRAMING error has occurred
		7	PARITY error has occurred
		8	OVERRUN error has occurred
		9	SIO channel has not been initialized

Description This function block indicates the number of bytes currently contained in the receive queue or send queue of the serial interface identified by 'CHANNEL', and shows the state of the oldest character.

2.2.5. Closing an interface (SIO_CLOSE)

SIO_CLOSE

Declaration

```

FUNCTION_BLOCK SIO_CLOSE
VAR_INPUT
    CHANNEL      :      INT;
END_VAR
VAR_OUTPUT
    STATE       :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 . . 3 , 255	Serial interface
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		9	SIO channel has not been initialized

Description

This function block ends operations with the serial interface identified by CHANNEL, but does not end the SVC protocol.
See SIO_CLOSESVC function block.

2.2.6. Initialising the SVC protocol on the cover (SIO_INITSVC)

SIO_INITSVC

```

Declaration      FUNCTION_BLOCK SIO_INITSVC
                  VAR_INPUT
                    CHANNEL      :      INT;
                  END_VAR
                  VAR_OUTPUT
                    STATE        :      INT;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	255	Serial interface on the cover
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid

Description This function block installs the SVC protocol on the serial interface, and ends any other protocol that may be installed on that interface.

2.2.7. Ending the SVC protocol (SIO_CLOSESVC)

SIO_CLOSESVC

```

Declaration      FUNCTION_BLOCK SIO_CLOSESVC
                  VAR_INPUT
                    CHANNEL      :      INT;
                  END_VAR
                  VAR_OUTPUT
                    STATE        :      INT;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	255	Serial interface on the cover
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		9	SVC protocol not installed

Description This function block deactivates the SVC protocol and releases the interface for other protocols.

3. CAN Interfaces (ECAN.LIB)

3.1. ECAN_V01.LIB

The current version documented here is ECAN_V10.LIB.

3.1.1. General Information

This library replaces the previous CAN.LIB library. Therefore, the ECAN.LIB library should be used for all new projects.

The Philips SJA1000 CAN controller is employed on the cell controllers. In contrast to the previously employed 82C200, this new component offers the user a wide variety of new functions, the most important of which are listed below:

- **EFF: 'Extended Frame Format'**
In other words, 29-bit identifiers can be sent and received.
- **LOM: 'Listen-only Mode'**
In other words, the CAN controller only listens in. There is no active receipt confirmation for CAN messages (no "Ack" slot).
- **STM: 'Self Test Mode'**
In other words, a CAN message is only ever sent once ("single shot"). Even in the case of transmission errors or arbitration losses there will be no automatic resend.
- **Expanded diagnostic options**
(access to error registries).

A further new property offered by the software driver is the option of changing the transmission sequence of the CAN messages in the send queue by means of prioritization.

It is strongly recommended that the ECAN.LIB be employed when using the SC_CAN on CANtrol *Powertrack* cell controllers, particularly as the expanded diagnostic options are very useful in this environment.

The ECAN.LIB can be employed as of the following firmware versions:

- CANtrol *Powertrack* cell controller FW 2.28
- Ethernet cell controller FW 2.10
- Standard cell controller FW 2.2X



If the ECAN.LIB is used with a cell controller whose firmware does not support this library, the cell controller will go into the error stop mode when the application program is downloaded. The CP1131 development environment's diagnostics function will then indicate unresolved references.

3.1.2. Brief Overview of the Most Important ECAN Function Blocks

The maximum of three CAN interfaces on the cell controllers can be served by a standardized user interface.

ECAN_INIT

Die CAN interface is initialized with this block.

There are 3 options for specifying the CAN interface baud rate:

- 1.) The baud rate is specified explicitly, e.g., 125000;
- 2.) The baud rate registers (BTR0, BTR1, CDR, OCR) are described explicitly;
- 3.) Using a preconfigured value (baud rate: =1). This can only be set via CNW.

We recommend option 3. This will ensure that the controller uses the same baud rate during the running application as it does in the boot loader mode. This is a particularly important consideration where the programming and service tools access the cell controller via the CANbus.

The following table lists the bit timing values for the standard baud rates of the ISO 11898 CAN interface:

Baud rate [KB/sec]	Register BTR0	Register BTR1
1000	0x00	0x14
500	0x00	0x1C
250	0x01	0x1C
125	0x03	0x1C
100	0x43	0x2F
50	0x47	0x2F
20	0x53	0x2F
10	0x67	0x2F

The following transmission speeds and parameters can be configured on the SC_CAN interface for the contact line (CANtrol Powertrack).

Line length [m]	Baud rate [KB/sec]	Register BTR1	Register BTR0	Register CDR	Register OCR
300	83,3	1C	05	82	DB
*	62,5	1C	07	83	DB
*	50,0	1C	09	84	DB
*	41,6	1C	0B	85	DB
*	35,7	1C	0D	86	DB

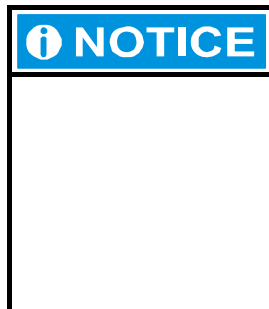
ECAN_SET_SWFILTER This block applies exclusively to the 11-bit identifier area. It controls the reception of CAN telegrams. While the CAN controller is ready to receive once the CAN interface has been successfully initialized, the application will not yet receive any CAN messages.

Only once messages have been released by the ECAN_SET_SWFILTER identifier will they be available to the application in a receive queue. The simplest method is to release the entire identifier range when the block is called up. However, this may result in performance problems. A better method is to have the identifier only allow those CAN messages which the application actually requires to pass. In other words, for example, 100 to 200, 352 and 712. This can be set up by multiple block call ups.

ECAN_RX

Receipt block for CAN messages, which only receives those 11-bit CAN messages which the ECAN_SET_SWFILTER block has permitted. 29-bit identifiers are received if the EFF: =TRUE parameter has been set in ECAN_INIT.

All received messages are placed in the receive queue. This queue can serve as an interim storage location for up to 200 CAN messages from all CAN interfaces. Calling up ECAN_RX retrieves the oldest CAN message or the message with the highest priority (*refer to 'ECAN_SET_SWFILTER'*) from the receive queue, making it available to the application. Depending on the communications volume, we recommend calling up the ECAN_RX more than once per PLC cycle.



Once the receive queue is completely filled, no additional CAN messages will be saved in the queue. This can result in data losses!

There are several reasons for this. For example, more CAN messages may be received than the ECAN_RX is able to make available to the application, or CAN messages may not be being retrieved from an initialized CAN interface.

The system characteristic of making only one receive queue available for all buses may also result in a buffer overrun at all initialized busses. Therefore steps must always be taken to ensure that an application which opens an input filter at a CANbus also actually retrieves the messages from the receive queue.

ECAN_TX

The transmit call-up for CAN messages places the message to be transmitted in a send buffer. The PLC application therefore does not need to wait until the CAN controller has successfully transmitted a message. The system drivers ensure the timely transmission of messages. Each CAN interface has a send queue for up to 64 CAN messages. In networks operating at relatively low baud rates such as 50 kBit/s, it is quite possible that the send queue may contain several messages.

To ensure that a CAN message is transmitted as quickly as possible even if the send buffer is full, priority control can be used to place this message at the head of the send queue. Naturally, this will influence the transmission sequence as well as the temporal behavior of the other CAN messages; something which must be taken into account during application conception.

29-bit identifier

29-bit identifier and 11-bit identifier messages can be used in parallel on the same CANbus interface. It should be noted that identifiers (COBID) in the 0...2047 range can be sent in both the standard as well as the extended frame format.

When CANbus messages with the same identifier are simultaneously sent on the same bus, the message sent in the standard format frame will always take priority over a message sent in extended frame format in the arbitration phase. Nonetheless, this dual-frame format option should be taken into account in the application. If you intend to work with both frame formats, a clear delineation among the identifiers with regard to both formats should be implemented for reasons of transparency.



Not all identifiers are available to the application!
 Each controller or control system can be configured or programmed from a central location. In order to ensure that these data will arrive at the correct controller, the so-called "service channel protocol" (SVC) is used with specific identifiers. These SVC identifiers lie in the ID1408 to 1664 range. In turn, this means that the identifiers in this range are not available to the application!

3.2. IEC61131 User Interface



So-called "identifiers" are used for CAN communications.
 In the following function blocks, these are indicated by the abbreviation, COB (Communication Object), e.g., COBID or COBSTART.

3.2.1. Initialization (ECAN_INIT)

ECAN_INIT
Declaration

```

FUNCTION_BLOCK ECAN_INIT
VAR_INPUT
    PORT          :    INT;
    BAUDRATE      :    DWORD;
    BTR0          :    BYTE;
    BTR1          :    BYTE;
    OCR           :    BYTE;
    CDR           :    BYTE;
    EFF           :    BOOL;
    LOM           :    BOOL;
    STM           :    BOOL;
END_VAR
VAR_OUTPUT
    ERROR         :    DWORD;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>	
Input parameters	PORT	0 . 2	CAN interface	
	BAUDRATE	0 , 1 , 10000 , 20000 , 35700 , 41600 , 50000 , 62500 , 83333 100000 , 125000 , 250000 , 500000 , 1000000	Specify baud rate via the BTR0, BTR1, OCR and CDR registers The baud rate from the EEPROM saved via CNW is used. 10 KB (Standard CAN) 20 KB (Standard CAN) 35,7 KB (CANtrol power track) 41,6 KB (CANtrol power track) 50 KB (Standard CAN + CANtrol power track) 62,5 KB (CANtrol power track) 83,3 KB (CANtrol power track) 100 KB (Standard CAN) 125 KB (Standard CAN) 250 KB (Standard CAN) 500 KB (Standard CAN) 1 MB (Standard CAN)	
	BTR0	0 . 255	Only valid if BAUDRATE:= 0	
	BTR1	0 . 255	Only valid if BAUDRATE:= 0	
	OCR	0 . 255	Only valid if BAUDRATE:= 0	
	CDR	0 . 255	Only valid if BAUDRATE:= 0	
	EFF	TRUE , FALSE	If EFF:=TRUE, the extended frame format (29-bit identifier) is also supported.	
	LOM	TRUE , FALSE	If LOM: =TRUE, data reception only (listen only mode).	
	STM	TRUE , FALSE	If STM: =TRUE, a telegram will only be sent once, even in case of transmission errors or arbitration loss (self test mode). No active telegram confirmation by other CAN-bus subscribers is required.	
	Output parameters	ERROR	0	Function successfully executed.
			1	Internal error
			2	PORT invalid
			3	BAUDRATE invalid
			4	STM, LOM and OCD must be FALSE. No other values can be set on this module.
			11	Invalid value in the serial EEPROM: BAUDRATE or BTR0/1
	Description	<p>Initializes the CAN interface identified by 'PORT' and sets it to the specified BAUDRATE.</p> <p>This call-up can also be used to rectify CAN controller error states such as BUSOFF. In doing this, the call-up deletes the entries in both the receive as well as the send queues. The input filter created via ECAN_SET_SWFILTER is also reset. In other words, when ECAN_INIT is called up, the application cannot yet receive a CAN telegram in the 11-bit format. Only after the input filter has again been put in place can CAN messages again be received.</p> <p>If the 'BAUDRATE' parameter has a value of '0', the BTR0, BTR1 and possibly OCR, CDR parameters are written directly to the corresponding CAN controller</p>		

registers (without a consistency check). If the 'BAUDRATE' parameter has a value of '1', the values stored in the controller's EEPROM are used (this is the recommended method). If the transmission rate is specified directly by the 'BAUDRATE' parameter value, the CAN controller is set accordingly. All other bit timing parameters in the BTR0, BTR1, OCR, CDR registers are set to the default values.

If the system software determines that the selected interface is a CANtrol *Powertrack*, only the transmission rates intended for this type of interface can be specified there. The system software independently selects the appropriate CANtrol *Powertrack* parameters. A transmission rate only intended for CANtrol *Powertrack* may not be specified for a standard interface.

The 'EFF' allows 29-bit identifiers (COBID) to be employed in addition to the 11-bit identifiers.

The 'LOM' parameter initializes the SJA1000 CAN controller in a "listen only mode". By activating this parameter, the CANbus will only be able to receive data. In this case, the CAN interface reacts passively at the bus. Neither an acknowledgment nor error frames will then be generated.

When activating the (Self Test Mode), no other subscriber who confirms CAN telegrams with an acknowledgement may be connected to the CANbus. This function is primarily required by large, segmented, dynamic CANbus networks such as are intended for CANtrol *Powertrack* in order to prevent the constant repetition of telegrams which are not retrieved at the CANbus.

A message sent on the CANbus is always handled as if it had been successfully sent even if no acknowledgement is returned by other subscribers.

When the controller starts up with the service channel activated, the associated CAN interface is set up with the values stored in the EEPROM. These settings are overridden by an initialization call-up in the application program. Any deviations in the two sets of settings can result in the service channel functions not working properly in all operating modes.

We therefore recommend calling up the initialization function with the BAUDRATE parameter = '1' where the service channel is activated (the initialization will then use the values from the EEPROM).

3.2.2. Starting/Ending CAN Messages (ECAN_SET_SWFILTER)

ECAN_SET_SWFILTER

```

Declaration      FUNCTION_BLOCK ECAN_SET_SWFILTER
                    VAR_INPUT
                        PORT          :      INT;
                        COBSTART      :      DWORD;
                        COBEND        :      DWORD;
                        PRIO           :      BYTE;
                        ENABLE         :      BOOL;
                    END_VAR
                    VAR_OUTPUT
                        ERROR          :      DWORD;
                    END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0..2	CAN interface
	COBSTART	0..2047	First identifier to be received
	COBEND	0..2047	Last identifier to be received
	PRIO	0..7	Sets the priority of the individual COBID area.
	ENABLE	TRUE	The identifiers between COBSTART and COBEND are received.
		FALSE	The identifiers between COBSTART and COBEND are <u>not</u> received.
Output parameters	ERROR	0	Function successfully executed.
		1	Internal error
		2	PORT invalid
		3	COBSTART invalid
		4	Invalid COBEND or wrong priority
	9	CAN interface not initialized.	

Description

This function block is used to control the software acceptance filter for the 11-bit standard identifier range of a CAN interface.

Reception of all CAN messages on the physical CAN interface 'PORT' with a COBID between 'COBSTART' and 'COBEND' (inclusive) is enabled ('ENABLE' = TRUE) or blocked ('ENABLE' = FALSE).

Once a COBID has been blocked, the CAN messages with this identifier are no longer placed in this interface's internal receive queue. All messages already in the receive queue are unaffected by this. Using 'PRIO', the COBID areas defined by calling up this block can be prioritized with regard to the receive queue. This means that messages with a higher priority but arriving later in the receive queue will be placed ahead of earlier messages with a lower priority. 'PRIO 0' is the highest priority, "PRIO 7" the lowest.



The software acceptance filtration is not employed for the reception of extended frame messages (29-bit). To receive standard frame messages (11-bit), the associated COBID's must be enabled after initialization.

3.2.3. Hardware Acceptance Filter (ECAN_SET_HWFILTER)



While this block is part of the ECAN.LIB, it is currently **not** implemented, in other words, the block will not function if it is called up!

ECAN_SET_HWFILTER

```

Declaration      FUNCTION_BLOCK ECAN_SET_HWFILTER
VAR_INPUT
    PORT          :      INT;
    AFM           :      BYTE;
    ACR           :      ARRAY[0..3] OF BYTE;
    AMR           :      ARRAY[0..3] OF BYTE;
END_VAR
VAR_OUTPUT
    ERROR         :      DWORD;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0..2	CAN interface
	AFM	0..1	Select a single or dual filters
	ACR	[0..255] [0..255] [0..255] [0..255]	First HW register filter
	AMR	[0..255] [0..255] [0..255] [0..255]	Second HW register filter
	Output parameters	ERROR	0 ??

Description This function block is used to control a CAN interface’s hardware COBID acceptance filter.

Reception of all CAN messages on the physical CAN interface ‘PORT’ with a COBID which can pass through the HW filter is enabled.

The HW filter should be configured as a dual filter with the aid of ‘AFM’. This will pass the extended COBIDs of the CAN messages received from the CAN transceiver through 2 filters.

While the ACR register defines the bit patterns of the extended COBID to be received, the AMR register defines those bit patterns which are “don’t care”.

$$FILTERED_COBID = ((RECEIVED_COBID \wedge ACR) \vee AMR)$$

The two LSB (Bit01, Bit1) of the ACR[3] and AMR[3] registers are not used.

Filtration of the RTR bit is performed in ‘ACR[3].Bit2’ and ‘AMR[3].Bit2’.

Thus, the MSB of the extended COBID is performed in bit positions ‘ACR[0].Bit7’ and ‘AMR[0].Bit7’.

3.2.4. Receiving CAN Messages (ECAN_RX)

ECAN_RX

Declaration

```

FUNCTION_BLOCK ECAN_RX
VAR_INPUT
    DATAPTR      :    DWORD;
    PORT         :    NT;
END_VAR
VAR_OUTPUT
    VALID        :    BOOL;
    COBID        :    DWORD;
    RTR          :    BOOL;
    EFF          :    BOOL;
    LENGTH       :    INT;
    DATA        :    ARRAY[0..7] OF BYTE;
    ERROR        :    DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0..2	CAN interface
	DATAPTR		Address of a data object
Output parameters	VALID	TRUE	CAN message received, The variables COBID, RTR, LENGTH and possibly DATA/DATAPTR are valid.
		FALSE	No CAN message received.
	COBID		CAN identifier
	RTR	TRUE	CAN remote frame received. The variables DATA/DATAPTR contain no data.
		FALSE	CAN data frame received. The variables DATA/DATAPTR may contain data.
	EFF	TRUE	The received message was transmitted us- ing the extended frame format.
		FALSE	The received message was transmitted us- ing the standard frame format.
	LENGTH	0..7	Indicates the number of data bytes received.
	DATA		Only valid if DATAPTR = 0. The number of received data bytes indicated by LENGTH begins as of index position 0.
	ERROR	0	Function successfully executed.
	1	Internal error	
	2	PORT invalid	
	3	Receive queue or CAN controller overrun.	
	5	CAN controller is 'bus OFF'.	
	7	Error in the CAN controller.	
	9	CAN interface not initialized.	
	10	Invalid range for DATAPTR.	

Description

The 'VALID' output variable can be used to check whether a CAN message was received. If 'VALID' = TRUE, the function module first retrieves the messages with the highest priority (refer to ECAN_SET_SWFILTER) from the physical 'PORT' CAN interface's receive queue. If prioritization is not used, this corresponds to the "oldest" message (FIFO principle). After retrieval, the message is deleted from the receive queue.

Once there are no further messages in the receive queue, the output variable is set to 'VALID' = FALSE.

The value, 'COBID', contains the received message's message identifier. Here, the 'EFF' variable indicates whether the identifier in question is an 11-bit or a 29-bit identifier.

The 'RTR' variable indicates whether the remote transmission request bit was set in this message.

'LENGTH' indicates the number of valid utility data bytes.

The received data are located either in the DATA array or under the address specified by DATAPTR.

3.2.5. Sending CAN Messages (ECAN_TX)



Do not assign the same identifier multiple times during transmission!

If 2 CAN nodes with the same send identifier initiate access to the network, transmission errors which may cause the CAN nodes to no longer be able to transmit (passive ERROR).

The CANbus is a multi-master bus. This means that each CAN node on a bus is entitled to transmit its messages at any desired time. The messages transmitted at the CANbus are identified by either an 11-bit or a 29-bit identifier. This identifier simultaneously gives the message a priority. This has an effect when 2 CAN nodes have a transmission request at the same time. In this case, the message with the higher priority will be sent first.

ECAN_TX

Declaration

```

FUNCTION_BLOCK ECAN_TX
VAR_INPUT
    DATAPTR      :    DWORD;
    PORT         :    INT;
    COBID        :    DWORD;
    RTR          :    BOOL;
    EFF          :    BOOL;
    SST          :    BOOL;
    LENGTH       :    INT;
    DATA        :    ARRAY[0..7] OF BYTE;
    PRIO         :    BYTE;
END_VAR
VAR_OUTPUT
    ERROR        :    DWORD;
END_VAR
  
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0..2	CAN interface
	DATAPTR		Address of a data object
	COBID		CAN identifier
	RTR	TRUE	Send CAN remote frame. The variables DATA/DATAPTR contain no data, LENGTH = 0.
		FALSE	Send CAN data frame. The variables DATA/DATAPTR may contain data.
	EFF	TRUE	The message being transmitted is sent in the extended frame format.
		FALSE	The message being transmitted is sent in the standard frame format.
	SST	TRUE	The message being transmitted is only placed on the CANbus once. There is no repetition in case of an error (physical transmission path fault or arbitration problem) or the simultaneous transmission of a higher priority CANbus message from another subscriber.

<i>Parameter</i>	<i>Value</i>	<i>Description</i>
	FALSE	Normal transmission
LENGTH	0..7	Indicates the number of data bytes to be sent.
DATA		Only valid if DATAPTR = 0. The number of data bytes to be sent is indicated by LENGTH begins as of index position 0.
PRIO	0..7	Selects the priority of the message to be sent. May result in an alteration in the sequence of the CANbus messages to be sent.

Output parameters	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
	ERROR	0	Function successfully executed.
	1	Internal error	
	2	PORT invalid	
	3	COBID invalid	
	4	LENGTH invalid	
	5	CAN controller is 'bus OFF'.	
	6	No free entry in the send queue	
	9	CAN interface not initialized.	
	10	Invalid range for DATAPTR.	

Description

This function block copies a CAN message into the "PORT" physical CAN interface's send queue.

The 'COBID' variable contains the message identifier for this message, while 'LENGTH' indicates the number of valid utility data bytes. Just as is the case with "ECAN_RX", 'EFF' indicates whether the message is to be sent with a standard or an extended identifier.

The data to be transmitted are located in the DATAPTR or DATA variables.

The 'RTR' variable indicates the state of the remote transmission request bit in the message being transmitted.

If 'SST:=TRUE', this results in a so-called "single shot transmission". The message is placed on the CANbus only once, regardless of whether or not this message's arbitration phase collides with the simultaneous transmission by another CANbus subscriber. Neither will any resend attempts be made in case of other errors such as a transmission path fault.

The 'PRIO' value can be used to suppress the normal FIFO sequence. A higher priority will sort the message into the FIFO according to its priority. In this case, the message recipient receives the higher priority telegrams prior to the lower priority telegrams. Transmitting all messages with the same priority results in the conventional transmission FIFO. 'PRIO 0' is the highest priority, "PRIO 7" the lowest.



If the priority of the transmission telegram is changed, the user must ensure that the message recipient makes the correct time reference with regard to this message.

3.2.6. Status Query (ECAN_STATE)

ECAN_STATE

Declaration

```

FUNCTION_BLOCK ECAN_STATE
VAR_INPUT
    PORT      :      INT;
END_VAR
VAR_OUTPUT
    TX_PEND   :      DWORD;
    RX_PEND   :      DWORD;
    TX_CNTERR :      DWORD;
    RX_CNTERR :      DWORD;
    ERROR     :      DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0 . 2	CAN interface
Output parameters	TX_PEND		Number of CAN messages in the PORT send queue.
	RX_PEND		Number of CAN messages in the PORT receive queue.
	TX_CNTERR		Incremental transmission error messages counter
	RX_CNTERR		Incremental receive error messages counter
	ERROR	0	Function successfully executed.
		1	Internal error
		2	PORT invalid
		3	Receive queue overrun.
		5	CAN controller is 'bus OFF'.
		9	CAN interface not initialized.

Description

Indicates the number of CAN messages currently in the receive or send queue of the 'PORT' physical CAN interface.

TX_CNTERR and RX_CNTERR indicate the total number of errors which occurred since the previous ECAN_INIT call-up.

3.2.7. Starting/Ending a Gateway (ECAN_BRIDGE)

ECAN_BRIDGE

Declaration

```

FUNCTION_BLOCK ECAN_BRIDGE
VAR_INPUT
    SOURCE      :      INT;
    DEST        :      INT;
    COBSTART    :      DWORD;
    COBEND      :      DWORD;
    ENABLE      :      BOOL;
END_VAR
VAR_OUTPUT
    ERROR       :      DWORD;
END_VAR
    
```

Input parameters

<i>Parameter</i>	<i>Value</i>	<i>Description</i>
SOURCE	0..2	The CAN message is received at this interface.
DEST	0..2	The CAN message is to be resent from this interface.
COBSTART COBEND		Identifier range between COBSTART and COBEND is transferred to DEST from SOURCE if ENABLE := TRUE.
ENABLE	TRUE	The identifiers between COBSTART and COBEND are to be transferred from SOURCE to DEST.
	FALSE	The identifiers between COBSTART and COBEND are to <u>not</u> be transferred from SOURCE to DEST.

Output parameters

ERROR	0	Function successfully executed.
	1	Internal error
	3	COBSTART invalid
	4	COBEND invalid
	6	SOURCE or DEST invalid
	9	One or both CAN interfaces not initialized.

Description

This function block is used to control the bridge functionality. CAN messages with an identifier between 'COBSTART' and 'COBEND' (inclusive) received on the 'SOURCE' physical CAN interface are automatically (at the system level) resent from the 'DEST' CAN interface if this option was set with 'ENABLE' = TRUE. Only 11-bit identifiers are permitted. This functionality can be blocked at any time for the associated identifier ranges ('ENABLE' = FALSE).

3.2.8. Querying the Own Node ID (ECAN_GET_NODEID)

ECAN_GET_NODEID

Declaration

```

FUNCTION_BLOCK ECAN_GET_NODEID
VAR_INPUT
    PORT      :      INT;
END_VAR
VAR_OUTPUT
    NODEID    :      BYTE;
    ERROR     :      DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0 . 2	CAN interface
Output parameters	NODEID	1 . 127	Node number
	ERROR	0	Function successfully executed.
		1	Internal error
		2	PORT invalid
	3	NODEID invalid	

Description

Provides this module's node ID at the 'PORT' physical CAN interface.

The node ID is used as the unique module identifier within a connected CAN network. It is saved to the module's EEPROM by the CNW configuration program.

3.2.9. Setting Error Limits (ECAN_SET_EWL)

ECAN_SET_EWL

```

Declaration      FUNCTION_BLOCK ECAN_SET_EWL
                  VAR_INPUT
                    PORT      :      INT;
                    EWL       :      BYTE;
                  END_VAR
                  VAR_OUTPUT
                    ERROR     :      DWORD;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0 . 2	CAN interface
	EWL	0 . 255	Error limit tolerance Default value = 96
Output parameters	ERROR	0	Function successfully executed.
		1	Internal error
		2	PORT invalid
		9	CAN interface not initialized.

Description This function block sets up a register in the CAN controller responsible for the send / receive error tolerance. The TXERR and RXERR output variable values in the ECAN_HWSTATE function block will only be incremented once the error frequency rises above 'EWL'. The 'EWL' default value is 96. The higher this value, the more "tolerant" the controller will be of CAN errors and vice versa.

3.2.10. Reading the Error Register (ECAN_HWSTATE)

ECAN_HWSTATE

Declaration

```

FUNCTION_BLOCK ECAN_HWSTATE
VAR_INPUT
    PORT      :      INT;
END_VAR
VAR_OUTPUT
    ERROR     :      DWORD;
    CTRL_TYPE :      E_ECAN_CTRL_TYPE;
    EWLRL     :      BYTE;
    RXERR     :      BYTE;
    TXERR     :      BYTE;
    ECCR      :      BYTE;
    ALCR      :      BYTE;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0..2	CAN interface
Output parameters	CTRL_TYPE	0	Philips 82C200 CAN controller
		1	Philips SJA1000 CAN controller
	EWLRL	0..255	Current error register status.
	RXERR	0..255	Number of faulty CAN messages received.
	TXERR	0..255	Number of faulty CAN messages transmitted.
	ECCR	0..255	Refer to the description below.
	ALCR	0..255	Refer to the description below.
	ERROR	0	Function successfully executed.
		1	Internal error
		2	PORT invalid
		9	CAN interface not initialized.

