SYS TEC-specific Modbus Function Block Library for OpenPCS

User Manual

Version 1.0

Edition February 2016

Document no.: L-1829e_01

SYS TEC electronic GmbH Am Windrad 2 D-08468 Heinsdorfergrund
Telefon: +49 (3765) 38600-0 Telefax: +49 (3765) 38600-4100
Web: http://www.systec-electronic.com Mail: info@systec-electronic.com

Status/Changes

Status: Released

Date/Version	Section	Changes	Author/Editor
2016/02/25 1.0	all	Creation ("TN_Modbus_FunctionBlockLibrary" reworked to this User Manual)	R. Sieber

Product names used in this manual which are also registered trademarks have not been marked extra. The missing © mark does not imply that the trade name is unregistered. Nor is it possible to determine the existence of any patents or protection of inventions on the basis of the names used.

The information in this manual has been carefully checked and is believed to be accurate. However, it is expressly stated that SYS TEC electronic GmbH does not assume warranty or legal responsibility or any liability for consequential damages which result from the use or contents of this user manual. The information contained in this manual can be changed without prior notice. Therefore, SYS TEC electronic GmbH shall not accept any obligation.

Furthermore, it is expressly stated that SYS TEC electronic GmbH does not assume warranty or legal responsibility or any liability for consequential damages which result from incorrect use of the hard or software. The layout or design of the hardware can also be changed without prior notice. Therefore, SYS TEC electronic GmbH shall not accept any obligation.

Inform yourselves:

Contact	**Direct**	**Your Local Distributor**
Address:	SYS TEC electronic GmbH Am Windrad 2 D-08468 Heinsdorfergrund GERMANY	Please find a list of our distributors under: http://www.systec-electronic.com/distributors
Ordering Information:	+49 (0) 37 65 / 38 600-0 info@systec-electronic.com	
Technical Support:	+49 (0) 37 65 / 38 600-0 support@systec-electronic.com	
Fax:	+49 (0) 37 65 / 38 600-4100	
Web Site:	http://www.systec-electronic.com	

1st Edition February 2016

Table of Contents

List of Tables

List of Figures

1 Introduction

This manual describes the application of the Modbus Function Block Library for OpenPCS. It describes the date types, functions and function blocks provided by the library. This library allows a PLC program for data exchange with other devices via Modbus protocol. The implementation supports Modbus/RTU as well as Modbus/TCP.

2 Modbus Overview and Basics

2.1 Modbus Function Overview

The Modbus extension is fully integrated in the PLC runtime system. It supports the following functions:

- Modbus RTU Master
- Modbus RTU Slave
- Modbus TCP Master
- Modbus TCP Slave

Table 1 shows an overview of the IEC61131-3 function blocks for Modbus. All components are realized as manufacturer-specific function blocks and therefore as a part of the PLC firmware.

Table 1: Overview of Function Blocks for Modbus

Function Block	Master Slave	Fnct. Code	Data Width	Meaning	Sect.
MODBUS_OPEN_INSTANCE	M / S			Open a new communication instance; the returned Handle is required for all other function blocks	5.2
MODBUS_CLOSE_INSTANCE	M / S			Close a communication instance	5.3
MODBUS_REGISTER_VAR_LIST	S			Register a list of PLC variables which should be linked to Modbus registers in Slave Mode	5.4
MODBUS_READ_REGS	M	03_H	16 bit word	Read multiple holding registers (e.g. configuration registers)	5.5
MODBUS_WRITE_SINGLE_REG	M	06_H	16 bit word	Write single holding register (e.g. configuration register)	5.6
MODBUS_WRITE_MULTI_REGS	M	10_H	16 bit word	Write multiple holding registers (e.g. configuration registers)	5.7
MODBUS_READ_WRITE_REGS	M	17_H	16 bit word	Read and Write multiple holding registers in a single Modbus transaction (e.g. configuration registers)	5.8
MODBUS_READ_INPUT_REGS	M	04_H	16 bit word	Read multiple input registers (e.g. analog inputs)	5.9

Function Block	Master Slave	Fnct. Code	Data Width	Meaning	Sect.
MODBUS_READ_DISCR_INPUTS	M	02_H	1 bit	Read multiple discrete input registers (e.g. digital inputs)	5.10
MODBUS_READ_COILS	M	01_H	1 bit	Read multiple coils (e.g. read back digital outputs)	5.11
MODBUS_WRITE_SINGLE_COIL	M	05_H	1 bit	Write single coil (e.g. digital output)	5.12
MODBUS_WRITE_MULTI_COILS	M	$0F_H$	1 bit	Write multiple coils (e.g. digital outputs)	5.13
MODBUS_RAW_PDU_REQUEST	M			Exchanges raw PDU telegrams with a Modbus Slave device	5.14

2.2 Modbus Master Basics

Using the PLC as Modbus Master, the PLC program can read and write registers on remote slave devices using the following Modbus master specific function blocks:

- *MODBUS_READ_REGS*
- *MODBUS_WRITE_SINGLE_REG*
- *MODBUS_WRITE_MULTI_REGS*
- *MODBUS_READ_WRITE_REGS*
- *MODBUS_READ_INPUT_REGS*
- *MODBUS_READ_DISCR_INPUTS*
- *MODBUS_READ_COILS*
- *MODBUS_WRITE_SINGLE_COIL*
- *MODBUS_WRITE_MULTI_COILS*
- *MODBUS_RAW_PDU_REQUEST*

Each of the listed function blocks above has an input parameter *SLAVE_ADDR*. The usage of this slave address depends on the communication type:

Modbus/RTU: On Modbus/RTU the master is connected to its slave devices on a serial bus (commonly a RS-485 bus). The handle for the communication instance created by function block *MODBUS_OPEN_INSTANCE* is related to the local serial interface only. The communication parameter string *COMM_PARAMS* of function block *MODBUS_OPEN_INSTANCE* (see section 5.2) specifies only communication parameters for the local serial interface such as interface number and baudrate. Thus, in case of Modbus/RTU the parameter *SLAVE_ADDR* of the master specific function blocks defines the device address of the remote slave on the serial bus. Valid slave addresses are in range 1..247. The address 0 can be used for broadcast messages.

Modbus/TCP: On Modbus/TCP the slave device is unequivocally addressed by its IP address and port number. Both values are already set in the communication parameter string *COMM_PARAMS* of function block *MODBUS_OPEN_INSTANCE* (see section 5.2). Thus, in case of Modbus/TCP the parameter *SLAVE_ADDR* of the master specific function blocks must be set to 255.

Exception: If the target device is reachable only via a gateway, then the IP address and port number specified with the communication parameter string *COMM_PARAMS* addresses the gateway and *SLAVE_ADDR* defines the slave device address in the destination network.

Using the PLC as Modbus Master requires the following steps:

(M1) Open a Modbus instance by calling the function block *MODBUS_OPEN_INSTANCE* (see section 5.2); on input *DEV_TYPE* either *MBDT_TCP_MASTER* or *MBDT_RTU_MASTER* has to be defined (see section 4.2)

(M2) Call a Modbus master specific function block, using the instance handle created in step (M1) and setting parameter *SLAVE_ADDR* as described above.

