

## **u-remote library for Beckhoff TwinCAT3**

### **Abstract:**

This library package contains function blocks and functions for easy use of Weidmüller u-remote modules with TwinCAT3. This document only explains the function blocks that are available to the END USER. Internal function blocks are not explained in more detail.

### Hardware reference

No.	Component name	Article No.	Hardware / Firmware version
1	UR20-1COM-232-485-422 V1	1315750000	FW 1.00.16
2	UR20-1COM-232-485-422-V2	2826800000	-
3	UR20-4COM-IO-LINK	1315740000	-

### Software reference

No.	Software name	Article No.	Software version
1	Beckhoff TwinCAT	-	V3 or higher

### File reference

No.	Name	Description	Version
1	-	-	-

### Contact

Weidmüller Interface GmbH & Co. KG  
Klingenbergstraße 26  
32758 Detmold, Germany  
[www.weidmueller.com](http://www.weidmueller.com)

For any further support please contact your  
local sales representative:  
<https://www.weidmueller.com/countries>

## Revision history

Library Version	Date	Change log	Author
1.0.0	2022-01	Draft	w010174
1.0.0	2022-01	Review	w010521
1.1.0	2022-04	added libWiUr204ComIOLink	w010521
1.3.0	2023-07	Updated 1COM V1, 4COM-IO-Link, added 1COM V2, ModBus RTU Master	w010174
1.4.0	2024-02	Added libWiUr20Diag	w010174

## Content

1	Warning and Disclaimer.....	5
2	Standardized behavior of the function blocks.....	6
2.1	Function block variants .....	6
2.2	Basic states .....	6
2.3	Inputs and Outputs .....	6
2.4	Simplified behavior model.....	8
2.5	Usage of the function blocks.....	8
3	libWiUr20ComRs_232_485_422.....	10
3.1	FB_UR20_1ComRs_232_485_422_Control .....	10
3.2	FB_UR20_1ComRs_232_485_422_ParamEcCanRead .....	12
3.3	FB_UR20_1ComRs_232_485_422_ParamEcCanWrite .....	13
4	libWiUr204ComIOLink .....	15
4.1	FB_UR20_4COM_IO_LINK_EtherCATReadWrite .....	15
5	libWiUr20ComRs_232_485_422_V2 .....	17
5.1	FB_UR20_1ComRs_232_485_422_V2_Serial .....	17
5.2	FB_UR20_1ComRs_232_485_422_V2_ModbusMaster .....	20
5.3	FB_UR20_1ComRs_232_485_422_V2_ModbusSlave .....	22
6	libWiUr20ModbusRTUMaster .....	25
6.1	FB_ModbusRTUMaster .....	25

# 1 Warning and Disclaimer

## Warning

Controls may fail in unsafe operating conditions, causing uncontrolled operation of the controlled devices. Such hazardous events can result in death and / or serious injury and / or property damage. Therefore, there must be provided safety equipment / electrical safety design or other redundant safety features that are independent from the automation system.

## Disclaimer

This software (programs, function blocks, functions, etc.) does not relieve you of the obligation to handle it safely during use, installation, operation and maintenance. Every user is responsible for the correct operation of his control system.

By using this software prepared by Weidmüller, you accept that Weidmüller cannot be held liable for any damage to property and / or personal injury that may occur because of the use.

## Note

This software does not represent customer-specific solutions, it is simply intended to help for typical tasks. The user is responsible for the proper operation of the used products. This software does not relieve you of the obligation of safe use, installation, operation and maintenance. This software is not binding and do not claim to be complete in terms of configuration as well as any contingencies.

By using this software, you acknowledge that Weidmueller cannot be held liable for any damages beyond the described liability regime. We reserve the right to make changes to this software at any time without notice.

We assume no liability for this software or the information contained in this document. Our liability, for whatever legal reason, for damages caused by the use of the software or instructions described in this document is excluded.

## Security notes

In order to protect equipment, systems, machines and networks against cyber threats, it is necessary to implement (and maintain) a complete state-of-the-art industrial security concept. The customer is responsible for preventing unauthorized access to his equipment, systems, machines and networks. Systems, machines and components should only be connected to the corporate network or the Internet if necessary and appropriate safeguards (such as firewalls and network segmentation) have been taken.

## 2 Standardized behavior of the function blocks

All Weidmüller function blocks work in a defined manner. This behavior is defined by an internal state machine, which includes a set of standardized inputs and outputs. The following part describes the basic interfaces and behavior of the function blocks, as well as procedures to activate and deactivate the function blocks and how to handle errors.

### 2.1 Function block variants

There are in total four different variations of the underlying behavior.

- A level-controlled function block changes state by the level of the input signals (enable and execute)
- An edge-controlled function block changes state by evaluating the positive edges of its control inputs (enable, disable, execute, abort)

Both variants can either be implemented as a finite (with additional output "q\_xDone") or as a continuous behavior. The finite behavior is used for any kind of action that comes to a defined end, while the continuous behavior is used for any action of an infinite nature.

### 2.2 Basic states

There are four basic states of a function block:

- **idle**: no action is performed (initial state of each function block).
- **standby**: the function block is initialized and ready to be executed
- **active**: the function block is performing its intended task
- **error**: an error occurred, and the function block had to be stopped.

The states **standby** and **active** are separated into different sub-states, since the function block may take some time to be initialized (entering state **standby**) or to safely shutdown an action (state **active**).

### 2.3 Inputs and Outputs

The inputs for the level-triggered behavior:

Input	Description
i_xEnable	Set <b>true</b> to enable the FB and start the initialization. The FB is initialized and ready ( <b>standby</b> ) if q_xStandby is <b>true</b> and q_xBusy is <b>false</b> . If set to <b>false</b> , all actions are stopped immediately, the FB is disabled and possible errors are reset.
i_xExecute	When the FB is initialized and ready ( <b>standby</b> ), setting i_xExecute to <b>true</b> starts the intended task and the FB becomes active. The output q_xActive indicates that the FB is being executed. If the FB implements a finite behavior, q_xDone indicates that the task is completed. If set to <b>false</b> , the active FB is stopped in a controlled way and the FB switches back to <b>standby</b> state. The output q_xBusy is <b>true</b> during shutdown.

The inputs for edge-triggered behavior:

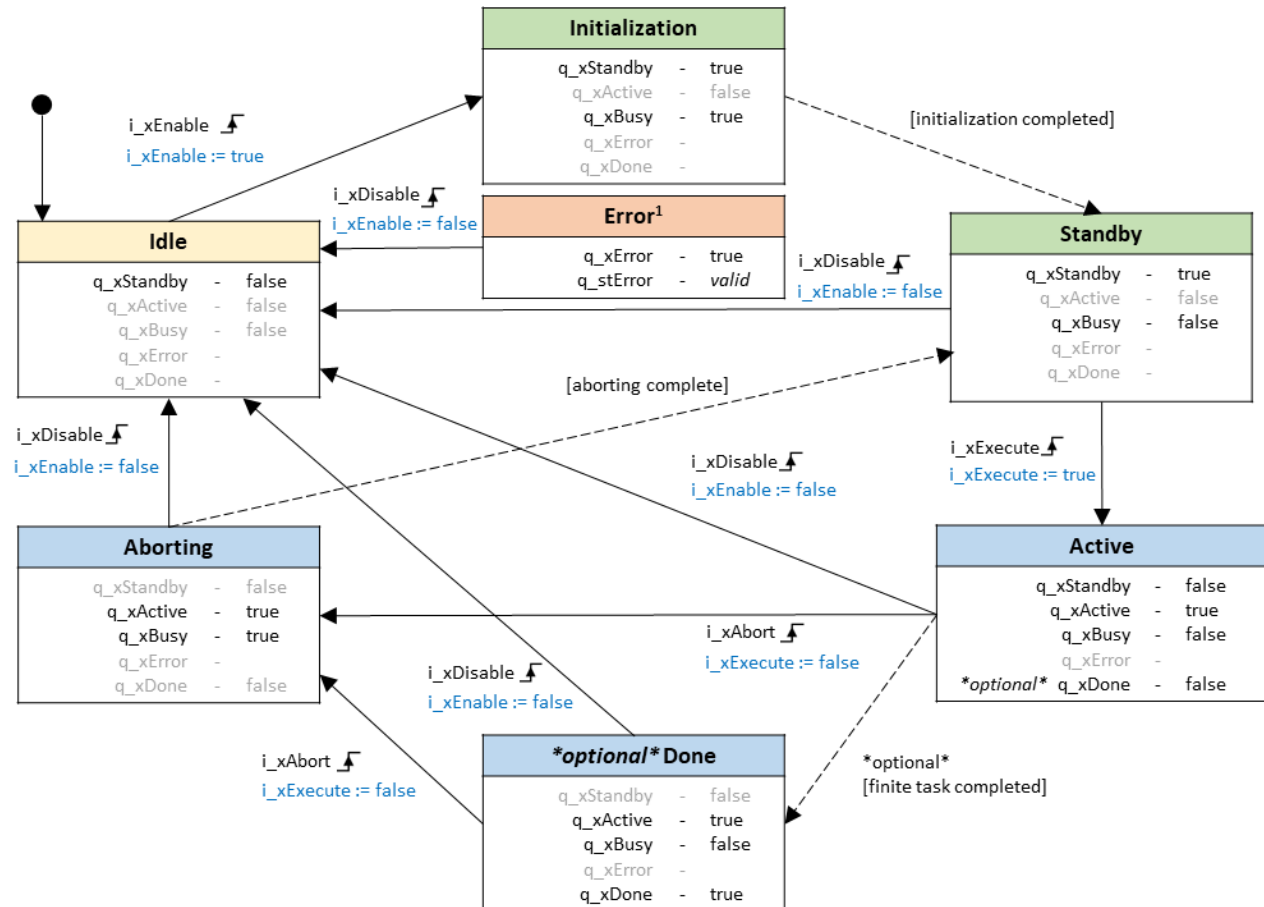
Input	Description
i_xEnable	A rising edge enables the FB and starts the initialization. The FB is initialized and ready ( <b>standby</b> ) if q_xStandby is <b>true</b> and q_xBusy is <b>false</b> .
i_xExecute	When the FB is initialized and ready ( <b>standby</b> ), a rising edge starts the intended task and the FB becomes active. The output q_xActive indicates that the FB is being executed. If the FB implements a finite behavior, q_xDone indicates that the task is completed.
i_xAbort	When the FB is active (q_xActive is <b>true</b> ), a rising edge stops the FB in a controlled way. The output q_xBusy is <b>true</b> during shutdown. The FB switches back to <b>standby</b> .
i_xDisable	A rising edge disables the FB. All actions are stopped immediately and possible errors are reset.

Outputs:

Output	Description
q_xStandby	If <b>true</b> , the FB is either initializing, which may take multiple cycles (output q_xBusy is <b>true</b> ), or already initialized and ready to be started (output q_xBusy is <b>false</b> )
q_xActive	If <b>true</b> , the FB is active. The FB is currently performing the desired task (q_xDone and q_xBusy are both <b>false</b> ), or has already completed the desired, finite task (q_xDone is <b>true</b> and q_xBusy is <b>false</b> ), or is currently shutting down an ongoing task in a controlled manner (q_xBusy is <b>true</b> )
q_xBusy	If <b>true</b> , the FB performs some internal tasks (shutdown or initialization) which may take multiple cycles. Wait until q_xBusy is <b>false</b> . Disabling (via inputs i_xEnable or i_xDisable) is always possible, but bypassing the usual shutdown sequence might result in undesired behaviour (depending on the specific function block).
q_xError	If <b>true</b> , an error has occurred and the FB is stopped.
q_stError	A struct that contains detailed information in case of an error (if q_xError is <b>true</b> ). Collecting relevant error details may take some time, so the information may not be available immediately after q_xError indicates an error. Therefore, the output q_stError.xErrorDataValid is set to <b>true</b> as soon as all information is available in q_stError. Prior to that, q_stError might contain outdated or incorrect information.
q_xDone	<b>(Finite behavior only)</b> If <b>true</b> , a finite task has been completed. Reset and back to <b>Standby</b> state (q_xStandby) by disabling i_xExecute.

## 2.4 Simplified behavior model

Combined view illustrating both level-triggered and edge-triggered behavior is shown on the figure. Blue labels at transitions with inputs correspond to level-triggered behavior<sup>1</sup>.



## 2.5 Usage of the function blocks

### ► Activation of the function block

The activation of an FB is controlled by the two boolean inputs of the function block, `i_xEnable` and `i_xExecute`. The activation process is split into two procedures and performed in the same way for every type of behavior:

1. Initialize the function block: switch from state **idle** to **standby**. By setting `i_xEnable`, parameters are validated, and the block is initialized. The parameter validation takes one single PLC cycle and results either into a start-up routine or an error. If needed, the start-up routine may perform some initial actions, which may take multiple PLC cycles and may also include interaction with devices, parameterization, communication, etc. After the input `i_xEnable` has been set, the output `q_xStandby` changes from **false** to **true**.

<sup>1</sup> The state "Error" is reached from any other state in case of a general error. For reasons of simplicity, the corresponding transitions are not shown in the diagrams.



As long as the function block performs its initialization routine, the output `q_xBusy` stays **true**. After the initialization is finished, `q_xBusy` changes back to **false**, which indicates the function block can be executed now.

2. Execute the function block and perform its intended task: switch from state **standby** to state **active**. Once the output `q_xStandby` indicates **true** and `q_xBusy` has been reset to **false**, the main operation can be started. This is always done by setting the input `i_xExecute` to **true**. The next step (internally performed) is to check whether the current process values are ok. If they are not ok, the activation results in an **error** state. If they are ok, the function block starts with its intended action. While this action is performed, the output `q_xActive` is **true**. After a function block that implements a finite behavior has performed its action completely, the output `q_xDone` is set to **true**. A continuous type function block stays in state **active** as long as no disable command is given.

► Deactivation of the function block

Once the intended action of the function block has been done (or an ongoing action needs to be aborted), the function block needs to be deactivated. The procedure differs with regard to the implemented behaviour of the function block:

- for a level-controlled function block, the input `i_xExecute` needs to be set to **false**.
- for an edge-controlled function block, a positive edge on the input `i_xAbort` is required.

Once the deactivation command has been received, the function block will change to state **standby**. In some cases, a shutdown routine is implemented and while this routine is executed, the output `q_xBusy` becomes **true** to indicate that the FB is shutting down. After the shutdown procedure has been finished, the outputs `q_xActive` and `q_xBusy` (and possibly `q_xDone`) will change to **false**, and `q_xStandby` will become **true** to indicate that the function block is now again in state **standby**.