Description

This function block allows internal CAN controller statistical functions to be utilized. Based on the information, assumptions regarding the network transmission quality can then be made.

If 'EWLRL' > 96 (unless changed by ECAN_SET_EWL), the number of transmission errors in 'TXERR' and the number of receipt errors in 'RXERR' are counted. 'ECCR' and 'ALCR' provide information concerning the location within the CAN telegram at which the transmission error occurred. Because these registers are extremely difficult to interpret (only for very serious CANbus problem), this document does not attempt to explain the various codes they contain. For more detailed information, please contact our Technical Support service.

If the CAN errors are serious enough to result in a 'bus OFF' state, 'RXERR' is set to '0'. 'TXERR' is then set to '127' and continuously incremented within the context of the bus OFF recovery (128 bus-free times). This allows the bus OFF recovery status to be monitored.

These statistical values are only available on the Philips SJA1000 CAN controller.

3.2.11. Reading the Expanded Controller Status (ECAN_EXTSTATE)

ECAN_EXTSTATE

Declaration

```

FUNCTION_BLOCK ECAN_EXTSTATE
VAR_INPUT
    PORT      :      INT;
END_VAR
VAR_OUTPUT
    ERROR      :      DWORD;
    TX_PEND    :      DWORD;
    RX_PEND    :      DWORD;
    RX_CNTERR  :      DWORD;
    TX_CNTERR  :      DWORD;
    CNTR_MODE  :      BYTE;
    CNTR_STATUS :      BYTE;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	0..2	CAN interface
Output parameters	TX_PEND		Number of CAN messages in the PORT send queue.
	RX_PEND		Number of CAN messages in the PORT receive queue.
	RX_CNTERR		Incremental transmission error messages counter
	TX_CNTERR		Incremental receive error messages counter
	CNTR_MODE	0..255	Please contact Tech. Support.
	CNTR_STATUS	0..255	Please contact Tech. Support.
	ERROR	0 1 2 9 10	Function successfully executed. Internal error PORT invalid CAN interface not initialized. Not a Philips SJA1000 CAN controller.

Description

Indicates the number of CAN messages currently in the receive or send queue of the 'PORT' physical CAN interface.

TX_CNTERR and RX_CNTERR indicate the total number of errors which occurred since the previous ECAN_INIT call-up.

'CNTR_MODE' and 'CNTR_STATUS' provide the current contents of the SJA1000 controller's mode register or status register.

3.2.12. Setting the Listen Only Mode (ECAN_SET_LOM)

ECAN_SET_LOM

Declaration

```
FUNCTION_BLOCK ECAN_SET_LOM
VAR_INPUT
    PORT    :    INT;
    INIT    :    BOOL;
    LOM     :    BOOL;
END_VAR
VAR_OUTPUT
    ERROR   :    DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	1	CAN interface
	INIT	TRUE	LOM set with a rising flank.
		FALSE	Block inactive.
Output parameters	ERROR	FALSE	Switch listen only mode off.
		TRUE	Switch listen only mode on.
		0	Function successfully executed.
		1	Internal error
		2	PORT invalid
		9	CAN interface not initialized.

Description

This block activates or deactivates the listen only mode without calling up 'ECAN_INIT'.

The mode change only occurs at the 'INIT' rising flank.



Telegrams may be lost when the mode is switched!

When the mode is switched, the CAN controller must be switched to the reset mode, thus preventing the CAN controller from communicating for a brief period. We therefore recommend only using this block during initialization.

3.2.13. Setting the Self Test Mode (ECAN_SET_STM)

ECAN_SET_STM

```

Declaration
FUNCTION_BLOCK ECAN_SET_STM
VAR_INPUT
    PORT      :      INT;
    INIT      :      BOOL;
    STM       :      BOOL;
END_VAR
VAR_OUTPUT
    ERROR     :      DWORD;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORT	1	CAN interface
	INIT	TRUE	STM set with a rising flank.
		FALSE	Block inactive.
	STM	FALSE	Switch self test mode off.
TRUE		Switch self test mode on.	
Output parameters	ERROR	0	Function successfully executed.
		1	Internal error
		2	PORT invalid
		9	CAN interface not initialized.

Description This block activates or deactivates the self test mode without calling up 'ECAN_INIT'.
The mode change only occurs at the 'INIT' rising flank.



Telegrams may be lost when the mode is switched!
When the mode is switched, the CAN controller must be switched to the reset mode, thus preventing the CAN controller from communicating for a brief period. We therefore recommend only using this block during initialization.

4. Analog Input/Output (QAIO.LIB)

4.1. Introduction

The QAIO.LIB can access up to 16 input channels and 4 output channels of an E-bus module. A signed 16-bit integer value is loaded in the process. These 16 input channels are converted automatically, one after the other, in the rhythm of the internal module-sampling cycle. As a rule, no one input signal will be any more than 4 ms old.

The measurement range is +/-10 volts and/or 0 ... 20 mA at 12-bit resolution including sign.

Data is written to the analog outputs in synchronisation with the module's internal sampling cycle.

4.2. IEC61131 User Interface

4.2.1. Reading analog input values (QAIO_ANALOGIN)

QAIO_ANALOGIN

```

Declaration
FUNCTION_BLOCK QAIO_ANALOGIN
VAR_INPUT
    POSITION      :      INT;
    CHANNEL     :      INT;
END_VAR
VAR_OUTPUT
    VALUE :      INT;
    STATE :      INT;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	POSITION	0 . . 6	Analog module's slot position in E-bus
	CHANNEL	0 . . 15	Channel number of desired analog input.
Output parameters	VALUE		Analog input voltage as signed 16-bit value (0.305 mV/6.104 µA/LSB) -32768 = -10.000 volt 0 = 0.000 volt = 0 mA 32752 = +9.995 volt = 19.991 mA
	STATE	0	Function successfully executed
		1	POSITION invalid
		2	CHANNEL invalid
		3	No analog module at POSITION

Description This function block indicates the current value of an analog input channel on a particular analog expansion module. This value is taken from the internal process image recorded cyclically over all multiplexer channels.

The analog value supplied in the output variable 'VALUE' is signed, and has a resolution of $10\text{ V}/32768 = 0.305\text{ mV}$ ($20\text{ mA}/32768 = 6.104\text{ µA}$) per LSB. The actual analog resolution supplied by the component is 12-bit, corresponding to 4.883 mV (9.766 µA), meaning the analog value jumps by 16 LSB each time.

4.2.2. Writing analog output values (QAIO_ANALOGOUT)

QAIO_ANALOGOUT

Declaration

```

FUNCTION_BLOCK QAIO_ANALOGOUT
VAR_INPUT
    POSITION      :      INT;
    CHANNEL      :      INT;
    VALUE       :      INT;
END_VAR
VAR_OUTPUT
    STATE       :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	POSITION	0 . . 6	Analog module's slot position in E-bus
	CHANNEL	0 . . 3	Channel number of desired analog output.
Output parameters	VALUE		Analog output value as signed 16-bit value (0.305 mV/6.104 µA/LSB) -32768 = -10.000 volt 0 = 0.000 volt = 0 mA 32752 = +9.995 volt = 19.991 mA
	STATE	0	Function successfully executed
		1	POSITION invalid
		2	CHANNEL invalid
	3	No analog module at POSITION	

Description

On the analog module's analog-output designated CHANNEL, this function block writes the value indicated in VALUE at the slot identified by POSITION. The internal system program inserts a delay of one module-sampling cycle (250 µsec) between two write operations to one and the same module.

The analog value supplied in output variable 'VALUE' is signed and has a resolution of $10\text{ V}/32768 = 0.305\text{ mV}$ ($20\text{ mA}/32768 = 6.104\text{ µA}$) per LSB. The actual analog resolution supplied by the module is 12-bit, corresponding to 4.883 mV (9.766 µA), meaning the analog value jumps by 16 LSB each time.

Blank page

5. CANopen Master (COMASTER.LIB)

5.1. COMASTER_V01.LIB

The current version is COMASTER_V01.LIB.

5.1.1. Introduction to CANopen Master Library

Users should have detailed knowledge of the CANbus and the CANopen protocol before working with the CANopen Master Library.

The CANbus, as is generally known, can be used to transmit up to 8 bytes of data, which are identified by an 11-bit identifier. It is for the user operating with the CANbus to interpret the data communicated. There are a number of ways of developing a user-specific protocol of the necessary kind, but attaching external devices to a proprietary application layer like this does take time and effort.

With the CANopen protocol, the manufacturers of CAN modules have defined a common foundation for interchange of data between CAN nodes with as little as possible in the way of problems. The CANopen protocol is an application layer that sits on the CANbus.

CANopen provides various different categories under which data is interchanged. Configuration data is an important example. This data is transmitted via a so-called **S**ervice **D**ata **O**bject (SDO). CANopen describes the identifier and the exact data format to use when transmitting an SDO. Process data is interchanged by means of a **P**rocess **D**ata **O**bject (PDO), which can also be transmitted in synchronous mode (SYNC). The network is supervised by a Network Management System (NMT).

The library is arranged into the following sections so that the CANopen master can use the relevant communications objects after an appropriate hardware initialisation:

- **Node Manager**
- **SDO Manager**
- **PDO Manager**
- **SYNC Manager**

Within the application, all four of these library sections behave in the same way:

- The first step is initialisation of an object, e.g. an SDO object.
The user specifies the appropriate initialisation data and, provided that it is correct, is given what is called a *handle* for this data.
- In the next step, the user can transmit or receive an SDO.
The *handle* given is always specified as a reference when doing so.

NOTICE

CAN interface 0 is not unconditionally available to the CANopen master.

Each controller in the control system can be configured and/or programmed from a central point. The Service Channel (SVC) protocol is used to make sure this data gets to the right controller.

A particular set of identifiers, the SDO identifiers, is used for this purpose when transmitting these SVC messages over the CANbus. Accordingly, the user can choose between the CANopen master and a service channel functionality on CAN interface 0. SVC is set up as default.

5.1.2. Initialising the CANopen interface

NOTICE

CANopen master initialisation:

The CANopen master functionality is initialised by calling various different function blocks. At the CP1131 firmware currently stands, the correct sequence of the basic initialisation process must be strictly observed.

1.) Initialisation of CANopen ports (CAN interfaces):

```
COM_PORT_INIT (CHANNEL := 0, ...);
COM_PORT_INIT (CHANNEL := 1, ...);
COM_PORT_INIT (CHANNEL := 2, ...);
```

The sequence in which the interfaces are initialised is not important.

2.) Initialisation of internal CANopen data structures:

```
CoM_Init (...);
```

This function block should be called just once, after initialisation of all ports occupied by CANopen.

The consequences of calling in the wrong sequence are as follows:

If COM_PORT_INIT is called after calling this function, the internal data structures will be reset (on the occasion of the first call), and the CANopen master will not work properly.

3.) Initialisation of remaining CANopen management blocks, for example:

```
Node Manager
SDO Manager
PDO Manager
```

5.1.3. Initialising the physical interface (COM_PORT_INIT)

COM_PORT_INIT

Declaration

```
FUNCTION_BLOCK COM_PORT_INIT
VAR_INPUT
    CHANNEL      :      INT;
    BAUDRATE     :      DWORD;
END_VAR
VAR_OUTPUT
    ERROR       :      DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
	BAUDRATE	1,	Corresponding to configuration in serial EEPROM
		10000,	10 Kbit
		20000,	20 Kbit
		50000,	50 Kbit
		100000,	100 Kbit
		125000,	125 Kbit
		250000,	250 Kbit
		500000,	500 Kbit
	1000000	1 Mbit	
Output parameters	ERROR (hex)	0	Function successfully executed
		10000002	Dynamic memory cannot be allocated
		10000004	Parameter value(s) invalid
		1010xxxx	Internal CAN driver error xxxx = internal error code
Description	This function block initialises the appropriate CAN port. The block must be called explicitly for each port in use. Channel CAN1 and Channel CAN2 will not operate at any higher than 125 Kbit.		

5.1.4. Restarting the physical interface (COM_PORT_RESTART)

COM_PORT_RESTART

Declaration

```
FUNCTION_BLOCK COM_PORT_RESTART
VAR_INPUT
    CHANNEL      :      INT;
END_VAR
VAR_OUTPUT
    ERROR :      DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
Output parameters	ERROR (hex)	0	Function successfully executed
		2000000E	Port not initialised
		10000004	Parameter value(s) invalid
		1010xxxx	Internal CAN driver error xxxx = internal error code

Description

If any one of the error codes 16#10100001, 16#10100003, 16#10100004 or 16#10100005 appears in a functioning CANopen application, the CANopen master can be re-started by calling this function block.

5.1.5. Initialising the CANopen master (COM_INIT)

COM_INIT

Declaration

```
FUNCTION_BLOCK COM_INIT
VAR_OUTPUT
    ERROR :      DWORD;
END_VAR
```

Output parameters

<i>Parameter</i>	<i>Value</i>	<i>Description</i>
ERROR (hex)	0	Function successfully executed
	20000005	Insufficient memory for number of nodes
	30000007	Insufficient memory for number of SDOs
	40000005	Insufficient memory for number of PDOs
	0000xxxx	Internal error xxxx = VRTXsa error code

Description

This function block initialises the CANopen master software, and may only be called once.

Once the function block has been executed successfully, the driver can administer up to 127 nodes, 254 Service Data Objects (SDOs), and 2048 Process Data Objects (PDOs).

The node guarding function and the generation of SYNC telegrams are not activated in the process. Specially designed function blocks are available to the user for this purpose.

5.2. Node Manager

5.2.1. Logging on new CANopen slave nodes (COM_ADD_NODE)

COM_ADD_NODE

Declaration

```
FUNCTION_BLOCK COM_ADD_NODE
VAR_INPUT
    CHANNEL      :      INT;
    NODEID       :      WORD;
    GUARDTIME    :      WORD;
    GUARDCOBID  :      WORD;
END_VAR
VAR_OUTPUT
    NODEHANDLE   :      WORD;
    ERROR        :      DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
	NODEID	1..127	Node number
	GUARDTIME		Number of 100 ms ticks
	GUARDCOBID	0 1..2047	Identifier conforming to DS 301 V.3.0 Identifier assigned for guarding
Output parameters	NODEHANDLE		Handle of logged-on node
	ERROR (hex)	0	Function successfully executed
		20000004	NODEID already exists
		20000006	Max. NODEID reached
		2000000D	Parameter(s) invalid
	2000000E	CAN port not initialised	

Description

Every CANopen node in the network must be logged on by means of the COM_ADD_NODE function block. When the function block is executed successfully, a *handle* is returned for the node in question, and must be referred to in all further function blocks specific to the node. The system can process at the most 127 nodes, which can be distributed over all of the CAN channels in use.

Where an entry other than zero is made under guard time, it is assumed that the driver can automatically transmit cyclical guarding telegrams to the nodes in question. Guarding is started by function block COM_GUARDING_ON. The guard-time is specified in the form of ticks based on a time of 100 ms. The identifier for the guarding telegram may be chosen at will from within the range 1 to 2047. If the value chosen is 0, the function block will use the identifier stipulated in CANopen Communication Profile DS-301, version 3.0 (1792 + NODEID).



A separate instance of this function block is required for each node logged on.

5.2.2. Changing state of a CANOpen slave (COM_CHANGE_STATE)

COM_CHANGE_STATE

Declaration

```

FUNCTION_BLOCK COM_CHANGE_STATE
VAR_INPUT
    NODEHANDLE : WORD;
    NODESTATE  : WORD;
END_VAR
VAR_OUTPUT
    ERROR : DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	NODEHANDLE		Handle of logged-on node
	NODESTATE	1,	Start node
		2,	Enter prepared state
		3,	Enter preoperational state
		4,	Reset node
5		Reset communication	
Output parameters	ERROR (hex)	0	Function successfully executed
		20000001	Unknown state
		20000002	Unknown handle
		2010xxxx	Internal CAN driver error xxxx = internal error code

Description This function block changes the communications state of the specified node. An unconfirmed CAN telegram is transmitted in the process (NMT Service). The actual state of the node can only be checked when the node guarding function is activated for the node in question.

5.2.3. Getting state of a CANopen slave (COM_GET_State)

COM_GET_State

```

Declaration      FUNCTION_BLOCK COM_GET_STATE
VAR_INPUT
    NODEHANDLE   :      WORD;
END_VAR
VAR_OUTPUT
    NODESTATE    :      WORD;
    ERROR        :      DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	NODEHANDLE		Handle of logged-on node
Output parameters	NODESTATE	1, 2, 3	operational prepared preoperational
	ERROR (hex)	0 20000002	Function successfully executed Unknown handle

Description This function block returns the communication state of the specified node. The actual state of the node can only be ascertained when the node guarding function is activated for the node in question.

When node guarding is disabled, the block returns the state most recently set up by means of the *COM_CHANGE_STATE* function block.



After a CANopen slave has received and executed the command "Reset", it automatically goes into "preoperational" state. A CANopen slave does exactly the same after it has been switched on.

5.2.4. Activating node guarding (COM_GUARDING_ON)

COM_GUARDING_ON

Declaration

```
FUNCTION_BLOCK COM_GUARDING_ON
VAR_INPUT
    NODEHANDLE : WORD;
END_VAR
VAR_OUTPUT
    ERROR : DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	NODEHANDLE		Handle of logged-on node
Output parameters	ERROR (hex)	0	Function successfully executed
		20000002	Invalid parameter
		2000000D	Invalid handle
		2000000F	Node guarding disabled
		2010xxxx	Internal error xxxx = VRTXsa error code

Description

This function block activates the node guarding function for the specified node. The time between two guarding telegrams is specified as a multiple of the pre-set guarding timebase (see *CoM_Add_Node* function block).



If COM_GUARDING_ON is called up sequentially for the network subscribers, the guarding telegrams will also be sent sequentially. This can result in a burst of guarding telegrams. To avoid, this we recommend calling up COM_GUARDING_ON at approx. 50 ms intervals.

5.2.5. Deactivating node guarding (COM_GUARDING_OFF)

COM_GUARDING_OFF

Declaration

```
FUNCTION_BLOCK COM_GUARDING_OFF
VAR_INPUT
    NODEHANDLE : WORD;
END_VAR
VAR_OUTPUT
    ERROR : DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	NODEHANDLE		Handle of logged-on node
Output parameters	ERROR (hex)	0	Function successfully executed
		20000002	Unknown handle

Description

This function block deactivates the node guarding function for the specified node.

5.2.6. Checking node guarding (COM_GUARDING_MSG)

COM_GUARDING_MSG

Declaration

```
FUNCTION_BLOCK COM_GUARDING_MSG
VAR_INPUT
    NODEHANDLE : WORD;
END_VAR
VAR_OUTPUT
    MESSAGE : DWORD;
    ERROR : DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	NODEHANDLE		Handle of logged-on node
Output parameters	MESSAGE (hex)	0	No message to hand
		20000008	Guarding – timeout error
		20000009	Guarding – toggle bit defective
		2000000A	Guarding – state has changed
	ERROR (hex)	0	Function successfully executed
		20000002	Invalid handle
		2000000F	Node guarding disabled

Description

The node guarding function checks here to see if a requested guarding telegram has been sent back correctly by the slave node. If no telegram is received from the slave node, a timeout error is signalled. A signalling message is also generated where the toggle bit in the telegram is not set to the expected value, or if the state of the node has been changed. The *COM_GUARDING_MSG* function block can be used to ask if there is any such message to hand for the specified node.

5.3. SDO Manager

5.3.1. Installing SDO (COM_INST_SDO)

COM_INST_SDO

Declaration

```

FUNCTION_BLOCK COM_INST_SDO
VAR_INPUT
    CHANNEL      :      INT;
    NODEID       :      WORD;
    RECID        :      WORD;
    TRAIID       :      WORD;
END_VAR
VAR_OUTPUT
    SDOHANDLE    :      WORD;
    ERROR        :      DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
	NODEID	1..127	Node number
	RECID	0, 1..2047	Automatic issuing of identifier (DS301 V3.0) Identifier for receive telegram (slave→master)
	TRAIID	0, 1..2047	Automatic issuing of identifier (DS301 V3.0) Identifier for transmit telegram (master→slave)
Output parameters	SDOHANDLE		Handle of correctly installed SDO
	ERROR (hex)	0	Function successfully executed
		30000003	Invalid node number (NODEID)
		30000004	Invalid identifier (RECID/TRAIID)
		30000005	CAN port not initialised
		30000006	SDO has already been initialised
		3000000D	Invalid parameter
		30000008	Max. number of SDOs reached

Description

This function block installs an SDO in the internal management structure. Where successfully executed, an SDO handle is sent back to the block.

All SDO communication blocks are based on a confirmed communication protocol. This means that the CANopen master first sends a request telegram with transmit identifier TRAIID. From the slave, the master then receives a response telegram with receive identifier RECID. The identifiers for the transmit telegram and receive telegram may be chosen individually from within a range of 1 to 2047. If 0 is entered as value for RECID/TRAIID, the identifier will automatically be chosen in conformity with CANopen Communication Profile DS 301, version 3.0 (RECID = 1408 + NODEID ; TRAIID = 1536 + NODEID).



A separate instance of this function block is required for each installed SDO. This feature permits the application to communicate via SDO with several slave nodes at the one time.

5.3.2. Transmitting up to 4 data bytes by SDO (COM_SEND_SDO_EDATA)

COM_SEND_SDO_EDATA

```

Declaration      FUNCTION_BLOCK COM_SEND_SDO_EDATA
                  VAR_INPUT
                    DATAPTR      :      DWORD;
                    SDOHANDLE    :      WORD;
                    INDEX       :      WORD;
                    SUBINDEX     :      WORD;
                    DATA        :      ARRAY [0..3] OF BYTE;
                    LENGTH       :      WORD;
                    TIMEOUT      :      DWORD;
                  END_VAR
                  VAR_OUTPUT
                    STATE       :      WORD;
                    ERROR       :      DWORD;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DATAPTR		Address of a data object
	SDOHANDLE		Handle of installed SDO
	INDEX		Taken from object directory in accordance with communications/device profile in use
	SUBINDEX		Sub-index assigned to index
	DATA		Valid if DATAPTR = 0 The number of bytes to be transmitted, indicated in LENGTH, is present at and from index position 0
	LENGTH	1 . . 4	Number of data bytes to be transmitted
	TIMEOUT	0 >0	No delay for response telegram Delay for response telegram (unit: 10 ms)
Output parameters	STATE	0 1 2 3 4 8 9 10	State could not be ascertained -> interpret error code Data transmission in progress Data transmission successfully completed Data transmission has been terminated Data transmission cannot be started because receive function is active. Internal error LENGTH invalid Invalid range for DATAPTR

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output parameters	ERROR (hex)	0	Data transmission successfully completed
		3010xxxx	Internal CAN driver error xxxx = CAN driver error code
		30200001	SDO transfer terminated
		30200002	SDO transfer protocol error
		30200003	Timeout for response telegram
		30200004	Invalid object directory entry
		05xyyyyy	CANopen error class 5 – service error xx = Error code yyyy = Additional code
		06xyyyyy	CANopen error class 6 – access error xx = Error code yyyy = Additional code

Description

This function block transmits a maximum of four data bytes to the specified slave node. The slave node is addressed by means of the SDO-handle whose properties have already been set up in COM_INST_SDO.

INDEX and SUBINDEX describe the data being transmitted. The meaning of this data can be looked up in the appropriate CANopen profiles.

The master first transmits a telegram via the CAN interface, and then waits for the response telegram from the slave. The maximum delay to be tolerated by the master is set up in TIMEOUT. Any exceeding of this time results in a two-stage error handling operation, in which, firstly, the state in question is signalled by sending ERROR, and, secondly, the master sends an abort telegram to the slave.

Variable STATE = 2 indicates to the application that communication has been completed successfully.



This function block permits communication in conformity with the Expedited Domain Download protocol specified in the CANopen communications profile.

5.3.3. Transmitting up to 256 data bytes by SDO (COM_SEND_SDO_DOMAIN)

COM_SEND_SDO_DOMAIN

```

Declaration      FUNCTION_BLOCK COM_SEND_SDO_DOMAIN
                  VAR_INPUT
                    DATAPTR      :      DWORD;
                    SDOHANDLE    :      WORD;
                    INDEX       :      WORD;
                    SUBINDEX    :      WORD;
                    DATA       :      ARRAY [0..255] OF BYTE;
                    LENGTH      :      WORD;
                    TIMEOUT     :      DWORD;
                  END_VAR
                  VAR_OUTPUT
                    STATE       :      WORD;
                    ERROR       :      DWORD;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DATAPTR		Address of a data object
	SDOHANDLE		Handle of installed SDO
	INDEX		Taken from object directory in accordance with communications/device profile in use
	SUBINDEX		Sub-index assigned to index
	DATA		Valid if DATAPTR = 0 The number of data bytes to be transmitted, indicated in LENGTH, is present at and from index position 0
	LENGTH	5 .. 256	Number of data bytes to be transmitted
	TIMEOUT	0 >0	No delay for response telegram Delay for response telegram (unit: 10 ms)
Output parameters	STATE	0 1 2 3 4 8 9 10	State could not be ascertained → interpret error code Data transmission in progress Data transmission successfully completed Data transmission has been terminated Data transmission cannot be started because receive function is active. Internal error LENGTH invalid Invalid range for DATAPTR)

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output parameters	ERROR (hex)	0	Data transmission successfully completed
		3010xxxx	Internal CAN driver error xxxx = CAN driver error code
		30200001	SDO transfer terminated
		30200002	SDO transfer protocol error
		30200003	Timeout for response telegram
		30200004	Invalid object directory entry
		30200005	Invalid toggle bit in telegram
		05xxyyyy	CANopen error class 5 – service error xx = Error code yyyy = Additional code
		06xxyyyy	CANopen error class 6 – access error xx = Error code yyyy = Additional code

Description

This function block transmits a maximum of 256 data bytes to the specified slave node. The slave node is addressed by means of the handle whose properties have already been set up in COM_INST_SDO.

INDEX and SUBINDEX describe the data being transmitted. The meaning of this data can be looked up in the appropriate CANopen profiles.

The master first sends a telegram via the CAN interface, and then waits for the response telegram from the slave. The maximum delay to be tolerated by the master is set up in TIMEOUT. Any exceeding of this time results in a two-stage error handling operation, in which, firstly, the state in question is signalled by sending ERROR, and, secondly, the master sends an abort telegram to the slave.

Variable STATE = 2 indicates to the application that communication has been completed successfully.


NOTICE

This function block permits communication in conformity with the Segmented Domain Download protocol specified in the CANopen communications profile.