2.3 Modbus Slave Basics

Using the PLC as Modbus Slave, the PLC program can register a set of PLC variables which should be accessible from a remote master. The runtime configuration of a single variable is described in the PLC structure *MODBUS_DATA_POINT* (resp. *MODBUS_DATA_POINT_VAR_REF*, see section 4.4). This structure includes the following information:

- Modbus object type (discrete input, coil, input register, holding register)
- Modbus register number
- Pointer to PLC variable

The individual *MODBUS_DATA_POINT* structures for each variable are joined together in an array. By adjusting the array size the user can link as many variables as needed to the Modbus stack. The whole array is registered to the Modbus stack by calling the function block *MODBUS_REGISTER_VAR_LIST* (see section 5.4).

The structure *MODBUS_DATA_POINT* in conjunction with the FB *MODBUS_REGISTER_VAR_LIST* provides a maximum flexibility to the user for designing its individual Modbus Slave register layout.

Using the PLC as Modbus Slave requires the following steps:

(S1) Open a Modbus instance by calling the function block *MODBUS_OPEN_INSTANCE* (see section 5.2); on input *DEV_TYPE* either *MBDT_TCP_SLAVE* or *MBDT_RTU_SLAVE* has to be defined (see section 4.2)

(S2) Create the slave variables list as *ARRAY[x..y] OF MODBUS_DATA_POINT* (see section 4.4)

(S3) Register the slave variables list registered to the Modbus stack by calling the function block *MODBUS_REGISTER_VAR_LIST* (see section 5.4); the Modbus instance handle created in step (S1) binds to variables list to the dedicated interface.

3 Error Codes of Modbus Function Blocks

Table 2 lists the Error Codes used by all function blocks of type *MODBUS_Xxx* for output *ERROR*.

Table 2: Error Codes of Modbus Function Blocks used for output ERROR

Error Code	Meaning
0	FB execution was successfully, no error occurred
1	Invalid parameter (e.g. structure has an unexpected size; enumerator has an unknown value etc.)
2	Pointer references an object with an unsupported data type
3	Out of resources (e.g. out of memory, or not more instances available)
4	A slave instance with requested connection already exists
5	Requested resource is busy
6	Connection to remote node failed
7	Master request failed (e.g. timeout)
8	Invalid handle
9	Size of variable in User/PLC program doesn't match with number of registers to read/write
10	Requested function only available for Master (requires handle for Master instance)
11	Requested function only available for Slave (requires handle for Slave instance)
12	Invalid data point configuration for ModBus slave variable
255	Function not implemented on the PLC

Table 3 lists the Error Info Codes used by all Modbus master function blocks for output *ERROR_INFO*. These codes are the internal error codes of the underlying Modbus protocol stack.

Table 3: Error Info Codes of Modbus Function Blocks used for output ERROR_INFO

Error Code	Meaning
0	No error
1	Illegal register address
2	Illegal argument
3	Porting layer error
4	Insufficient resources
5	I/O error
6	Protocol stack in illegal state
7	Retry I/O operation
8	Timeout error occurred
10	Illegal function exception
11	Illegal data address

Error Code	Meaning
12	Illegal data value
13	Slave device failure
14	Slave acknowledge
15	Slave device busy
16	Memory parity error
17	Gateway path unavailable
18	Gateway target device failed to respond

4 Type Definitions for Modbus Function Blocks

All types shown in this section are defined global in the *OpenPCS Programming System* environment. They must not be redeclared in the User/PLC program. All of this global defined Modbus types can be used within User/PLC program as common standard types, e.g. *INT* or *POINTER*.

4.1 Data Type HMODBUS

The data type *HMODBUS* is globally defined in *OpenPCS*. It is used as a *"Handle to Modbus Instance"*. It is returned by function block *MODBUS_OPEN_INSTANCE* (see section 5.2) and is used as input parameter for all other function blocks.

```
TYPE
  HMODBUS : DINT;
END_TYPE
```

4.2 Enum MODBUS_DEV_TYPE

The enumerator *MODBUS_DEV_TYPE* is globally defined in *OpenPCS*. It is specifies the kind of Modbus Instance for the own device. It is used to create a new instance by calling the function block *MODBUS_OPEN_INSTANCE* (see section 5.2).

```
GLOBAL_TYPE_BEGIN
  TYPE
    MODBUS_DEV_TYPE :
    ( MBDT_NOT_SET,           (* 0: Device Type uninitialized / not set *)
      MBDT_TCP_SLAVE,         (* 1: Device is ModBus TCP Slave *)
      MBDT_RTU_SLAVE,         (* 2: Device is ModBus RTU Slave *)
      MBDT_TCP_MASTER,        (* 3: Device is ModBus TCP Master *)
      MBDT_RTU_MASTER         (* 4: Device is ModBus TCP Master *)
    ) := MBDT_NOT_SET;
  END_TYPE
GLOBAL_TYPE_END
```

The device type defined by the enumerator *MODBUS_DEV_TYPE* refers to the **own device**. It is used for creating the appropriate instance handle with *MODBUS_OPEN_INSTANCE*. Table 4 explains the enumerator values of *MODBUS_DEV_TYPE*.

Table 4: Description of MODBUS_DEV_TYPE

Type	Description
MBDT_TCP_MASTER, MBDT_RTU_MASTER	The own PLC wants to act as master using the master function blocks listed in Table 1
MBDT_TCP_SLAVE, MBDT_RTU_SLAVE	The own PLC wants to run as slave linking a list of PLC variables to Modbus registers via function block *MODBUS_REGISTER_VAR_LIST* (see section 5.4)

4.3 Enum MODBUS_OBJ_TYPE

The enumerator *MODBUS_OBJ_TYPE* is globally defined in *OpenPCS*. It specifies to which register category a User/PLC variable should be linked in Modbus Slave Mode. This type is used for registering PLC variables with function block *MODBUS_REGISTER_VAR_LIST* (see section 5.4).

```
GLOBAL_TYPE_BEGIN
  TYPE
    MODBUS_OBJ_TYPE :
    ( MBOT_NOT_SET,           (* 0: Variable uninitialized / not set *)
      MBOT_DISCRETE_INPUT,    (* 1: Bit Variables of an I/O System *)
      MBOT_COILS,             (* 2: Bit Variables of an Application *)
      MBOT_INPUT_REGISTERS,   (* 3: 16bit Word Variables of an I/O System *)
      MBOT_HOLDING_REGISTERS  (* 4: 16bit Word Variables of an Application *)
    ) := MBOT_NOT_SET;
  END_TYPE
GLOBAL_TYPE_END
```

4.4 Structure MODBUS_DATA_POINT_Xxx

The both alternative structures *MODBUS_DATA_POINT* and *MODBUS_DATA_POINT_VAR_REF* are globally defined in *OpenPCS*. They specify mapping information to link a User/PLC variable to a Modbus register in Slave Mode. These structures are used for registering PLC variables with function block *MODBUS_REGISTER_VAR_LIST* (see section 5.4).

The more effective structure *MODBUS_DATA_POINT* uses a *POINTER* for a direct addressing of the variable associated with this Modbus data point (member *'m_pDataVar'*, see section 4.4.1). Using this feature effectuates an easy and clear program structure. However, *POINTER* in structures is a new feature of *OpenPCS* which is only available since *OpenPCS* version 6.8.0.

For backward compatibility to *OpenPCS* versions before 6.8.0 (up to and including *OpenPCS* version 6.7.4) the alternative structure *MODBUS_DATA_POINT_VAR_REF* is supported. This structure doesn't contain a member of type *POINTER*, but the 4 equivalent members '*m_udiDataVarMemAdd*', '*m_usiDataVarBitAdd*', '*m_usiDataVarCbeType'* and '*m_uiDataVarSize'* (see section 4.4.2). The information content of these 4 members is exactly the same as for the one member *'m_pDataVar'* in structure *MODBUS_DATA_POINT*. The Firmware Function *POINTER_TO_VARREF* converts a given pointer variable into these 4 separate parameters (see example in section 4.4.2).