A deactivation can also be achieved by

- setting the `i_xEnable` to **false** (in case of a level-controlled function block), or
  - providing a rising edge on the input `i_xDisable` (in case of an edge-controlled function block).
- This way of deactivating the function block sets it back to its idle state. All outputs are set to **false** and possible errors are reset. This bypasses the usual shutdown sequence and might result in undesired behaviour (depending on the specific function block). For reactivation, the complete activation procedure is required.

► Detect and reset an error

In some cases, errors might occur, and the function block will switch into an **error** state. During this state, the error information is generated and provided via the `q_stError` output. The process of collecting error information is finished once `q_stError.xErrorDataValid` becomes **true**. While this value is **false**, any information contained in `q_stError` is not valid and should not be processed, even if `q_xError` already indicates that an error has occurred. To reset the function block after an error, it needs to be disabled (`i_xEnable` = **false** or positive edge on `i_xDisable`).

### 3 libWiUr20ComRs\_232\_485\_422

The UR20-1COM-232-485-422 module features a UART for serial data transmission via either \*RS232\*, \*RS485\* or \*RS422\*.

#### 3.1 FB\_UR20\_1ComRs\_232\_485\_422\_Control

The UR20-1COM-232-485-422 control function block sends the TX data in iq\_arbTransmit-Data via the UR20-1COM-232-485-422 module or it receives serial data via the UR20-1COM-232-485-422 and provides them in iq\_arbReceiveData.

The FB\_UR20\_1ComRs\_232\_485\_422\_Control derives from WI's level controlled finite behavior model. Accordingly, the user application and the fb interact as follows:

The application provides parametrization and TX data to the fb. Then it sets i\_xEnable to **true** to initialize the FB. The FB validates the parametrization and enters its state **standby** if the parameters are valid or it enters its state **error** otherwise.

In **state** standby, the FB monitors the UR20-1COM-232-485-422 module for incoming serial data or a pending RX buffer overflow. If there are incoming serial data, the FB sets its output q\_xReceivedNewMessage to true. If the RX buffer has less than 10 free bytes left, the FB sets q\_xReceivedBufferNearlyFull to **true**.

The user application can now use the FB and the UR20-1COM-232-485-422 module to *send* or to *receive* serial data.

*Receive* serial data:

The user application has initialized the FB and the FB is in **standby**. The user application has set i\_etReadWriteMode to ET\_UR20\_1ComRs\_232\_485\_422\_ReadWriteMode.ReadFix and waits until the FB signals incoming serial data (indicated by q\_xReceivedNewMessage = **true**).

Then the user application sets i\_xExecute to true to activate the FB. The FB collects the incoming serial data from the module. While the transfer is ongoing, the FB signals its ongoing activity with q\_xBusy = **true**. After the FB has collected i\_diTransactionSize bytes of serial data from the UR20-1COM-232-485-422 module, it signals completion of its task with q\_xBusy = **false** and q\_xDone = **true**.

Note that any incoming serial data exceeding i\_diTransactionSize bytes remain in the UR20-1COM-232-485-422's RX buffer until the user application deactivates and activates the FB again to fetch the next batch of RX data from the UR20-1COM-232-485-422 module.

*Send* serial data:

The user application can select two TX modes provided by the UR20-1COM-232-485-422 module:

- direct sending: the 1COM module sends the bytes as they arrive in the module
- triggered sending: the 1COM module collects the bytes in its TX buffer and only starts sending on the serial line after the FB has told it to do so.

Choose by setting `ip_stControlParameter.etTxBufferBehavior` to `<...>.DirectSending` or `<...>.TriggeredSending`.

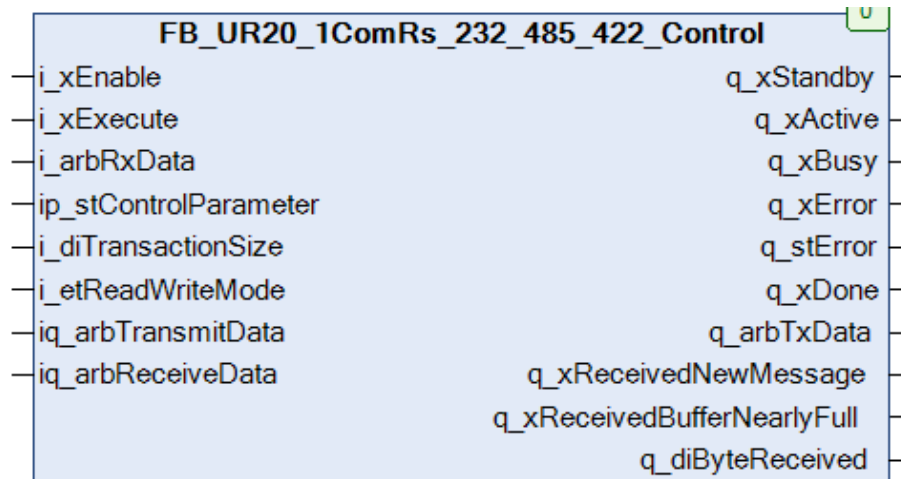
The user application has initialized the FB and the FB is in **standby**. The user application has set `i_etReadWriteMode` to `ET_UR20_1ComRs_232_485_422_ReadWriteMode.Write` and has provided serial data to send in `iq_arbTransmitData`. The user application sets `i_xExecute` to **true** to activate the FB. The FB signals its ongoing activity by setting `q_xBusy` to **true** and transfers `i_diTransactionSize` bytes of serial data to the UR20-1COM-232-485-422 module. It also starts the `ip_stControlParameter.tTransmissionWatchdogTime` transfer timeout.

After the FB has transferred `iq_arbTransmitData` bytes of TX data, it stops the transfer timeout timer.

If the user application chose triggered sending, the FB signals the UR20-1COM-232-485-422 module to send the TX data on the serial line.

The FB waits out the `ip_stControlParameter.tWaitTimeEnableTX_HWBUFFER` time, then it sets `q_xBusy` to **false** and `q_xDone` to **true** to signal completion of its activity.

If the transfer timeout timer triggers before the FB has transferred all TX data to the UR20-1COM-232-485-422 module, the FB enters its error state and reports the error on its error interface.



## Inputs and Outputs

Scope	Name	Type	Comment
Input	<code>i_xEnable</code>	BOOL	enables the function block by switching from idle to standby
	<code>i_xExecute</code>	BOOL	activates the function block by switching from standby to active
Input	<code>i_arbRxData</code>	ARRAY [0..15] OF USINT	connection to hardware / process data input
	<code>ip_stControlParameter</code>	ST_UR20_1ComRs_232_485_422_ControlParam	contains parameter for operating the module
	<code>i_diTransactionSize</code>	DINT	amount of bytes/segments to be send or read
	<code>i_etReadWriteMode</code>	ET_UR20_1ComRs_232_485_422_ReadWriteMode	defines the type of operation
Input	<code>iq_arbTransmitData</code>	POINTER TO BYTE	input for all data to be send

Scope	Name	Type	Comment
	iq_arbReceiveData	POINTER TO BYTE	output of all the data to be read
Output	q_xStandby	BOOL	waiting for activation
	q_xActive	BOOL	function block is activated
	q_xBusy	BOOL	function block is activated and doing its supposed task
	q_xError	BOOL	function block in error state
	q_stError	ST_ErrorInfo	detailed error information
Output	q_xDone	BOOL	execution has come to its supposed end
Output	q_arbTxData	ARRAY [0..15] OF USINT	connection to hardware / process data output
	q_xReceivedNewMessage	BOOL	status from module, new data available
	q_xReceivedBufferNearlyFull	BOOL	status from module, only 10 bytes left in buffer
	q_diByteReceived	DINT	number of bytes received during read command

### Possible Errors

Possible error messages on FB output q\_stError

Error interface (Reason)	Description
Communication error	Communication error detected during operation
Frame too long during read	A frame that was received by the module is too long for being read into the output array of the function block
Frame too long during write	A frame that shall be send is too long for the input buffer of the module
Transaction size is larger than the provided TX data	i_diTransactionSize is larger than the actual size of iq_arbTransmitData
Transmission watchdog exceeded	The module did not response within a specific time during a transmission
Invalid parameter	Watchdog time is too short ( must be > 0)
Watchdog for buffer flush expired	The time to flush the buffer was too long.