5.3.4. Receiving up to 4 data bytes by SDO (COM_RECEIVE_SDO_EDATA)

COM_RECEIVE_SDO_EDATA

```

Declaration      FUNCTION_BLOCK COM_RECEIVE_SDO_EDATA
                    VAR_INPUT
                        DATAPTR      :      DWORD;
                        SDOHANDLE    :      WORD;
                        INDEX       :      WORD;
                        SUBINDEX     :      WORD;
                        TIMEOUT      :      DWORD;
                    END_VAR
                    VAR_OUTPUT
                        DATA       :      ARRAY[0..3] OF BYTE;
                        LENGTH      :      WORD;
                        STATE       :      WORD;
                        ERROR       :      DWORD;
                    END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DATAPTR		Address of a data object
	SDOHANDLE		Handle of installed SDO
	INDEX		Taken from object directory in accordance with communications/device profile in use
	SUBINDEX		Sub-index assigned to index
	TIMEOUT	0 >0	No delay for response telegram Delay for response telegram (unit: 10 ms)
Output parameters	DATA		Valid if DATAPTR = 0 The number of received data bytes, indicated in LENGTH, is present at and from index position 0
	LENGTH	1..4	Number of received data bytes
	STATE	0	State could not be ascertained → interpret error code
		1	Data transmission in progress
		2	Data transmission successfully completed
		3	Data transmission has been terminated
		4	Data transmission cannot be started because receive function is active.
	8	Internal error	
	10	Invalid range for DATAPTR	

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output parameters	ERROR (hex)	0	Data transmission successfully completed
		3010xxxx	Internal CAN driver error xxxx = CAN driver error code
		30200001	SDO transfer terminated
		30200002	SDO transfer protocol error
		30200003	Timeout for response telegram
		30200004	Invalid object directory entry
		05xyyyyy	CANopen error class 5 – service error xx = Error code yyyy = Additional code
		06xyyyyy	CANopen error class 6 – access error xx = Error code yyyy = Additional code

Description

This function block receives a maximum of 4 data bytes from the specified slave node. The slave node is addressed by means of the SDO handle whose properties have already been set up in COM_INST_SDO.

INDEX and SUBINDEX describe the data being transmitted. The meaning of this data can be looked up in the appropriate CANopen profiles.

The master first transmits a telegram via the CAN interface, and then waits for the response telegram from the slave. The maximum delay to be tolerated by the master is set up in TIMEOUT. Any exceeding of this time results in a two-stage error handling operation, in which, firstly, the state in question is signalled by sending ERROR, and, secondly, the master sends an abort telegram to the slave.

Variable STATE = 2 indicates to the application that communication has been completed successfully.



This function block permits communication in conformity with the Expedited Domain Upload protocol specified in the CANopen communications profile.

5.3.5. Receiving up to 256 data bytes by SDO (COM_RECEIVE_SDO_DOMAIN)

COM_RECEIVE_SDO_DOMAIN

```

Declaration      FUNCTION_BLOCK COM_RECEIVE_SDO_DOMAIN
                  VAR_INPUT
                    DATAPTR      :      DWORD;
                    SDOHANDLE    :      WORD;
                    INDEX       :      WORD;
                    SUBINDEX    :      WORD;
                    TIMEOUT     :      DWORD;
                  END_VAR
                  VAR_OUTPUT
                    DATA       :      ARRAY[0..255] OF BYTE;
                    LENGTH     :      WORD;
                    STATE      :      WORD;
                    ERROR      :      DWORD;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DATAPTR		Address of a data object
	SDOHANDLE		Handle of installed SDO
	INDEX		Taken from object directory in accordance with communications/device profile in use
	SUBINDEX		Sub-index assigned to index
	TIMEOUT	0 >0	No delay for response telegram Delay for response telegram (unit: 10 ms)
Output parameters	DATA		Valid if DATAPTR = 0 The number of received data bytes, indicated in LENGTH, is present at and from index position 0
	LENGTH	5..256	Number of received data bytes
	STATE	0	State could not be ascertained → interpret error code
		1	Data transmission in progress
		2	Data transmission successfully completed
	3	Data transmission has been terminated	
	4	Data transmission cannot be started because receive function is active.	
	8	Internal error	
	10	Invalid range for DATAPTR	

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output parameters	ERROR (hex)	0	Data transmission successfully completed
		3010xxxx	Internal CAN driver error xxxx = CAN driver error code
		30200001	SDO transfer terminated
		30200002	SDO transfer protocol error
		30200003	Timeout for response telegram
		30200004	Invalid object directory entry
		05xyyyyy	CANopen error class 5 – service error xx = Error code yyyy = Additional code
		06xyyyyy	CANopen error class 6 – access error xx = Error code yyyy = Additional code

Description

This function block receives a maximum of 256 data bytes from the specified slave node. The slave node is addressed by means of the SDO handle whose properties have already been set up in COM_INST_SDO.

INDEX and SUBINDEX describe the data being transmitted. The meaning of this data can be looked up in the appropriate CANopen profiles.

The master first transmits a telegram via the CAN interface, and then waits for the response telegram from the slave. The maximum delay to be tolerated by the master is set up in TIMEOUT. Any exceeding of this time results in a two-stage error handling operation, in which, firstly, the state in question is signalled by sending ERROR, and, secondly, the master sends an abort telegram to the slave.

Variable STATE = 2 indicates to the application that communication has been completed successfully.



This function block permits communication in conformity with the Segmented Domain Upload protocol specified in the CANopen communications profile.

5.4. PDO Manager

5.4.1. Installing PDO (COM_INST_PDO)

COM_INST_PDO

```

Declaration
FUNCTION_BLOCK COM_INST_PDO
VAR_INPUT
    CHANNEL      :      INT;
    COBID       :      WORD;
    PDOTYPE     :      WORD;
END_VAR
VAR_OUTPUT
    PDOHANDLE   :      WORD;
    ERROR      :      DWORD;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
	COBID	1..2047	Identifier for PDO telegram (also see chapter on Introduction to CANopen)
	PDOTYPE	1, 2	Receive PDO Transmit PDO
Output parameters	PDOHANDLE		Handle of correctly installed PDO
	ERROR (hex)	0	Function successfully executed
		40000003	Invalid identifier
		40000004	PDO has already been installed
		40000006	Maximum number of PDOs reached
		40000008	Invalid parameter
4000000B	CAN port not initialised		

Description

This function block installs a PDO in the internal management structure. A PDO handle is sent back to the block when installation is completed successfully. PDOTYPE indicates to the CANopen master whether the initialised PDO is a receive PDO, or a transmit PDO. The identifiers for transmit PDOs and receive PDOs may be chosen individually from within a range of 1 to 2047 (COBID). The PDO identifier should be chosen in conformity with CANopen Communication Profile DS 301, version 3.0, which presents issued identifiers from the slave device's viewpoint. The following is a brief summary of identifiers, showing the two receive PDOs and two transmit PDOs defined in the profile.

PDO1: PDOTYPE: receive/slave transmit: 384 + node number
 PDO1: PDOTYPE: transmit/slave receive: 512 + node number
 PDO2: PDOTYPE: receive/slave transmit: 640 + node number
 PDO2: PDOTYPE: transmit/slave receive : 768 + node number



A separate instance of this function block is required for each installed PDO.

5.4.2. Transmitting PDO telegram (COM_SEND_PDO)

COM_SEND_PDO

Declaration

```

FUNCTION_BLOCK COM_SEND_PDO
VAR_INPUT
    DATAPTR      :      DWORD;
    PDOHANDLE    :      WORD;
    DATA        :      ARRAY[0..7] OF BYTE;
    LENGTH       :      WORD;
END_VAR
VAR_OUTPUT
    STATE        :      WORD;
    ERROR        :      DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DATAPTR		Address of a data object
	PDOHANDLE		Handle of installed PDO
	DATA		Valid if DATAPTR = 0 The number of data bytes to be transmitted, indicated in LENGTH, is present at and from index position 0
	LENGTH	0 . . 7	Number of data bytes to be transmitted
Output parameters	STATE	0	State could not be ascertained → interpret error code
		8	Internal error
		9	LENGTH invalid
		10	Invalid range for DATAPTR
	ERROR (hex)	0	Function successfully executed
		40000007	Unknown handle
		40000008	Invalid parameter
		4000000A	Not a transmit PDO
	4010xxxx	Internal CAN driver error xxxx = internal CAN driver error code	

Description

This function block transmits a maximum of 8 data bytes by PDO telegram. The corresponding properties must first be set up in COM_INST_PDO.

5.4.3. Requesting PDO telegram (COM_REQUEST_PDO)

COM_REQUEST_PDO

```

Declaration      FUNCTION_BLOCK COM_REQUEST_PDO
                    VAR_INPUT
                        PDOHANDLE    :    WORD;
                    END_VAR
                    VAR_OUTPUT
                        ERROR    :    DWORD;
                    END_VAR
  
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PDOHANDLE		Handle of installed PDO
Output parameters	ERROR (hex)	0	Function successfully executed
		40000007	Unknown handle
		4000000A	Remote Request not allowed
		4010xxxx	Internal CAN driver error xxxx = internal CAN driver error code

Description This function block transmits a request for a PDO telegram. The relevant properties must first have been set up in COM_INST_PDO. The requested PDO must be installed as receive PDO to allow this request to operate by Remote Frame. The response to the request can then be received by COM_RECEIVE_PDO.

5.4.4. Receiving PDO telegram (COM_RECEIVE_PDO)

COM_RECEIVE_PDO

Declaration

```

FUNCTION_BLOCK COM_RECEIVE_PDO
VAR_INPUT
    DATAPTR      :    DWORD;
    PDOHANDLE    :    WORD;
END_VAR
VAR_OUTPUT
    STATE       :    WORD;
    DATA       :    ARRAY[0..7] OF BYTE;
    LENGTH      :    WORD;
    ERROR       :    DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DATAPTR		Address of a data object
	PDOHANDLE		Handle of installed PDO
Output parameters	STATE	0	No PDO received
		1	PDO received
		8	Internal error
		10	Invalid range for DATAPTR
	DATA		Valid if DATAPTR = 0 The number of received data bytes, indicated in LENGTH, is present at and from index position 0
	LENGTH	0..7	Number of data bytes to be transmitted
	ERROR (hex)	0	Function successfully executed
	40000007	Unknown handle	
	4000000A	Not a receive PDO	
	4000000C	Buffer overrun	

Description

This function block receives at the most 8 data bytes by PDO telegram. The relevant properties must first be set up in COM_INST_PDO.



This block has an internal data buffer for ten telegrams per installed PDO. Where communications traffic is heavier, the application should make sure the data buffer does not overrun. This could be done, for example, by calling the block several times during a cycle.

5.4.5. Updating the data of a synchronous transmit PDO (COM_UPDATE_PDO)

COM_UPDATE_PDO

Declaration

```

FUNCTION_BLOCK COM_UPDATE_PDO
VAR_INPUT
    DATAPTR      :      DWORD;
    PDOHANDLE    :      WORD;
    DATA        :      ARRAY [0..7] OF BYTE;
    LENGTH       :      WORD;
END_VAR
VAR_OUTPUT
    STATE       :      WORD;
    ERROR       :      DWORD;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DATAPTR		Address of a data object
	PDOHANDLE		Handle of installed PDO
	DATA		Valid if DATAPTR = 0 The number of data bytes to be transmitted, indicated in LENGTH, is present at and from index position 0
	LENGTH	0..7	Number of data bytes to be transmitted
Output parameters	STATE	0	State could not be ascertained → interpret error code
		8	Internal error
		9	LENGTH invalid
		10	Invalid range for DATAPTR
	ERROR (hex)	0	Function successfully executed
		40000007	Unknown handle
		40000008	Invalid parameter
	4000000A	Not a transmit PDO	
	4010xxxx	Internal CAN driver error xxxx = internal CAN driver error code	

Description

This function block updates the max. 8 data bytes of a synchronous transmit PDO. The relevant properties must first be set up in COM_INST_PDO. It is also essential to set up the exact synchronous transmission mode in advance, using function-block COM_CFG_SYNC_PDO.



Synchronous PDOs are only transmitted after their data content has been updated with COM_UPDATE_PDO.

5.4.6. Configuring a synchronous transmit PDO (COM_CFG_SYNC_PDO)

COM_CFG_SYNC_PDO

```

Declaration      FUNCTION_BLOCK COM_CFG_SYNC_PDO
                    VAR_INPUT
                        PDOHANDLE      :      WORD;
                        TRATYPE        :      WORD;
                        TRIGGER        :      WORD;
                    END_VAR
                    VAR_OUTPUT
                        ERROR          :      DWORD;
                    END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PDOHANDLE		Handle of installed transmit PDO
	TRATYPE	0 . . 240	Number of sync telegrams between renewed transmissions of synchronous transmit PDO
	TRIGGER	0 . . TRATYPE-1	Time-offset transmission of PDO telegrams with same TRATYPE.
Output parameters	ERROR (hex)	0	Function successfully executed
		40000007	Unknown handle
		40000008	Invalid parameter
		4000000A	Not a transmit PDO

Description This function block configures a synchronous transmit PDO. The TRA_TYPE parameter indicates how many SYNC signals lie between two transmitted telegrams. The TRIGGER parameter may be used to transmit different transmit PDOs of the same transmission type at staggered intervals.

NOTICE	<p>Sync telegrams are used to interchange data between CANopen devices at identical intervals of time, making it possible to create a process image. It may sometimes be necessary to transmit certain data frequently, while other data need only be sent less often.</p> <p><u>Example:</u> The sync telegram is transmitted every 50 ms for digital output modules, so that these modules can obtain their output data in that same timeframe. Where various different analog devices are to receive data at the same time, an update period of 200 ms would normally be sufficient. This is where the TRATYPE variable comes into play. Using TRATYPE, the synchronous transmit PDO can be configured so that the transmit PDO for analog data will only be sent with every fourth sync telegram.</p> <p>If several other analog transmit PDOs of this kind are also to be transmitted, they may be distributed over the four available sync telegrams. The appropriate distribution can be set up with the variable TRIGGER.</p>
---------------	---

Sync	01	02	03	04	05	06	07	08	09	10	11	12	TRATYPE	TRIGGER
PDO1	X	X	X	X	X	X	X	X	X	X	X	X	0	0
PDO2				X				X				X	4	0
PDO3			X				X				X		4	3

5.5. SYNC Manager

5.5.1. Installing sync telegram (COM_INST_SYNC)

COM_INST_SYNC

Declaration

```
FUNCTION_BLOCK COM_INST_SYNC
VAR_INPUT
    CHANNEL      :      INT;
    COBID :      WORD;
    SYNCTIME     :      WORD;
END_VAR
VAR_OUTPUT
    SYNCHANDLE   :      WORD;
    ERROR :      DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
	COBID	0 1..2047	Identifier: 128 conforming to DS301 V3.0 Identifier for the sync telegram
	SYNCTIME	0 >= 1	Receive external sync telegram Period in which sync telegram is to be transmitted (unit 5 ms)
Output parameters	SYNCHANDLE		Handle of correctly installed SYNC
	ERROR (hex)	0	Function successfully executed
		50000003	Invalid identifier
		50000004	SYNC has already been installed
		50000005	Insufficient memory
		50000007	Invalid parameter
		50000008	CAN port not installed

Description

This function block installs the SYNC telegram of a CAN interface in the internal management structure. A SYNC handle is sent back to the block on successful execution. SYNC signals can be received or transmitted over each one of the three possible CAN ports. The time base for generation of SYNC signals is 5 ms. The COBID for the SYNC telegram may be chosen at will from within a range of 1 to 2047. If 0 is entered as value for ID, the identifier will be set to 128 in conformity with CANopen Communication Profile DS 301, version 3.0.



A separate instance of this function block is required for each installed SYNC.

5.5.2. Starting sync telegram (COM_START_SYNC)

COM_START_SYNC

Declaration

```
FUNCTION_BLOCK COM_START_SYNC
VAR_INPUT
    SYNCHANDLE : WORD;
END_VAR
VAR_OUTPUT
    ERROR : DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	SYNCHANDLE		Handle of correctly installed SYNC
Output parameters	ERROR (hex)	0	Function successfully executed
		50000006	Unknown handle

Description This function block starts the processing of SYNC telegrams

5.5.3. Stopping sync telegram (COM_STOP_SYNC)

COM_STOP_SYNC

Declaration

```
FUNCTION_BLOCK COM_STOP_SYNC
VAR_INPUT
    SYNCHANDLE : WORD;
END_VAR
VAR_OUTPUT
    ERROR : DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	SYNCHANDLE		Handle of correctly installed SYNC
Output parameters	ERROR (hex)	0	Function successfully executed
		50000006	Unknown handle

Description This function block stops the processing of SYNC telegrams

5.6. Error Codes

The following chapter describes a selection of the error codes to be found in the function blocks of the CANopen library.

5.6.1. CANopen error codes

The following error codes stem from CANopen Communication Profile V4.00. The CANopen services of the CANopen master may display error codes like these for two main reasons. Firstly, a service that has been transmitted by the master may contain data whose type, format or content does not suit the addressed slave. Secondly, the slave may be in a device state in which it is unable, or is not permitted, to perform particular instructions at the time in question. Note always that this kind of error code will have been transmitted by the slave.

Abort code (hex)	Description
0503 0000	Toggle bit not alternated.
0504 0000	SDO protocol timed out.
0504 0001	Client/server command specifier not valid or unknown.
0504 0002	Invalid block size (block mode only).
0504 0003	Invalid sequence number (block mode only).
0504 0004	CRC error (block mode only).
0504 0005	Out of memory.
0601 0000	Unsupported access to an object.
0601 0001	Attempt to read a write only object.
0601 0002	Attempt to write a read only object.
0602 0000	Object does not exist in the object dictionary.
0604 0041	Object cannot be mapped to the PDO.
0604 0042	The number and length of the objects to be mapped would exceed PDO length.
0604 0043	General parameter incompatibility reason.
0604 0047	General internal incompatibility in the device.
0606 0000	Access failed due to a hardware error.
0607 0010	Data type does not match, length of service parameter does not match.
0607 0012	Data type does not match, length of service parameter too high.
0607 0013	Data type does not match, length of service parameter too low.
0609 0011	Sub-index does not exist.
0609 0030	Value range of parameter exceeded (only for write access).
0609 0031	Value of parameter written too high.
0609 0032	Value of parameter written too low.
0609 0036	Maximum value is less than minimum value.
0800 0000	General error.

Abort code (hex)	Description
0800 0020	Data cannot be transferred or stored to the application.
0800 0021	Data cannot be transferred or stored to the application because of local control.
0800 0022	Data cannot be transferred or stored to the application because of the present device state.
0800 0023	Object dictionary dynamic generation fails or no object dictionary is present (e.g. object dictionary is generated from file and generation fails because of a file error).

5.6.2. Internal CAN driver errors xxxx = internal error code

The two lowest-order bytes of the CANopen error code (when calling the COM_PORT_RESTART function block, for example) contain additional information which may assist with error location. These bytes are generated by the CAN driver (CANDRV). This software layer underlies the CANopen library, and performs the actual operation of accessing the CAN chip.

This error code for the CAN driver is recognisable from the place-holder 'xxxx'.

Code xxxx (hex)	Meaning
0000	No errors
0001	Driver function cannot be executed
0002	Wrong parameter
0003	CAN chip – "bus-off"
0004	Identifier already defined
0020	Transmit buffer full

5.6.3. Internal error xxxx = VRTXsa error code

These error codes are generated by the VRTXsa operating system, and indicate errors in the software driver. Please take a note of the code and report it to our Support Service along with the corresponding error description.

5.6.4. Error handling

In principle, errors signalled by the CANopen library are divisible into three classes. Class one comprises errors attributable to the program itself (for example, incorrect data length, invalid interface). These errors can be rectified by making the appropriate changes to the program.

- The second class of errors may occur during ongoing operations. Characteristically, these will consist of errors in the communications system, for example a defective or briefly disconnected cable, or a communications partner switched off, and they especially comprise the CANDRV errors referred to above.
- Provided it is permitted to do so by the application, the program can react to these errors with due tolerance. Following electrical repair work, the program can resume its activities again after renewed initialisation of its interfaces.
- The third class of errors is caused by incorrect operation of CANopen slave devices. These errors are identified by the CANopen error codes described above. Normally, they too are rectified by making a software change either at the CP1131 end, or in the slave device.

6. String conversion (STRCNV.LIB)

6.1.1. Converting DINT to STRING (CNV_DINT_TO_STR)

CNV_DINT_TO_STR

Declaration

```

FUNCTION_BLOCK CNV_DINT_TO_STR
VAR_INPUT
    NUMBER      :      DINT;
    DIGIT       :      INT;
    SIGN        :      BOOL;
    LEFTALIGN   :      BOOL;
    ZEROFILL    :      BOOL;
    HEXFORMAT   :      BOOL;
END_VAR
VAR_OUTPUT
    STATE       :      INT;
    NUMSTRING   :      STRING;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	NUMBER		Number to be converted
	DIGIT		Number of digits to be created
	SIGN		Sign also for positive numbers
	LEFTALIGN		Left-aligned
	ZEROFILL		Leading zeros
	HEXFORMAT		Hexadecimal notation
Output parameters	STATE	0	No errors
		1	Too few digits to represent number
	NUMSTRING		Converted value in STRING
Description	This function block converts a value of DINT type into a STRING of desired length.		

NOTICE

The number of digits (identified by DIGITS) must be at least one higher than the number of figures (figures + sign).

The SIGN flag is ignored when numbers are shown in hexadecimal form, because hexadecimal numbers do not have an explicit sign. Negative numbers are shown in two's complement notation.

The following examples show the results (in "<>" brackets) obtained with differently set Boolean variables.

```

NUMBER = 12345 DIGITS = 9
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 0 < 12345>
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 0 < +12345>
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 0 <12345 >
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 0 <000012345>
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 1 < 3039>
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 0 <+12345 >
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 0 <+00012345>
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 1 < 3039>
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 0 <12345 >
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 1 <3039 >
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 1 <000003039>
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 0 <+12345 >
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 1 <3039 >
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 1 <000003039>
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 1 <3039 >
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 1 <3039 >

```

```

NUMBER = -12345 DIGITS = 9
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 0 < -12345>
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 0 < -12345>
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 0 <-12345 >
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 0 <-00012345>
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 1 < FFFFCFC7>
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 0 <-12345 >
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 0 <-00012345>
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 0 HEXFORMAT = 1 < FFFFCFC7>
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 0 <-12345 >
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 1 <FFFCFC7 >
SIGN = 0 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 1 <0FFFCFC7>
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 0 <-12345 >
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 0 HEXFORMAT = 1 <FFFCFC7 >
SIGN = 1 LEFTALIGN = 0 ZEROFILL = 1 HEXFORMAT = 1 <0FFFCFC7>
SIGN = 0 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 1 <FFFCFC7 >
SIGN = 1 LEFTALIGN = 1 ZEROFILL = 1 HEXFORMAT = 1 <FFFCFC7 >

```

6.1.2. Converting REAL to STRING (CNV_REAL_TO_STR)

CNV_REAL_TO_STR

Declaration

```

FUNCTION_BLOCK CNV_REAL_TO_STR
VAR_INPUT
    REALNUMBER : REAL;
    ALLDIGITS   : INT;
    DECIDIGITS  : INT;
    SIGN        : BOOL;
    LEFTALIGN   : BOOL;
    EXPFORMAT   : BOOL;
END_VAR
VAR_OUTPUT
    STATE : INT;
    REALSTRING: STRING;
END_VAR
    
```

	Parameter	Value	Description
Input parameters	REALNUMBER		REAL number to be converted
	ALLDIGITS		Number of digits to be created (minimum)
	DECIDIGITS		Exact number of decimal digits
	SIGN		Sign also for positive numbers
	LEFTALIGN		Left-aligned
	EXPFORMAT		Exponential format
Output parameters	STATE	0	No errors
		1	Too few digits to represent number
	REALSTRING		Converted value in STRING

Description

This function block converts a value of REAL type into a STRING of desired length.

i NOTICE

The variables ALLDIGITS and DECIDIGITS indicate the minimum number of digits that should be created. If the number contains more digits than specified, all digits will be jointly accommodated in the string.

Examples:

```

REALNUMBER = -123.456001 ALLDIGITS = 20 DECDIGITS = 9
SIGN = 0 LEFTALIGN = 0 EXPFORMAT = 0 < -123.456001282>
SIGN = 1 LEFTALIGN = 0 EXPFORMAT = 0 < -123.456001282>
SIGN = 0 LEFTALIGN = 1 EXPFORMAT = 0 <-123.456001282 >
SIGN = 0 LEFTALIGN = 0 EXPFORMAT = 1 < -1.234560013E+002>
SIGN = 1 LEFTALIGN = 1 EXPFORMAT = 0 <-123.456001282 >
SIGN = 1 LEFTALIGN = 0 EXPFORMAT = 1 < -1.234560013E+002>
SIGN = 0 LEFTALIGN = 1 EXPFORMAT = 1 <-1.234560013E+002 >
SIGN = 1 LEFTALIGN = 1 EXPFORMAT = 1 <-1.234560013E+002 >

REALNUMBER = 123.456001 ALLDIGITS = 20 DECDIGITS = 9
SIGN = 0 LEFTALIGN = 0 EXPFORMAT = 0 < 123.456001282>
SIGN = 1 LEFTALIGN = 0 EXPFORMAT = 0 < +123.456001282>
SIGN = 0 LEFTALIGN = 1 EXPFORMAT = 0 <123.456001282 >
SIGN = 0 LEFTALIGN = 0 EXPFORMAT = 1 < 1.234560013E+002>
SIGN = 1 LEFTALIGN = 1 EXPFORMAT = 0 <+123.456001282 >
SIGN = 1 LEFTALIGN = 0 EXPFORMAT = 1 < +1.234560013E+002>
SIGN = 0 LEFTALIGN = 1 EXPFORMAT = 1 <1.234560013E+002 >
SIGN = 1 LEFTALIGN = 1 EXPFORMAT = 1 <+1.234560013E+002 >
    
```

6.1.3. Converting STRING to DINT (CNV_STR_TO_DINT)

CNV_STR_TO_DINT

```

Declaration      FUNCTION_BLOCK CNV_STR_TO_DINT
                  VAR_INPUT
                    NUMSTRING      :      STRING;
                  END_VAR
                  VAR_OUTPUT
                    STATE           :      INT;
                    NUMBER          :      DINT;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	REALSTRING		STRING containing a DINT number
Output parameters	STATE	0	No errors
		1	No DINT values convertible from STRING
	REALNUMBER		DINT value from STRING

Description This function block extracts a DINT value from a string.



The number contained in the string must be of type INT or DINT. Hexadecimal numbers cannot be converted.

6.1.4. Converting STRING to REAL (CNV_STR_TO_REAL)

CNV_STR_TO_REAL

```

Declaration      FUNCTION_BLOCK CNV_STR_TO_REAL
                  VAR_INPUT
                    REALSTRING     :      STRING;
                  END_VAR
                  VAR_OUTPUT
                    STATE           :      INT;
                    REALNUMBER     :      REAL;
                  END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	REALSTRING		STRING containing a REAL number
Output parameters	STATE	0	No errors
		1	No real value convertible from STRING
	REALNUMBER		REAL value from STRING

Description This function block extracts a REAL value from a string.