The structure *MODBUS_DATA_POINT_VAR_REF* is maintained even in newer versions of *OpenPCS*. Thus, PLC programs using structure *MODBUS_DATA_POINT_VAR_REF* can also used in with newer version of *OpenPCS* without any changes.

4.4.1 Structure MODBUS_DATA_POINT

The structure *MODBUS_DATA_POINT* specifies mapping information to link a User/PLC variable to a Modbus register in Slave Mode. This structure is used for registering PLC variables with function block *MODBUS_REGISTER_VAR_LIST* (see section 5.4).

```
GLOBAL_TYPE_BEGIN
  TYPE
    MODBUS_DATA_POINT : STRUCT
      m_ObjType    : MODBUS_OBJ_TYPE;   (* ModBus Object Type *)
      m_uiDataAddr : UINT;              (* Register/Coil Number (0..65535) *)
      m_pDataVar   : POINTER;           (* Pointer to associated Variable *)
    END_STRUCT;
  END_TYPE
GLOBAL_TYPE_END
```

Important: This structure is applicable only from *OpenPCS* version 6.8.0. For older *OpenPCS* versions up to and including version 6.7.4 the alternative structure *MODBUS_DATA_POINT_VAR_REF* has to be used (see section 4.4.2).

Example:

The following code snippet shows how to create a data point for a variable using the structure *MODBUS_DATA_POINT*. The ***bold red*** marked part handles the pointer addressing.

```
VAR

    (* ModBus Data Variable *)
    xModbusDataVar              : BOOL := TRUE;

    (* Slave Variables List *)
    aModBus_DataPoint_List      : ARRAY[1..1] OF MODBUS_DATA_POINT;

END_VAR

    aModBus_DataPoint_List[1].m_ObjType    := MBOT_DISCRETE_INPUT;
    aModBus_DataPoint_List[1].m_uiDataAddr := 1;
    aModBus_DataPoint_List[1].m_pDataVar   := &xModbusDataVar;
```

4.4.2 Structure MODBUS_DATA_POINT_VAR_REF

The structure *MODBUS_DATA_POINT_VAR_REF* is used for backward compatibility to support Modbus functionality even for *OpenPCS* versions before 6.8.0 (up to and including *OpenPCS* version 6.7.4). For *OpenPCS* 6.8.0 or newer the alternative and more comfortable *MODBUS_DATA_POINT* is recommended (see section 4.4.1).

The structure *MODBUS_DATA_POINT_VAR_REF* specifies mapping information to link a User/PLC variable to a Modbus register in Slave Mode. This structure is used for registering PLC variables with function block *MODBUS_REGISTER_VAR_LIST* (see section 5.4). The runtime values for members '*m_udiDataVarMemAdd*', '*m_usiDataVarBitAdd*', '*m_usiDataVarCbeType*' and '*m_uiDataVarSize*' has to be requested by using the firmware function *POINTER_TO_VARREF*. This converts a given pointer variable into these 4 parameters (see example below).

```
GLOBAL_TYPE_BEGIN
  TYPE
    MODBUS_DATA_POINT_VAR_REF : STRUCT
      m_ObjType    : MODBUS_OBJ_TYPE;   (* ModBus Object Type *)
      m_uiDataAddr : UINT;              (* Register/Coil Number (0..65535) *)
      m_udiDataVarMemAdd  : UDINT;      (* -+                              *)
      m_usiDataVarBitAdd  : USINT;      (*  | Reference to associated Var.  *)
      m_usiDataVarCbeType : USINT;      (*  | as 'embedded' struct 'VAR_REF' *)
      m_uiDataVarSize     : UINT;       (* -+                              *)
    END_STRUCT;
  END_TYPE
GLOBAL_TYPE_END
```

Note: For *OpenPCS* systems newer then version 6.7.4 the more comfortable and easier-to-use structure *MODBUS_DATA_POINT* is recommended (see section 4.4.1).

Example:

The following code snippet shows how to create a data point for a variable using the structure *MODBUS_DATA_POINT_VAR_REF*. The ***bold red*** marked part handles the pointer reference addressing.

```
VAR

    (* ModBus Data Variable *)
    xModbusDataVar              : BOOL := TRUE;
    VarRef                      : VAR_REF;
    pVarPtr                     : POINTER;

    (* Slave Variables List *)
    aModBus_DataPoint_List      : ARRAY[1..1] OF MODBUS_DATA_POINT_VAR_REF;

END_VAR

    (* Get 'VarRef' for ModBus Data Variable *)
    pVarPtr := &xModbusDataVar;
    VarRef  := POINTER_TO_VARREF (pVarPtr);

    aModBus_DataPoint_List[1].m_ObjType           := MBOT_DISCRETE_INPUT;
    aModBus_DataPoint_List[1].m_uiDataAddr        := 1;
    aModBus_DataPoint_List[1].m_udiDataVarMemAdd  := VarRef.m_udiMemAddress;
    aModBus_DataPoint_List[1].m_usiDataVarBitAdd  := VarRef.m_usiBitAddress;
    aModBus_DataPoint_List[1].m_usiDataVarCbeType := VarRef.m_usiCbeType;
    aModBus_DataPoint_List[1].m_uiDataVarSize     := VarRef.m_uiDataSize;
```

4.5 Representation of PLC Variables on Modbus

The memory layout of the variables in the PLC program (little-endian, big-endian) depends on the CPU type of the PLC device. However, on Modbus all 16 bit register word are transmitted with high-byte first (big-endian). Therefore the PLC firmware converts the data representation from the host memory layout into the Modbus standard big-endian format.

Table 5 in Appendix A describes in detail how PLC variables are mapped to Modbus registers.

5 Function Blocks

5.1 Synchronization between the Modbus Function Blocks and PLC Program

Internally most of the Modbus library function blocks are executed asynchronous to the PLC program. The synchronization between the function blocks and the PLC program is performed by the *ENABLE* input and *CONFIRM* output. The interaction of both signals is shown in Figure 1.

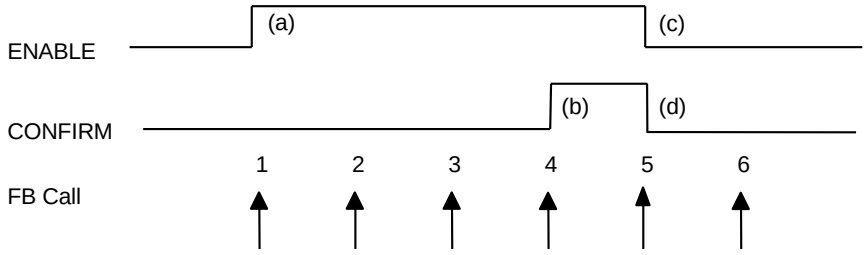


Figure 1: Process synchronization between Function Block and PLC Program

The complete execution cycle of a function block asynchronously to the PLC program is performed by following steps:

1. After the PLC program has provided all data to the inputs, it set *ENABLE* to *TRUE* and calls the function block (call 1). The function block recognizes a rising edge at *ENABLE* and samples all input data (step (a)). The requested operation is started internally and the function block returns to the PLC program. The started operation itself runs asynchronously in the background.

2. The PLC program calls the function block in the following PLC cycles and holds *ENABLE* on *TRUE*. The function block internally runs the requested operation in background (calls 2 and 3).