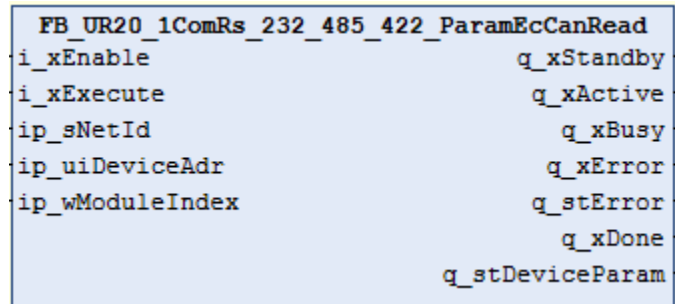
## 3.2 FB\_UR20\_1ComRs\_232\_485\_422\_ParamEcCanRead

This function block reads the parameters from a UR20\_1COM\_232\_485\_422 module connected to an EtherCAT or CANopen fieldbus coupler. The input parameters ip\_uiDeviceAdr and ip\_udiIndex are fixed after enabling the function block.

After the user activates the function block, it begins reading the module parameters. During reading, the function block checks for errors. After all parameters are read, the function block changes to state **done**.

During the transition back to **standby**, the function block deletes the output values in q\_stDeviceParam.

Note that the behavior of the function block bases on the FB\_LevelControlledFiniteBehavior Model. Please consult the behavior model library description for more information.



### Inputs and Outputs

Scope	Name	Type	Comment
Input	i_xEnable	BOOL	enables the function block by switching from idle to standby
	i_xExecute	BOOL	activates the function block by switching from standby to active
Input	ip_sNetId	T_AmsNetId	string containing the AMS network ID of the EtherCAT master device.
	ip_uiDeviceAdr	UINT	address of CANOpen or EtherCAT device [coupler]
	ip_wModuleIndex	WORD	start address/index of the module
Output	q_xStandby	BOOL	waiting for activation
	q_xActive	BOOL	function block activated
	q_xBusy	BOOL	function block is activated and doing its supposed task
	q_xError	BOOL	function block is in error state
	q_stError	ST_ErrorInfo	detailed error information
Output	q_xDone	BOOL	execution has come to its supposed end
Output	q_stDeviceParam	ST_UR20_1ComRs_232_485_422_DeviceParam	parameter set that was read

### Possible Errors

Possible error messages on FB output q\_stError

Error interface (Reason)	Description
Reading Ethercat Index Parameter	Problem triggered by communication level function block.
Connection lost during action	Connection error while reading the parameters

### 3.3 FB\_UR20\_1ComRs\_232\_485\_422\_ParamEcCanWrite

The function block writes the parameter set in ip\_stDeviceParam to a UR20\_1COM\_232\_485-422 module connected to an EtherCAT or CANopen fieldbus coupler. The input parameter ip\_stDevice and ip\_udiIndex are fixed after enabling the function block.

After the user has activated the function block, it fixes the parameters to be written, then it starts writing the parameters to the module. The function block checks for errors during writing. After the function block has written all parameters, it changes to state **done**.

Note that the behavior of the function block bases on the FB\_LevelControlledFiniteBehavior model. For more information, please consult the behavior model description.

FB_UR20_1ComRs_232_485_422_ParamEcCanWrite	
i_xEnable	q_xStandby
i_xExecute	q_xActive
ip_sNetId	q_xBusy
ip_uiDeviceAdr	q_xError
ip_wModuleIndex	q_stError
ip_stDeviceParam	q_xDone

### Inputs and Outputs

Scope	Name	Type	Comment
Input	i_xEnable	BOOL	enables the function block by switching from idle to standby
	i_xExecute	BOOL	activates the function block by switching from standby to active
Input	ip_sNetId	T_AmsNetId	string containing the AMS network ID of the EtherCAT master device.
	ip_uiDeviceAdr	UINT	address of CANOpen or EtherCAT device [coupler]
	ip_wModuleIndex	WORD	start address/index of the module
	ip_stDeviceParam	ST_UR20_1ComRs_232_485_422_DeviceParam	parameter set to be written
Output	q_xStandby	BOOL	waiting for activation
	q_xActive	BOOL	function block is activated
	q_xBusy	BOOL	function block is activated and doing its supposed task
	q_xError	BOOL	function block is in error state
	q_stError	ST_ErrorInfo	detailed error information
Output	q_xDone	BOOL	execution has come to its supposed end

### Possible Errors

Possible error messages on FB output q\_stError

Error interface (Reason)	Description
Writing EtherCAT Index Parameter	Problem triggered by communication level function block.
Connection lost during action	Connection error while writing the parameters

## 4 libWiUr204ComIOLink

This library provides function blocks for non-cyclic communication with an UR20-4COM-IO-LINK module connected to an EtherCAT fieldbus coupler.

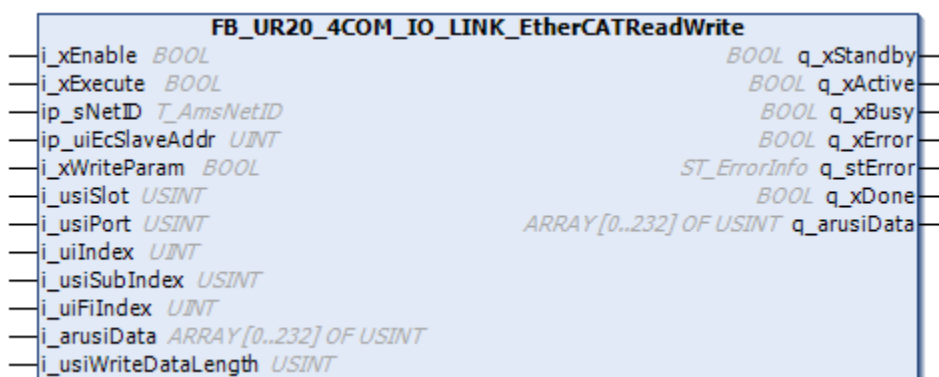
### 4.1 FB\_UR20\_4COM\_IO\_LINK\_EtherCATReadWrite

#### Functional Description

The function block provides non-cyclic communication with UR20-4COM-IO-LINK module and IO-Link slave device over Ethercat. It is possible to read and write data from/to a connected IO-Link device.

The inputs of the function block must contain the position of the IO-Link device and which parameter (Index+Subindex) shall be read or written.

Please read the IO-Link Device manual to get more information about the parameter index.