6.1.5. Converting STRING to array (CNV_STR_TO_ARRAY)

CNV_STR_TO_ARRAY

```

Declaration      FUNCTION_BLOCK CNV_STR_TO_ARRAY
                  VAR_INPUT
                    BYTESTRING :    STRING;
                    LENGTH      :    INT;
                  END_VAR
                  VAR_OUTPUT
                    STATE :    INT;
                    BYTEARRAY  :    ARRAY[0..80] OF BYTE;
                  END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	BYTESTRING		String comprising individual characters (BYTES)
	LENGTH	1..80	Number of characters to be copied from the string to the array
Output parameters	STATE	0	No errors
		1	LENGTH invalid
	BYTEARRAY		Array with the characters of the STRING

Description This function block files the individual characters of a string in an array (ARRAY) of BYTE type.

6.1.6. Converting array to STRING (CNV_ARRAY_TO_STR)

CNV_ARRAY_TO_STR

```

Declaration      FUNCTION_BLOCK CNV_ARRAY_TO_STR
                  VAR_INPUT
                    BYTEARRAY  :    ARRAY[0..80] OF BYTE;
                    LENGTH      :    INT;
                  END_VAR
                  VAR_OUTPUT
                    STATE :    INT;
                    BYTESTRING :    STRING;
                  END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	BYTEARRAY		Array with individual characters
	LENGTH	1..80	Number of characters to be filed in the string
Output parameters	STATE	0	No errors
		1	LENGTH invalid
	BYTESTRING		String comprising the individual characters of the array

Description This function block files the individual characters of an array in a string.

6.1.7. Converting real string to decimal number(CNV_REALSTR_TO_DINT)

CNV_REALSTR_TO_DINT

Declaration

```

FUNCTION_BLOCK CNV_REALSTR_TO_DINT
VAR_INPUT
    REALSTRING :      STRING;
    MULTIPLIKATOR :    DINT;
    DIVISOR :        DINT;
END_VAR
VAR_OUTPUT
    STATE :      INT;
    NUMBER :     DINT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	REALSTRING		String comprising individual characters (BYTES)
	MULTIPLIKATOR		Multiplier of real number
	DIVISOR		Divisor of real number
Output parameters	STATE	0	No errors
		1	No real value convertible from real string
		2	Division by zero
	NUMBER		Result of calculation (REAL * MULTIPLIKATOR)/DIVISOR

Description

This function block converts a string, comprising a real number, into a decimal number. The real number contained in the string is multiplied by the MULTIPLIER and divided by the DIVISOR. The result is stored as 32-bit decimal number, with decimal places truncated. The two operators MULTIPLIER and DIVISOR represent scaling factors.

7. TCP/UDP Protocol Driver via IP

7.1. TCP_UDP_V01.LIB

The library has been expanded to include the IP_TCP_DISCONNECT_TIMEOUT call-up.

If cycle errors occasionally occur on the Ethernet cell controller, these may be caused by calling up the IP_TCP_DISCONNECT function block. The function block forces the closure of a TCP link. If there are still data in the send buffer at this time, the system attempts to transmit them to the recipient but, if necessary, will terminate this action with a TIMEOUT. Up to firmware versions < V2.10 this TIMEOUT is set at 2 seconds. Naturally, the TIMEOUT affects the cycle time monitoring and, when the max. cycle time default setting of 1 second is used, can result in cycle times being exceeded. Therefore, the IP_TCP_DISCONNECT_TIMEOUT function block was introduced as of firmware version V2.10. Here, the TIMEOUT can be defined in one-second intervals. A wait time which will guarantee rapid processing (approx. 30 ms when the buffer is full) can also be defined. However, in this case all data in the buffer will be deleted immediately.

Alternately, the cycle time can also be changed. The new 'SYSTIME_V01.LIB' library has been provided for this. Unfortunately, the previous 'SYSTIME.LIB' version contained an error precisely where the cycle time setting was concerned.

7.1.1. Introduction

The function blocks described in this document permit data interchange based on the TCP/IP and UDP/IP protocol, using modules with Ethernet interface.

The following describes the function blocks required to exchange data blocks by UDP telegram between information source and sink.

An interface with the network should be declared to begin with, using function block IP_UDP_CREATE_CONNECTOR – an unused UDP port will be needed for this purpose. Ports occupied by the system are described later on in the document. Initialisation having been completed successfully, a HANDLE is returned by means of which other UDP function blocks can identify the interface.

Using the interface identified by HANDLE, function block IP_UDP_WRITE transmits data to a remote station. The remote station is uniquely defined by PEERIP and PORTNUMBER, while the data block to be transmitted is defined by the start address DATAPTR, and the size indicated in the DATASIZE block.

IP_UDP_STATE or IP_UDP_READ are used to detect the reception of telegrams from an interface.

IP_UDP_STATE checks the interface identified by HANDLE, to find if a new telegram can be transmitted, or if a telegram has been received. SEND_READY is set to TRUE if a new telegram can be transmitted. The size of the received telegram is indicated in RXCOUNT, and RX_PAKET is also set to TRUE.



If function block IP_UDP_STATE is the first to be called when a data block has been received, the block is copied into a buffer memory in the interface so that its size can be determined. The maximum acceptable size of data block is 2048 bytes in this case, the remaining data in any larger blocks being discarded by the system. Any bigger data blocks than this can only be received by using function block IP_UDP_READ, which has no restrictions in respect of block size.

The buffer memory of the interface identified by HANDLE can be cleared with function block IP_UDP_CLEAR_BUFFER.

A data block received through the interface identified by HANDLE is copied to the specified address (DATAPTR) when using function block IP_UDP_READ. The maximum amount of data written is specified by DATASIZE and displayed by RXCOUNT. The data packet is discarded after the function block is called, so if the data block is any larger than DATASIZE, the uncopied portion of the data in the data packet will be lost.



The only way to avoid the discarding of data where DATASIZE is too small is to use IP_UDP_STATE to wait for reception of a packet, and leave the calling of IP_UDP_READ until then.

In this case, the data can be read in several stages from the 2048-byte-sized buffer memory. The continued presence of data in the buffer memory is indicated by STATE = 258.

An interface can be closed and the occupied UDP-port released again by calling IP_UDP_DELETE_CONNECTOR.

7.1.2. Transmission Control Protocol (TCP)

The function blocks for TCP are divided into the following sub-areas:

Client function block:

- IP_TCP_CONNECT_TO_SERVER

Connection function blocks:

- IP_TCP_WRITE
- IP_TCP_STATE
- IP_TCP_READ
- IP_TCP_DISCONNECT

Server function blocks:

- IP_TCP_REGISTER_SERVER
- IP_TCP_SERVERSTATE
- IP_TCP_CONNECT_NEW_CLIENT
- IP_TCP_UNREGISTER_SERVER

A server is installed by function block IP_TCP_REGISTER_SERVER at port PORTNUMBER, provided no other server is operating there, and waits for a client to establish connection through the port. A HANDLE is returned so that this server can be operated by other function blocks too.

Function block IP_TCP_SERVERSTATE signals a client's establishment of a connection – through a server port identified by HANDLE – by setting NEW_CLIENT to TRUE. The connection is not yet actually available at this stage, because function block IP_TCP_CONNECT_NEW_CLIENT must first be called to arrange issue of a new CLIENTHANDLE by means of which the connection can be addressed. The HANDLE returned by IP_TCP_CONNECT_NEW_CLIENT can be used for all of the connection function blocks.

A server can be deleted again by IP_TCP_UNREGISTER_SERVER – all connections established through that server then remain intact.

A connection to the server is established by function block IP_TCP_CONNECT_TO_SERVER, the server being selected through address SERVERHOSTIP and server port PORTNUMBER. This function block initiates a connection attempt without waiting for reactions from the server. The procedure returns a HANDLE which is needed so that other TCP connection function blocks can identify the connection in question.

NOTICE

A connection set-up times out after 75 seconds, or may be rejected or confirmed by the server.

The state of a connection can be checked with IP_TCP_STATE, HANDLE indicating the connection that is to be checked. CONNECT is set to TRUE once the connection is established. SEND_READY is set to TRUE if a new data block can be transmitted. RX_PAKET set to TRUE indicates that a data block has been transmitted from a remote station.

NOTICE

Releasing or clearing of the connection by the remote station can only be detected by reading or writing data.

IP_TCP_READ writes a data block, received through the connection identified by HANDLE, to the specified address DATAPTR. The maximum amount of data written is determined by DATASIZE and indicated in RXCOUNT.

Function block IP_TCP_WRITE transmits data to the remote station over the connection identified by HANDLE. The data block being transmitted is defined by start address DATAPTR and the size of the DATASIZE block.

The connection is cleared down with IP_TCP_Disconnect.

7.1.3. Internet Protocol (IP)

The module's IP address can be set up by the CNW.

During the life of the application, this address can be read out by function block IP_OWN_NUMBER, and the MAC address can be read by IP_OWN_MAC_ADR.

7.1.4. Ports used

UDP port 7000 is occupied in the system.

Ports 1 to 1024 are reserved ports that are used by system applications, and should therefore be avoided in so far as possible.

7.1.5. System restrictions

The maximum total number of connections and interfaces (TCP and UDP) is limited to 8.

NOTICE

If a TCP connection is cleared down by the remote station, this can only be registered by a write or read access.

7.1.6. General aspects

The IP address is represented in array [0..3] of bytes, with the first byte of the array corresponding to the first IP address byte, so IP address 10.1.2.3 is represented as [0]=10; [1]=1; [2]=2; [3]=3.

NOTICE

Where a function block is ended with STATE other than 0, all its other output parameters remain undefined.

7.2. UDP/IP Function Blocks

7.2.1. Clearing the interface's buffer memory (IP_UDP_CLEAR_BUFFER)

IP_UDP_CLEAR_BUFFER

```

Declaration      FUNCTION_BLOCK IP_UDP_CLEAR_BUFFER
                  VAR_INPUT
                    HANDLE      :      WORD;
                  END_VAR
                  VAR_OUTPUT
                    STATE :      INT;
                  END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
Output parameters	STATE	0	No errors
		5	The interface indicated by HANDLE is not of UDP type

Description This function block deletes the received data block held in the buffer memory, but not any other data blocks present in the system.

7.2.2. Setting up a UDP interface (IP_UDP_CREATE_CONNECTOR)

IP_UDP_CREATE_CONNECTOR

Declaration

```

FUNCTION_BLOCK IP_UDP_CREATE_CONNECTOR
VAR_INPUT
    PORTNUMBER : WORD;
END_VAR
VAR_OUTPUT
    STATE : INT;
    HANDLE : WORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORTNUMBER	0-65535	UDP/IP port through which packets are to be received
Output parameters	HANDLE	0-255	Unique key for this interface
	STATE	0	No errors
		2	The specified port number is occupied
		3	Too many interfaces opened
		6	Internal error
		30	Error when setting up (creating) interface

Description This function block sets up a UDP/IP interface at the specified port. A HANDLE is issued for this interface so that other function blocks can address it unambiguously.



If 0 is entered as PORTNUMBER, the system searches for an available port number

7.2.3. Closing the interface (IP_UDP_DELETE_CONNECTOR)

IP_UDP_DELETE_CONNECTOR

Declaration

```
FUNCTION_BLOCK IP_UDP_DELETE_CONNECTOR
VAR_INPUT
    HANDLE      :      WORD;
END_VAR
VAR_OUTPUT
    STATE      :      INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0 - 255	Interface's unique key
Output parameters	STATE	0	No errors
		5	The interface identified by HANDLE is not of UDP type

Description

This function block closes the UDP/IP interface specified by HANDLE. Data still present in the transmit memory is transmitted before the interface is closed.

7.2.4. Reading a data block from an interface (IP_UDP_READ)

IP_UDP_READ

Declaration

```

FUNCTION_BLOCK IP_UDP_READ
VAR_INPUT
    HANDLE      :      WORD;
    DATAPTR     :      DWORD;
    DATASIZE    :      WORD;
END_VAR
VAR_OUTPUT
    STATE      :      INT;
    PEERIP     :      ARRAY [0..3] OF BYTE;
    PORTNUMBER :      WORD;
    RXCOUNT   :      WORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>	
Input parameters	HANDLE	0-255	Interface's unique key	
	DATAPTR		Start address at and from which a received data block is filed.	
	DATASIZE	1-65535	Maximum size of data block to be received, given in bytes.	
Output value	STATE	0	No errors	
		5	Interface identified by HANDLE is not of UDP type.	
		8	The specified start address is invalid, the size of the data block is too large, or the data block extends beyond the admissible data area.	
		258	Data still present in receive memory	
	PEERIP	259	No data received	
		0-255, 0-255, 0-255, 0-255	Sender's IP address	
		PORTNUMBER	1-65535	Sender's IP port number
		RXCOUNT	0-65535	Size of copied, received data block in bytes.

Description

This function block writes a received data block to the specified data memory, beginning at start address DATAPTR.

If the data memory is not large enough to store all of the data received, the copying operation is limited to DATASIZE bytes. Where the received data is present in the buffer memory and not all data has been copied, STATE = 258 indicates that there is still some data in the buffer memory; if the data packet is not in the buffer memory, any of the packet's characters that have not actually been read are discarded. The number of bytes copied is displayed in RXCOUNT.

7.2.5. State of an interface (IP_UDP_STATE)

IP_UDP_STATE

Declaration

```

FUNCTION_BLOCK IP_UDP_STATE
VAR_INPUT
    HANDLE      :      WORD;
END_VAR
VAR_OUTPUT
    STATE      :      INT;
    RXCOUNT   :      WORD;
    RX_PAKET   :      BOOL;
    SEND_READY :      BOOL;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
Output parameters	RXCOUNT	0-2048	Number of bytes held in receive buffer
	RX_PAKET	TRUE	Data in receive buffer
		FALSE	No data in receive buffer
	SEND_READY	TRUE	No data in transmit memory
		FALSE	Still data in transmit memory
	STATE	0	No errors
		5	Interface identified by HANDLE is not of UDP type
		6	Internal error

Description

This function block determines the state of the specified interface. If the interface has received a data packet, RX_PAKET is set to TRUE, and the size of the packet (in bytes) is indicated in RXCOUNT.

The function block copies a received data packet to a temporary buffer memory for the purpose of determining the size of the packet. The maximum packet size accepted is 2048 bytes, any larger packets being limited to this maximum packet size. IP_UDP_STATE cannot be used if it is intended to receive larger data packets. IP_UDP_READ has no restrictions in respect of data packet size, and may be used instead of IP_UDP_STATE.

7.2.6. Writing data (IP_UDP_WRITE)

IP_UDP_WRITE

Declaration

```

FUNCTION_BLOCK IP_UDP_WRITE
VAR_INPUT
    HANDLE      :      WORD;
    DATAPTR     :      DWORD;
    DATASIZE    :      WORD;
    PEERIP      : ARRAY [0..3] OF BYTE;
    PORTNUMBER  :      WORD;
END_VAR
VAR_OUTPUT
    STATE :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
	DATAPTR		Start address of data block being transmitted
	DATASIZE	1-65535	Size of block being transmitted, given in bytes.
	PEERIP		Receiver's IP address
	PORTNUMBER	0-65535	Receiver's IP port number
Output parameters	STATE	0	No errors
		4	Insufficient temporary system memory available
		5	The interface identified by HANDLE is not of UDP type
		6	Internal error
		8	The specified start address is invalid, the data block is too large in size, or the data block extends beyond the valid data area
		10	Only part of data packet could be transmitted.
		13	There is still a data packet in the transmit memory, the new data packet has not been transmitted.

Description

This function block transmits a data block of size DATASIZE (beginning at start address DATAPTR) to subscriber PEERIP/PORTNUMBER. Completed transmission of the block is signalled by STATE = 0. If an error occurs, its cause is described in STATE

7.3. TCP/IP Function Blocks - Server functions

7.3.1. Confirming a new client connection (IP_TCP_CONNECT_NEW_CLIENT)

IP_TCP_CONNECT_NEW_CLIENT

Declaration

```

FUNCTION_BLOCK IP_TCP_CONNECT_NEW_CLIENT
VAR_INPUT
    HANDLE      :      WORD;
END_VAR
VAR_OUTPUT
    STATE :      INT;
    CLIENTIP   :      ARRAY [0..3] OF BYTE;
    CLIENTPORTNUMBER :      WORD;
    CLIENTHANDLE :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Unique key for the server interface
Output parameters	STATE	0 3 5 6 11	No errors Too many connections open. The interface identified by HANDLE is not a TCP server Internal error No connection request pending from a client
	CLIENTIP	0-255, 0-255, 0-255, 0-255	Client's IP address
	CLIENTPORTNUMBER	1-65535	Client's port number
	CLIENTHANDLE	0-255	Unique key for new connection.

Description

This function block confirms a connection from a client and gives that connection a key (handle).

A new key cannot be allocated if too many interfaces are open, but the connection in question remains in suspense until one is in fact allocated.

7.3.2. Setting up a server (IP_TCP_REGISTER_SERVER)

IP_TCP_REGISTER_SERVER

```

Declaration      FUNCTION_BLOCK IP_TCP_REGISTER_SERVER
                    VAR_INPUT
                        PORTNUMBER :    WORD;
                    END_VAR
                    VAR_OUTPUT
                        STATE :    INT;
                        HANDLE    :    WORD;
                    END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	PORTNUMBER	1-65535	Number of IP port at which the server is to be set up.
Output parameters	STATE	0 1 2 3 6	No errors Error when setting up interface Port is occupied Too many interfaces/connections open Internal error
	HANDLE	0-255	Unique key for server set up.

Description The function block sets up a TCP server so that it will accept connection requests from clients through the relevant PORTNUMBER.

7.3.3. Determining the state of a TCP server (IP_TCP_SERVERSTATE)

IP_TCP_SERVERSTATE

Declaration

```

FUNCTION_BLOCK IP_TCP_SERVERSTATE
VAR_INPUT
    HANDLE      :      WORD;
END_VAR
VAR_OUTPUT
    STATE      :      INT;
    NEW_CLIENT  :      BOOL;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Unique key for the server interface
Output parameters	STATE	0	No errors
		5	The interface identified by HANDLE is not a TCP server
		6	Internal error
	NEW_CLIENT	TRUE	A client has established a new connection
	FALSE	No new connection established	

Description This function block checks to see if a TCP client has established a connection through the specified server interface.

7.3.4. Closing a TCP server (IP_TCP_UNREGISTER_SERVER)

IP_TCP_UNREGISTER_SERVER

Declaration

```

FUNCTION_BLOCK IP_TCP_UNREGISTER_SERVER
VAR_INPUT
    HANDLE      :      WORD;
END_VAR
VAR_OUTPUT
    STATE :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
Output parameters	STATE	0	No errors
		5	The interface identified by HANDLE is not a TCP server

Description This function block closes the specified TCP server. Connections established through that server remain intact.

7.4. TCP/IP Function Blocks - Client Functions

7.4.1. Establishing a connection to a server (IP_TCP_CONNECT_TO_SERVER)

IP_TCP_CONNECT_TO_SERVER

Declaration

```

FUNCTION_BLOCK IP_TCP_CONNECT_TO_SERVER
VAR_INPUT
    SERVERHOSTIP      :    ARRAY [0..3] OF BYTE;
    PORTNUMBER        :    WORD;
END_VAR
VAR_OUTPUT
    STATE             :    INT;
    HANDLE            :    WORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	SERVERHOSTIP	0-255, 0-255, 0-255, 0-255	Server's IP address
	PORTNUMBER	0-65535	Server's TCP/IP port number
Output parameters	HANDLE	0-255	Unique key for this interface
	STATE	0	No errors
		2	The specified port number is already occupied
		3	Too many interfaces open
		4	Insufficient temporary system memory available
6	Internal error		

Description

This function block attempts to establish a TCP connection to the specified server.

Because it can take some time to establish, the connection may not yet be actually set up straight after calling the function block.

The connection from the server is not actually established until function block IP_TCP_STATE CONNECT signals TRUE.

7.5. TCP/IP Function Blocks - Connection Functions

7.5.1. Clearing down a connection (IP_TCP_DISCONNECT)

IP_TCP_DISCONNECT

```

Declaration      FUNCTION_BLOCK IP_TCP_DISCONNECT
                  VAR_INPUT
                    HANDLE      :      WORD;
                  END_VAR
                  VAR_OUTPUT
                    STATE :      INT;
                  END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
Output parameters	STATE	0	No errors
		5	The connection identified by HANDLE is not a TCP connection

Description This function block closes the specified connection. Any data held in the transmit memory is transmitted. If the recipient can no longer accept data, the TIMEOUT has been predefined at 2 seconds. Attention: This can result in cycle errors.

7.5.2. Timeout Control During Line Disconnection (IP_TCP_DISCONNECT_TIMEOUT)

IP_TCP_DISCONNECT_TIMEOUT

Declaration

```

FUNCTION_BLOCK IP_TCP_DISCONNECT_TIMEOUT
VAR_INPUT
    HANDLE : WORD;
    TIMEOUT: DINT
END_VAR
VAR_OUTPUT
    STATE  : INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
	TIMEOUT	0-255	TIMEOUT in seconds
Output parameters	STATE	0	No errors
		5	The connection identified by HANDLE is not a TCP connection

Description

Setting the TIMEOUT for IP_TCP_DISCONNECT.
 No waiting time (Value 0) can also be set, which will guarantee rapid processing (approx. 30 ms when the buffer is full) can also be defined. However, in this case all data in the buffer will be deleted immediately.

7.5.3. Reading data (IP_TCP_READ)

IP_TCP_READ

Declaration

```

FUNCTION_BLOCK IP_TCP_READ
VAR_INPUT
    HANDLE      :      WORD;
    DATAPTR     :      DWORD;
    DATASIZE    :      WORD;
END_VAR
VAR_OUTPUT
    STATE      :      INT;
    RXCOUNT   :      WORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
	DATAPTR		Start address at and from which a received data block is filed.
	DATASIZE	1-65535	Maximum size of data block to be received, given in bytes.
Output value	STATE	0	No errors
		5	The connection identified by HANDLE is not a TCP connection.
		6	Internal error has occurred
		7	Connection has been closed.
		8	The specified start address is invalid, the data block is too large in size, or the data block extends beyond the valid data area.
		12	Connection has timed out.
		14	Connection not yet established
		259	No data received
RXCOUNT	0-65535	Size of copied, received data block in bytes.	

Description

This function block writes a received data block to the specified data memory, beginning at start address DATAPTR.

At the most DATASIZE bytes of the received data packet are copied, the number of copied bytes being displayed in RXCOUNT.

7.5.4. Determining state of a connection (IP_TCP_STATE)

IP_TCP_STATE

Declaration

```

FUNCTION_BLOCK IP_TCP_STATE
VAR_INPUT
    HANDLE      :      WORD;
END_VAR
VAR_OUTPUT
    STATE      :      INT;
    RX_PAKET   :      BOOL;
    CONNECT    :      BOOL;
    SEND_READY :      BOOL;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0 - 255	Interface's unique key
Output parameters	RX_PAKET	TRUE	Data in receive memory
		FALSE	No data in receive memory
	CONNECT	TRUE	Connection confirmed by remote station.
		FALSE	Connection not yet established, or cleared down again.
	SEND_READY	TRUE	No data in transmit memory
		FALSE	Transmit memory still contains data
	STATE	0	No errors
		5	The connection identified by HANDLE is not a TCP connection.
		6	Internal error
		7	Connection has been closed.
	12	Connection has timed out.	
	260	Connection to TCP server could not be established.	

Description

This function block determines the state of the specified interface. RX_PAKET is set to TRUE if the interface has received a data packet.

A connection clear-down can only be detected if signalled in function blocks IP_TCP_READ or IP_TCP_WRITE after they have been called.

7.5.5. Transmitting data (IP_TCP_WRITE)

IP_TCP_WRITE

Declaration

```

FUNCTION_BLOCK IP_TCP_WRITE
VAR_INPUT
    HANDLE      :      WORD;
    DATAPTR     :      DWORD;
    DATASIZE    :      WORD;
END_VAR
VAR_OUTPUT
    STATE :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	HANDLE	0-255	Interface's unique key
	DATAPTR		Start address at and from which the data to be transmitted is stored.
	DATASIZE	1-65535	Size of data block to be transmitted, given in bytes.
Output value	STATE	0	No errors
		4	Insufficient temporary system memory available
		5	The connection identified by HANDLE is not a TCP connection.
		6	Internal error
		7	Connection has been closed.
		8	The specified start address is invalid, the data block is too large in size, or the data block extends beyond the valid data area.
		10	Only part of data packet could be transmitted.
		12	Connection has timed out.
	13	Transmit memory still contains a data packet, new data packet not transmitted.	
	14	Connection not yet established	

Description

Beginning at the start address, this function block transmits a data block of size DATASIZE over the specified connection. STATE = 0 signals that the block has been transmitted successfully. If an error occurs, its cause is described in STATE.

7.6. IP Function Blocks

7.6.1. Reading own IP address (IP_OWN_NUMBER)

IP_OWN_NUMBER

Declaration

```
FUNCTION_BLOCK IP_OWN_NUMBER
VAR_INPUT
END_VAR
VAR_OUTPUT
    STATE :      INT;
    HOSTIP  :      ARRAY [0..3] OF BYTE;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output parameters	HOSTIP	0-255, 0-255, 0-255, 0-255	Module's IP address
	STATE	0 50	No errors Function not available on this module

Description

The module's IP address can be set up by the CNW (Control Node Wizard), version 1.40 and later. During the life of the application, this address can be read off by using function block IP_OWN_NUMBER.

7.6.2. Reading own Ethernet MAC address (IP_OWN_MAC_ADR)

IP_OWN_MAC_ADR

Declaration

```
FUNCTION_BLOCK IP_OWN_NUMBER
VAR_INPUT
END_VAR
VAR_OUTPUT
    STATE :      INT;
    MACADDR :      ARRAY [0..5] OF BYTE;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output parameters	MACADDR	0-255, 0-255, 0-255, 0-255, 0-255, 0-255	Module's MAC address
	STATE	0 50	No errors Function not available on this module

Description

This function block returns the module's MAC address. The highest-order byte of the MAC address is stored in MACADDR[0], the lowest-order in MACADDR[5].

7.6.3. Warning codes

STATE	Warnings
258	Receive memory still contains data.
259	No data received.
260	Connection to TCP server not yet established.

7.6.4. Error codes

STATE	Error Message
1	Error when setting up interface.
2	Specified port number is occupied.
3	Too many interfaces/connections open.
4	Insufficient temporary system memory available, close interface/connection.
5	The interface identified by HANDLE is not of UDP type. The connection identified by HANDLE is not a TCP connection. The interface identified by HANDLE is not a TCP server.
6	Internal error.
7	Connection has been closed.
8	Specified start address is invalid, data block too large in size, or data block extending beyond valid data area.
10	Only part of data packet could be transmitted.
11	No connection requested by a client.
12	Connection has timed out.
13	Transmit memory still contains a data packet, the new data packet has not been transmitted.
14	Connection not yet established.
30	Error when setting up interface.
50	Function not available on this module.

8. Cycle Time (SYSTIME.LIB)

8.1. SYSTIME_V01.LIB

In the SYSTIME.LIB library, the data type of the PLC_SETMAXCYCLETIME function block call-up was "integer" (INT). However, the associated firmware expects a DWORD data type. This is why the change to the max. cycle time was not processed correctly. The call-up parameter's data type has therefore now been changed to DWORD.

8.1.1. Introduction

The cycle time is the time the controller requires to process the cyclical user program. The cycle time comprises the sum of the program parts called in cycle, the IEC61131 tasks additionally performed during the cycle, processing times for interruptions and communication, and the time required to produce the process image of inputs and outputs. The minimum cycle time on a cell controller is ≤ 1 msec. The maximum cycle time is a measure of the user program's response time. The cycle time is monitored. The controller goes into error stop mode if the maximum cycle time is exceeded. The cycle monitoring time is set to one second by default.

A series of statistical data on cycle duration (cycle statistics) can be recorded for the purpose of estimating the cycle time required by a user program. Library functions are available to assist with initialising cycle statistics, reading off statistical data, and enabling, disabling or resetting the recording of statistical data.

Provision is also made for setting up the cycle monitoring time.

Cyclical-statistics data is recorded to within a precision of $\pm 250 \mu\text{s}$.
The maximum cycle time monitoring facility operates to within a precision of 10 ms.

8.2. IEC61131 User Interface

8.2.1. Setting up the cycle monitoring time (PLC_SetMaxCycleTime)

PLC_SetMaxCycleTime

```

Declaration      FUNCTION_BLOCK PLC_SetMaxCycleTime
                  VAR_INPUT
                    MaxCycleTime      :      DWORD;
                  END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	MaxCycleTime		New cycle monitoring time in milliseconds

Description This function block can be used to change the cycle monitoring time and, with that, to re-define the maximum admissible cycle time. A cycle monitoring time of one second is set up by default. The maximum cycle time monitoring facility operates to within a precision of 10 ms.