3. If the requested operation has been finished (either successfully or with an error), the outputs *CONFIRM*, *ERROR* and *ERRORINFO* are set accordingly. On a successfully completion the output *CONFIRM* is set to *TRUE*. In this case, the data outputs of the function block contain valid data. In case of any error, the outputs *ERROR* and *ERRORINFO* signals the error reason (step (b), call 4).

4. The PLC program reads the output data and after that it confirms the function block by setting input *ENABLE* to *FALSE*. On the next call the function block reset it's internally state and clears all outputs (step (c), call 5). By setting output *CONFIRM* to *FALSE* the function block signals it's readiness for the next operation request (step (d)).

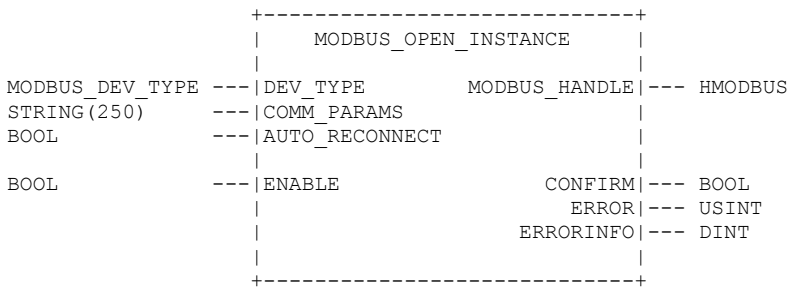
The output *CONFIRM* is only set to *TRUE*, if the requested operation has been completed successfully. In case of any error, the outputs *ERROR* or *ERRORINFO* are setting accordingly. Therefore it's necessary that a PLC program always checks *CONFIRM* as well as *ERROR*.

If the PLC program calls the function block with *ENABLE := FALSE* during an asynchronously operation is running in the background, the active operation will canceled and the function resets it's internally state to inactive and clears all outputs.

5.2 Function Block MODBUS_OPEN_INSTANCE

The function block *MODBUS_OPEN_INSTANCE* opens a new communication instance.

Prototype of the Function Block



Meaning of Operands

DEV_TYPE	Specifies which role the **own device** gets for the connection to be created. This parameter specifies the kind of destination node. Possible values are defined as enumerator *MODBUS_DEV_TYPE* (see section 4.2).
COMM_PARAMS	String with open parameters for the communication interface. The parameter string depends on the used interface. For details see text below.
AUTO_RECONNECT	TRUE: instructs the Modbus stack to auto-reconnect to the slave node in case of a communication interruption FALSE: an interrupted communication is not automatically reconnected by the Modbus stack
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information (max. number of possible connections, see text below)

Description

This function block opens a new communication instance. The parameter *DEV_TYPE* specifies the type of the destination node. The possible values are defined as enumerator *MODBUS_DEV_TYPE* (see section 4.2). For example, *DEV_TYPE := MBDT_TCP_SLAVE* is used to open a communication instance to a Modbus TCP Slave device.

The parameter *COMM_PARAMS* specifies the communication parameter necessary to open the connection to the destination node. The structure depends on the communication interface type (Ethernet for Modbus/TCP or serial interface for Modbus/RTU):

- **Communication Parameter string for Master to open a connection to a Modbus/TCP Slave:**

  Scenario: own PLC operates as Master
  remote devices operates as Slave

  `Format:` `ETH<IfNum>#<IpAddr>:<PortNum>`

  IfNum: Interface number to communicate with destination slave (e.g. "0" for ETH0)
  IpAddr: IP address of destination slave (e.g. "192.168.1.27")
  PortNum: Port number on destination slave (e.g. "502" for default Modbus port)

  Example: ETH0#192.168.1.27:502

  In this example the local interface 'ETH0' is used to establish a connection from Master to the Slave device with IP address '192.168.1.27', port number 502.

  Simplifications:

  COMM_PARAMS:= 'ETH0#192.168.1.27'; The specified Ethernet device with the specified IP address and Modbus default port (502) is used.

- **Communication Parameter string for Master to open a connection to a Modbus/RTU Slave:**

  Scenario: own PLC operates as Master
  remote devices operates as Slave

  `Format:` `SIO<IfNum>#<BaudRate>,<Parity>,<FrameTimeout>,<RespTimeout>`

  IfNum: Interface number to communicate with destination slave (e.g. "2" for SIO2)
  BaudRate: Baudrate (e.g. "19200" or "115200")
  Parity: 'N' = none parity (2 stop bits are used, see note below)
  'O' = odd parity (1 stop bit is used, see note below)
  'E' = even parity (1 stop bit is used, see note below)
  FrameTimeout: Optional frame timeout in number of characters. This value specifies a number of characters for the silent interval between two frames. Depending on baudrate this number of characters is converted in the internal wait time.
  If no explicit frame timeout is set in the communication parameter string then a standard timeout of 10 ms is used.
  RespTimeout: Optional response timeout in ms. This value specifies the maximum time period within the master expects a response from the slave. If no response was received from the slave within this period, than a communication timeout is signaled (output *ERRORINFO* is set to *"Timeout error occurred"*, see Table 3 in section 3).
  If no explicit timeout is set in the communication parameter string then a standard timeout of 500 ms is used

  Example: SIO2#115200,E,20,1000

  In this example the local interface 'SIO2' on Master is used. The serial port is configured to 115200 baud with even parity. The frame timeout is set to 20 ms and the response timeout is configured with 1000 ms.

  Note: In a Modbus frame each character is encoded with 11 bits. The character starts with 1 start bit, followed by 8 data bits. If parity is used, then 1 parity bit and 1 stop bit are appended. If no parity is used, then 2 stop bits are appended.

- **Communication Parameter string for Slave to run own device as Modbus/TCP Slave:**

  The Communication Parameter string for Modbus/TCP Slave mode doesn't include an IP address section. In Modbus/TCP Slave mode the PLC always uses the configured system IP address of the selected Ethernet interface. The procedure for configuring the system IP address depends on the target. The respective System Manual describes the necessary details on this (e.g. configuring the IP address in Bootloader on Linux devices or setting the IP address in the Configuration Command Shell).

  Scenario: own PLC operates as Slave
  remote devices operates as Master

  `Format:` `ETH<IfNum>#:<PortNum>`

  IfNum: Interface number (e.g. "0" for ETH0); the configured system IP address of this interface is used for Modbus Slave
  PortNum: Port number (e.g. "502" for default Modbus port)

  Example: ETH0#:502

  In this example the local interface 'ETH0' on Slave is used. The Slave is assigned to port number 502.

  Simplifications:

  COMM_PARAMS:= ''; On an empty parameter string the default Ethernet device (typically 'ETH0') with Modbus default port (502) is used.

  COMM_PARAMS:= 'ETH0'; The specified Ethernet device with its configured system IP address and Modbus default port (502) is used.

- **Communication Parameter string for Slave to run own device as Modbus/RTU Slave:**

  Scenario: own PLC operates as Slave
  remote devices operates as Master

  `Format:` `SIO<IfNum>#<BaudRate>,<Parity>,<FrameTimeout>@<NodeAddr>`

  IfNum: Interface number (e.g. "2" for SIO2)
  BaudRate: Baudrate (e.g. "19200" or "115200")
  Parity: 'N' = none parity (2 stop bits are used, see note below)
  'O' = odd parity (1 stop bit is used, see note below)
  'E' = even parity (1 stop bit is used, see note below)
  FrameTimeout: Optional frame timeout in number of characters. This value specifies a number of characters for the silent interval between two frames. Depending on baudrate this number of characters is converted in the internal wait time.
  If no explicit frame timeout is set in the communication parameter string then a standard timeout of 10 ms is used.
  NodeAddr: Own Modbus/RTU Slave address (e.g. "12" for running own device as Node 12)

  Example: SIO2#115200,E,20@12

  In this example the local interface 'SIO2' on Slave is used. The serial port is configured to 115200 baud with even parity. The frame timeout is set to 20 ms. The Slave is assigned to node address 12.