#### Possible Errors

Reason	Description
Invalid parameter	AMSNetID and Ethercat slave address need a valid handle to UR20_4COM_IO_LINK.
Invalid process value	a) <code>i_usiSlot</code> < 1 or > 64 b) <code>i_usiPort</code> < 1 or > 4 c) <code>i_uiIndex</code> < 1 d) <code>i_xWriteParam</code> is true and ( <code>usiWriteDataLength</code> <= 0 or > 2333)
Read / Write timeout	The Read/Write operation takes more than 5s, please check communication settings.
Read / write error	The sub-fb reports an error during the io-link data write or read process in active state.
Received error via IO-Link call	IO-Link Module responds with an error.

**Inputs and Outputs**

Scope	Name	Type	Comment
Output	q_xStandby	BOOL	waiting for activation
	q_xActive	BOOL	function block is activated
	q_xBusy	BOOL	function block is activated and doing its supposed task
	q_xError	BOOL	function block is in error state
	q_stError	ST_ErrorInfo	detailed error information
Input	i_xEnable	BOOL	enables the function block by switching from idle to standby
	i_xExecute	BOOL	activates the function block by switching from standby to active
Output	q_xDone	BOOL	execution has come to its supposed end
Input	ip_sNetID	T_AmsNetID	AMSNetID of Ethercat Master device
	ip_uiEcSlaveAddr	UINT	Ethercat slave address
	i_xWritePram	BOOL	False = Read; True = Write
	i_usiSlot	USINT	Hardware slot of the station where the UR20 module is connected; 1..64
	i_usiPort	USINT	hardware port of the UR20 module where IO-Link device is connected; 1 ... 4
	i_uiIndex	UINT	IO-Link Device parameter Index (See IO-Link slave manual)
	i_uiSubIndex	UINT	IO-Link Device parameter Sub-Index (See IO-Link slave manual)
	i_uiFiIndex	UINT	IO-Link FI Index (See IO-Link slave manual)
	i_arusiData	ARRAY [0..232] OF USINT	Data to be written to the IO-Link Device (see IO-Link slave manual)
	i_usiWriteDataLength	USINT	Data length to be written to the IO-Link Device
Output	q_arusiData	ARRAY [0..232] OF USINT	Output data from IO-Link device(see IO-Link slave manual)



## 5 libWiUr20ComRs\_232\_485\_422\_V2

This library provides hardware interfaces in the form of function blocks. The implemented interfaces enable the user to control the UR20-1COM-232-485-422 V2 module.

### 5.1 FB\_UR20\_1ComRs\_232\_485\_422\_V2\_Serial

The function block provides an interface to the UR20-1Com-RS-232-485-422-V2 module in the custom mode configuration. In the custom mode it is possible to create and adjust a serial communication between the module and another serial device connected via RS232, RS422 or RS485.

The plc application on the one hand and the function block on the other hand exchange serial data via two arrays with variable size (`iq_arbTransmitData`, `iq_arbReceiveData`). Please declare the array variables you connect here with sufficient size for the data you expect to transfer between the UR20-1Com-RS-232-485-422-V2 module and your serial device.

The function block expects and provides the hardware process data in two fixed size arrays: `i_arbProcessInputData` and `q_arbProcessDataOutputData`.

The input `i_etSendReceiveMode` : `ET_UR20_1ComRs_232_485_422_V2_SerialOpMode` configures the following *tasks* for serial data communication:

1. Transmit: The `FB_UR20_1ComRs_232_485_422_V2_Serial` sends a defined amount (`i_uiTransactionSize`) of data bytes (`iq_arbTransmitData`) to a connected serial device.
2. ReceiveFixedCycles: The `FB_UR20_1ComRs_232_485_422_V2_Serial` reads a defined number (`i_uiTransactionSize`) of cycles. Every cycle, a maximum of 16 bytes or one telegram can be read.
3. ReceiveTelegrams: The `FB_UR20_1ComRs_232_485_422_V2_Serial` reads one telegram.
4. ReceiveBuffer: The `FB_UR20_1ComRs_232_485_422_V2_Serial` reads all data available in the receive buffer of the module.

Once the user sets `i_xEnable` to **true**, the function block enters its **standby** state and clears the internal RX and TX buffer of the module.

After the user transitions the function block to the state **active** by also setting `i_xExecute`, the function block freezes the selected *task* and performs it until the transaction defined by the selected task is complete. The function block indicates completion of the transaction by setting its output `q_xDone` to **true**.

The function block's output `q_xRxBufferNotEmpty` indicates whether new data is available in the receive buffer of the module.

While the function block is reading data, `q_uiBytesRead` shows the amount of read bytes.

The telegram configuration is done in the hardware configuration of the module. Please refer to the u-remote manual for further details.

**Inputs and Outputs**

Scope	Name	Type	Comment
Input	i_xEnable	BOOL	enables the function block by switching from idle to standby
	i_xExecute	BOOL	activates the function block by switching from standby to active
Input	i_arbProcessInputData	ARRAY [0 .. 18] OF BYTE	input for process data send by the hardware to the plc
	i_etSendReceiveMode	ET_UR20_1ComRs_232_485_422_V2_SerialOpMode	user input to select between sending and receiving of data
	i_uiTransactionSize	UINT	amount of data that shall be handled in current action. can be segments of bytes; depends on selected operation mode
Output	q_xStandby	BOOL	waiting for activation
	q_xActive	BOOL	function block is activated
	q_xBusy	BOOL	function block is activated and doing its supposed task
	q_xError	BOOL	function block is in error state
	q_stError	ST_ErrorInfo	detailed error information
Output	q_xDone	BOOL	execution has come to its supposed end
In/Out	iq_arbTransmitData	ARRAY [*] OF BYTE	array of variable size to provide the data that shall be sent
	iq_arbReceiveData	ARRAY [*] OF BYTE	array of variable size that contains the data received after read
Output	q_arbProcessDataOutputData	ARRAY [0 .. 18] OF BYTE	output for process data sent by the plc to the hardware
	q_xRxBufferNotEmpty	BOOL	the receive buffer of the module contains data that has not been read
	q_xRxBufferFull	BOOL	the receive buffer of the module cannot take any more data, data loss possible
	q_uiBytesRead	UINT	the amount of bytes read during this transaction

## Possible Errors

The FB may indicate the following error messages on output q\_stError:

Error interface (Reason)	Description
Input data conversion	The input data is corrupted and cannot be processed further. Please check if the fb's process input data are correctly mapped to the module's process inputs.
Output data conversion	The output data is corrupted. The error is internal and cannot be fixed by the user. Please contact the Weidmüller service.
Frame to long during read	The frame that is read from the module is longer than the input array. The problem can be solved by extending iq_arbReceiveData, if possible.
Reading procedure to long, watchdog exceeded	No response from the module within timeframe during the reading sequence.
Frame to long during write	The frame that shall be written to the module is longer than the input array. Check the transaction size: you attempt to transmit data that are not within the borders of iq_arbTransmitData.
Writing procedure to long, watchdog exceeded	No response from the module within timeframe during the writing sequence.
Shutdown procedure to long, watchdog exceeded	No response from the module within timeframe during the shutdown sequence
No response while clearing the module.	No response from the module during flushing the receive and transmit buffers.

## 5.2 FB\_UR20\_1ComRs\_232\_485\_422\_V2\_ModbusMaster

The function block provides an interface to the UR20-1Com-RS-232-485-422-V2 module in its modbus master configuration. The module allows the user to parametrize up to 255 modbus master channels to exchange data. Every channel has its own message configuration with a function code, a data offset, a length and a cycle time. The function block transfers these parameters to the UR20-1Com-RS-232-485-422-V2 module in the standby initialize state and before the main action is started.