8.2.2. Retriggering cycle monitoring (PLC_RetriggerCycleTime)

PLC_RetriggerCycleTime

```

Declaration      FUNCTION_BLOCK PLC_RetriggerCycleTime
                  no parameters

```

Description When called, this function block re-triggers the cycle monitoring time, meaning it re-starts the timer for the monitoring operation. From the time the function block is called, the maximum admissible cycle time is extended (for the current cycle) by the value set up for the cycle monitoring time. Note also that the cycle in which this function block is called is not included in the cycle time statistics.

8.2.6. Reading cycle statistics (PLC_GetCycleStatistics)

PLC_GetCycleStatistics

```

Declaration      FUNCTION_BLOCK PLC_GetCycleStatistics
                    VAR_OUTPUT
                        CycleCounter      :    DWORD;
                        CurrCycleTime     :    DWORD;
                        LastCycleTime     :    DWORD;
                        MinCycleTime      :    DWORD;
                        MaxCycleTime      :    DWORD;
                        AverageCycleTime  :    DWORD;
                    END_VAR
    
```

<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output value CycleCounter		Number of cycles performed since most recent initialisation of cycle statistics
CurrCycleTime		The cycle time elapsed since the last cycle boundary in the cycle currently being processed (in µsec)
LastCycleTime		The duration of the preceding cycle (in µsec)
MinCycleTime		The minimum cycle time since the most recent initialisation of the cycle statistics (in µsec)
MaxCycleTime		The maximum cycle time since the most recent initialisation of the cycle statistics (in µsec)
AverageCycleTime		The average value for cycle time (in µsec)

Description All values recorded in the cycle statistics can be read off by calling this function block. The statistical data comprises:

- the number of cycles that have been performed since the most recent initialisation of the cycle statistics
- the cycle time elapsed since the last cycle boundary in the cycle currently being processed
- the duration of the preceding cycle
- the minimum and the maximum cycle time since the most recent initialisation of the cycle statistics
- the average value for cycle time. Calculation of the average value is based on the cycle times of the last 512 cycles recorded in the statistics.

The minimum, maximum and average value for cycle time are updated only when the cycle time statistics feature is activated. The number of cycles, the duration of the preceding cycle, and the time elapsed in the current cycle are updated on all occasions, whether or not the cycle time statistics feature is activated.

Calculating the average value:

Calculation of the average value is based on the following algorithm: Each time the statistical data is updated, the value recorded for the most recent cycle is stored in a system-internal cyclic buffer. This buffer can accommodate a maximum of 512 values. Once the buffer is full, the oldest value in it is dropped and the respective new value is stored. In addition, each time the data is updated, totals are produced for the values (max. 512) contained in the buffer. When function block PLC_GetCycleStatistics is called, the average value is calculated by dividing the totals by the number of values stored in the buffer. In practice, this means that, as a rule, the average value is always derived from the cycle times of the most recent 512 cycles.



The cycle statistics facility only records "normal" cycles. If the recording of a cycle period has been distorted by re-triggering or re-starting the cycle monitoring time, for example, the data on that cycle will not be included in the statistics. This prevents the statistics from being influenced by "maverick" values. However, this does mean that if, say, the cycle monitoring time is repeatedly re-triggered, little or no data will be recorded for statistical purposes.

8.2.7. Reading the system time (SYS_GETTIMER)**SYS_GETTIMER****Declaration**

```
FUNCTION_BLOCK SYS_GETTIMER
VAR_OUTPUT
    TIMER :      DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output value	TIMER		Current system time in μsec

Description

The system provides an internal timer having a resolution of 250 μsec . This timer can be read off by calling this function block, to let the user make very precise measurements of time.

Blank page

9. TPU Blocks for Rapid Inputs/Outputs

The following function blocks have been specially designed for components in which the Motorola Time Processing Unit (TPU) is accessible to the user. The TPU is a self-programmable co-processor specially devised for rapid I/Os and for measurement of time. The controller's system programs use the TPU for this purpose, but it does still have some unused inputs and outputs which can therefore be made available to the user.

9.1. Discrete I/O programming of TPU (TPUDIO.LIB)

9.1.1. Initialising a TPU channel as digital input or output (DIO_INIT)

DIO_INIT

Declaration

```
FUNCTION_BLOCK DIO_INIT
VAR_INPUT
    CHANNEL      :    INT;
    INTERFACE    :    BYTE;
END_VAR
VAR_OUTPUT
    STATE       :    INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 - 7	CHANNEL 0-3 corresponding to OUT 1-4 CHANNEL 4-7 corresponding to IN 1-4
	INTERFACE	0	Signal from/to V422 interface
		1	Signal from/to 24 V interface
Output parameters	STATE	0	Function successfully executed
		1	No TPU channels available on module
		2	CHANNEL invalid
		3	Module does not have the selected INTERFACE
		4	Error when initialising, call function block again.
	5	Internal error	

Description

This function block installs a digital input or output on the selected CHANNEL. The signal path is indicated by INTERFACE. Any other TPU feature that may have been active on this CHANNEL is deactivated.

9.1.2. Ending discrete programming of inputs or outputs (DIO_CLOSE)

DIO_CLOSE

Declaration

```
FUNCTION_BLOCK DIO_CLOSE
VAR_INPUT
    CHANNEL      :      INT;
END_VAR
VAR_OUTPUT
    STATE :      INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 - 7	CHANNEL 0-3 corresponding to OUT 1-4 CHANNEL 4-7 corresponding to IN 1-4
Output parameters	STATE	0	Function successfully executed
		1	No TPU channels available on module
		2	CHANNEL invalid
		3	CHANNEL is not a digital input/output

Description

This function block ends the input/output facility on the selected CHANNEL. If the selected input/output was a 24 V port, it is restored to the normal I/O system.

9.1.3. Setting an output (DIO_SET)

DIO_SET

```

Declaration      FUNCTION_BLOCK DIO_SET
                    VAR_INPUT
                        CHANNEL      :      INT;
                        DATA       :      BOOL;
                    END_VAR
                    VAR_OUTPUT
                        STATE        :      INT;
                    END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 - 3	Select desired output
	DATA	FALSE, TRUE	Level of output
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	CHANNEL is not a digital input/output

Description This function block writes the value of DATA to the output designated in CHANNEL.

9.1.4. Reading an input (DIO_GET)

DIO_GET

```

Declaration      FUNCTION_BLOCK DIO_GET
                    VAR_INPUT
                        CHANNEL      :      INT;
                    END_VAR
                    VAR_OUTPUT
                        DATA       :      BOOL;
                        STATE        :      INT;
                    END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	4 - 7	Select desired input
Output parameters	DATA	FALSE, TRUE	Level of selected input
	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
	3	CHANNEL is not a digital input/output	

Description This function block reads the value of DATA to the input designated in CHANNEL.

9.2. PWM Generation (TPUPWM.LIB)

9.2.1. Setting up a PWM generator (PWM_INIT)

PWM_INIT

Declaration

```

FUNCTION_BLOCK PWM_INIT
VAR_INPUT
    CHANNEL      :      INT;
    DESTINATION  :      BYTE;
    PERIOD       :      REAL;
    HIGHTIME     :      REAL;
END_VAR
VAR_OUTPUT
    STATE :      INT;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 - 3	Outputs OUT 1-4
	DESTINATION	0	Output of signal at V422 interface
		1	Output of signal at 24 V interface
	PERIOD		Period length in seconds
	HIGHTIME		Duration of ones signal in seconds
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	Parameter incorrect
		4	Occurrence of error when initialising (call function block again)
		5	Module does not have a PWM generator

Description

This function sets up a PWM generator on the channel identified by CHANNEL. The PWM generator can operate within two ranges, the first of which achieves a period length of 10µs to 5 ms with a resolution of 160 ns, while the second has a period length of 10µs to 80 ms with a resolution of 3 µs. The duration of the HIGH signal within a period length is limited only by the resolution of the range in question.

Where PERIOD = HIGHTIME, the output is HIGH;
 where HIGHTIME = 0, the output is LOW.

Depending on the particular application, the module may produce a measurable jitter where the values for PERIOD are < 100µs.

The range providing the best resolution is chosen when initialising the generator. If a 24 V interface is selected as output, it is deactivated in the IO system.

9.2.2. Changing a PWM parameter (PWM_CHANGE)

PWM_CHANGE

Declaration

```

FUNCTION_BLOCK PWM_CHANGE
VAR_INPUT
    CHANNEL      :    INT;
    PERIOD       :    REAL;
    HIGHTIME     :    REAL;
END_VAR
VAR_OUTPUT
    STATE       :    INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 – 3	Number of PWM generator
	PERIOD		Period length in seconds
	HIGHTIME		Duration of ones signal in seconds
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	No PWM generator installed
	4	One or more parameters are incorrect	

Description

This function block changes a parameter for a PWM generator. The change takes effect at the beginning of the next period.



This function block cannot be used to change the range of a PWM generator.

9.2.3. Ending PWM generator (PWM_CLOSE)

PWM_CLOSE

Declaration

```
FUNCTION_BLOCK PWM_CLOSE
VAR_INPUT
    CHANNEL      :      INT;
END_VAR
VAR_OUTPUT
    STATE :      INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 - 3	Number of PWM generator
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	CHANNEL not installed as PWM generator

Description

This function block stops the specified PWM generator. If a 24 V interface had been specified as output, it is restored to the IO system.

9.3. Counters (COUNTER.LIB)

9.3.1. Setting up a counter (COUNTER_INIT)

COUNTER_INIT

Declaration

```

FUNCTION_BLOCK COUNTER_INIT
VAR_INPUT
    CHANNEL      :    INT;
    SOURCE       :    INT;
    EDGE        :    INT;
END_VAR
VAR_OUTPUT
    STATE       :    INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	4 - 7	Inputs IN 1-4
	SOURCE	0	Input of signal at V422 interface
		1	Input of signal at 24 V interface
		2	Input of signal at 24 V interface
	EDGE	0	Count all edges
		1	Count all rising edges
2		Count all falling edges	
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	Error when initialising, call function block again.
		4	Module has no counter inputs
		5	SOURCE or EDGE outside the admissible value range

Description

This function block sets up a counter on the channel identified by CHANNEL. The counter has a capacity of 16 bits, and overflows from 16#0000FFFF to 16#00000000.

9.3.2. Setting a counter (COUNTER_SET)

COUNTER_SET

```

Declaration      FUNCTION_BLOCK COUNTER_SET
                    VAR_INPUT
                        CHANNEL      :      INT;
                        DATA       :      DWORD;
                    END_VAR
                    VAR_OUTPUT
                        STATE        :      INT;
                    END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	4 - 7	Inputs IN 1-4
	DATA		Value to which counter is set.
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	Input not installed as counter
		5	DATA outside admissible value range

Description This function block sets a counter to the value indicated in DATA.

9.3.3. Reading a counter (COUNTER_GET)

COUNTER_GET

```

Declaration      FUNCTION_BLOCK COUNTER_GET
                    VAR_INPUT
                        CHANNEL      :      INT;
                    END_VAR
                    VAR_OUTPUT
                        STATE        :      INT;
                        COUNT       :      DWORD;
                    END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	4 - 7	Inputs IN 1-4
Output parameters	COUNT		Value of counter
	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
	3	Input not installed as counter	

Description This function block reads the counter-value of the channel identified by CHANNEL.

9.4. Quadrature Encoder (POSREL.LIB)

9.4.1. Selecting signal source for quadrature encoder (POSITION_SOURCE_SELECT)

ENCODER_SOURCE_SELECT

```

Declaration      FUNCTION_BLOCK POSITION_SOURCE_SELECT
                   VAR_INPUT
                       ENCODER      :      INT;
                       SOURCE       :      INT;
                       CODE        :      INT;
                   END_VAR
                   VAR_OUTPUT
                       STATE       :      INT;
                   END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	ENCODER	0..1	Selection of encoder interface
	SOURCE	0	Signal applied to V422 interface
		1	Signal applied to 24 V interface
Output parameters	CODE	0	Has no function at the present time, set to 0.
	STATE	0	Function successfully executed
		1	Internal error
		2	Value for ENCODER invalid
		3	Module does not support source-selection, or does not provide an input for a quadrature encoder.
	4	Value for SOURCE invalid	

Description This function block can be used to select the signal source for a quadrature encoder. The quadrature encoder's counter is set to 0. Any other function that may have been activated at the inputs occupied by the encoder are ended, and the quadrature encoder is installed.

9.4.2. Setting a quadrature encoder value (POSITION_REL_SET)

POSITION_REL_SET

Declaration

```

FUNCTION_BLOCK POSITION_REL_SET
VAR_INPUT
    ENCODER      :      INT;
    POSITION      :      WORD;
END_VAR
VAR_OUTPUT
    STATE :      INT;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	ENCODER	0 . . 1	Selection of encoder interface
	POSITION		Encoder value which is to be set
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	Value for ENCODER invalid

Description This function block can be used to set the value of the encoder to POSITION.

9.4.3. Reading a quadrature encoder value (POSITION_REL_GET)

ENCODER_SOURCE_GET

Declaration

```

FUNCTION_BLOCK POSITION_REL_GET
VAR_INPUT
    ENCODER      :      INT;
END_VAR
VAR_OUTPUT
    POSITION      :      WORD;
    STATE :      INT;
END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	ENCODER	0 . . 1	Selection of encoder interface
Output parameters	POSITION		The current encoder value
	STATE	0	Function successfully executed
		1	Internal error
		2	ENCODER invalid

Description This function block reads the current value of the encoder into POSITION.

10. Analog Library for Dialog Controllers (AnalogIn.LIB)

10.1. AnalogIn_V01.LIB

The current version is AnalogIn_V01.LIB.

10.1.1. Introduction

The analog library can only be used for the two analog inputs on the CDISP29 and CEDISP29 dialog controllers. The call-up parameters have been based on the QAIO.LIB. The QAIO.LIB is used solely for analog E-bus expansion modules.

10.1.2. Reading the Analog Value (AnalogIn)

ANALOGIN

Declaration

```
FUNCTION_BLOCK ANALOGIN
VAR_INPUT
    POSITION:    INT;
    CHANNEL:   INT;
    MODE:      WORD;
VAR_OUTPUT
    VALUE:     INT;
    STATE:    INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	POSITION	1	First channel
		2	Second channel
	CHANNEL	Always 0	
	MODE		Analog input operating mode:
		0	Current input
		1	Voltage input
Output parameters	VALUE	0 .. 32767	Analog value of the analog input as an integer value with preceding sign.
	STATE	0	Call-up successfully executed
		1	POSITION invalid
		2	No corresponding analog channel detected.
		3	CHANNEL invalid
4		MODE invalid	
	5	Analog channel is still busy switching from a voltage to a current input and can therefore not provide VALUE with a valid analog value.	

Description

This function provides the current analog value of the specified input. The analog converter resolution is 10 bits. The maximum measurable voltage is 10 V. Thus, the following formula can be used to calculate the analog voltage from the VALUE variable:

$$\text{Voltage, in millivolts} = (10000 \text{ mV} * \text{VALUE}) / 32767$$

Thus, the LSB of the AD converter corresponds to a voltage of $10\text{V} / (2^{10} - 1) \approx 9.8 \text{ mV}$, and represents the lower voltage resolution limit!

The maximum measurable current is 20mA. Thus, the following formula can be used to calculate the correct current value with the aid of the VALUE variables:

$$\text{Current, in milliamps} = (20 \text{ mA} * \text{VALUE}) / 32767$$

Thus, the LSB of the AD converter corresponds to a current of $20\text{mA} / (2^{10} - 1) \approx 19.6 \mu\text{A}$, and represents the current voltage resolution limit!

NOTICE

The calculations do not take system-related measurement errors by the AD converter and the associated analog circuit into account!

11. System Information (SysInfo.LIB / SysInfo-CP1131.Lib)

11.1. SysInfo_V02.LIB and SysInfo-CP1131_V01.LIB

The current versions documented here are SysInfo_V02.LIB and SysInfo-CP1131_V01.LIB.

11.1.1. Introduction

The 'SysInfo' libraries provide function blocks with which system information can be queried.

There are two different implementations of the 'SysInfo' libraries:

SysInfo (firmware-based) This is the preferred default version of the library. It is part of the firmware and is faster and more reliable than SysInfo-CP1131.LIB. In addition, this version provides a larger function scope and includes all the necessary information for all supported firmware versions. The SysInfo-CP1131 implementation can be used for older firmware versions.

SysInfo-CP1131

In this version, the library is completely written in CP1131 and can therefore be employed with older firmware versions. However, this implementation provides no guarantee that it contains complete information for all module and firmware combinations as these are determined at runtime and errors can occur. Compared to the firmware-based Sysinfo library, this version contains only a portion of the functionalities.

Functions implemented in both libraries have the same call-up parameters. This allows the SysInfo-CP1131.LIB to be replaced with the SysInfo.LIB without any problems.



All functions blocks conforming to the 'SYSINFO_VERSION_XX' naming convention are used for internal version administration and thus provide no functionalities to the user. Therefore, they are not included in the following description of the function blocks.

11.1.2. Module Type Query (SYSINFO_GET_CPUTYPE)

SYSINFO_GET_CPUTYPE

```

Declaration      FUNCTIONBLOCK SYSINFO_GET_CPUTYPE
                    VAR_OUTPUT
                      id          :      DINT;          (*CPU ID*)
                      name        :      STRING(30);    (*CPU Name*)
                      serialno    :      STRING(14);    (*Serial no as an
                                                         ASCII String*)
                      hwversion   :      WORD;          (*Hardware
                                                         version*)
                      bitfield    :      ARRAY[0..15] OF BYTE; (*Internal*)
                      state       :      SYSINFO_STATES := SYSINFO_STATE_INIT;
                    END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	None		
Output parameters	id		The module type identification number
	name		The module name as an ASCII string
	serialno		The serial number as an ASCII string
	hwversion		The module hardware version
	bitfield		Internal system data
	state		Status of the most recent FB call-up: SYSINFO_STATE_INIT: FB initializing, data <u>not yet</u> valid. SYSINFO_STATE_RUN: Query running, data <u>not yet</u> valid. SYSINFO_STATE_READY: Action successful, returned data are valid. SYSINFO_STATE_NOTFOUND: The desired data could <u>not</u> be determined.
Description	The function block call-up returns the module type, the serial number as well as the module hardware version number.		

11.1.3. Firmware Version Query (SYSINFO_GET_FWVERSION)

SYSINFO_GET_FWVERSION

```

Declaration      FUNCTION_BLOCK SYSINFO_GET_FWVERSION
                   VAR_OUTPUT
                     version      :      BYTE;          (*Primary version*)
                     release      :      BYTE;          (*Sub-version*)
                     revision     :      UINT;          (*Revision, binary*)
                     rev_ascii    :      STRING(2);     (*Revision as ASCII*)
                     text         :      STRING(100);  (*Additional designation*)
                     fwType       :      STRING(50);   (*Firmware type*)
                     state        :      SYSINFO_STATES := SYSINFO_STATE_INIT;
                   END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	None		
Output parameters	version		Primary versions number: e.g., '2' for firmware version 2.40.
	release		Release number: e.g., '40' for firmware version 2.40
	revision		Revision letter as a number. Always '0' for official releases. For beta versions, the ASCII value of the letter (e.g., 16#61 for 'a').
	rev_ascii		Revision letter as a string.
	text		Additional firmware version text. Displayed under 'Comments' in the CNW when the firmware version is queried.
	fwType		Firmware type as an ASCII string. Indicates the firmware type shown under 'Type' in the CNW when the firmware version is queried. Possible values include 'CECPU_S', 'CCPU_S' or 'CCPU_S_SC'
	state		Status of the most recent FB call-up: SYSINFO_STATE_INIT: FB initializing, data <u>not yet</u> valid. SYSINFO_STATE_RUN: Query running, data <u>not yet</u> valid. SYSINFO_STATE_READY: Action successful, returned data are valid. SYSINFO_STATE_NOTFOUND: The desired data could <u>not</u> be determined.

Description This function block permits data related to the firmware to be queried.

11.1.4. Library Data Query (SYSINFO_GET_VERSION)

SYSINFO_GET_VERSION

Declaration

```
FUNCTION_BLOCK SYSINFO_GET_VERSION
VAR_OUTPUT
    version      :      DWORD;
    comment      :      POINTER TO STRING;
END_VAR
```

Input parameters

<i>Parameter</i>	<i>Value</i>	<i>Description</i>
None		

Output parameters

version		Library version, e.g., '2' for SysInfo v02.
comment		Pointer to the string with library comments

Description

This function block permits the version and comments related to the SysInfo library to be queried.

11.1.5. Querying Information from E-Bus Modules (SYSINFO_GET_QBUSMODTYPE)

NOTICE

This FB is only available in the firmware-based SysInfo library.

SYSINFO_GET_QBUSMODTYPE

Declaration

```
FUNCTION_BLOCK SYSINFO_GET_QBUSMODTYPE
VAR_INPUT
    position      :      BYTE;          (*Module position*)
END_VAR
VAR_OUTPUT
    id            :      DINT;          (*Module ID*)
    name          :      STRING(30);    (*Module name*)
    serialno      :      STRING(14);    (*Serial no as an
                                        ASCII string*)
    state         :      SYSINFO_STATES := SYSINFO_STATE_INIT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	position		Valid value range: 0 ... 6. Position of the E-bus module on the E-bus.
Output parameters	id		E-bus module type identification number.
	name		E-bus module name as an ASCII string.
	serialno		E-bus module serial number as an ASCII string.
	state		Status of the most recent FB call-up: SYSINFO_STATE_INIT: FB initializing, data <u>not yet</u> valid. SYSINFO_STATE_RUN: Query running, data <u>not yet</u> valid. SYSINFO_STATE_READY: Action successful, returned data are valid. SYSINFO_STATE_NOTFOUND: The desired data could <u>not</u> be determined (e.g., if there is no module at the specified module position).

Description

This function block permits the serial number and type of an E-bus expansion module to be queried. This FB allows data regarding the connected E-bus expansion modules (also known as QBUS modules) to be queried.

Blank page

12. Application ID (SVCAppId.LIB)

The application ID is used for module identification in the CAN network and only for the extended service channel (extended SVC)

12.1.1. Reading the Application ID (SVC_GET_APPID)

SVC_GET_APPID

Declaration

```
FUNCTION_BLOCK SVC_GET_APPID
VAR_OUTPUT
    appid      :      WORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	None		
Output parameters	appid	1..65535	Module's application ID
Description	This function block returns the module's currently set application ID.		

12.1.2. Writing the Application ID (SVC_SET_APPID)

SVC_SET_APPID

Declaration

```
FUNCTION_BLOCK SVC_SET_APPID
VAR_INPUT
    appid      :      WORD;
END_VAR
VAR_OUTPUT
    status     :      WORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	appid	1..65535	Module's application ID
Output parameters	status	0 1 2 4	Function successfully executed. Transferred value outside the value range Error saving the value to the EEPROM Unknown error
Description	This function block writes the application ID directly to the EEPROM. The new application ID is immediately valid.		

Blank page

13. CP1131 Multitasking (TaskManager.LIB)

NOTICE

Multitasking projects with the Taskmanager.lib can become much more complex to handle than single task projects with PLC_PRG.

We therefore recommend employing multitasking only in very special cases and to contact our Support Service prior to using multitasking.

CP1131 projects can be built up with a different task structure. The default setting is for a single task structure with the cyclic call-up of PLC_PRG as the startup program. From there, the application branches off to the various blocks. However, a multitasking system can also be employed in order to improve application structuring or for faster event reactions. There are two instances of the multitasking system which are differentiated in particular by the task scheduling:

1. **Standard multitasking** (supported by CP1131 (CoDeSys)):

One task is completed, after which the scheduler decides which task will be executed next on the basis of the task priority and the wait time (refer to the description, 'CP1131 User's Manual').
2. **Preemptive multitasking** (accesses the cell controller's operating system):

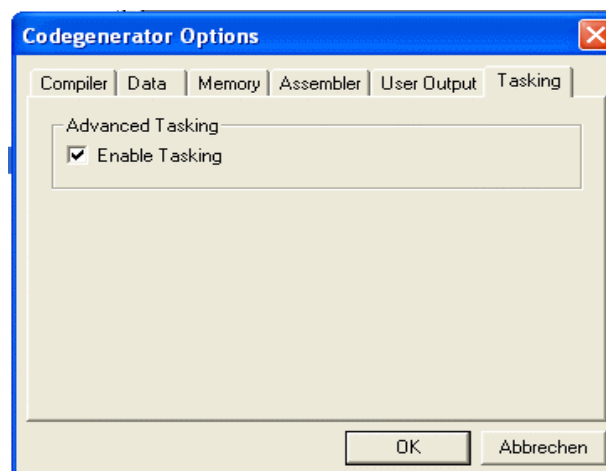
While a task is being executed, the scheduler is activated when a block is called up from within the source code. If at this time (block call-up) a higher priority task is active, the previous task is interrupted and the higher priority task is executed. This allows applications to react to event very quickly despite computing-intensive tasks running.

NOTICE

The code generator options allow the selection of one of the two tasking models. Preemptive multitasking is the default setting (refer to the image below).

Preemptive multitasking is supplemented by the 'TaskManager.lib' CP1131 library. The library contains important functions to support and monitor scheduling. Example: The reaction time to events is not optimal in applications with time-intensive program loops or few block call-ups. The reaction time can be improved by the 'PLC_ReSchedule()' library function, because this function explicitly activates the scheduler. The 'TaskManager.lib' library should therefore be integrated as soon as preemptive multitasking is used.

Code generator options: Tasking



2VF100198DG00_en.gif

NOTICE

The following sections describe the behavior of preemptive multitasking. Standard multitasking is described in the *CP1131 User's Manual*.