Note: In a Modbus frame each character is encoded with 11 bits. The character starts with 1 start bit, followed by 8 data bits. If parity is used, then 1 parity bit and 1 stop bit are appended. If no parity is used, then 2 stop bits are appended.

The parameter *AUTO_RECONNECT* instructs the Modbus stack either to automatically reconnet an interrupted communication to a Modbus Slave device or not.

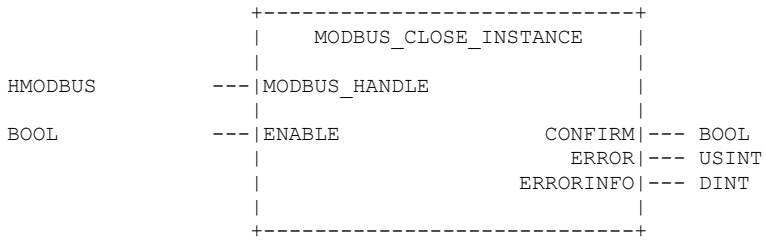
If the function block was able to open the connection successfully, an appropriate handle is returned on output *MODBUS_HANDLE*. This handle is requested for further calls of all other function blocks.

If the function block set its output *CONFIRM* to *TRUE*, the requested operation has been finished successfully. Otherwise the output *ERROR* signals an appropriate error code according to Table 2. Additionally, output *ERROR_INFO* is set to the maximum number of possible connections.

5.3 Function Block MODBUS_CLOSE_INSTANCE

The function block *MODBUS_CLOSE_INSTANCE* closes an existing communication instance.

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to close
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information

Description

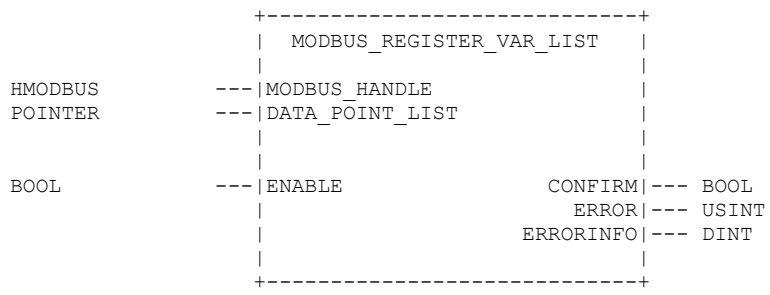
This function block closes a communication instance, opened prior by function block *MODBUS_OPEN_INSTANCE* (see section 5.2).

If the function block set its output *CONFIRM* to *TRUE*, the requested operation has been finished successfully. Otherwise the output *ERROR* signals an appropriate error code according to Table 2.

5.4 Function Block MODBUS_REGISTER_VAR_LIST

The function block *MODBUS_REGISTER_VAR_LIST* registers a set of PLC variables which should be accessible from a remote master.

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle is to request by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
DATA_POINT_LIST	Pointer to slave the slave variables list, defines as *ARRAY[x..y] OF MODBUS_DATA_POINT* (see section 4.4)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information (number of the array element which has caused this error, see text below)

Description

This function block registers a set of PLC variables which should be accessible from a remote master. Each single variable is defined by its individual instance of the *MODBUS_DATA_POINT* structure (see section 4.4). All data point instances are joined together in an array:

```
aModBus_DataPoint_List  :  ARRAY[x..y] OF MODBUS_DATA_POINT;
```

By adjusting the array size (lower/upper boundary *x, y*) the user can link as many variables as needed to the Modbus stack. The whole array is registered to the Modbus stack by calling the function block *MODBUS_REGISTER_VAR_LIST*. The function block internally checks the data point array for validity. If there is any incorrectness, then the function block stops checking on the first faulty element. The output *ERROR* describes the error reason (see Table 2 in section 3) and the output *ERRORINFO* is set to the number of the array element which has caused this error.

The parameter *MODBUS_HANDLE* identifies the communication instance to use. The handle is created by function block *MODBUS_OPEN_INSTANCE* (see section 5.2). To create a handle suitable for a Modbus slave instance, the function block *MODBUS_OPEN_INSTANCE* has to be called by setting input *DEV_TYPE* either to *MBDT_TCP_SLAVE* or to *MBDT_RTU_SLAVE* (see section 4.2).

Section 2.3 describes basics about using the PLC as Modbus Slave.

Example:

The following code snippet shows how to create and register a slave variables list:

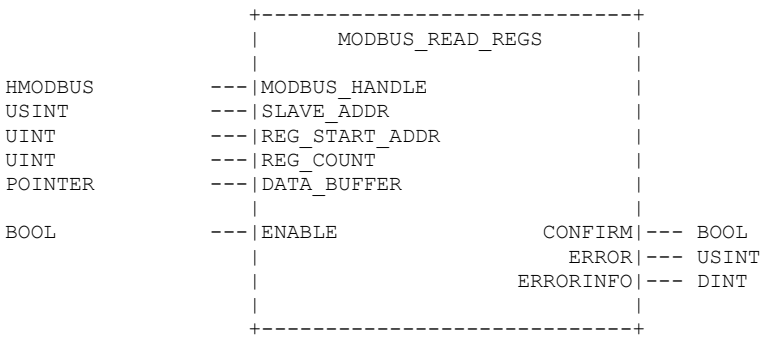
```
VAR

    (* ModBus Data Variables *)
    xModbusDataVar_DigiIn       : BOOL;
    uiModbusDataVar_AnalogIn    : UINT;
    wModbusDataVar_ModeConfig   : WORD;

    (* Slave Variables List *)
    aModBus_DataPoint_List      : ARRAY[1..3] OF MODBUS_DATA_POINT;
    paModBus_DataPoint_List     : POINTER;
    FB_ModBusRegisterVarList    : MODBUS_REGISTER_VAR_LIST;

END_VAR


    (* Link 'xModbusDataVar_DigiIn' as Reg1 @ DISCRETE_INPUT *)
    aModBus_DataPoint_List[1].m_ObjType    := MBOT_DISCRETE_INPUT;
    aModBus_DataPoint_List[1].m_uiDataAddr := 1;
    aModBus_DataPoint_List[1]. m_pDataVar  := &xModbusDataVar_DigiIn;

    (* Link 'uiModbusDataVar_AnalogIn' as Reg10 @ INPUT_REGISTERS *)
    aModBus_DataPoint_List[2].m_ObjType    := MBOT_INPUT_REGISTERS;
    aModBus_DataPoint_List[2].m_uiDataAddr := 10;
    aModBus_DataPoint_List[2]. m_pDataVar  := &uiModbusDataVar_AnalogIn;

    (* Link 'wModbusDataVar_ModeConfig' as Reg20 @ HOLDING_REGISTERS *)
    aModBus_DataPoint_List[3].m_ObjType    := MBOT_HOLDING_REGISTERS;
    aModBus_DataPoint_List[3].m_uiDataAddr := 20;
    aModBus_DataPoint_List[3]. m_pDataVar  := &wModbusDataVar_ModeConfig;


    (* Register ModBus DataPoint List *)
    paModBus_DataPoint_List := &aModBus_DataPoint_List;
    FB_ModBusRegisterVarList (
                    MODBUS_HANDLE   := hModbusInst,
                    DATA_POINT_LIST := paModBus_DataPoint_List,
                    ENABLE          := TRUE;
```

5.5 Function Block MODBUS_READ_REGS

The function block *MODBUS_READ_REGS* reads multiple holding registers from a Modbus slave device (e.g. configuration registers).