The channels are configured with a variable array named `iq_arstModbusChannels`. To use it, the user needs to define its size. Every communication channel used, needs an array element. If for example the user wants to use three channels of the module, the size needs to be configured like `[0..2]`. The elements contain both, the channel configuration and the modbus register/coil data. After the configuration data is set, the user needs to enable the function block with the `i_xEnable` input. During this stage, the block resets all module data and writes the new configuration. A read data channel cyclically reads the data specified by variables `uiOffsetRead` and `uiLengthRead` in `iq_arstModbusChannels` and stores them in `arwRegisterData` or `arxCoilData` in `iq_arstModbusChannels`. A write data channel has two modes of operation: cyclic or on demand.

Cyclic: set `iq_arstModbusChannels[<channel number>].uiCycleTime` to the desired cycle time in milliseconds. The UR20\_1COM\_232\_485\_422\_V2 module will send the associated data every `iq_arstModbusChannels[<channel number>].uiCycleTime` milliseconds.

On demand: set `iq_arstModbusChannels[<channel number>].uiCycleTime` to zero. The UR20\_1COM\_232\_485\_422\_V2 module will only send data when the FB\_UR20\_1ComRs\_232\_485\_422\_V2 transfers new data to the module, i.e., when the associated `xTriggerTransaction` flag is set.

After the communication channels were successfully parametrized, the function block confirms entry into the state **standby** by setting `q_xStandby` and waits for the execute signal `i_xExecute` to start its main operation in its state **active**. While the block is **active**, the channel configuration cannot be changed. The FB starts the Modbus communication mode of the UR20\_1COM\_232\_485\_422\_V2 module, then it cyclically iterates over the configured Modbus channels. For those channels with a read configuration, the FB transfers channel read data from the UR20\_1COM\_232\_485\_422\_V2 module into `arwRegisterData` or `arxCoilData` in `iq_arstModbusChannels`. For those channels with a write configuration, the FB checks if `iq_arstModbusChannels[<channel number>].xTriggerTransaction` is set. If so, it transfers `arwRegisterData` or `arxCoilData` in `iq_arstModbusChannels[<channel number>]` to the associated channel in the UR20\_1COM\_232\_485\_422\_V2 module and clears the associated `xTriggerTransaction` flag.

The behavior is depending on the Modbus function codes that the user has provided in the channel configuration. Some function codes exhibit write behavior and others exhibit read

behavior. Thus, some channels read data from, and others write data to the Modbus slave devices connected to the UR20\_1COM\_232\_485\_422\_V2 module. If there are both write- and read channels, the FB will alternate between processing a write channel, then a read channel, then a write channel and so on.

To send *new* cyclic data or any on-demand data, write a new payload to `arwRegisterData` or `arxCoilData` in `iq_arstModbusChannels`, or just keep the old payload. Set `iq_arstModbusChannels[<channel number>].xTriggerTransaction` to **true**. The function block finishes any ongoing update of a read channel's data, then it writes the new data to the module. After the FB has written the new data to the module, it sets `iq_arstModbusChannels[<channel number>].xTriggerTransaction` to **false**. For channels that are configured as "write on demand", the UR20\_1COM\_232\_485\_422\_V2 module will send the data after you have set `iq_arstModbusChannels[<channel number>].xTriggerTransaction` to true and the FB has written the data to the module. If the transmit buffer is empty, the UR20\_1COM\_232\_485\_422\_V2 module will send the data right away. If another channel is transmitting cyclic data, the UR20\_1COM\_232\_485\_422\_V2 module will send the on-demand data as soon as the ongoing transmit has finished.

The timing of the data transfer depends on the data size and on the cycle time of your field bus.

### Inputs and Outputs

Scope	Name	Type	Comment
Input	i_xEnable	BOOL	enables the function block by switching from idle to standby
	i_xExecute	BOOL	activates the function block by switching from standby to active
Input	i_arbProcessInputData	ARRAY [0 .. 18] OF BYTE	input for process data send by the hardware to the plc
Output	q_xStandby	BOOL	waiting for activation
	q_xActive	BOOL	function block is activated
	q_xBusy	BOOL	function block is activated and doing its supposed task
	q_xError	BOOL	function block is in error state
	q_stError	ST_ErrorInfo	detailed error information
Output	q_xDone	BOOL	execution has come to its supposed end
In/Out	iq_arstModbusChannels	ARRAY [*] OF STRUCT	contains the channel configuration, control bits and transaction data
Output	q_arbProcessDataOutput Data	ARRAY [0 .. 18] OF BYTE	output for process data send by the plc to the hardware

### Possible Errors

The FB indicates the following error messages on output q\_stError:

Error interface (Reason)	Description
input data conversion	The input data is corrupted and cannot be processed further. It should be considered to check the mapped data from the module
output data conversion	The output data is corrupted. The error is internal can not be fixed by the user. Weidmüller service needs to be contacted
no response while clearing or configuring the module	Before the module can be used, all channels are configured by this fb. the module stopped responding during this procedure.
reading/writing procedure to long, watchdog exceeded	The module stopped the communication during reading and writing the channels. The modbus may continue running, as the error only addresses the plc to module communication.
shutdown procedure to long, watchdog exceeded	The module stopped the communication during the shutdown sequence. The modbus may continue running, as the error only addresses the plc to module communication.
wrong channel response during read	At the beginning of every readout of a channel, the module writes the current channelnumber into the datastream. during this procedure an error happened.
read function code occurred during write sequence	Somehow a read functioncode was found during a write sequence. Internal error.
difference in channel configuration between module and fb detected	The check sequence after channel configuration returned a wrong amount of configured channels.

## 5.3 FB\_UR20\_1ComRs\_232\_485\_422\_V2\_ModbusSlave

This function block provides an interface to the UR20-1Com-RS-232-485-422-V2 module in its Modbus slave configuration. In this mode the module acts as a Modbus slave device.

The module allows the user to parametrize up to 255 Modbus slave channels to exchange data. Each channel has its own data register which can be accessed via Modbus commands. The data is stored in form of registers (16Bit) that can be accessed either as coils or registers.

The channels are configured in the variable-size array `iq_arstModbusChannels`: The user configures one array element per communication channel. E.g. for usage of three channels of the module, implement an array `[0..2]` and connect it to `iq_arstModbusChannels`.

The array element associated to a channel contains both the configuration and the Modbus register/coil data. The user first sets up the configuration data, then enables the function block by setting `i_xEnable` to **true**. The function block will then reset all module data and write the new configuration to the module.

After the function block has parametrized the desired communication channels, the function block confirms entry into its state **standby** by setting output `q_xStandby` to **true**. Next, it waits for the signal `i_xExecute` before it enters the state **active** and starts its main operation. While the block is **active**, the channel configuration cannot be changed.

During the startup of the main operation the function block initially writes the register data in `iq_arstModbusChannels[<channel number>].wRegisters` to the module.

In ongoing **active** state, the function block cyclically checks the state of xTriggerTransaction for each member of iq\_arstModbusChannels. The function block exchanges data with the 1COM V2 module when xTriggerTransaction of a channel is set to **true**. The transfer direction depends on iq\_arstModbusChannels[<channel number>].xTransactionRead:

If iq\_arstModbusChannels[<channel number>].xTransactionRead is **true**, the function block will read channel data from the 1COM V2 module and store it in iq\_arstModbusChannels[<channel number>].wRegisters.