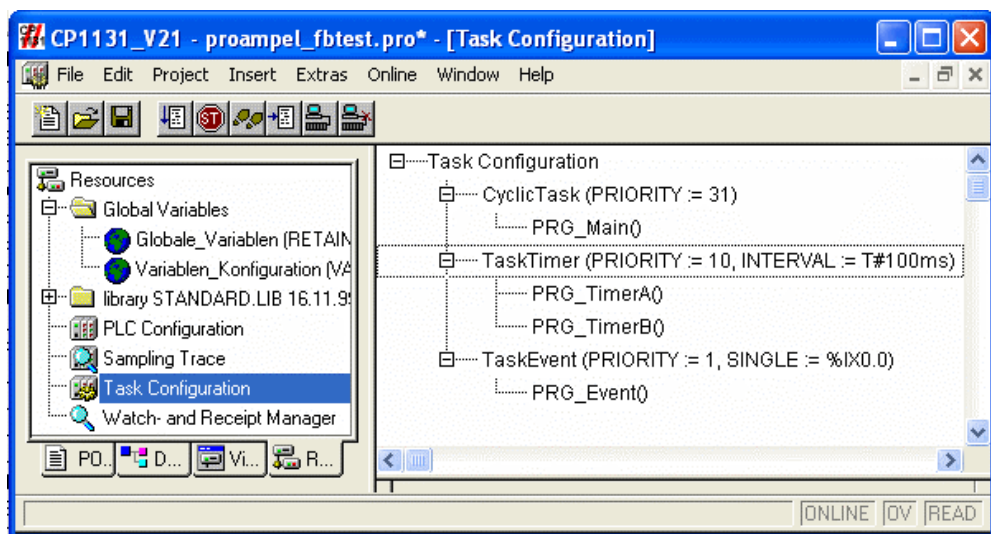
13.1. Task Configuration

Just as is the case with standard multitasking, preemptive multitasking is also based on the CP1131 task configuration (CP1131 register card resources). A task declaration consisting of the following elements is inserted for each task:

- Task designation;
- Task priority;
- Time interval for time-controlled tasks;
- Boolean event for event oriented tasks.

Programs are associated with tasks, that is, as soon as the task is active the associated programs are carried out sequentially.

Example:
Task configuration



2VF100199DG00_en.gif

The above task configuration defines 3 tasks:

- A cyclic task (TASK_ZYKLUS) with the lowest priority of 31.
- A time-controlled task (100 milliseconds) with a medium priority (TASK_TIMER_100MS) and two associated programs.
- An event-controlled task (TASK_HIGHPRIO) with the highest priority of 0 which is activated at input 0.0 with the rising flank.

NOTICE

As soon as multitasking has been activated via the task administration (the declaration of a task is sufficient for this), PLC_PRG becomes a program just like any other. In other words, if PLC_PRG is to be executed, it must be associated with a task.

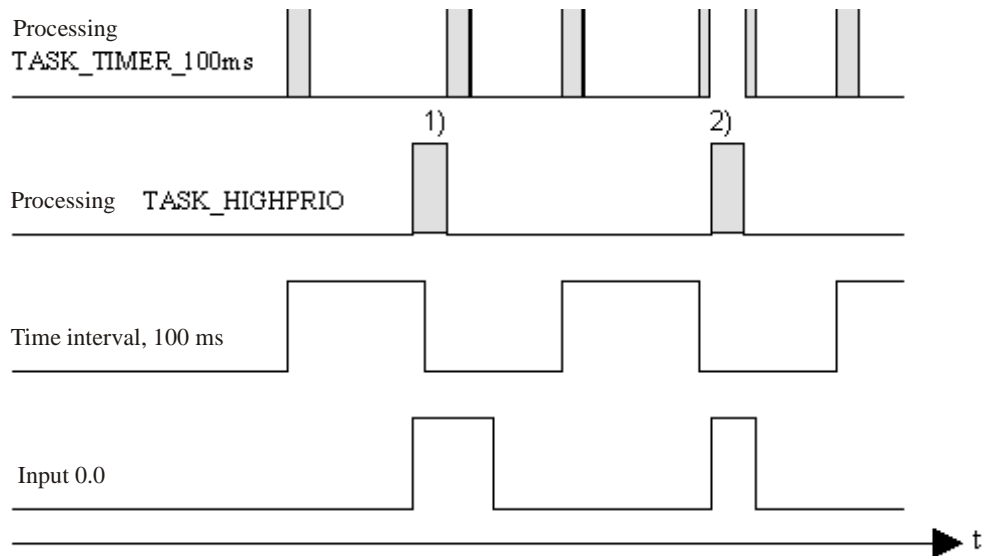
Time-controlled task

A time can be entered in the 'Interval' field to activate the task at equidistant time intervals. Any delays in the execution of the task or the time required to perform the task itself are not included in the calculation of the next activation time point.

In the example, the time-controlled task, 'TASK_TIMER_100MS', is activated every 100 milliseconds. Execution can be delayed by a rising flank at input 0.0 which activates the highest priority task, 'TASK_HIGHPRIO'.

The following chart illustrates the temporal behavior.

Time chart for time-controlled tasks



2VF100200DG00_en.cdr

1. The execution of the interval-controlled tasks is delayed because a higher priority task is currently being executed. Nonetheless, the interval to execute the task will not be extended.
2. The execution of the interval-controlled tasks is interrupted because 'TASK_HIGHPRIO' is being executed. This has no influence on the interval time.



If the execution of a time-controlled task is delayed long enough for the specified start time to be reached again, a call-up error will occur.

The controller goes into the 'STOP' state and the cause of the error (error message: 'LOSTTIMEREVENT') can be evaluated using diagnostics.

If the program processing is interrupted by a "breakpoint", the times for time-controlled tasks remain "frozen" until the controller returns to the 'RUN' mode.

Event-controlled task If an input or a global Boolean property are entered in the “Property” field of the task properties, the task will be activated whenever a rising flank occurs as an event.

Flank evaluation:

1. The evaluation of the digital inputs is independent of the process image. These input data are updated in 1 ms time periods to allow very fast reactions to input events.
The timely call-up of the task is also monitored for this type of task activation. If the input again exhibits a rising flank while execution of the previous task has not yet been completed, a call-up error will be detected and the controller goes to the 'STOP' state.
2. Evaluation of global variables is not executed for each write access to the variable, but rather, only when the scheduler is active, in other words at the end of the cycle or when 'PLC_RESCCHEDULE' is called up.

Cyclic task If neither an interval nor an event are entered in the task properties, this indicates that the requirement for activating the task is always met. Typically, these types of tasks have a low priority and are always executed when no event-controlled or time-controlled task is active. If there are several cyclic tasks with the same priority they will be sequentially processed based on the wait time.

System task The system contains tasks with predefined names. If such a system task is declared, the system event associated with it is used as the execution condition.

The following system task is supported: 'PLC_ONSTART'

This system task is always called up at the start of cyclic program processing. This event can occur after the controller is switched on or when an application program is started for the first time via the CP1131 development environment. All functions which only need to be executed at the controller start can be programmed in this task.

If this task is not declared the system event is ignored.

13.2. The Library Functions

13.2.1. Internal Function (CALLHOOK)

CALLHOOK

Declaration

```
FUNCTION CALLHOOK: BOOL
VAR_INPUT
    POUNR    : WORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	POUNR		Not to be transferred by the user.

Output parameters	CALLHOOK		No significance
--------------------------	----------	--	-----------------

Description This internal function is not called up by the user, but instead it serves code generation as a reference for calling up task administration for function blocks or function call-ups.

13.2.2. Application Program Reaction Time (PLC_GetMaxTaskDelay)

PLC_GetMaxTaskDelay

Declaration

```
FUNCTION PLC_GetMaxTaskDelay : DWORD
VAR_INPUT
    DUMMY    : INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DUMMY	0	In order to ensure compatibility with newer versions this parameter should always be transferred as '0'.

Output parameters	PLC_GetMaxTaskDelay		Time, in ms, corresponding to the maximum delay of a high priority task.
--------------------------	---------------------	--	--

Description This function returns the “worst case” reaction time on the part of the application program with respect to high priority events measured by the system. This time is the longest program runtime between function blocks or function call-ups measured since the program was started. This time represents the maximum time which might pass in the worst case between the occurrence of an event and the execution of the associated task.



The time returned by 'PLC_GetMaxTaskDelay()' is a measured time taken during ongoing operation. However, it can only be viewed as a reference value since only runtimes of program paths which were actually carried out are considered. If the application program contains program paths which result in still worse reaction times but which have not yet been processed, reaction times to events may also be slower.

13.2.3. Recalling the Task Scheduler (PLC_RESCCHEDULE)

PLC_RESCCHEDULE

Declaration

```
FUNCTION PLC_ReSchedule : BOOL
VAR_INPUT
    DUMMY : INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	DUMMY	0	In order to ensure compatibility with newer versions this parameter should always be transferred as '0'.
Output parameters	PLC_ReSchedule		No significance

Description

Calling up the library function also calls up the task administration's scheduler mechanism. On the one hand, this calculates the flanks of all global variables linked to a task and, on the other, runnable high priority tasks are nested. Calling up this function is practical where there are long-running loops. In addition, the function can be used to immediately start a task activated with a rising global variable flank.

Example:

```
PROGRAM PLC_PRG
    IF Condition THEN
        MyBoolVar := TRUE;
        PLC_ReSchedule(0); (*Perform task immediately*)
    END_IF

PROGRAM PRG_EVENT

    MyBoolVar := FALSE;
    .....
```

13.2.4. Task Identification (PLC_GetTaskID)

PLC_GetTaskID

Declaration

```
FUNCTION PLC_GetTaskID : INT
VAR_INPUT
    TaskName : STRING;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	TaskName		Task name in the task configuration
Output parameters	PLC_GetTaskID	0 .. 32767	ID of the 'TaskName' task
		-1	No task named 'TaskName' found

Description

A task identifier is assigned in the controller to each task in the application program. This library function provides the task identifier associated with the specified task name. This task identifier is required in other library functions in order to reference individual tasks

13.2.5. Task Information (PLC_GetTaskInfo)

PLC_GetTaskInfo

Declaration

```
FUNCTION PLC_GetTaskInfo : BOOL
VAR_INPUT
    TaskID : INT;
    Info   : POINTER TO TaskInfo;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	TaskID		Task ID (return value of the 'PLC_GetTaskID' function).
	Info	POINTER	
	.TaskID		Task identifier
	.TaskName		Task name as it was specified in the task configuration.
	.TaskStatus	0	RUNNING: Task is currently being processed.
		1	READY: The task execution condition has been met.
		2	WAITING: The task execution condition has <u>not</u> been met.
		3	SUSPENDED: The task has been suspended; the task execution conditions will <u>not</u> be checked.
	.CountLostEvents		How frequently could a task not be executed in time.
	.TimeLastCall		The system time at which the task was most recently executed.
	.TimeWaiting		The time, in milliseconds, between the execution conditions being met and the actual task execution. If the task has not yet been executed, the time between the event and the call-up of the 'PLC_GetTaskInfo' library function is displayed.
Output parameters	PLC_GetTaskInfo	TRUE	A task with the corresponding task ID exists; the pointer info information is valid.
		FALSE	A task with the corresponding task ID <u>does not</u> exist; the pointer info information is invalid.

Description

The 'Info.CountLostEvents' pointer provides information about how frequently a task was unable to be executed on time before the execution condition was again met. In this case, the event will be lost. This variable provides information about how often events have been lost.

13.2.6. Blocking a Task (PLC_SuspendTask)

PLC_SuspendTask

Declaration

```
FUNCTION PLC_SuspendTask : BOOL
VAR_INPUT
    TaskID : INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	TaskID	0 .. 32767	Task ID (return value of the 'PLC_GetTaskID' function).
Output parameters	PLC_SuspendTask	TRUE	Successful execution, Task with ID 'TaskID' exists.
		FALSE	No task with this ID exists.

Description

This library function suspends the task with the associated task ID. This means the task can no longer be executed until it is recalled by the 'PLC_ResumeTask' library function.

Any events which occur during this time and which would normally result in the task being activated are ignored. No error is indicated nor are the events counted in the 'Info.CountLostEvents' of the 'PLC_GetTaskInfo' element.

13.2.7. Releasing a Task (PLC_ResumeTask)

PLC_ResumeTask

Declaration

```
FUNCTION PLC_ResumeTask : BOOL
VAR_INPUT
    TaskID : INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	TaskID	0 .. 32767	Task ID (return value of the 'PLC_GetTaskID' function).
Output parameters	PLC_ResumeTask	TRUE	Successful execution, Task with ID 'TaskID' exists.
		FALSE	No task with this ID exists.

Description

A suspended task can be recalled by calling up this library function. After recall, the task will again be processed by the scheduler and will be activated once the execution condition is met.

Any events which occurred while the task was suspended are ignored. The task is therefore not automatically activated after PLC_ResumeTask() if an event previously occurred.

13.2.8. Stopping the Cell Controller (PLC_STOP)

PLC_STOP

Declaration

```
FUNCTION PLC_STOP : BOOL
VAR_INPUT
    USERCODE : DWORD;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	USERCODE		Entry which appears in the diagnostics information.
Output parameters	PLC_STOP		No significance

Description

This library function stops the application program from being executed. The call-up can contain a "user code" which is saved to the diagnostics information entry.

If the user declares the 'PLC_ONERROR' system task, any remaining program code will be processed to completion before the controller goes to the 'STOP' state.

13.2.9. Restarting the Cell Controller (PLC_REBOOT)

PLC_REBOOTv

Declaration

```
FUNCTION PLC_REBOOT : BOOL
VAR_INPUT
    DUMMY : INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	Dummy		In order to ensure compatibility with newer versions this parameter should always be transferred as '0'.
Output parameters	PLC_REBOOT		No significance

Description

This library function results in the restart of the cell controller. Variables declared as 'RETAIN' are saved, but all other variable contents filed in RAM are lost.

14. Saving Application Data to the Flash Memory (FOD_Vxx.LIB)

Terminology

Application data are saved to the flash memory on the application's demand or by the tools. This function is therefore referred to as **Flash on Demand (FoD)**.

14.1. FoD Application Area



Improper use of FoD can destroy the flash memory.

Flash memories cannot be written to an infinite number of times. The component may become inoperative after more than 100,000 write accesses! Care should be taken that FoD is only used once to write to the flash memory!

Example: If write access occurs every 10 seconds, the flash memory can be destroyed within only 12 days!



Saving with FoD can increase the cycle time by up to 2 seconds.

This will result in a 2-second delay in the controller reaction time. Therefore, you should only employ the FoD blocks when the machinery or equipment are in a state which does not require a fast controller reaction. The maximum permissible cycle time must be set accordingly.

FoD can always be used whenever dynamically determined data – in other words, data generated during the IEC61131 program's runtime – must be permanently available on the controller. These data should change very rarely, if ever.

A typical application area would be for machinery or equipment configuration parameters. The machine's installer adjusts the parameters of a newly installed machine to meet the local conditions and characteristics of the machine. Once set-up has been completed, the determined parameters are saved and are thus available at every restart.

14.1.1. When Can FoD Not be Used?

FoD should not be used for continuously changing data. If these are continuously saved during ongoing operations, the result will be long cycle times or flash memory faults. In this type of application the data are best declared to be retain data. Retain data are only saved in case of a power outage.

14.2. FoD Functional Overview

FoD offers a comprehensive concept for processing FoD data online or offline. The FOD_Vxx.lib CP1131 library as well as the two tools, CP1131 and CNW, are available for this.

CP1131 library

The FOD_Vxx.lib CP1131 library provides blocks with which data values can be saved from a CP1131 program and can be read, edited and deleted. In addition FoD data status information can be read from the cell controllers.

CP1131 / CNW

Using CNW / CP1131, the FoD data of a controller can be administered, read and written. With the tools, FoD allows entire records to be loaded, and this in both directions. Thus, a machine parameter set can be saved from the controller to a PC. And, in the same way, a record can be downloaded from a PC to a controller. This allows parameters that have been determined once to be subsequently loaded to other cell controllers. This simplifies not only commissioning, but also spare parts stocks.

14.3. FoD Data Structure

FoD data configuration in CP1131 / CNW operates on the basis of data structures. CP1131 / CNW stores the data structure, together with the data types and data values in a file (*.FOD).

CP1131 memory subdivision limits the maximum size of any given data structure to 32 KB. The flash memory in the controller is subdivided into 64 KB blocks. Thus, each flash memory block can hold up to two data structures. If the data structures are smaller than 32 KB, any remaining space up to the next 32 KB remains unused.

The table shows the example of 2 flash blocks made accessible to FoD.

Block size	Data structure 1	Data structure 2
64 KB	max 32 KB	max 32 KB
64 KB	max 32 KB	max 32 KB

Attributes of an FoD block

Attribute	Description
Index	Unique identifier (number) for the FoD block
Size1	Size of the first data structure in the block, in bytes
Structure1	Name of the first data structure
Size2	Size of the second data structure in the block, in bytes
Structure2	Name of the second data structure
Data size	Number of bytes actually saved
Version	Version identifier

The FoD data structure is uniquely specified by **Index**, **Size1** and **Size2**. Optionally, designating attributes **Version**, **Structure1** and **Structure2** are also available.



FoD blocks are considered to be **identical** if their Index, Size1 and Size2 attributes are the same, even if their version information and structure names differ.

14.4. FoD File Structure

CP1131 / CNW places all FoD data on the PC in an FoD file. The name of this FoD file is made up of the CP1131 project name and the extension, ".FOD". FoD files can be edited with a simple editor or even with Microsoft Excel.

An FoD file can be linked to icon information, or not. In the example below, the FoD file is linked to icon information. A flash block has been initialized. It contains the two data structures, 'sFOD1' and 'sFOD2', each taking up 42 bytes of memory. The data structure is indicated by the structure variable name, the data type and the placeholder for the data value. The data values can now be loaded from the cell controller online. In the cell controller, the data values are sequentially placed in the memory without any structure information. During upload, the data values are then simply filed sequentially in the FoD file.

The second method for transferring data values takes place offline. Here, data values are simply copied sequentially into the FoD file from binary or other FoD files. The associated CP1131 / CNW menus are illustrated in the following sections.



The `_Align` variable is a special case in the sample file below.

This variable does not, in fact, belong to the data structure and was inserted automatically by CP1131 / CNW, otherwise the next variable from the data structure would lie at an uneven address. The system always places variables on an even address. Thus, a structure declared in this form by the user actually takes up more memory than necessary.

```

;=====
[Block 1]
Structure1 = sFOD1
StructSize1 = 42
Structure2 = sFOD2
StructSize2 = 42
Version =
;Block XXXX Struct Name Type Len FMT = Data ...
Block 1 Struct1 Version DWORD 4 HEX =
Block 1 Struct1 Comment STRING 21 HEX =
Block 1 Struct1 _Align BYTE 1 HEX =
Block 1 Struct1 s1.Val0 BYTE 1 HEX =
Block 1 Struct1 _Align BYTE 1 HEX =
Block 1 Struct1 s1.Val2 WORD 2 HEX =
Block 1 Struct1 s1.Val1 WORD 2 HEX =
Block 1 Struct1 s1.Val3 BYTE 1 HEX =
Block 1 Struct1 _Align BYTE 1 HEX =
Block 1 Struct1 s2.Val0 BYTE 1 HEX =
Block 1 Struct1 _Align BYTE 1 HEX =
Block 1 Struct1 s2.Val2 WORD 2 HEX =
Block 1 Struct1 s2.Val1 WORD 2 HEX =
Block 1 Struct1 s2.Val3 BYTE 1 HEX =
Block 1 Struct1 _Align BYTE 1 HEX =

Block 1 Struct2 Version DWORD 4 HEX =
Block 1 Struct2 Comment STRING 21 HEX =
Block 1 Struct2 _Align BYTE 1 HEX =
Block 1 Struct2 s1.Val0 BYTE 1 HEX =
Block 1 Struct2 _Align BYTE 1 HEX =
Block 1 Struct2 s1.Val2 WORD 2 HEX =
Block 1 Struct2 s1.Val1 WORD 2 HEX =
Block 1 Struct2 s1.Val3 BYTE 1 HEX =
Block 1 Struct2 _Align BYTE 1 HEX =
Block 1 Struct2 s2.Val0 BYTE 1 HEX =
Block 1 Struct2 _Align BYTE 1 HEX =
Block 1 Struct2 s2.Val2 WORD 2 HEX =
Block 1 Struct2 s2.Val1 WORD 2 HEX =
Block 1 Struct2 s2.Val3 BYTE 1 HEX =
Block 1 Struct2 _Align BYTE 1 HEX =
;=====