Modbus Function Code: **03_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_START_ADDR	Address of the first 16-bit holding register to read
REG_COUNT	Number of 16-bit holding registers to read
DATA_BUFFER	Address of an object (single variable or array) for receiving the register data to read (see section 4.5)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block reads multiple holding registers from a Modbus slave device (e.g. configuration registers). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *DATA_BUFFER* addresses a data object which acts as receive buffer for the requested data (see section 4.5 for details about representation of PLC variables on Modbus).

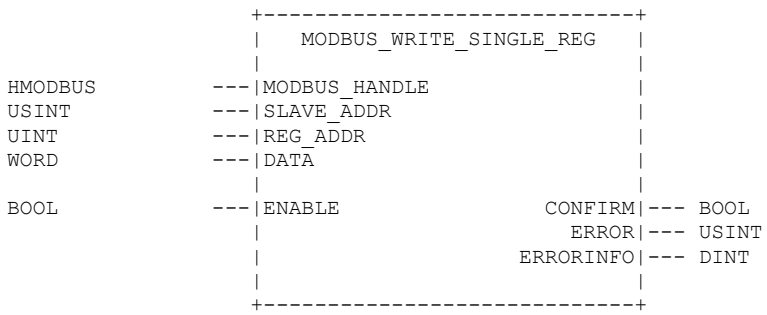
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the reading request is still running. The function block signals a successfully completion of the read operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.6 Function Block MODBUS_WRITE_SINGLE_REG

The function block *MODBUS_WRITE_SINGLE_REG* writes a single holding register to a Modbus slave device (e.g. configuration register).

Modbus Function Code: **06_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_ADDR	Address of the 16-bit holding register to write
DATA	Register data to send
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block writes a single holding register to a Modbus slave device (e.g. configuration register). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The input *DATA* contains the register data to send.

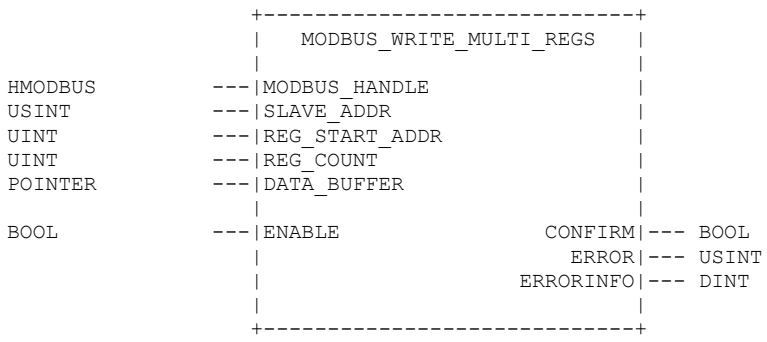
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the transmission request is still running. The function block signals a successfully completion of the write operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.7 Function Block MODBUS_WRITE_MULTI_REGS

The function block *MODBUS_WRITE_MULTI_REGS* writes multiple holding registers to a Modbus slave device (e.g. configuration registers).

Modbus Function Code: **10_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_START_ADDR	Address of the first 16-bit holding register to write
REG_COUNT	Number of 16-bit holding registers to write
DATA_BUFFER	Address of an object (single variable or array) which contains the register data to send (see section 4.5)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block writes multiple holding registers to a Modbus slave device (e.g. configuration registers). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *DATA_BUFFER* addresses a data object which contains the register data to send (see section 4.5 for details about representation of PLC variables on Modbus).

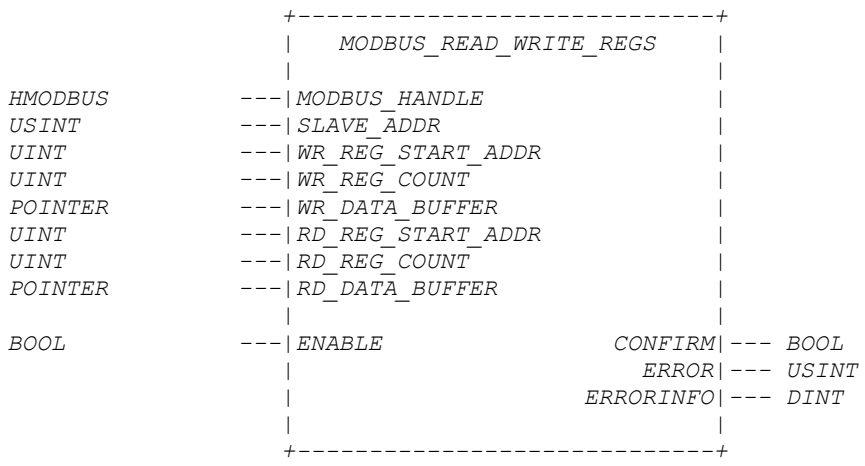
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the transmission request is still running. The function block signals a successfully completion of the write operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.8 Function Block MODBUS_READ_WRITE_REGS

The function block *MODBUS_READ_WRITE_REGS* reads and writes multiple holding registers to a Modbus slave device in a single Modbus transaction (e.g. configuration registers).

Modbus Function Code: **17_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
WR_REG_START_ADDR	Address of the first 16-bit holding register to write
WR_REG_COUNT	Number of 16-bit holding registers to write
WR_DATA_BUFFER	Address of an object (single variable or array) which contains the register data to send (see section 4.5)
RD_REG_START_ADDR	Address of the first 16-bit holding register to read
RD_REG_COUNT	Number of 16-bit holding registers to read
RD_DATA_BUFFER	Address of an object (single variable or array) for receiving the register data to read (see section 4.5)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block reads and writes multiple holding registers to a Modbus slave device in a single Modbus transaction (e.g. configuration registers). The write operation is performed before the read operation. The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *WR_DATA_BUFFER* addresses a data object which contains the register data to send. The pointer *RD_DATA_BUFFER* addresses a data object which acts as receive buffer for the requested data (see section 4.5 for details about representation of PLC variables on Modbus).

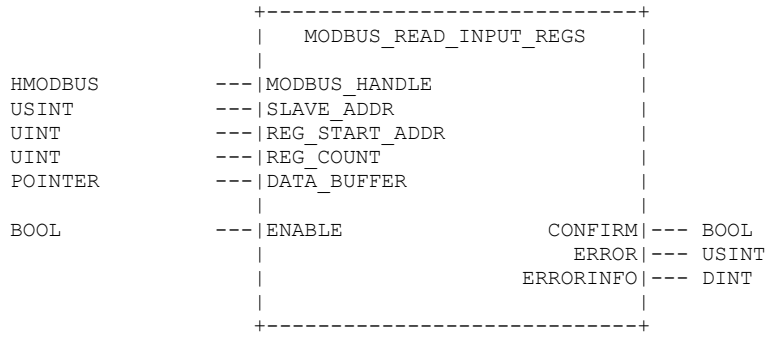
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the Modbus data transfer is still running. The function block signals a successfully completion of the operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.9 Function Block MODBUS_READ_INPUT_REGS

The function block *MODBUS_READ_INPUT_REGS* reads multiple input registers from a Modbus slave device (e.g. analog inputs).