If iq\_arstModbusChannels[<channel number>].xTransactionRead is false, the function block will write channel data from iq\_arstModbusChannels[<channel number>].wRegisters into the 1COM V2 module.

After the data transfer is complete, the function block sets the xTriggerTransaction flag for that channel back to false.

To use Modbus function codes FC1, FC3, FC5, FC6, FC15, FC16, FC23 activate the following flags in the channel configuration: iq\_arstModbusChannels[<channel number>].xRead and iq\_arstModbusChannels[<channel number>].xWrite.

To use Modbus function code FC2, FC4, enable iq\_arstModbusChannels[<channel number>].xRead only.

### Inputs and Outputs

Scope	Name	Type	Comment
Input	i_xEnable	BOOL	enables the function block by switching from idle to standby
	i_xExecute	BOOL	activates the function block by switching from standby to active
Input	i_arbProcessInputData	ARRAY [0 .. 18] OF BYTE	input for process data sent by the hardware to the plc
Output	q_xStandby	BOOL	waiting for activation
	q_xActive	BOOL	function block is activated
	q_xBusy	BOOL	function block is activated and doing its supposed task
	q_xError	BOOL	function block is in error state
	q_stError	ST_ErrorInfo	detailed error information
Output	q_xDone	BOOL	execution has come to its supposed end
In/Out	iq_arstModbusChannels	ARRAY [*] OF STRUCT	contains the channel configuration, control bits and transaction data
Output	q_arbProcessDataOutputData	ARRAY [0 .. 18] OF BYTE	output for process data sent by the plc to the hardware

## Possible Errors

The FB indicates the following error messages on output q\_stError:

Error interface (Reason)	Description
Input data conversion	The input data is corrupted and cannot be processed further. Please check the 1COM V2 module's process data mapping.
Output data conversion	The output data is corrupted. The error is internal can not be fixed by the user. Please contact the Weidmüller service.
no response while clearing or configuring the module	Before the module can be used, all channels are configured by this fb. the module stopped responding during this procedure.
read/write procedure too long, watchdog timeout exceeded	The module stopped the communication during reading and writing the channels. The modbus may continue running, as the error only addresses the plc to module communication.
shutdown procedure too long, watchdog timeout exceeded	The module stopped the communication during the shutdown sequence. The modbus may continue running, as the error only addresses the plc to module communication.
wrong channel response during read	At the beginning of every readout of a channel, the module writes the current channel number into the datastream. during this procedure an error happened.
difference in channel configuration between module and fb detected	The check sequence after channel configuration returned a wrong amount of configured channels.



## 6 libWlUr20ModbusRTUMaster

### 6.1 FB\_ModbusRTUMaster

The function block FB\_ModbusRTUMaster implements a Modbus RTU master that provides communication with a Modbus slave via u-remote module UR20\_1COM\_232\_485\_422. This function block derives from the level-controlled finite behavior model. Please consult the Standard Behavior library documentation for further information.

The following paragraphs explain the interaction of the user's application (referred to as "user") with the function block ("FB").

The user enables the FB and the FB does a check of the `ip_stParameterInputs` parameters during the transition from **idle** to **standby**. If the parameters are not ok, the FB will enter its state **error** instead of **standby**.

The user activates the FB and the FB does a process value check during the transition from **standby** to **active**. If the process values are not ok, the FB will enter its error state instead of the state **active**.

In **active** state, the FB performs a ModBus RTU data transfer: Depending on the ModBus function code that the user has provided, the FB either sends data to a ModBus RTU slave device or it reads data from a ModBus RTU slave device. After completion of the transfer, the FB enters its state **active-done**. If the transfer fails, the FB enters the state **error**.

If the user wants to initiate another transfer, the user de-activates the FB, waits for the FB to enter its state **standby** and then sets up the new transfer and activates the FB, again.

If the user wants the FB to leave the state **error**, the user disables the FB and enables it, again. In state **error**, the FB will provide additional error information on its output `q_stError`.

*Send data:*

The user sets up the transfer with the following information on the inputs of the FB:

- one of the ModBus RTU Function Codes 5, 6, 8, 15 or 16 on the FB's input `i_enFcCode`.
- the ModBus RTU slave device's address on the FB's input `i_usiSlaveAdr`.
- the start address of the ModBus RTU registers to write on the FB's input `i_uiStartAdr`.
- the amount of data to send on the FB's input `i_uiQuantityData`.
- the data to send in `iq_arxCoilData` for the ModBus RTU Function Codes 5 and 15, or
- the data to send in `iq_arwRegisterData` for the ModBus RTU Function Codes 6, 8 and 16.

Then the user activates the FB and waits until the FB signals completion of the transfer, or its failure.

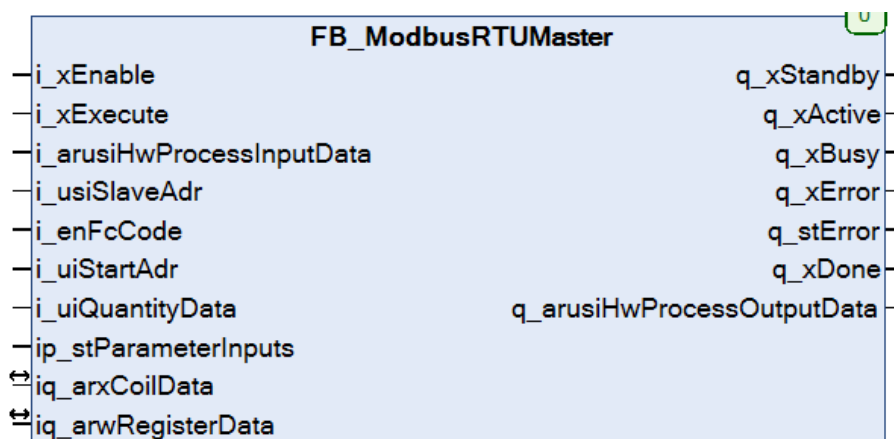
*Read data:*

The user sets up the transfer with the following information on the inputs of the FB:

- one of the ModBus RTU Function Codes 1, 2, 3 or 4 on the FB's input i\_enFcCode.
- the ModBus RTU slave device's address on the FB's input i\_usiSlaveAdr.
- the start address of the ModBus RTU registers to read, on the FB's input i\_uiStartAdr.
- the amount of data to fetch on the FB's in-/output i\_uiQuantityData.

Then the user activates the FB and waits until the FB signals completion of the transfer, or its failure. After successful completion of the transfer, the FB provides the read data on its associated in-/output. This is:

- iq\_arxCoilData for the ModBus RTU Function Codes 1 and 2 or
- iq\_arwRegisterData for the ModBus RTU Function Codes 3 and 4.