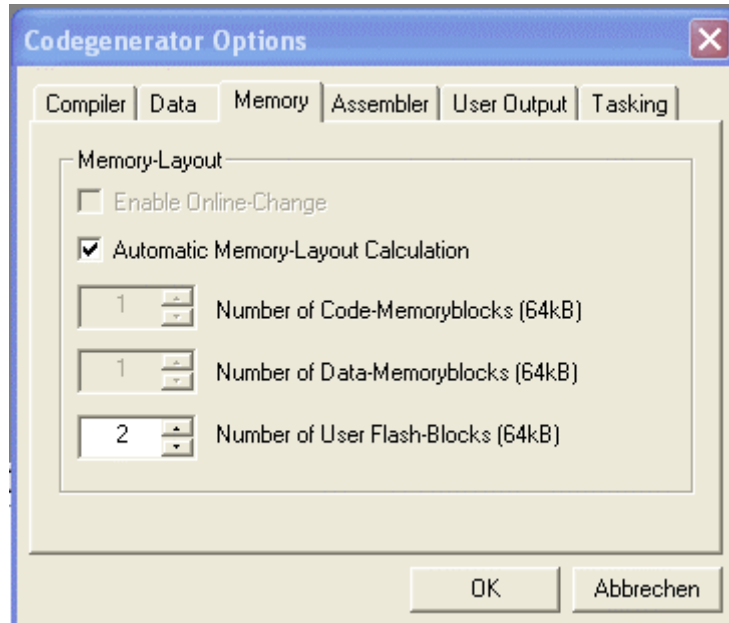
```

14.5. Activating FoD

The controller's memory layout can be adjusted using the code generator "Memory" option. The "Number of User Flash-Blocks" is the decisive factor for activating FoD. Once this number > 0 this memory is available to FoD and FoD is activated.

The total number of flash blocks available to the application depends on the individual controller type. It should be documented in the hardware manuals.

Code generator
memory option

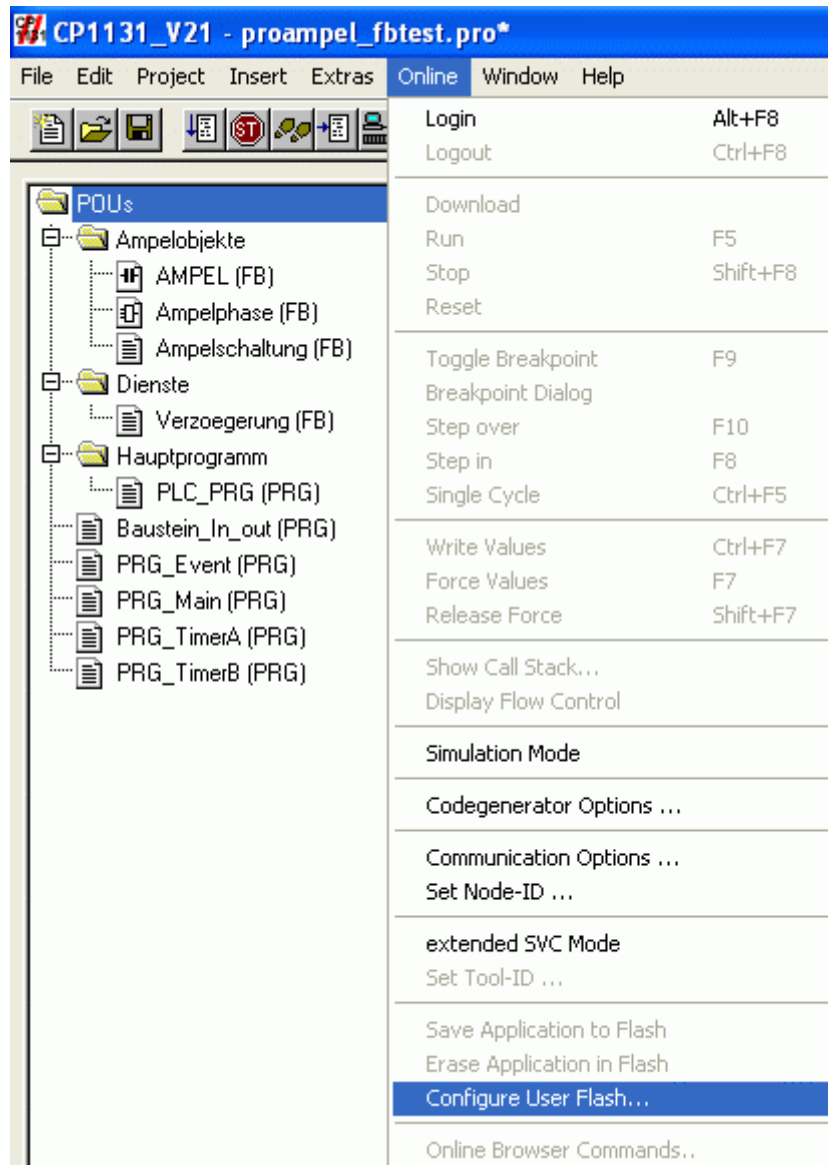


2VF100201DG00_en.gif

14.6. FoD Data Configuration (User Flash Configuration)

Prerequisite Only when FoD is activated (Refer to the section, 'Activating FoD') will the FoD data configuration menu be active ('User Flash Configuration').

Select 'User Flash Configuration'

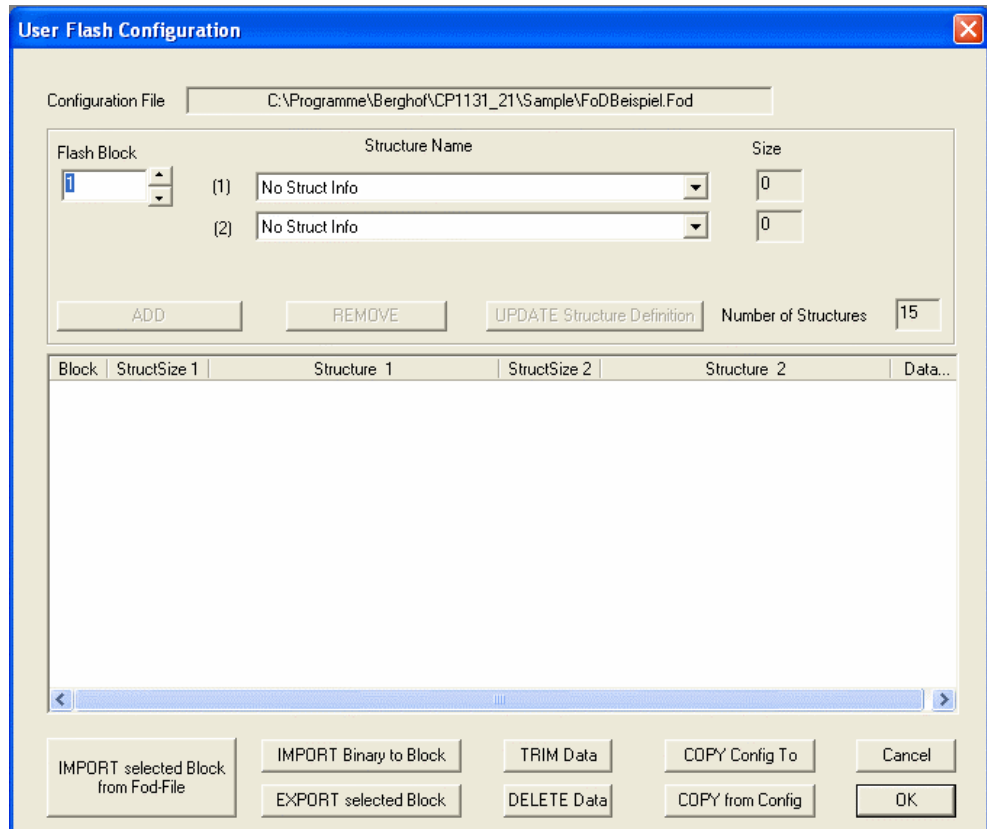


2VF100202DG00_en.gif

The 'User Flash Configuration' (refer to the user flash configuration image) is used to administer (create, import, edit) the FoD data structures. When the menu is called up, a configuration file with the extension ".FOD" is created, unless one already exists. The file has the same name as the project and is stored in the same directory. In the following image, this is the file, 'FoDBeispiel.Fod'.

Initially, only the FoD blocks together with their data structures or data types are created in the FoD file. Subsequently, data can also be stored in the file. These data can come directly from the controller or can be imported from other files.

Select 'User Flash Configuration'

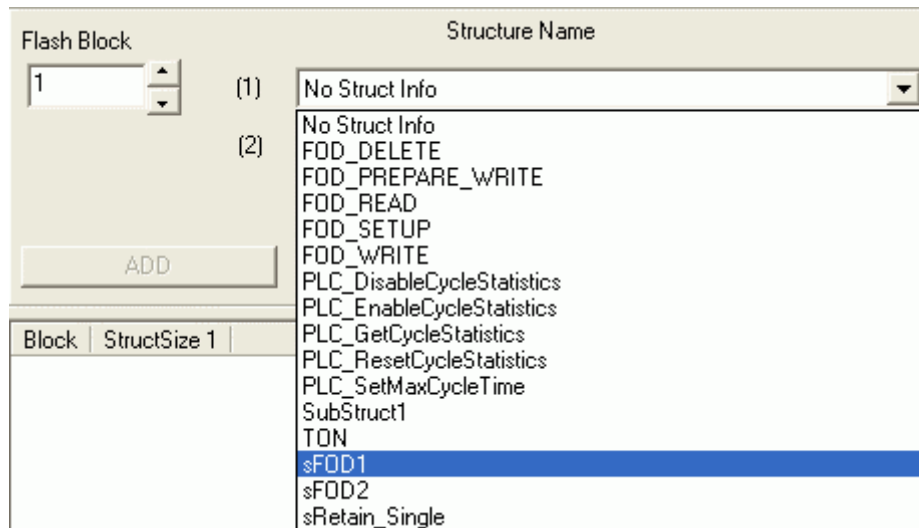


2VF100203DG00_en.gif

14.6.1. Creating an FoD Data Configuration

A maximum of 2 data structures (Structure 1, Structure 2) can be selected for each flash block (refer to the 'Structure Name' image). Only the data types in the project itself or those defined via libraries are available. No elementary data types are offered.

Selecting a data structure



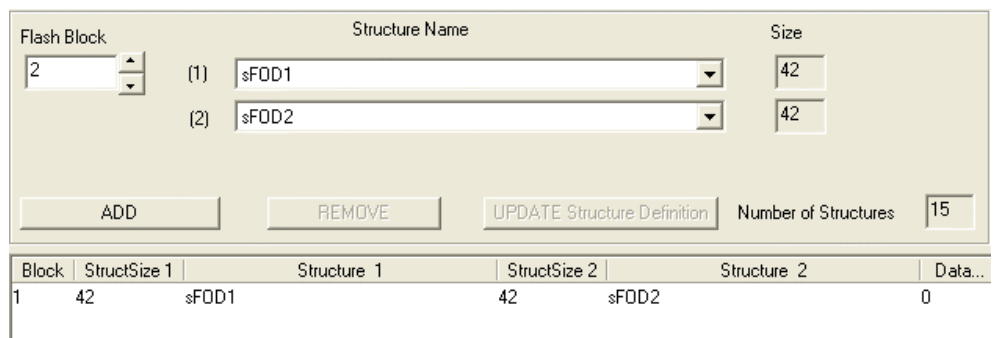
2VF100204DG00_en.gif



The flash block number is used as the FoD attribute index and thus provides a unique identification.

The block, together with its data structure is added to the FoD configuration by clicking on the 'ADD' button (refer to the 'Structure Name' image). The configured block now appears in the overview list. When adding, the FoD block number is automatically incremented by 1. The data size is always 0 for newly created blocks.

Overview list of FoD blocks and their data structures



2VF100205DG00_en.gif

Editing blocks

Blocks which appear in the configuration list must first be removed before they can be edited or redefined.

14.6.2. Editing an FoD Data Configuration

The 'User Flash Configuration' dialog performs two functions: On the one hand, functions for data structure configuration and, on the other, functions which access the data values.

First, let us look at the functions used to edit FoD blocks and data structures.

REMOVE

An FoD block can be removed from the configuration.

UPDATE Structure Definition

If there are changes to the data structures defined for the project, previous structure information in an FoD configuration can be updated.

COPY Config To

Copies of an FoD configuration can be saved under any other desired name.

COPY from Config

FoD configurations can be loaded to the current configuration file.

The following functions are related to the data stored in the FoD file. This dialog can be used to export or delete these data.

DELETE Data

Data belonging to an FoD block are deleted from the FoD file.

EXPORT selected Block

Data from the activated FoD block are exported to a binary file.

Saving

When exiting the menu by clicking on 'OK', the configuration is automatically saved in the FoD file. If you do not wish to save your changes, simply click on the 'Cancel' button to exit.

14.6.3. Importing FoD Data

There are 2 import functions with which defined FoD blocks or data structures can be filled with data from other files:

1. Import data from any desired binary file;
2. Import data from an existing FoD file.

IMPORT Binary to Block

When importing from any desired binary file, the desired file is chosen from a file selector box and imported to the data in the currently selected block. In doing this, the data are sequentially copied to Structure 1 followed by Structure 2. The data size does not need to match the total size of the two structures. However, the 'Trim Data' can be used to adjust the size.

IMPORT selected Block from Fod-File

When importing an existing FoD file, the file can be chosen from a file selector box. The data with the corresponding block number (of the currently selected block) are then imported. In doing this, the data are sequentially copied to Structure 1 followed by Structure 2. The data size does not need to match the total size of the two structures. However, the 'Trim Data' can be used to adjust the size.

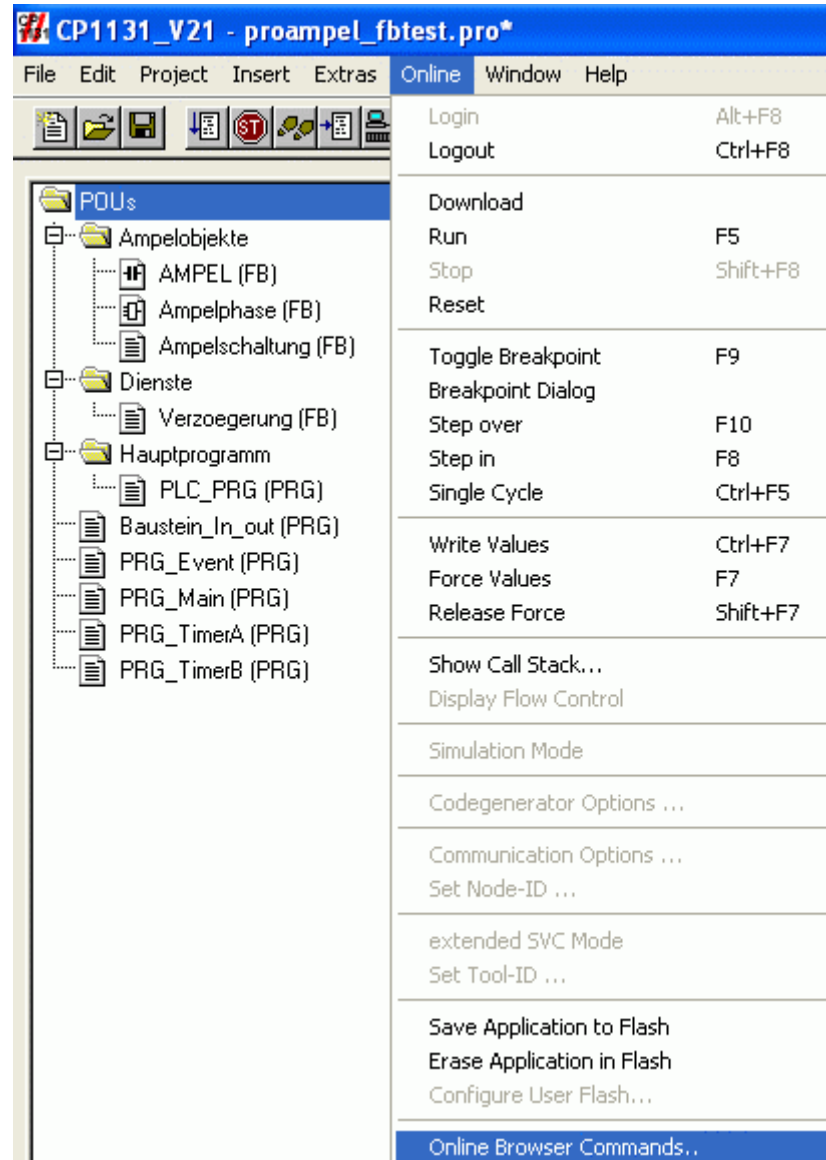
TRIM Data

The 'Trim Data' command can be used to adjust the imported data to the predefined structure size. Extraneous data are then deleted or, if required, the data are extended with 'n-times 0x00'.

14.6.4. Downloading/Uploading FoD Data with CP1131

The dialog in the CP1131 used to download/upload FoD data can be accessed via the 'Online Browser Commands' menu.

'Online Browser
Commands' menu



2VF100206DG00_en.gif

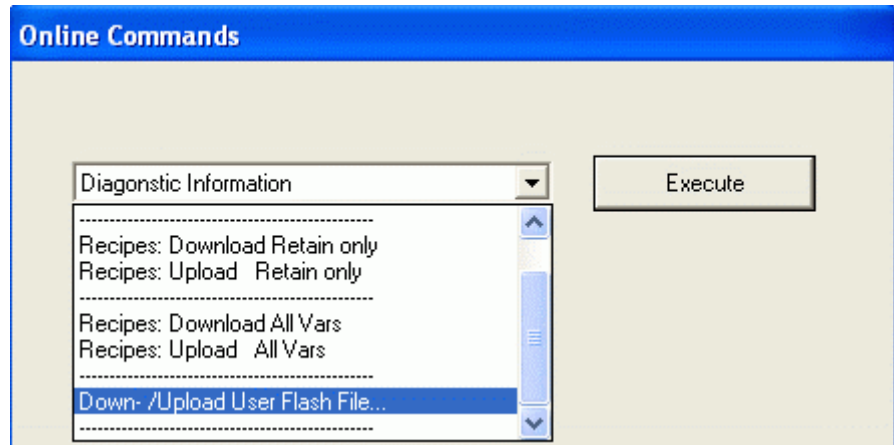
NOTICE

In order for the 'Online Browser Commands' menu to be active, CP1131 must be logged onto the controller.

In order to ensure consistency of the loaded data, the controller should be in the CP1131 'STOP' operating mode (ERRORSTOP also works).

Select the 'Down-/ Upload User Flash File' command from the list of available commands under 'Online Browser Commands'.

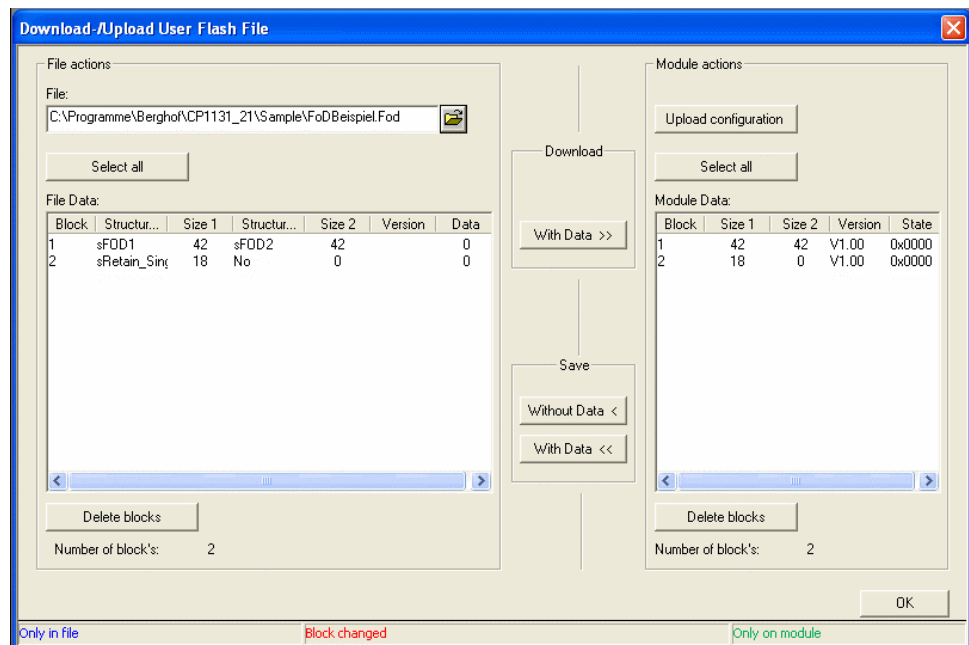
Select 'Online Browser Commands'



2VF100207DG00_en.gif

The subsequent 'Down-/ Upload User Flash File' dialog provides an overview of the FoD data currently saved on the controller.

'Down-/ Upload User Flash File' dialog



2VF100208DG00_en.gif

The dialog is subdivided into two areas.

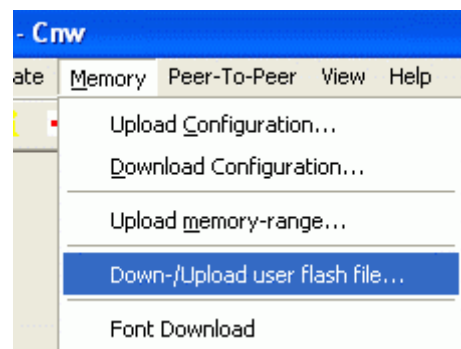
1. The left side of the dialog contains the so-called PC or FoD file actions. The FoD file containing the defined data structures and the loaded data is shown on the top. The overview window below this displays the data structures. In the image above, the data are still set to '0'. It can therefore be assumed that no data have yet been uploaded from the controller. After uploading, Block1 should indicate size 84 and Block 2 size 18.
2. The right side of the dialog contains the controller or module actions.

Using the buttons, data can now be uploaded out of the controller and saved in the FoD file. Correspondingly, data can also be downloaded from an FoD file to the module. If necessary, individual FoD data can also be deleted.

14.6.5. Downloading/Uploading FoD Data with CNW

CNW provides the identical dialog to 'Down-/ Upload User Flash File' as in CP1131. (refer to the section, 'Downloading/Uploading FoD Data with CP1131').

Select from the
'Save' menu



2VF100209DG00_en.gif

14.7. The FoD Library (FOD_Vxx.LIB)

FOD_V10.lib	In order to utilize the FoD functionality in the CP1131 program the user must establish a link to the FoD library. This will make the following function blocks available:
FOD_SETUP	Parameter settings to influence behavior.
FOD_READ	The data from a data block (specified by the index) are copied to a specified address in RAM by the flash.
FOD_WRITE	The data from a data block (specified by the index) are saved to the flash from a specified address in RAM. Prerequisite: FOD_PREPARE_WRITE was previously executed.
FOD_PREPARE_WRITE	The data from a data block (specified by the index) are prepared for writing. This minimizes the time required for the actual writing. Prerequisite for FOD_WRITE.
FOD_READ_INFO	The index, version information and size status information are returned for each existing data block.
FOD_GET_MAXBLOCK	Determines the largest index of the available FoD data.
FOD_RESET	The FoD data block associated with the specified index is initialized with zero.
FOD_DELETE	The FoD data block associated with the specified index is deleted.

14.7.1. FOD_SETUP

FOD_SETUP

Declaration

```
FUNCTION_BLOCK FOD_SETUP
VAR_INPUT
    SUSPENDMODE:      WORD;
END_VAR
VAR_OUTPUT
    STATE:            INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	SUSPENDMODE	TRUE	FoD function block execution can be interrupted.
		FALSE	FoD function block execution cannot be interrupted.
Output parameters	STATE	0	Success
		1	Internal error

Description

If the suspend mode has been set, the FoD functions return 'BUSY' if the execution has not yet been completed.

14.7.2. FOD_READ

FOD_READ

Declaration

```

FUNCTION_BLOCK FOD_READ
VAR_INPUT
    Index:      BYTE;
    ADDRESS1:   DWORD;
    ADDRESS2:   DWORD;
    SIZE1:      DWORD;
    SIZE2:      DWORD;
    FORMAT:     BYTE;
END_VAR
VAR_OUTPUT
    STATE:      INT;
    VERSINFO:   DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	INDEX	1 .. MAX	FoD block index
	ADDRESS1		Address of the first FoD structure
	ADDRESS2		Address of the second FoD structure
	SIZE1		Size of the first FoD structure, in bytes
	SIZE2		Size of the second FoD structure, in bytes
	FORMAT		Reserved
	Output parameters	STATE	0
		1	Function BUSY
		2	Invalid parameters
		3	No data available
		4	Data inconsistency
		5	Data out of date (previous write attempt failed)
		6	Internal error
	VERSINFO		FoD block version information

Description

The data in the block are copied from the flash to the specified addresses. If the second address is not used, the ADDRESS2 and SIZE2 input variables can remain empty or can be initialized with zero.

14.7.3. FOD_READ_INFO

FOD_READ_INFO

Declaration

```

FUNCTION_BLOCK FOD_READ_INFO
VAR_INPUT
    Index:      BYTE;
END_VAR
VAR_OUTPUT
    STATE:      INT;
    SIZE1:      DWORD;
    SIZE2:      DWORD;
    FORMAT:     BYTE;
    VERSINFO:   STRING[20];
    PREPARED:   BOOL;
    ADDRESS1:   DWORD;
    ADDRESS2:   DWORD;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	INDEX	1 .. MAX	FoD block index
Output parameters	STATE	0	Valid data available
		1	Invalid parameters
		2	No data available
		3	Data inconsistency
		4	Data out of date (previous write attempt failed)
		5	Block deleted.
		6	Internal error
	SIZE1		Size of the first FoD structure, in bytes
	SIZE2		Size of the second FoD structure, in bytes
	FORMAT		
	VERSINFO		FoD block version information
	PREPARED	TRUE	Block prepared for writing
		FALSE	Block not prepared
	ADDRESS1		Address of the first FoD structure in the flash
	ADDRESS2		Address of the second FoD structure in the flash

Description

Status information of the block (specified by the index) are read.

ADDRESS1 and ADDRESS2 provide the addresses of the FoD data in the flash (if data are available). A pointer can be used to access these data directly without having to copy them to RAM via FOD_READ.

However, in this case the data cannot be changed in the application and cannot be saved via FOD_WRITE.

14.7.4. FOD_GET_MAXINDEX**FOD_GET_MAXINDEX**

Declaration

```

FUNCTION_BLOCK FOD_GET_MAXINDEX
VAR_INPUT
END_VAR
VAR_OUTPUT
    INDEX:          BYTE;
    STATE:          INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	-	-	-
Output parameters	INDEX	1..MAX	Largest FoD index encountered
	STATE	0	Success
		1	Internal error

Description Returns the index of the largest FoD block in the flash.

14.7.5. FOD_WRITE

FOD_WRITE

Declaration

```

FUNCTION_BLOCK FOD_WRITE
VAR_INPUT
    Index:          BYTE;
    VERSINFO        STRING[20];
    ADDRESS1:       DWORD;
    ADDRESS2:       DWORD;
    SIZE1:          DWORD;
    SIZE2:          DWORD;
    FORMAT:         BYTE;
END_VAR
VAR_OUTPUT
    STATE:          INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	INDEX	1..MAX	FoD block index
	VERSINFO		FoD block version information
	ADDRESS1		Address of the first FoD structure
	ADDRESS2		Address of the second FoD structure
	SIZE1		Size of the first FoD structure, in bytes
	SIZE2		Size of the second FoD structure, in bytes
	FORMAT		Reserved
	Output parameters	STATE	0
		1	Function BUSY
		2	Invalid parameters
		3	Block not prepared (execute FOD_PREPARE_WRITE)
		4	Internal error

Description

The data are saved to the FoD block.
If the second address is not used, the ADDRESS2 and SIZE2 input variables can remain empty or can be initialized with zero.

14.7.6. FOD_PREPARE_WRITE

FOD_PREPARE_WRITE

```

Declaration      FUNCTION_BLOCK FOD_PREPARE_WRITE
                    VAR_INPUT
                        Index:      BYTE;
                    END_VAR
                    VAR_OUTPUT
                        STATE: INT;
                    END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	INDEX	1 . . MAX	FoD block index
Output parameters	STATE	0	FoD block successfully prepared
		1	Busy
		2	Invalid parameters
		3	Data out of date
		4	Data inconsistency (data deleted)
		5	Internal error

Description

The FoD block data are prepared for writing. This command must be executed prior to each write procedure (FOD_WRITE).

FoD blocks that are writable are alternately saved to two different flash blocks. This ensures that the previous data consistency can be maintained in case of an error during the write procedure.

In order to write to a flash block it must first be deleted. The delete routine for a flash block takes approx. 1 sec. Thus, a FOD_PREPARE_WRITE takes at least 1 sec. if the flash block must first be deleted.

Due to the long time required for deletion, the FOD_PREPARE_WRITE process is broken down. The interruptability of the FoD call-ups can be defined via FB FOD_SETUP.

The FOD_PREPARE_WRITE command deletes the block so that this action does not need to be subsequently carried out when writing with FOD_WRITE. This results in data write times of only a few milliseconds. In turn, this allows write procedures to be accelerated during time-critical periods, while the preparation can take place during less critical times.

14.7.7. FOD_RESET

FOD_RESET

Declaration

```

FUNCTION_BLOCK FOD_RESET
VAR_INPUT
    Index:      BYTE;
    VERSINFO    STRING[20];
    SIZE1:      DWORD;
    SIZE2:      DWORD;
    FORMAT:     BYTE;
END_VAR
VAR_OUTPUT
    STATE:     INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	INDEX	1 .. MAX	FoD block index
	VERSINFO		FoD block version information
	SIZE1		Size of the first FoD structure, in bytes
	SIZE2		Size of the second FoD structure, in bytes
	FORMAT		Reserved
Output parameters	STATE	0	FoD block successfully initialized with 0
		1	Busy
		2	Invalid parameters
		3	Block not prepared (execute FOD_PREPARE_WRITE)
		4	Internal error

Description

The FoD block specified by the index is initialized with zero.

14.7.8. FOD_DELETE**FOD_DELETE****Declaration**

```

FUNCTION_BLOCK FOD_DELETE
VAR_INPUT
    Index:      BYTE;
END_VAR
VAR_OUTPUT
    STATE:     INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	INDEX	1 .. MAX	FoD block index
Output parameters	STATE	0	FoD block successfully removed
		1	Busy
		2	Invalid parameters
		3	Internal error

Description

The FoD block specified by the index is removed.

15. Notes

15.1. Data interchange via addresses (DATAPTR) and markers

The cell controllers have various different communications interfaces “on board”. There are always at least two CAN interfaces, and there is at least one serial interface. An Ethernet interface may also be added. The feature these interfaces share in common is the ability to receive and transmit data. The following is a representative example.

15.1.1. Transmitting data with an ARRAY at the CAN interface

The corresponding CP1131 function block, CAN_TRANSMIT, provides an input parameter (VAR_INPUT) for this purpose, called DATA. DATA is an 8-byte-sized array, and is therefore of data type ARRAY[0..7] OF BYTE. The application can copy data byte-by-byte into an array of this kind, and then transmit the data.

```
PROGRAM BspArraySend
VAR
    CanSend          : CAN_TRANSMIT;          (* instance of transmit
                                              function block *)
    CanSendBuffer    : ARRAY[0..7] OF BYTE;  (* transmit data array *)
...
END_VAR

...
CanSendBuffer[0] := 10;                      (*value assignments,
                                              transmit data array*)
CanSendBuffer[1] := 15;
CanSendBuffer[2] := 20;
CanSendBuffer[3] := 25;
...
CanSend    (DATAPTR    := 0,                (*equals 0, so it is not used*)
           CHANNEL    := 0,                (* transmission via CAN on
                                              the cover *)
           COBID      := 100,            (*with identifier 100*)
           RTR         := FALSE,          (*data frame*)
           LENGTH     := 4,              (*4 bytes transmitted*)
           DATA       := CanSendBuffer  (*the data itself*)
           );
...

```

15.1.2. Transmitting data with an address pointer at the CAN interface

The CP1131 function block CAN_TRANSMIT provides the parameter DATAPTR in addition to the input parameter DATA. DATAPTR is an address within the valid address space. Problems are not normally to be anticipated when working with addresses of variables that have been created by the compiler. The address of any such variable can be determined with the address operator ADR. Addresses are of data type DWORD, and are therefore 32 bits long.

```

PROGRAM BspPointerSend
VAR
    CanSend          : CAN_TRANSMIT;    (* instance of transmit
                                         function block*)
    CanSendVar       : DINT;            (*transmit variable*)
    AdressVar        : DWORD;          (*address buffer*)
...
END_VAR

...

CanSendVar          := 10152025;        (*value assignments,
                                         transmit variable*)
AdressVar           :=ADR (CanSendVar);  (*use address operator
                                         ADR to determine the
                                         address of CanSendVar*)

...
CanSend (DATAPTR    := AdressVar,      (*address buffer*)
        CHANNEL     := 0,              (*transmit via CAN on
                                         the cover*)
        COBID       := 100,           (*with identifier 100*)
        RTR         := FALSE,        (*data frame*)
        LENGTH      := 4,            (*4 bytes transmitted*)
                                         (*DATAPTR <> 0,
                                         so DATA cannot be used*)
        );
...

```

Thanks to this design, data (when receiving, for instance) can immediately be filed anywhere within a large array, at and from the address designated by DATAPTR.

```

CanReceiveBuffer    := ARRAY[0..500] OF BYTE;
AdressVar           := ADR (CanReceiveBuffer [i+8]);

```

A complicated copying operation would be needed if the array DATA (CanSend-Buffer) were to be used here.

NOTICE

Make sure there is sufficient memory space available at the address indicated by 'DATAPTR', because memory areas could otherwise be overwritten.

15.1.3. Transmitting data with markers at the CAN interface

Markers globally created by the system provide another means of working with fixed addresses within CP1131. Several variables can be entered in any one marker, so it becomes possible to access the same data with different variables with different data types. An example would be I/O data which has to be processed, transmitted, and supplemented by other data. In modern general languages, the data structure UNION may also be used for this purpose.