Modbus Function Code: **04_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_START_ADDR	Address of the first 16-bit input register to read
REG_COUNT	Number of 16-bit input registers to read
DATA_BUFFER	Address of an object (single variable or array) for receiving the register data to read (see section 4.5)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block reads multiple input registers from a Modbus slave device (e.g. analog inputs). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *DATA_BUFFER* addresses a data object which acts as receive buffer for the requested data (see section 4.5 for details about representation of PLC variables on Modbus).

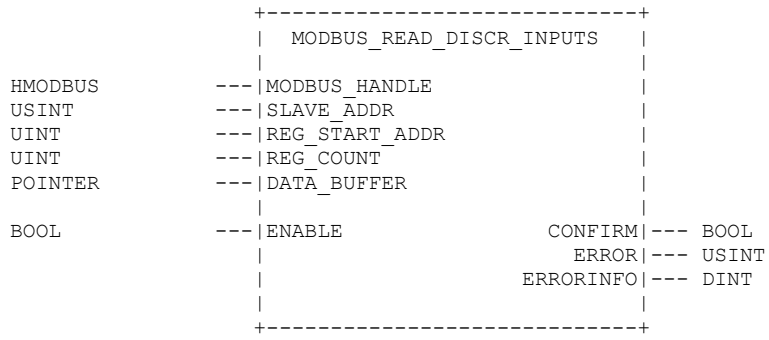
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the reading request is still running. The function block signals a successfully completion of the read operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.10 Function Block MODBUS_READ_DISCR_INPUTS

The function block *MODBUS_READ_DISCR_INPUTS* reads multiple discrete input registers from a Modbus slave device (e.g. digital inputs).

Modbus Function Code: **02_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_START_ADDR	Address of the first 1-bit discrete input register to read
REG_COUNT	Number of 1-bit discrete input registers to read
DATA_BUFFER	Address of an object (single variable or array) for receiving the register data to read (see section 4.5)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block reads discrete input registers from a Modbus slave device (e.g. digital inputs). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *DATA_BUFFER* addresses a data object which acts as receive buffer for the requested data (see section 4.5 for details about representation of PLC variables on Modbus).

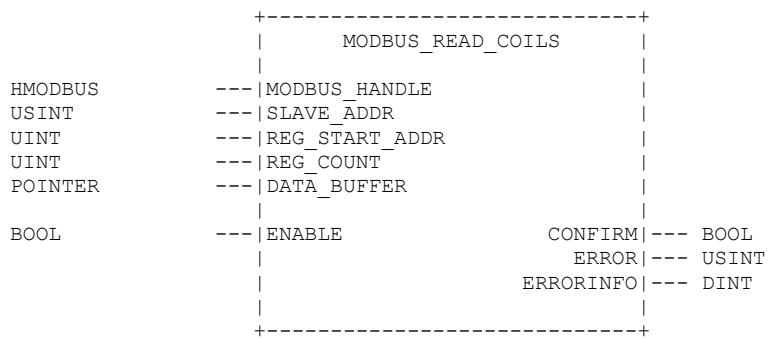
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the reading request is still running. The function block signals a successfully completion of the read operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.11 Function Block MODBUS_READ_COILS

The function block *MODBUS_READ_COILS* reads multiple coils from a Modbus slave device (e.g. read back digital outputs).

Modbus Function Code: **01_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_START_ADDR	Address of the first 1-bit coil to read
REG_COUNT	Number of 1-bit coil registers to read
DATA_BUFFER	Address of an object (single variable or array) for receiving the register data to read (see section 4.5)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function reads multiple coils from a Modbus slave device (e.g. read back digital outputs). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *DATA_BUFFER* addresses a data object which acts as receive buffer for the requested data (see section 4.5 for details about representation of PLC variables on Modbus).

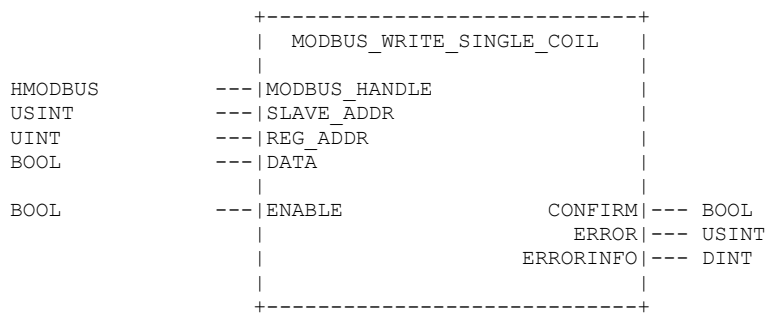
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the reading request is still running. The function block signals a successfully completion of the read operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.12 Function Block MODBUS_WRITE_SINGLE_COIL

The function block *MODBUS_WRITE_SINGLE_COIL* writes a single coil to a Modbus slave device (e.g. digital output).

Modbus Function Code: **05_H**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_ADDR	Address of the 1-bit coil to write
DATA	Register data to send
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block writes a single coil to a Modbus slave device (e.g. digital output). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The input *DATA* contains the register data to send.

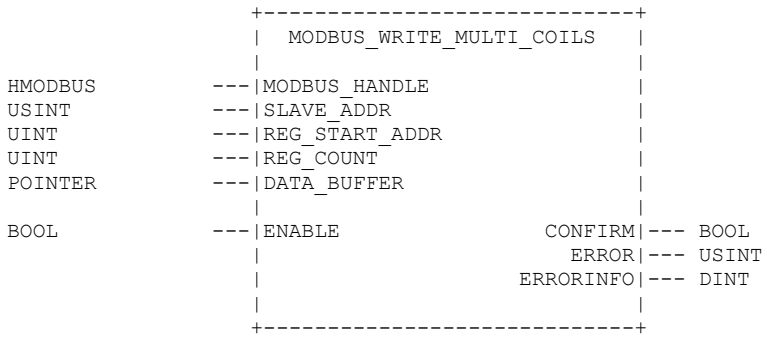
As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the transmission request is still running. The function block signals a successfully completion of the write operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.13 Function Block MODBUS_WRITE_MULTI_COILS

The function block *MODBUS_WRITE_MULTI_COILS* writes multiple coils to a Modbus slave device (e.g. digital outputs).

Modbus Function Code: **$0F_H$**

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
REG_START_ADDR	Address of the first 1-bit coil to write
REG_COUNT	Number of 1-bit coil registers to write
DATA_BUFFER	Address of an object (single variable or array) which contains the register data to send (see section 4.5)
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

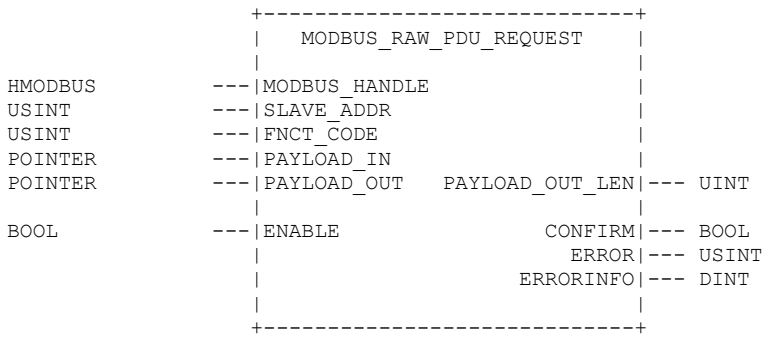
This function block writes multiple coils to a Modbus slave device (e.g. digital outputs). The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *DATA_BUFFER* addresses a data object which contains the register data to send (see section 4.5 for details about representation of PLC variables on Modbus).