### Inputs and Outputs

Scope	Name	Type	Initial	Comment
Input	i_xEnable	BOOL		enables the function block by switching from idle to standby
	i_xExecute	BOOL		activates the function block by switching from standby to active
Input	i_arusiHwProcessInputData	ARRAY [0..15] OF USINT		array of process inputs from UR20-1Com Module
	i_usiSlaveAdr	USINT	1	modbus slave address 1-247
	i_enFcCode	ET_UR20_ModbusRTU_1ComRs_FunctionCode		modbus function code
	i_uiStartAdr	UINT	0	starting address of modbus register 0-65535
	i_uiQuantityData	UINT	1	quantity of data 1-2000 coils or 1-125 registers
	ip_stParameterInputs	ST_UR20_ModbusRTU_1ComRs_ControlParameter		struct contains control parameter
Inout	iq_arxCoilData	ARRAY [1..2000] OF BOOL		read and send buffer for bit-oriented Modbus coils and discrete inputs
	iq_arwRegisterData	ARRAY [1..125] OF WORD		read and send buffer for word-oriented Modbus registers

Scope	Name	Type	Initial	Comment
Output	q_xStandby	BOOL		waiting for activation
	q_xActive	BOOL		function block is activated
	q_xBusy	BOOL		function block is activated and doing its supposed task
	q_xError	BOOL		function block is in error state
	q_stError	ST_ErrorInfo		detailed error information
Output	q_xDone	BOOL		execution has come to its supposed end
Output	q_arusiHwProcessOutputData	ARRAY [0..15] OF USINT		array of process outputs to UR20-1Com Module

### Supported Modbus Function Codes

Function Code	Register Type	Explanation
1	Read Coil Status	Reads binary outputs (coils) from a connected slave. The data is stored in Array iq_arxCoilData[1..2000] of BOOL.
2	Read Input Status	Reads binary inputs from a connected slave. The data is stored in Array iq_arxCoilData[1..2000] of BOOL.
3	Read Holding Registers	Reads register-data from a connected slave. The data is stored in Array iq_arwRegisterData[1..125] of WORD.
4	Read Input Registers	Reads input registers from a connected slave. The data is stored in Array iq_arwRegisterData[1..125] of WORD.
5	Write Single Coil	Sends a binary output (Coil) to a connected slave. The value from array iq_arxCoilData [1] is written to the slave.
6	Write Single Register	Sends a single data word to a connected slave. The value from array iq_arwRegisterData[1] is written to the slave.
8	Diagnostics	Sends a diagnostics request with a user defined subfunction code to a connected slave. The subfunction code is passed to the function by the parameter i_uiStartAdr. Additional data can be passed by Array iq_arwRegisterData[1].
15	Write Multiple Coils	Sends several binary outputs (Coils) to a connected slave. The data must be provided in array iq_arxCoilData[1..2000] of BOOL. The maximum number of coils are 0x07B0.
16	Preset Multiple Registers	Sends data to a connected slave. The data must be provided in array arwRegisterData[1..125] of WORD. The maximum number of data are 0x007B.

### Possible Errors

Possible error messages on FB output q\_stError:

Error interface (Reason)	Description
Invalid parameter	The FB's parameters must be.. a. ip_stParameterInputs.tReqTimeout > T#0S b. ip_stParameterInputs.tHwMemFlushTimer > T#0S
Invalid process value	Please check the FB's input values and -ranges to be.. a. iq_arxCoilData is a valid reference b. iq_arwRegisterData is a valid reference c. 1 <= i_usiSlaveAdr <= 247

Error interface (Reason)	Description
	d. $1 \leq i\_uiQuantityData \leq 2000$ for FC1 and FC2 e. $1 \leq i\_uiQuantityData \leq 125$ for FC3 and FC4 f. $1 \leq i\_uiQuantityData \leq 1968$ for FC15 g. $1 \leq i\_uiQuantityData \leq 123$ for FC16 h. $i\_enFcCode$ is in the supported-list above
Error 1COM driver	Evaluate the forwarded error message.
Timeout 1Com-Driver	Sub-FB doesn't return an answer, please check the MB-Slave communication settings.
Invalid Response	Modbus Slave returns an invalid value
Slave Error Response	Modbus Slave returns an Error code

## 7 libWiUr20Diag

This library provides diagnosis message handling for WI u-remote modules attached to an EtherCAT fieldbus coupler.

### 7.1 FB\_UR20\_DiagData\_EtherCAT

Many u-remote modules can generate diagnostic data, e.g. in the event of a channel error. The diagnostic data is always an array of 0..46 bytes. Please take a look at the u-remote manual for further information for which modules provide diagnostic alarm messages and for the meaning of the diagnostic messages.

The function block FB\_UR20\_DiagData\_Ethercat can read the diagnostic data from a u-remote module connected to an EtherCAT coupler. The diagnostic functionality must be activated inside the EtherCAT coupler. Without activation the function block returns an error message.

The function block reads all available messages at once and put the messages in the array q\_arstDeviceDiagData. The output array member q\_arstDeviceDiagData[0] shows the newest, q\_arstDeviceDiagData[20] the oldest diagnostic message.

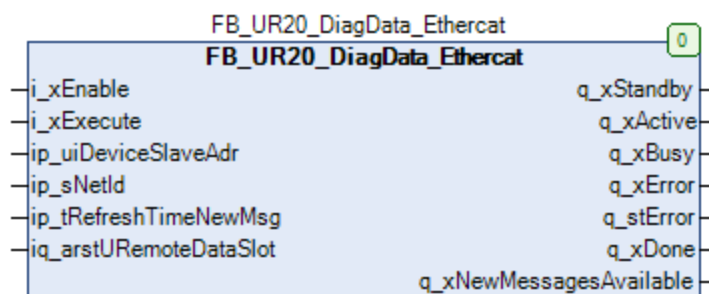
The other output q\_usiDiagDataCnt gives an information how many diagnostic data messages are available and copied to the output array.

After all messages are loaded, the function block changes to "done" state.

The input parameter ip\_uiDeviceSlaveAdr must contain the ethercat slave address of the coupler.

Supported Modules:

- 0x62=UR20\_1Com\_V2,
- 0x70=Digital Input,
- 0x71=Analog Input,
- 0x73=Analog Output,
- 0x74=Analog Input UI,
- 0x75=Analog Input PWR,
- 0x76=Counter /Funktionmodul,
- 0x77=SIL-4DI/4DO,
- 0x78=SIL-PF-Baugruppe,
- 0x79=Communication Modules,
- 0x7A=SIL-8DI,
- 0x7B=IO-Link,
- 0x7C=SAI Counter,
- 0x7D=Stepper Module



## Inputs and Outputs

Scope	Name	Type	Init	Comment
Input	i_xEnable	BOOL		enables the function block by switching from idle to standby
	i_xExecute	BOOL		activates the function block by switching from standby to active
Input	ip_uiDeviceSlaveAdr	UINT		device slave address from Ethercat coupler
	ip_sNetId	T_AMSNETID		device slave AMS Net ID from Ethercat coupler
	ip_tRefreshTimeNewMsg	TIME		refresh time, if new diag messages are available
Inout	iq_arstURemoteDataSlot	ARRAY[*] OF ST_UR20_DiagDataEcUremote		array of diag data for the single slots
Output	q_xStandby	BOOL		waiting for activation
	q_xActive	BOOL		function block is activated
	q_xBusy	BOOL		function block is activated and doing its supposed task
	q_xError	BOOL		function block is in error state
	q_stError	ST_ErrorInfo		detailed error information
Output	q_xDone	BOOL		execution has come to its supposed end
Output	q_xNewMessagesAvailable	BOOL		new diag message is available

## Possible Errors

Possible error messages on FB output q\_stError:

Error interface (Reason)	Description
Invalid parameter	The parameter "ip_uiDeviceSlaveAdr" requires an input mandatorily.
Invalid process value	"ip_uiDeviceSlaveAdr": The input device could not be found during transition to state active.
sub-FB returns an error	Please check the error parameter output for more details.