```

PROGRAM BspArraySend
VAR
    CanSend                : CAN_TRANSMIT;    (instance of transmit
                                                function block*)
    CanSendBuffer AT %MB0  : ARRAY [0..3] OF BYTE;
                                                (*transmit data array*)
    CanSendVar AT %MB0     : DINT;            (*transmit variable*)
    CanSendText AT %MB4    : STRING (4);     (*transmit variable*)
    AdressVar              : DWORD;          (*address buffer*)
    Ausgang01 AT %MX0.1    : BOOL;           (*create individual
                                                marker bit as
                                                variable*)
END_VAR

```

```

...
CanSendText      := 'OUT1' ;
CanSendVar       := 0;
                                                (*value assignments,
                                                transmit variable*)

%MX0.0           := TRUE;
first
AdressVar        := ADR (CanSendVar);
                                                (*use address operator
                                                ADR to determine
                                                address of Can
                                                SendVar*)

...
...
                                                (*transmit the following data: 1,0,0,0,'O','U','T','1' (Type-mix)*)
CanSend (DATAPTR   := AdressVar,          (*address buffer*)
        CHANNEL    := 0,                    (*transmit via CAN on
the cover*)
        COBID      := 100,                  (*with identifier 100*)
        RTR        := FALSE,                (* data frame *)
        LENGTH     := 8,                    (* 4 bytes transmitted
*)
                                                (* DATAPTR <> 0, so
                                                DATA cannot be
                                                used*)
        );
...

```

NOTICE

When using markers, make sure that the bit-oriented access to markers (%MX2.0) is always set up for word orientation. %MX2.0 => first marker in MarkerWORD 2 is not in MarkerBYTE 2, but is already MarkerBYTE 4.

15.2. CAN interfaces (CAN.LIB)

15.2.1. Introduction

The cell controllers are provided with up to three CAN interfaces. The following is a list of the available CAN interfaces:

- CAN0: Wide CAN, accessible through Min-D jack or connector on the cover of the cell controller.
- CAN1: E-bus CAN, connected to the outside through the 40-pole, side-mounted E-bus connector. Used for local addition of remote modules.
- CAN2: Local CAN, available via Min-D jack or connector on special cell controllers.

All three CAN interfaces can be operated with a single user interface using IEC61131 function blocks. (Initialisation, reception, transmission, etc.).

Each of these function blocks has a receive queue and a send queue, each of which are fixed in size.

A baud rate is normally specified when initialising a CAN interface. In the software driver, this baud rate is converted into register values for the CAN controller, these being the values for bit-timing register 0 and 1 (BTR0, BTR1). The following table shows the bit-timing values the automation system uses for the various standard baud rates:

Baud Rate (kbit/sec)	BTR0	BTR1
1000	0x00	0x14
500	0x00	0x1C
250	0x01	0x1C
125	0x03	0x1C
100	0x43	0x2F
50	0x47	0x2F
20	0x53	0x2F
10	0x67	0x2F

A cell controller equipped with a CAN interface for the contact conductor (CANtrol *power track*) would be set up with the following transmission rates and parameters.

Line length [m]	Baud rate [kbit/s]	Register BTR1	Register BTR0	Register CDR	Register OCR
76	125.0	1C	03	81	DB
307	83.3	1C	05	82	DB
*	62.5	1C	07	83	DB
*	50.0	1C	09	84	DB
*	41.6	1C	0B	85	DB
*	35.7	1C	0D	86	DB

The CANtrol *powertrack* baud rate can only be adjusted by writing a complete register record. The CAN.LIB can only access this register record if the correct CANtrol *powertrack* baud rate has already been predefined via CNW. If the application requires that the registers be manually written, this can only be done with the ECAN.LIB.

NOTICE

Not all identifiers are available to the application.

Every controller in the control system can be configured and/or programmed from a central point. The Service Channel (SVC) protocol is used to make sure this data gets to the right controller. Particular identifiers are used for this purpose when transmitting these SVC messages over the CANbus. These SVC identifiers are chosen from within the range ID1409 to 1663, which means that identifiers from this range are not available to the application.

15.3. IEC61131 User Interface



CAN communication uses so-called 'identifiers', represented, in the following function blocks, by the abbreviation COB (Communication Object); for example COBID, or COBSTART.

15.3.1. Initialisation (CAN_INIT)

CAN_INIT

Declaration

```

FUNCTION_BLOCK CAN_INIT
VAR_INPUT
    CHANNEL      :      INT;
    BAUDRATE     :      DWORD;
    BTR0         :      BYTE;
    BTR1         :      BYTE;
END_VAR
VAR_OUTPUT
    STATE       :      INT;
END_VAR
    
```

Input parameters

Parameter	Value	Description
CHANNEL	0..2	CAN interface
BAUDRATE	0,	Activate bit-timing register BTR0/1
	1,	Conforming to configuration in serial EEPROM
	10000,	10 kbit (standard CAN)
	20000,	20 kbit (standard CAN)
	35700,	35.7 kbit (CANtrol powertrack)
	41600,	41.6 kbit (CANtrol powertrack)
	50000,	50 kbit (standard CAN+CANtrol powertrack)
	62500,	62.5 kbit (CANtrol powertrack)
	83333	83.3 kbit (CANtrol powertrack)
	100000,	100 kbit (standard CAN)
	125000,	125 kbit (standard CAN+CANtrol powertrack)
	250000,	250 kbit (standard CAN)
	500000,	500 kbit (standard CAN)
	1000000	1 Mbit (standard CAN)
	BTR0	0..255
BTR1	0..255	Valid only if BAUDRATE = 0

Output parameters

STATE	0	Function successfully executed
	1	Internal error
	2	CHANNEL invalid
	3	BAUDRATE invalid
	11	Value in serial EEPROM invalid: BAUDRATE or BTR0/1

Description

This function block sets the CAN interface identified by 'CHANNEL' to the desired BAUDRATE, and empties both the receive queue and the send queue.

The block also resets the input filter generated by CAN_DEFINE_COBID. The application will not be able to receive any more CAN messages until this input filter has been restarted.

When the 'BAUDRATE' parameter is set at value '0', the two parameters BTR0 and BTR1 are written directly (without consistency check) to the corresponding registers of the 82C200 or SJA1000 CAN controller.

When the value of the 'BAUDRATE' parameter is greater than '0', the CAN controller is set to the corresponding baud rate; all remaining bit-timing parameters in the BTR0 and/or BTR1 registers are set to their default values.

If the system software detects that the selected interface is of the CANtrol *power-track* type, it will only be able to set up the transmission rates intended for that kind of interface.

The system software selects the appropriate parameters of its own accord.

A transmission rate that is intended solely for CANtrol *power-track* must not be selected on a standard interface.

Do not use for new product development

15.3.2. Logging CAN messages on and off (CAN_DEFINE_COBID)

CAN_DEFINE_COBID

```

Declaration      FUNCTION_BLOCK CAN_DEFINE_COBID
                    VAR_INPUT
                        CHANNEL      :      INT;
                        COBSTART     :      WORD;
                        COBEND       :      WORD;
                        ENABLE        :      BOOL;
                    END_VAR
                    VAR_OUTPUT
                        STATE         :      INT;
                    END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
	COBSTART	0..2047	First identifier to be received
	COBEND	0..2047	Last identifier to be received
	ENABLE	TRUE	Identifiers between COBSTART and COBEND received
		FALSE	Identifiers between COBSTART and COBEND <u>not</u> received
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	COBSTART invalid
		4	COBEND invalid
		9	CAN interface not initialised

Description This function block is used to control the COBID acceptance filter of a CAN interface.

The block respectively enables ('ENABLE' = TRUE) or disables ('ENABLE' = FALSE) the reception, at the physical CAN interface identified by 'CHANNEL', of all CAN messages having a COBID between 'COBSTART' and 'COBEND' (inclusive).

After a COBID has been disabled, the internal receive queue of the interface in question will cease to accept any more CAN messages having the message identifier in question. This will not affect any messages already held in the receive queue.

15.3.3. Receiving CAN messages (CAN_RECEIVE)

CAN_RECEIVE

Declaration

```

FUNCTION_BLOCK CAN_RECEIVE
VAR_INPUT
    DATAPTR      :    DWORD;
    CHANNEL      :    INT;
END_VAR
VAR_OUTPUT
    VALID        :    BOOL;
    COBID        :    WORD;
    RTR          :    BOOL;
    LENGTH       :    INT;
    DATA        :    ARRAY[0..7] OF BYTE;
    STATE        :    INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
	DATAPTR		Address of a data object
Output parameters	VALID	TRUE	CAN message received, variables COBID, RTR, LENGTH and possibly DATA/DATAPTR being valid.
		FALSE	No CAN messages received
	COBID	0..2047	CAN identifier
	RTR	TRUE	CAN remote frame received, no data contained in variables DATA/DATAPTR
		FALSE	CAN data frame received, variables DATA/DATAPTR may contain data
	LENGTH	0..7	Indicates the number of data bytes received
	DATA		Valid only if DATAPTR = 0 The number of received data bytes indicated in COUNT are present at and from index position 0
	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
	3	Receive queue overrun	
	5	CAN controller is bus-off	
	9	CAN interface not initialised	
	10	Invalid range for DATAPTR	

Description

The output variable 'VALID' can be used to check if a CAN message has been received. If 'VALID' = TRUE, the function block fetches the 'oldest' CAN message having up to 8 data bytes from the receive queue of the physical CAN interface identified by 'CHANNEL'. This message is then deleted from the receive queue.

The output variable 'VALID' is set to FALSE if there are no messages left in the receive queue.

The value 'COBID' contains the message identifier of the message received.

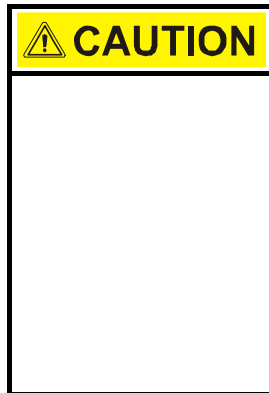
The variable 'RTR' indicates if the remote transmission request bit was set in this message.

'LENGTH' indicates the number of valid user-data bytes.

The received data is held in the variables DATAPTR or DATA.

Do not use for new product development

15.3.4. Transmitting CAN messages (CAN_TRANSMIT)



Do NOT duplicate identifiers when transmitting. This will cause transmission errors, as a result of which CAN nodes might be unable to continue transmitting.

The CANbus is a multimaster bus. This means that each CAN node on this bus is entitled to transmit its messages at any time – the only restriction is that a message which is already in course of transmission cannot be interrupted. Messages transmitted on the CANbus are identified by an 11-bit identifier. This identifier simultaneously assigns a priority to the message, which comes into play when two CAN nodes wish to transmit at the same time, by arranging for the message having the higher priority to be transmitted first. Transmission errors which could render the CAN nodes incapable of further transmission would occur if two CAN nodes having the same transmit identifier were to access the network at the one time.

CAN_TRANSMIT

Declaration

```
FUNCTION_BLOCK CAN_TRANSMIT
VAR_INPUT
    DATAPTR      :    DWORD;
    CHANNEL      :    INT;
    COBID        :    WORD;
    RTR          :    BOOL;
    LENGTH       :    INT;
    DATA        :    ARRAY[0..7] OF BYTE;
END_VAR
VAR_OUTPUT
    STATE       :    INT;
END_VAR
```

Input parameters

Parameter	Value	Description
CHANNEL	0..2	CAN interface
DATAPTR		Address of a data object
COBID	0..2047	CAN identifier
RTR	TRUE	Transmit CAN remote frame, no data contained in variables DATA/DATAPTR, LENGTH = 0
	FALSE	Transmit CAN data frame, variables DATA/DATAPTR may contain data
LENGTH	0..7	Indicates the number of data bytes to be transmitted
DATA		Valid only if DATAPTR = 0 The number of data bytes to be transmitted, as indicated in COUNT, is present at and from index position 0

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	COBID invalid
		4	LENGTH invalid
		5	CAN controller is bus-off
		6	No free entries in send queue
		9	CAN interface not initialised
		10	Invalid range for DATAPTR
		Description	<p>This function block copies a CAN message into the send queue of the physical CAN interface identified by 'CHANNEL'.</p> <p>The variable 'COBID' contains the message identifier for this message, while 'LENGTH' indicates the number of valid user-data bytes.</p> <p>The data to be transmitted is held in variables DATAPTR or DATA.</p> <p>The variable 'RTR' contains the status of the remote transmission request bit for the message being transmitted.</p>

Do not use for new product development

15.3.5. State request (CAN_STATE)

CAN_STATE

Declaration

```

FUNCTION_BLOCK CAN_STATE
VAR_INPUT
    CHANNEL      :      INT;
END_VAR
VAR_OUTPUT
    TXCOUNT    :      INT;
    RXCOUNT    :      INT;
    STATE       :      INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0 . . 2	CAN interface
Output parameters	TXCOUNT		Number of CAN messages in the send queue of CHANNEL
	RXCOUNT		Number of CAN messages in the receive queue of CHANNEL
	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	Receive queue overrun
		5	CAN controller is bus-off
		9	CAN interface not initialised

Description

This function block indicates the number of CAN messages currently contained in the receive queue and/or send queue of the physical CAN interface identified by 'CHANNEL'.

15.3.6. Logging a gateway on and off (CAN_DEFINE_GATEWAY)

CAN_DEFINE_GATEWAY

```

Declaration      FUNCTION_BLOCK CAN_DEFINE_GATEWAY
                    VAR_INPUT
                        SOURCE      :      INT;
                        DEST        :      INT;
                        COBSTART    :      WORD;
                        COBEND      :      WORD;
                        ENABLE      :      BOOL;
                    END_VAR
                    VAR_OUTPUT
                        STATE        :      INT;
                    END_VAR
    
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>	
Input parameters	SOURCE	0..2	The CAN message in question is received at this interface	
	DEST	0..2	The CAN message in question is automatically re-transmitted at this interface	
	COBSTART	0..2047		
	COBEND	0..2047		
	ENABLE	TRUE		Permits reception of the identifiers between COBSTART and COBEND
		FALSE		Prohibits reception of the identifiers between COBSTART and COBEND
Output parameters	STATE	0	Function successfully executed	
		1	Internal error	
		3	COBSTART invalid	
		4	COBEND invalid	
		-6	SOURCE and/or DEST invalid	
		9	CAN interface not initialised	

Description This function block is used to control gateway functionality. CAN messages with a message identifier chosen from between 'COBSTART' and 'COBEND' (inclusive), and received at the physical CAN interface 'SOURCE', are automatically (at system level) re-transmitted at the CAN interface 'DEST' – if this option has been enabled by setting 'ENABLE' to TRUE. This facility can be disabled again at any time for any desired range of message identifiers ('ENABLE' = FALSE).

15.3.7. Requesting own node ID

Node ID

Declaration

```
FUNCTION_BLOCK CAN_MYNODEID
VAR_INPUT
    CHANNEL      :      INT;
END_VAR
VAR_OUTPUT
    NODEID       :      BYTE;
    STATE        :      INT;
END_VAR
```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	CHANNEL	0..2	CAN interface
Output parameters	NODEID	1..127	Node number
	STATE	0	Function successfully executed
		1	Internal error
		2	CHANNEL invalid
		3	NODEID invalid

Description

Returns node ID of module in question at the physical CAN interface identified by 'CHANNEL'.

The node ID is used to identify modules uniquely within an interconnected CAN network.

This value is loaded to the module (using the configuration program, for example) and stored there in non-volatile form.

15.4. Quadrature Encoder (INCR.LIB)

15.4.1. Introduction

Various cell controllers have an interface for the connection of two incremental encoders having differential quadrature signals.

This position value can be sampled or set at any time during the program, and any change takes effect immediately. As a result, these values are independent of the cyclical behaviour of the controller program.

The range of values of this counter comprises 16 bits, and at counter reading 16#FFFF, it overflows to value 16#0000 (or vice versa).



This chapter has been included solely for reasons of compatibility. The INCR.LIB should no longer to be used for new developments.

Further cell controllers have been added to the automation system in the meantime, and new libraries have had to be introduced to cover the new range of functions:

- POSREL.LIB
- TPUDIO.LIB
- TPUPWM.LIB
- COUNTER.LIB

15.5. IEC61131 User Interface

15.5.1. Sampling the position value (POSITION_REL_GET)

POSITION_REL_GET

Declaration

```
FUNCTION_BLOCK POSITION_REL_GET
VAR_INPUT
    ENCODER      :      INT;
END_VAR
VAR_OUTPUT
    POSITION      :      WORD;
    STATE       :      INT;
END_VAR
```

Input parameters

Parameter	Value	Description
ENCODER	0..1	

Output parameters

POSITION		Contains the current counter-reading of the quadrature encoder
STATE	0	Function successfully executed
	1	Internal error
	2	ENCODER invalid

Description

This function block returns the current counter-reading of quadrature encoder 'ENCODER' (relative position).

15.5.2. Setting the position value (POSITION_REL_SET)

POSITION_REL_SET

Declaration

```

FUNCTION_BLOCK POSITION_REL_SET
VAR_INPUT
    ENCODER      :    INT;
    POSITION      :    WORD;
END_VAR
VAR_OUTPUT
    STATE :    INT;
END_VAR

```

	<i>Parameter</i>	<i>Value</i>	<i>Description</i>
Input parameters	ENCODER	0..1	
	POSITION		Contains the quadrature encoder counter-reading which is to be set
Output parameters	STATE	0	Function successfully executed
		1	Internal error
		2	ENCODER invalid

Description

This function block can be used to set the current counter-reading of quadrature encoder 'ENCODER' to the defined value indicated by 'POSITION'.

Blank page

16. Annex

16.1. Environmental Protection

16.1.1. Emission

When used correctly, our modules do not produce any harmful emissions.

16.1.2. Disposal

At the end of their service life, modules may be returned to the manufacturer against payment of an all-inclusive charge to cover costs. The manufacturer will then arrange for the modules to be recycled.

16.2. Maintenance/Upkeep



Do not insert, apply, detach or touch connections while in operation – risk of destruction or malfunction.

Disconnect all incoming power supplies before working on our modules; this also applies to connected peripheral equipment such as externally powered sensors, programming devices, etc. All ventilation openings must always be kept free of any obstruction.

The modules are maintenance-free when used correctly.
Clean only with a dry, non-fluffing cloth.
Do not use detergents.

16.3. Repairs/Service



Repair work may only be carried out by the manufacturer or its authorised service engineers.

16.3.1. Warranty

Sold under statutory warranty conditions. Warranty lapses in the event of unauthorised attempts to repair the equipment and/or product, or in the event of any other form of intervention.

16.4. Nameplate

Erklärungen zu den Typenschildern (Beispiel)
nameplate descriptions (example)

Barcode ①
 Identifizierungs-Nr.
identification-no.

Modul-Typ ②
module type

Identifizierungs-Nr. ③
identification-no.

Modell / Bestell-Nr. ④
model / order-number

Version ⑤

Versorgungsspannung ⑥
supply voltage

Datum / Date ⑦

CE Kennzeichnung ⑧
CE mark

①
 ② CDIO 16/16-0,5 -1131
 ③ Num. : 20110300300329
 ④ Modell : 2011030
 ⑤ /version: 03
 ⑥ SELV 24V DC; 12A max. ⑧

⑦
 ①
 ② CDIO 16/16-0,5-1131
 ③ Num. : 20122302000001
 ④ Modell : 2012230
 ⑤ Version: 20
 ⑥ SELV 24V DC; 12 A max.

①
 ③ 00836400001073 ② KS800-CAN
 ④ Typ:9407 481 60001
 ⑦ Nr. :8346
 ⑤ Version: 2.1
 ⑥ 24V DC; 5W intern ⑧
 Made in Germany

2VF100080DG01.cdr

- ① **Barcode**
same as identification number.
- ② **Module type**
plain-text name of module.
- ③ **Identification no.**
module's identification number.
- ④ **Model/order no.**
You only need to give this number when ordering a module. The module will be supplied in its current hardware and software version.
- ⑤ **Version**
defines the design-level of the module as supplied ex-works.
- ⑥ **Supply voltage**
- ⑦ **Date**
internal code.
- ⑧ **CE mark**



The 'Version' (supply version) panel specifies the design-level of the module as supplied ex-works.

When replacing a module, users, with the CNW (Control Node Wizard) tool, can read off the current software version of the newly supplied module, and then re-load their 'own' software version for a particular project if necessary. With the latter in mind, before the download you should always keep a record of the existing software levels in your project documentation (software version, node IDs, baud rate, etc.).

16.5. Addresses and Bibliography

16.5.1. Addresses

CiA 'CAN in Automation', international manufacturers and users organisation for CAN users in the field of automation:

CiA - CAN in Automation e.V.
Am Weichselgarten 26
D-91058 Erlangen /Germany
e-mail: headquarters@can-cia.de
<http://www.can-cia.de>

DIN-EN Standards Beuth Verlag GmbH or VDE-Verlag GmbH
10772 Berlin 10625 Berlin

IEC Standards VDE Verlag GmbH or Internet search
10625 Berlin <http://www.iec.ch/>

16.5.2. Standards/Bibliography

IEC61131-1/EN61131-1	Programmable controllers Part 1: General information
IEC61131-2/EN61131-2	Programmable controllers Part 2: Equipment requirements and tests
IEC61131-3/EN61131-3	Programmable controllers Part 3: Programming languages
IEC61131-4/EN61131B1	Programmable logic controllers Supplementary Sheet 1: User guidelines
EN 50081 Parts 1+2	German EMC Act: Emitted interference
EN 50082 Parts 1+2	German EMC Act: Noise immunity
ISO/DIS 11898	Draft International Standard: Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high-speed communication
EN 954-1	Safety of machinery: Safety-related parts of control systems (Part 1)
Bibliography	A variety of specialist publications on the CANbus is available from specialist bookshops, or can be obtained through the CiA users' organisation.

NOTICE

Our Technical Support team will be glad to provide other literature references on request.

17. Index

A

Analog Library for Dialog Controllers	119
AnalogIn.LIB	119
Application ID	127

C

CALLHOOK	133
CAN interfaces	162
CAN Interfaces	21
CAN.LIB	162
CAN_DEFINE_COBID	166
CAN_DEFINE_GATEWAY	172
CAN_INIT	164
CAN_RECEIVE	167
CAN_STATE	171
CAN_TRANSMIT	169
CANopen Master	45
CNV_ARRAY_TO_STR	79
CNV_DINT_TO_STR	75
CNV_REAL_TO_STR	77
CNV_REALSTR_TO_DINT	80
CNV_STR_TO_ARRAY	79
CNV_STR_TO_DINT	78
CNV_STR_TO_REAL	78
COM_ADD_NODE	50
COM_CFG_SYNC_PDO	69
COM_CHANGE_STATE	51
COM_GET_State	52
COM_GUARDING_MSG	54
COM_GUARDING_OFF	53
COM_GUARDING_ON	53
COM_INIT	49
COM_INST_PDO	64
COM_INST_SDO	55
COM_INST_SYNC	70
COM_PORT_INIT	47
COM_PORT_RESTART	48
COM_RECEIVE_PDO	67
COM_RECEIVE_SDO_DOMAIN	62
COM_RECEIVE_SDO_EDATA	60
COM_REQUEST_PDO	66
COM_SEND_PDO	65
COM_SEND_SDO_DOMAIN	58
COM_SEND_SDO_EDATA	56
COM_START_SYNC	71
COM_STOP_SYNC	71
COM_UPDATE_PDO	68
COMASTER.LIB	45
COUNTER_GET	116
COUNTER_INIT	115
COUNTER_SET	116
CP1131 Multitasking	129
Cycle time	103

D

DIO_CLOSE	110
DIO_GET	111
DIO_INIT	109
DIO_SET	111

E

ECAN.LIB	21
ECAN_BRIDGE	34
ECAN_EXTSTATE	38
ECAN_GET_NODEID	35
ECAN_HWSTATE	37
ECAN_INIT	24
ECAN_RX	29
ECAN_SET_EWL	36
ECAN_SET_HWFILTER	28
ECAN_SET_LOM	39
ECAN_SET_STM	40
ECAN_SET_SWFILTER	27
ECAN_STATE	33
ECAN_TX	31
Error codes	72, 102

F

FOD_DELETE	158
FOD_GET_MAXINDEX	154
FOD_PREPARE_WRITE	156
FOD_READ	152
FOD_READ_INFO	153
FOD_RESET	157
FOD_SETUP	151
FOD_Vxx.LIB	139, 151
FOD_WRITE	155

I

Internal CAN driver errors	73
Internal error	73
IP_OWN_MAC_ADR	101
IP_OWN_NUMBER	101
IP_TCP_CONNECT_NEW_CLIENT	91
IP_TCP_CONNECT_TO_SERVER	95
IP_TCP_DISCONNECT	96
IP_TCP_DISCONNECT_TIMEOUT	97
IP_TCP_READ	98
IP_TCP_REGISTER_SERVER	92
IP_TCP_SERVERSTATE	93
IP_TCP_STATE	99
IP_TCP_WRITE	100
IP_UDP_CLEAR_BUFFER	85
IP_UDP_CREATE_CONNECTOR	86
IP_UDP_DELETE_CONNECTOR	87
IP_UDP_READ	88
IP_UDP_STATE	89
IP_UDP_WRITE	90

N			
Node ID	173		
P			
PLC_DisableCycleStatistics	105		
PLC_EnableCycleStatistics	105		
PLC_GetCycleStatistics	106		
PLC_GetMaxTaskDelay	133		
PLC_GetTaskID	135		
PLC_GetTaskInfo	136		
PLC_REBOOT	138		
PLC_RESCHEDULE	134		
PLC_ResetCycleStatistics	105		
PLC_ResumeTask	137		
PLC_RetriggerCycleTime	104		
PLC_SetMaxCycleTime	104		
PLC_STOP	138		
PLC_SuspendTask	137		
POSITION_REL_GET	174		
POSITION_REL_SET	118, 175		
POSITION_SOURCE_GET	118		
POSITION_SOURCE_SELECT	117		
PWM_CHANGE	113		
PWM_CLOSE	114		
PWM_INIT	112		
Q			
QAIO.LIB	41		
QAIO_ANALOGIN	42		
QAIO_ANALOGOUT	43		
Quadrature encoder (INCR.LIB)	174		
S			
Saving Application Data to the Flash Memory ..	139		
Serial Interfaces	13		
SIO.LIB	13		
SIO_CLOSE	19		
SIO_CLOSESVC	20		
SIO_INIT	15		
SIO_INITSVC	20		
SIO_RECEIVE	16		
SIO_STATE	18		
SIO_TRANSMIT	17		
SVC_GET_APPID	127		
SVC_SET_APPID	127		
SVCAppId.LIB	127		
SYS_GETTIMER	107		
SysInfo.LIB / SysInfo-CP1131.Lib	121		
SYSINFO_GET_CPUTYPE	122		
SYSINFO_GET_FWVERSION	123		
SYSINFO_GET_QBUSMODTYPE	125		
SYSINFO_GET_VERSION	124		
System Information	121		
SYSTIME.LIB	103		
T			
TaskManager.LIB	129		
TCP/UDP protocol driver	81		
TPU blocks for rapid inputs/outputs	109		
Transmission Control Protocol	82		
W			
Warning codes	102		