As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the transmission request is still running. The function block signals a successfully completion of the write operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

5.14 Function Block MODBUS_RAW_PDU_REQUEST

The function block *MODBUS_RAW_PDU_REQUEST* exchanges raw PDU telegrams with a Modbus slave device.

Prototype of the Function Block



Meaning of Operands

MODBUS_HANDLE	Handle of the communication instance to use; this handle has to be requested by calling *MODBUS_OPEN_INSTANCE* (see section 5.2).
SLAVE_ADDR	Slave address of the remote node (see section 2.2)
FNCT_CODE	Modbus Function Code for the requested operation
PAYLOAD_IN	Address of a byte array which contains the payload data to send
PAYLOAD_OUT	Address of a byte array for receiving the payload data to read
PAYLOAD_OUT_LEN	Number of bytes valid in receive buffer addressed by *PAYLOAD_OUT*
ENABLE	Input for enabling or disabling the FB (see section 5.1)
CONFIRM	Output for ready signaling by the FB (see section 5.1)
ERROR	Execution result of the FB; possible error codes are defined in Table 2
ERRORINFO	Additionally error information; possible codes are defined in Table 3

Description

This function block exchanges raw PDU telegrams with a Modbus slave device. That allows using any functions that are not supported by this library. The required handle for parameter *MODBUS_HANDLE* has to be created by using the function block *MODBUS_OPEN_INSTANCE* (see section 5.2). The usage of the input parameter *SLAVE_ADDR* is described in section 2.2. The pointer *PAYLOAD_IN* addresses a byte array which contains the payload data to send. Equivalently the pointer *PAYLOAD_OUT* addresses a byte array which acts as receive buffer for the requested data.

For the both parameters *PAYLOAD_IN* and *PAYLOAD_OUT* only pointers to *ARRAY* of *BYTE*, *USINT* and *SINT* are accepted. The PLC firmware doesn't convert any of this data. That means that the PLC programmer itself is responsible for the correct data format.

The Transmit PDU is build from Function Code and payload data addressed by parameters *PAYLOAD_IN*. The Function Code is used as first byte of PDU, followed by the payload data (see Figure 2).

The first byte of the Receive PDU specifies the length of following payload data in bytes. This length information is set on output PAYLOAD_OUT_LEN as well as kept as first byte of the receive data block, stored in buffer addressed by parameter *PAYLOAD_OUT* (see Figure 2).

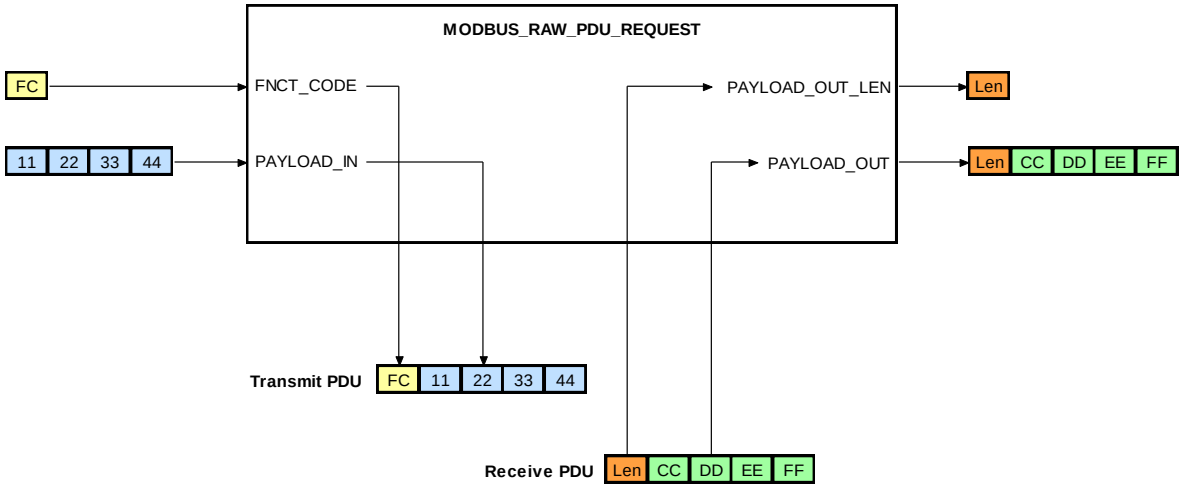


Figure 2: Structure of Transmit and Receive Raw PDU

As long as output *CONFIRM* remains *FALSE* and no error code is set on output *ERROR*, the requested operation is still running. The function block signals a successfully completion of the operation by setting of output *CONFIRM* to *TRUE*. In case of an error the output *CONFIRM* keeps *FALSE* and the output *ERROR* is set to an appropriate error code according Table 2.

6 Index

Appendix A: Mapping PLC Variables to Modbus Registers

Table 5: Mapping PLC Variables to Modbus Registers

Modbus Type	PLC Data Type	Modbus Register Representation
1 bit register	BOOL	Reg[n] := Var
	BYTE SINT USINT	Reg[n] := Var.0 Reg[n+1] := Var.1 ... Reg[n+7] := Var.7
	WORD INT UINT	Reg[n] := Var.0 Reg[n+1] := Var.1 ... Reg[n+15] := Var.15
	DWORD DINT UDINT	Reg[n] := Var.0 Reg[n+1] := Var.1 ... Reg[n+31] := Var.31
	BOOL[0..a]	Reg[n] := Var[0] ... Reg[n+a] := Var[a]
	BYTE[0..a] SINT[0..a] USINT[0..a]	Reg[n] := Var[0].0 Reg[n+1] := Var[0].1 ... Reg[n+(a*8)+7] := Var[a].7
	WORD[0..a] INT[0..a] UINT[0..a]	Reg[n] := Var[0].0 Reg[n+1] := Var[0].1 ... Reg[n+(a*16)+15] := Var[a].15
	DWORD[0..a] DINT[0..a] UDINT[0..a]	Reg[n] := Var[0].0 Reg[n+1] := Var[0].1 ... Reg[n+(a*32)+31] := Var[a].31
16 bit register	WORD INT UINT	Reg[n] := Var
	DWORD DINT UDINT	Reg[n] := Var.LowWord Reg[n+1] := Var.HighWord
	WORD[0..a] INT[0..a] UINT[0..a]	Reg[n] := Var[0] Reg[n+1] := Var[1] ... Reg[n+a] := Var[a]
	DWORD[0..a] DINT[0..a] UDINT[0..a]	Reg[n] := Var[0].LowWord Reg[n+1] := Var[0].HighWord ... Reg[n+(a*2)] := Var[a].LowWord Reg[n+(a*2)+1] := Var[a].HighWord

Document:	**SYS TEC Specific Modbus Function Block Library for OpenPCS**
Document number:	**L-1829-01, February 2016**

Do you have any suggestions for improving this manual?

Have you discovered any errors in this manual? Page

Sent from:

Customer number:

Name:

Company:

Address:

Send to:

SYS TEC electronic GmbH
Am Windrad 2
D – 08468 Heinsdorfergrund
GERMANY
Fax: +49 (0) 37 65 / 38600-4100
Email: info@systec-electronic.com